# An Automated Blackbox Noncompliance Checker for QUIC Server Implementations

Kian Kai Ang
The University of Adelaide
Australia
kiankai.ang@adelaide.edu.au

Guy Farrelly
The University of Adelaide
Australia
guy.farrelly@adelaide.edu.au

Cheryl Pope
The University of Adelaide
Australia
cheryl.pope@adelaide.edu.au

Damith C. Ranasinghe
The University of Adelaide
Australia
damith.ranasinghe@adelaide.edu.au



## ABSTRACT

We develop QUICTester, an automated approach for uncovering non-compliant behaviors in the ratified QUIC protocol implementations (RFC 9000/9001). QUICTester leverages active automata learning to abstract the behavior of a QUIC implementation into a finite state machine (FSM) representation. Unlike prior noncompliance checking methods, to help uncover state dependencies on event timing, QUICTester introduces the idea of state learning with event timing variations, adopting both valid and invalid input configurations, and combinations of security and transport layer parameters during learning. We use pairwise differential analysis of learned behaviour models of tested QUIC implementations to identify non-compliance instances as behaviour deviations in a property-agnostic way. This exploits the existence of the many different QUIC implementations, removing the need for validated, formal models. The diverse implementations act as cross-checking test oracles to discover non-compliance. We used QUICTester to analyze 186 learned models from 19 QUIC implementations under the *five* security settings and discovered 55 implementation errors. Significantly, the tool uncovered a QUIC specification ambiguity resulting in an easily exploitable DoS vulnerability, led to 5 CVE assignments from developers, and two bug bounties thus far.

**Code & PoCs**: https://github.com/QUICTester.

## CCS CONCEPTS

• **Networks** → **Network protocols; Protocol testing and verification**; • **Security and privacy**;

## KEYWORDS

QUIC, Noncompliance, Differential Analysis, Active Learning


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Ratified in May 2021, QUIC is a performance-optimized, secure, reliable transport protocol for the Internet and a core part of the HTTP/3 protocol. QUIC is a ground up re-design aiming to reduce latency and connection overhead associated with the use of TLS (Transport Layer Security) [22] over TCP for secure transport [52] and achieve *inherently* secure communication channels—ensuring message confidentiality, integrity, and availability for Internet applications. According to [5], as of November 2023, 27% of all websites use HTTP/3 employing QUIC for the transport protocol—including Google, Meta, Amazon and all major browsers—with use cases also extending to the Domain Name System (DNS) [34]. Further, with the significant growth in Internet of Things applications, projected to be more than 29 billion by 2027 [36], we can expect QUIC to dominate secure, reliable data transfer over the future Internet [25, 32, 45].

Despite significant efforts to investigate secure protocols—such as TLS [12, 14, 20, 59], Datagram Transport Layer Security (DTLS) [28, 29], OpenVPN [19] and the 802.11 4-Way Handshake [48]—there is a gap in reliable *tools* to scrutinize specification conformance of QUIC implementations and consequential vulnerabilities. Our motivation is to address this gap.

Prior to ratifying the QUIC specification [40, 62], early efforts made advances to develop methods and tools to validate QUIC implementations [18, 42, 49, 54]. But, the tools are specific to Google-QUIC, are no longer actively maintained, not open source or are limited in their scope and suitability for evaluating the ratified QUIC protocol as we discuss in Section 7. An effective and noncompliance testing method is important for verifying the security promised by QUIC is delivered by implementations.

In the absence of an open-source testing method for the QUIC specification [40, 62] to identify non-compliances and potential security vulnerabilities, *our work presents the first open-source, comprehensive design and implementation of a framework for analyzing implementations of the ratified QUIC standard.*

**Our Work.** Our goal is to provide a tool that automates the task of uncovering: i) non-conforming protocol behaviors; and ii) security vulnerabilities exploitable by crafting specific message sequences. To this end, we design and build QUICTester—a comprehensive, automated, black-box tester for uncovering non-compliant behaviors and security vulnerabilities in QUIC implementations. Notably, QUICTester has the desirable property of being agnostic to the

QUIC implementation and run-time environment, such as the programming language, operating system, and CPU instruction set architecture. We address a number of key challenges in realizing an effective *QUIC-specific* noncompliance checker.

*(1) Automatically Learning a Behavior Model Under Valid and Invalid Inputs.* Analyzing non-conforming behavior of a protocol implementation requires construction of a model of the underlying implementation and comparison with a formal model of a protocol by a domain expert. The process requires a significant manual effort and scaling the effort to a large number of implementations is often impractical. In addition to potential human error, the process is made more difficult given specification ambiguities, under-specification, the length of specifications (the QUIC specification spans 80,000 words incorporating five different security levels, as well as several security and transport parameter options).

Our approach uses active automata learning to overcome the lengthy task of eliciting the behavior abstractions of a QUIC implementation to a finite state machine (FSM) representation. Active automata learning aims to infer a system's state space and possible transitions between those states by sending a series of input sequences to the system and observing the corresponding outputs [53, 63]. In the process, all input sequences and corresponding output sequences explored in a model learning phase are inferred into an FSM description of the target system. Although past efforts used active automata learning to build testing tools for network protocols [6, 19, 20, 27, 28, 30, 35, 48, 61], an active learner for the QUIC specification does not yet exist. We design such a Learner.

To allow automata learning algorithms to generate tests—a series of input sequences, *we construct the first comprehensive QUIC-specific Learner for all secure handshake configurations through careful examination of the QUIC RFC 9000/9001. The result is a highly expressive learner capable of generating both valid and invalid packet configurations as well as various security and transport parameter combinations in the learning phase.*

*(2) Learning Time-Dependent Behavior Models.* Further, we recognize, network protocols like QUIC, are typically time sensitive due to timeout behaviors. Therefore, we can expect timing variations, such as intervals between packet transmissions, to impact protocol behavior and potentially expose hidden states and weaknesses in protocol implementations that may be exploitable. In contrast to prior methods to analyze protocols, *we introduce the idea of exploring temporal dependencies on protocol states. We parameterize the symbols with time to allow the learner to self-select timing variations.*

*(3) Test Harness.* Because the learning algorithms are protocol agnostic and only operate on the symbols and parameters, a protocol-specific test harness is needed to translate parameterized symbols from a learner into protocol messages and vice versa. This harness manages the interactions between a model Learner and the target QUIC implementation under test.

The complexity of the QUIC protocol—supporting 5 security levels, with the combination of transport parameters and cryptographic negotiations to manage various types of packets and frames—makes building a test harness a significant challenge. *We built a comprehensive QUIC protocol test harness to enable testing all, five secure handshake configurations in QUIC implementations.*

*(4) Automating Analysis.* A significant challenge is to identify non-compliant behavior. To reduce the cumbersome, error-prone, manual effort in the analysis phase, we combine two strategies to achieve an effective automated analysis method. *First*, we propose a set of *optimizations* to eliminate redundant information from learned models while preserving the captured behavior. *Second*, we construct a differential analysis method—pair-wise testing—to automatically identify non-conforming behaviors. Differential analysis simplifies the task of identifying non-compliant behavior to the task of identifying *deviating behaviors* based on comparing models against each other. Through the results of differential testing and our manual analysis and validation, we are able to contribute a curated library of reference FSM models for the QUIC specification for each security configuration. The models represent behavior that conforms to the specification. *These models serve developers to employ QUICTester to effectively and efficiently identify non-conformance behaviors of a target implementation* (see Fig. 10).

**Scope.** We focus our tests on the more impactful server-side implementations of QUIC as server failures affect multiple active connections; for example, Denial of Service attacks. Notably, the same protocol library is used by clients and servers. Further, our focus is on security. Hence, we test the handshake component *crucial* for establishing secure, multi-stream, connections with QUIC.

**Contributions.** In this work, in summary:

- We propose QUICTester, a blackbox testing framework for QUIC to *automatically* identify specification deviation (non-compliance checking), uncover logical flaws (functional bugs) and security vulnerabilities without needing a formal reference model description.
- We introduce the idea of learning a model under *event timing variations*. We craft a conceptually simple, yet protocol-agnostic means to achieve it.
- We design and implement a comprehensive QUIC-specific Test Harness adhering to RFC 9000/9001 to support the complex Learner sequence combinations (valid, invalid, time dependent message with varying protocol parameters, both transport and security) and all of the security configurations to test QUIC implementations—see Table 1.
- To alleviate the manual analysis burden, we automate analysis of learned models. We manually validate our method with 186 learned models and curate conforming models for various security configurations to serve as reference models. The reference models with QUICTester can support practitioners' use of our tool to efficiently test and analyze QUIC implementation targets (see Section 7 and Fig. 10).
- We open-source QUICTester, at https://github.com/QUICTester.

**Findings.** We used QUICTester with publicly available QUIC server releases to assess its effectiveness and help improve security and interoperability of QUIC implementations.

- We test 19 QUIC implementations summarized in Table 1 under the *five* different *security* configurations from a range of providers. We analyzed 186 learned models to uncover 55 faults.

Our work has been validated with a *bug bounty award* and 5 CVE assignments thus far—see Table 2, and in *Appendix* Table 7.

- Importantly, we discovered QUIC specification ambiguities in connection management that expose a DoS vulnerability and propose an amendment to address the issue—see Section 5.4.

**Table 1: Tested QUIC implementations and their *security* configurations. These include Google (Google-quiche), Mozilla (Neqo), Lite Speed Technologies (LSQUIC), Meta (Mvfst), Microsoft (MsQuic), Cloudflare (Quiche), Amazon (S2n-quic) and Alibaba (XQUIC).**

| Name | Commit Version | Tested Configurations | URL |
|---|---|---|---|
| Aioquic | 239f99b8 | Basic, Retry, PSK | https://github.com/aiortc/aioquic |
| Google-quiche | 42dab6be | Basic, ClientAuth, PSK | https://github.com/google/quiche |
| Kwik | 745fd4e2 | Basic, Retry, PSK | https://bitbucket.org/pjtr/kwik/src/master/ |
| LSQUIC | 1b113d19 | Basic, PSK | https://github.com/litespeedtech/lsquic |
| MsQuic | 5c070cdc | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/microsoft/msquic |
| Mvfst | a76144e1 | Basic, ClientAuth, PSK | https://github.com/facebook/mvfst |
| Neqo | aaabc1c1 | Basic, Retry | https://github.com/mozilla/neqo |
| Ngtcp2 | f65399b5 | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/ngtcp2/ngtcp2 |
| Picoquic | d2f01093 | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/private-octopus/picoquic |
| PQUIC | 841c8228 | Basic, PSK | https://github.com/p-quic/pquic |
| Quant | 511d91c3 | Basic, Retry, PSK | https://github.com/NTAP/quant |
| Quiche | 24a959ab | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/cloudflare/quiche |
| Quiche4j | ea5effce | Retry | https://github.com/kachayev/quiche4j |
| Quic-go | f78683ab | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/quic-go/quic-go |
| Quicly | d44cc8b2 | Basic, Retry, PSK | https://github.com/h2o/quicly |
| Quinn | 4395b969 e1e1e6e3 | Basic, Retry, ClientAuth, RetryClientAuth, PSK | https://github.com/quinn-rs/quinn |
| Quiwi | b7b5dadb | Basic, Retry, PSK | https://github.com/goburrow/quic |
| S2n-quic | ec651875 | Basic, Retry | https://github.com/aws/s2n-quic |
| XQUIC | 00f62288 | Basic, PSK | https://github.com/alibaba/xquic |

**Basic**: Basic handshake. **Retry**: Handshake with client address validation. **ClientAuth**: Handshake with client authentication. **RetryClientAuth**: Handshake with client address validation and authentication. **PSK**: Handshake with pre-shared key.

**Responsible Disclosure.** Following the practice of responsible disclosure, we shared our findings with corresponding development teams by sending bug reports to vendors/developers following their reporting policies. We summarize the current state of disclosures and vendor responses in Table 7 within the *Appendix*.

## 2 BACKGROUND

We provide a brief overview of the QUIC protocol handshake (secure connection establishment prior to application data exchange) and active automata learning addbefore delving into our framework in Section 3.

### 2.1 QUIC Protocol Transport Layer Security

To understand the process of secure connection establishment in QUIC, we begin with a primer on connection establishment in a client-server setting. An entity using QUIC must complete a handshake with its endpoint before it can communicate. QUIC combines both transport and cryptographic parameter negotiations into a single handshake. Our work focuses on this QUIC handshake, which is responsible for establishing *secure* multi-stream connections.

QUIC provides **five** different security configurations, we briefly discuss each and the respective packets and frames employed. A simplified illustration of QUIC handshake message exchange for the 5 different secure connection configurations is shown in Figure 1. These handshakes include:

(1) Basic.
(2) Client address validation (without client authentication).
(3) Client authentication and without address validation.
(4) Client address validation and authentication.
(5) Handshake with a pre-shared key.

The handshakes are realized in QUIC using: Initial packets, Handshake packets, 1-RTT (Round Trip Time) packets and Retry packets. These packets carry the necessary information in frames and QUIC messages to complete the transport and cryptographic parameter negotiations. Each frame is defined in [40, Section 12.4] to

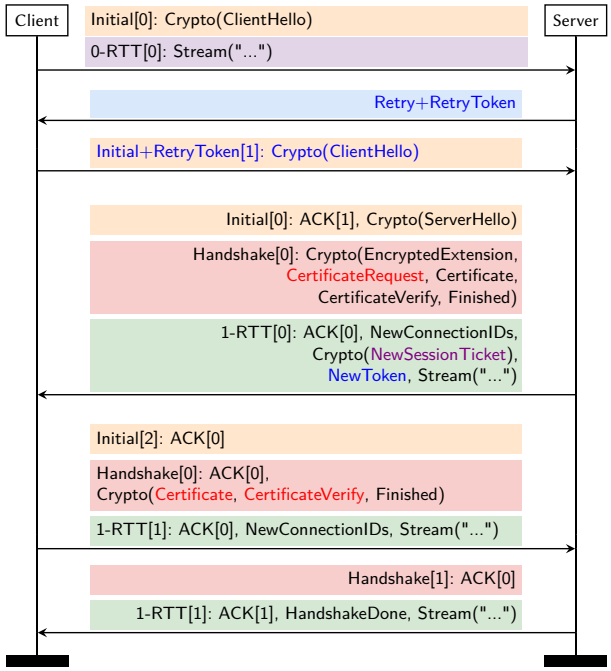



**Figure 1: A *simplified* overview of handshake security configurations consisting of Initial packets, 0-RTT packets, Retry packets, Handshake packets and 1-RTT packets. Frames for Address validation are in blue text and for Client Authentication are red text. Messages in purple text carries the negotiated parameters to derive the pre-shared key for 0-RTT encryption in a future connection. Packet space numbers for each packet type are within square brackets.**

carry different types of data. For example, cryptographic parameters are in a CRYPTO frame. To simplify our explanations, we refer to frames used for connection establishment and the messages encapsulated within the frames such as CRYPTO frames as simply *messages*.

**(1) Basic Handshake.** It is the most fundamental handshake [40, Section 7.1]. First, a client sends the server an Initial packet with a ClientHello message with application protocol negotiation, transport parameters, and cryptographic information to perform the key exchange. The server continues with an Initial packet and a Handshake packet. The Initial packet contains a ServerHello message containing cryptographic information to complete the key exchange. The Handshake packet carries an EncryptedExtensions message containing transport parameters and the negotiated application protocol version, a Certificate message containing the server's certificate, a CertificateVerify message used for requesting the client to verify the server's certificate, and a Finished message.

Notably, QUIC can combine multiple different types of packets into a single UDP datagram for transmission. Once the server sends the Finished message, it can transmit application data using the Stream frame in 1-RTT packets. The client continues the handshake by verifying the server's certificate, and sending a Finished message to the server. Both parties will verify the Finished message received to ensure the previous handshake messages have not been modified. After receiving a Finished message from the client, the server will send a 1-RTT packet with a HandshakeDone message to confirm the handshake and connection establishment.

**(2) Client Address Validation.** Given that a QUIC server responds with many messages to a small initial request, it is at risk of exploitation for amplification attacks. QUIC provides an optional mechanism to validate a client's address, minimizing data sent to spoofed client IP addresses [40, Section 8]. Upon receiving the first ClientHello message, the server responds with a Retry packet containing a RetryToken. A client that receives a Retry packet must include the RetryToken in all further Initial packets sent for the rest of the handshake. The server will validate the RetryToken contained in the client's subsequent Initial packets. If the server fails to validate the RetryToken, the server should immediately close the connection with a ConnectionClose message. Moreover, after the server sends the Finished message, the server can optionally send a NewToken message with an address validation token that can be used for address validation in future connections.

**(3) Client Authentication.** The server can select to authenticate a client by including a CertificateRequest message in the Handshake packet prior to sending the Finished message [62, Section 4.4]. If the client receives a CertificateRequest message, it must send a Certificate message that contains the client's certificate and a CertificateVerify message for client authentication. The server verifies the client certificate before sending the HandshakeDone message. If the server fails to verify the client certificate, the server must close the connection with a ConnectionClose.

**(4) Client Address Validation and Authentication.** The complete handshake, using both client address validation and client authentication simultaneously, is illustrated in Figure 1. The handshake incorporates the basic, client address validation and authentication protocol flow discussed above.

**(5) Pre-shared key.** A client can send 0-RTT packets carrying early (application) data to a server prior to handshake completion [62, Section 2.1]. The pre-shared key used to encrypt these packets is derived from the NewSessionTicket message in the previous connection. This provisions for a faster, secure connection.

**Encryption Keys and Packet Number Spaces.** Instead of sequence numbers, QUIC uses three separate packet number spaces to track different packet types. Initial packets use Initial packet number space; Handshake packets use Handshake packet number space; 0-RTT and 1-RTT packets share Application data packet number space [40, Section 12.3]. Packets in different number spaces use different encryption keys [62, Section 4]. The Initial packet uses the Initial key to provide the Initial encryption level with no confidentiality or integrity protection, the Handshake packet uses the Handshake key in the Handshake encryption level, the 0-RTT packet uses the 0-RTT encryption key in 0-RTT encryption level, and the 1-RTT packet uses the 1-RTT encryption key in the 1-RTT encryption level. The Handshake, 0-RTT and 1-RTT encryption levels provide confidentiality and integrity.

## 2.2 Automata Learning

While automata learning can be categorized into passive and active learning, we focus on active learning [9, 31]. In this setting, a learner constructs a deterministic finite automaton by generating queries to infer the behavior of a system by observing the resulting responses. Learner generates input sequences based on a dictionary of choices to probe a black-box system and observe the output symbol. To ensure a deterministic state model that precisely mirrors the behavior of a given black-box system, learning cycles through 2 phases: (i) *hypothesis construction*; and (ii) *conformance testing*.

In the hypothesis construction phase, a series of input sequences and corresponding input-output observations are used to construct a hypothesis, a minimal deterministic state machine model that accurately reflects the recorded observations until this point for a possible FSM. The learner actively improves the hypothesis until conditions for convergence are fulfilled. For the input sequences that the learner had not sent and observed before, the hypothesis predicts an output by extrapolating from the recorded observations. To ensure this prediction accurately reflects the behavior of the black-box system, the learner proceeds to the conformance testing phase to validate the hypothesis. If the hypothesis is not supported by the behavior of the black-box system when a new input sequence is tested, the learner reverts back to the hypothesis construction phase to generate a more refined hypothesis. Alternatively, if the hypothesis matches the observed behavior of the black-box system for all the conformance tests, the learner considers it as the final learned state machine model, and the learning ends. Effectively, the learner treats the learned FSM as an equivalent oracle and searches for counterexamples to invalidate this assumption.

# 3 NONCOMPLIANCE CHECKING

In this section, we provide a high-level description of noncompliance checking framework and the design and implementation challenges along with our proposed solutions.

We illustrate the QUICTESTER framework modules in Figure 2: i) the Learner; ii) Test Harness; iii) Optimizer; iv) Crash Logger; and iv) Differential Analyzer. The Learner generates test inputs from

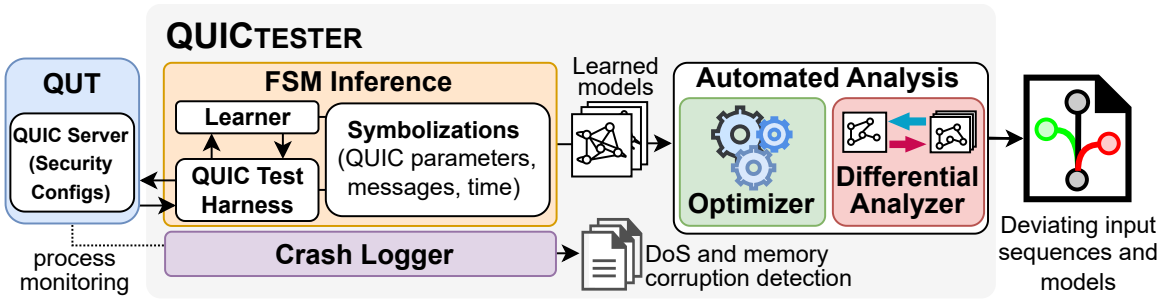



**Figure 2: An overview of QUICTESTER. FSM Inference with active learning and Automated Analysis method for identifying deviating behaviors.**

symbolized QUIC protocol parameters, messages and event timing definitions. The Test Harness, responsible for maintaining all protocol state with a test target, constructs the QUIC messages based on the symbolic instruction sequence for transmission through a UDP connection to the QUIC implementation under test (QUT). Subsequently, the Test Harness awaits for the responses from the QUT within time intervals defined by the parameters in the input symbols. The Test Harness deconstructs the received responses and reconstructs a set of output symbols using the symbolized messages to return to the Learner. The response symbols provide feedback to the Learner to explore and infer a FSM representation of the QUT–the *learned model*.

We employ an Optimizer to simplify the complex models generated from the Learner to generate a simpler, easier to analyze automatically and more readable form of the learned model for analysis. Subsequently, the Differential Analyzer automates the analysis process by using differential testing strategies to identify protocol non-compliance and potential security vulnerabilities. We discuss the design and implementation challenges (**C1**-**C7**) of the framework in the following.

## 3.1 Automatically Learning a Behavior Model (Learner)

**C1 Generating test sequences (valid and invalid messages and parameters).** Fundamentally, in a blackbox setting, no prior knowledge is required. However, a *protocol-specific symbolization* of possible inputs to combine and outputs for a target protocol must be defined to allow the Learner to generate tests—a series of input sequences. These sequences allow the effective exploration and inference of the FSM of the target system through not only valid but *invalid* packet configurations as well as various security and transport parameter combinations in the learning phase. However, a symbolic dictionary for the purpose, extracted from a careful examination of the QUIC specification to test all 5 security settings in a QUIC handshake does not currently exist. Notably, symbolization is not straightforward. It demands a deep understanding of an extensive, technical specification to express and use protocol behaviors to facilitate uncovering an FSM and non-compliance.

**Methods for resolving C1.** To automatically define all the necessary symbols, we attempted to employ an LLM model to generate the symbols. However, the results returned by the LLM model were unsatisfactory—see results and discussion in Section A within the *Appendix*—we reverted to manually extract the symbols from the specification. An author requires approximately 30-40 hours to read the specifications [40, 62] and manually defines symbols for protocol parameters and messages for QUIC. Multiple researchers examined the RFC individually and agreed with all the symbol constructions, to avoid ambiguities. Although the construction phase is time-intensive, the symbolic dictionary needs to be constructed only once and can be applied in further studies on QUIC, thereby reducing the manual effort required to redefine the symbols. For constructing test inputs and modeling outputs, we include 30 symbols. While Section 2.1 provides an overview of messages and their constituent parts, such as frames, we defer our justifications and details of symbolization to Appendix A.1.

**C2 Learning Time-Dependent Behavior.** We introduce the idea of testing with different time variations to uncover time-dependent vulnerabilities (such as **M**-4 in Table 2). However, using random timeouts is not beneficial as it can confuse the learner's observations and lead to non-determinism and failure to infer a FMS or complete the active learning process.

**Methods for resolving C2.** To observe time-dependent behavior, we parameterize the *input* symbols. We determine two timeouts parameters, a *short* timeout and a *long*, is adequate; as we explain below. The timeout parameter represent the duration the Test Harness is required to wait prior to the subsequent message transmission after receiving a response; with *short* representing the minimum wait time to receive QUT's response for a request sent and *long* is selected to be much longer, 10× *short*. The time parameterized symbols we curate are summarized in Table 8.

Importantly, the timeouts are discovered automatically by the Test Harness prior to model learning. Notably, in contrast to an adversarial interpretation, the time parameter can also elicit behavior of the QUT under network delays experienced in practice. Two time settings are sufficient for capturing protocol states as they can capture results from when protocol implementation timeouts *do* or *do not* occur. Further timeout settings were found to increase learning time without leading to additional state discovery.

**C3 Non-determinism during learning.** Model learning depends on observing deterministic behavior from the QUT to infer a valid FSM model. We expect the QUT to generate deterministic responses to multiple repetitions of the same input sequence. However, since the Learner is unaware of time-related artifacts in the rest of the system, non-determinism can manifest when **(C3.1)** the Test Harness is not capable of receiving all the QUT responses during an allocated

time period, for example, due to various execution speeds of QUIC implementations; and **(C3.2)** the Learner sends an input before the QUT is fully initialized or **(C3.3)** after the QUT crashes during learning, for example, if the Learner receives responses to a given input sequence in one step but fails to receive any when replaying the sequence because the QUT crashed before it can respond, the active learning task can fail. These sources of non-determinism can lead the Learner to infer an incorrect behavior or fail to complete the active learning task.

**Methods for resolving C3.** To address **(C3.1)**, we adopt implementation-specific minimum time delays for the Test Harness to wait to capture responses from the QUT to ensure that no responses are missed. Recall, the time for the Test Harness to capture responses will be defined by the Learner as introduced in Section 3.1. Hence, we use the Test Harness to record the longest time needed to capture expected packets in a handshake to determine the value for *short*, specific to each QUIC implementation, prior to model learning. For **(C3.2)**, we augment a delay after starting the QUT and validate its availability before sending the first input. This ensures the QUT is initialized and ready to accept new connections. For **(C3.3)**, we implement a Crash Logger to monitor the status of the QUT after each test sequence iteration and restart the QUT if it crashes. These modifications ensure the Test Harness and Learner behavior does not lead to non-deterministic outcomes and any observations of non-determinism are due to QUIC implementation defects.

**Learner Implementation.** We implemented the Learner using LearnLib, a Java library of active automata learning algorithms [39]. We selected the TTT algorithm [38]; it requires less queries compared to other algorithms [37] and Wp-method algorithm [16] as the conformance testing algorithm.

## 3.2 QUIC Test Harness (Test Harness)

**C4 Building a Test Harness to Support All Security Configurations, Valid and Invalid Messages, and Learning Time Dependent Behaviors.** As described in Section 2, the Test Harness is a protocol-specific test harness responsible for exchanging messages with the QUT. It constructs and transmits QUIC messages sequentially, based on the test input symbol sequence determined by the Learner. The symbolic representation cannot be directly sent to a QUT. The Test Harness must be functionally and logically correct on: **(C4.1)** translating state and state transitions explored by the model Learner to QUIC protocol message exchanges with QUT; **(C4.2)** constructing symbolic representations of protocol messages; whilst **(C4.3)** maintaining state cohesion with the target QUT to achieve successful progression of a test from message to message. Further, as discussed in Section 2.1, QUIC supports *five* different secure connection establishments and requires deriving and installing 3 different encryption keys during a handshake. Therefore, building a Test Harness is a significant undertaking. To the best of our knowledge, a QUIC-specific test harness to fulfill these requirements does not currently exist.

**Methods for resolving C4.** To manage the effort in addressing the challenges and building a comprehensive *QUIC-specific test harness*, we extend and modify the Aioquic [1] library. To address:

- **(C4.1)** we modify and extend the library functions to generate packets based on a given input symbol by the Learner. Further, to comprehensively test all *five* different secure connection establishments, we implement functions for client authentication, which is currently *not* supported by the library.
- **(C4.2)** we extend with functions to parse incoming packets from the QUT to output symbols.
- **(C4.3)** we utilize the state machine to maintain the state cohesion with the QUT. Importantly, we configured the state machine to not discard any installed encryption keys. This enables the Test Harness to generate QUIC messages with the encryption key from the previous encryption level irrespective of the current encryption level. This capability is valuable for testing the server's ability to handle encryption keys across different encryption levels (Section 2.1). For example, after the handshake is confirmed (1-RTT encryption level), the Test Harness can still generate and transmit an Initial packet (at the Initial encryption level) to the QUT.

## 3.3 Automating Analysis (Optimizer & Differential Analyzer)

**C5 Learned Models are Difficult to Interpret and Analyze.** The learned model generated by the Learner is unnecessarily difficult to analyze due to artifacts from the testing phase and protocol complexity.

**Methods for resolving C5 (Optimizer).** To ease the analysis process, we design an optimization routine to simplify the learned model by incorporating the following observation. The model description can contain numerous edges (state transitions) that do not provide useful information. In particular, the following types of edges observed in models are redundant:

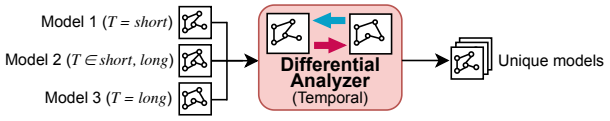
- Edges that do not transition to a different state.
- Edges with the same input but different timeouts transitioning to the same next state from a given state.

Therefore, we first, we remove all the edges that do not transition to a different state from the current state, as these edges indicate no progress on the QUT. Then, we merge all edges with the same input type but different timeouts, if they have the same current and next state. Doing so allows us to easily identify deviations given the same input with a different timeout. Including these optimizations in QUICTESTER allowed up to 90% of edges to be removed from the original learned model in the best case, while testing the Quinn server (Figure 16) as shown in the *Appendix*. The overall result is a considerable simplification to aid automated analysis and a significant reduction in the effort required to read and analyze models and to identify anomalous behavior.

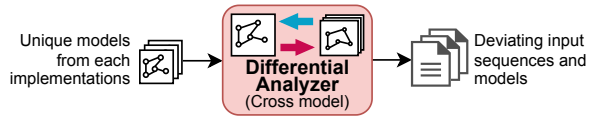
**C6 Cost and Problems with Manual Deviation Analysis.** One of the significant problems with black-box noncompliance checking using automata learning in the previous studies [20, 28] is the subsequent manual effort needed to analyze learned models by a domain expert.

**Methods for resolving C6 (Differential Analyzer).** We develop an automated state machine comparison approach for the problem. A simplified example illustrating the automated analysis approach is shown in Figure 4.

We prune all the output symbols from the optimized learned models and use the Labeled Transition System Differential (LTS_Diff) algorithm [64] in two scenarios: i) we compare models originating




**(a) Comparison between models originating from the same target (server) and configuration but under different time parameter settings.**




**(b) Fully automated testing method to mitigate the need for a reference model. Comparison between all unique models from Figure 3a to extract all input sequences that lead to deviations along with model descriptions (FSMs).**

**Figure 3: Automated analysis using Differential Analyzer.**

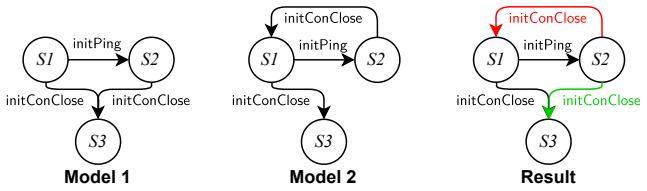



**Figure 4: Example illustrating a Differential Analyzer result. Given two optimized models as inputs (Model 1 and Model 2), unique state transitions (the green edge denotes the unique edge in Model 1 while the red edge denotes it is unique to Model 2) are identified by the algorithm.**

from the same target (server) and configuration but in different timeout settings (Figure 3a); and ii) we cross check all unique models obtained from (i) against all other targets (Figure 3b) and extract *all* the deviations automatically. During the comparison, we identify and highlight deviations—additional (green) or missing (red)—states and state transitions. Figure 4 shows an example; the highlighted deviations indicate at least one of the implementations may not be compliant with the specification. To further analyze the deviations, we automatically extract the shortest input sequence with its corresponding output leading to deviations. To this end, we ***only need to analyze the extracted deviations*** from (ii) to identify the non-compliant implementation. The approach significantly reduces the manual effort to inspect all the learned models. We discuss the reduced manual effort obtained from the approach in Section 4.

Significantly, our extensive testing regime has allowed us to curate conforming learned models to serve as reference models. The use of QUICTESTER with the reference models can automatically extract non-conformance behaviors from the learned model we discuss in Section 7. Consequently, our automated approach in the Differential Analyzer can support developers in the future by reducing the burden in identifying the non-compliant implementations.

**C7 Detecting DoS.** If the QUT exhibits a memory-corruption bug or a logical flaw, unexpected behavior such as a crash can occur. In some cases, the crash can be hidden in the learned model (as exemplified in **M**-8 in Table 2) and pose a challenge for detecting DoS vulnerabilities.

**Methods for resolving C7 (Crash Logger).** We employ a Crash Logger to monitor the aliveness of a QUT by checking its `PID` after each learning step. If the QUT crashes, usually indicated by a missing or invalid `PID`, the Crash Logger saves the `stderr` and `stdout` from the QUT with the corresponding *input sequences* for subsequent analysis. The logging function and data allow identifying DoS attacks and memory-corruption bugs.

## 4 EVALUATION

We tested and analyzed 19 open-source QUIC implementations summarized in Table 1 using our QUICTESTER implementation. Our implementation effort is summarized in Table 6 in Appendix E.

**Test Environment.** All experiments are conducted using Ubuntu 20.04 with an AMD Ryzen 9 5950X CPU and 128 GB of RAM.

**QUIC Configurations.** We test QUIC implementations with all mandatory and recommended cipher suites [56] implemented by the targets. Since all of the QUIC implementations we employed provide servers in their repository, we use the provided tools to configure and run these servers as the QUT. As mentioned in Section 2.1, our aim is to test all 5 different handshake configurations. Where an implementation did not support all 5 configurations, we tested the implemented configurations as summarized in Table 1.

**Differential Testing With Time Parameters.** In our experiments, we conduct learning on each implementation with three timeout settings: $T \in short, long, mixed(short + long)$. In our differential analysis, we compared the learned models generated from these three different timeout settings to identify state transitions that were influenced by temporal factors.

**Identifying Anomalous Behaviors.** We examined: i) differential test results with QUICTESTER and validated the deviating behaviors discovered with the extracted input sequences to identify non-compliant behaviors; and ii) crashes, trace data and crashing seeds from the QUICTESTER's crash logger. Subsequently, we examined the server code of these QUIC implementations for *root cause analysis* and bug disclosures.

## 5 RESULTS AND ANALYSIS

In this section, we discuss our results from analyzing the 186 learned models. With the model optimization technique, our framework can generate a more readable learned model for analysis. To simplify the presentation of learned models, in the following analysis: i) we group the input and output symbols on a transition to a connection close, and label it with the Other symbol; and ii) when all input and output symbols from a given state transition to the same next state, we also group these symbols, replacing them with Other, as these transitions do not provide any new information. Further, we have highlighted valid paths to complete a handshake in blue and highlighted invalid paths denoting adverse behavior in red. An example to illustrate the interpretation of a learned model is given in Appendix B. Here, we present a series of case studies on fault discoveries impacting the security or availability of servers.

**Threat Model.** Notably, our analysis is based on the threat model described in [40] and [57]; wherein a key aspect of QUIC is to build mechanisms to mitigate DoS attacks. In the following, we present a series of summarized case studies on fault discoveries impacting

**Table 2: Overview of identified faults (detailed in Table 7).**

| Server | Fault Description | Type-ID |
|---|---|---|
| Aioquic | Incorrect handling of unexpected frame type. | **S**-1 |
| Kwik | Retention of the unused encryption keys. | **S**-2 |
| | Implementation without a state machine. | **S**-3 |
| | Process CRYPTO frame in a 0-RTT packet. | **S**-4 |
| | Exceeds the operating system's maximum number of memory mappings for a single process (100,000) when receiving PING frame from 50,000 clients. | **M**-1 |
| Lsquic (Lite Speed) | Retention of the unused encryption keys (PSK configuration). | **S**-5 |
| | Incorrect handling of re-transmission, leaving a half-opening connection on the client side (PSK configuration in v4.0.2). | **L**-1 |
| MsQuic (Microsoft) | Does not issue its initial_source_connection_id at the correct connection state. | **S**-6 |
| Neqo (Mozilla) | NULL pointer dereference when getting the primary path. | **M**-2 |
| | Limited connections due to a hardcoded value. | **M**-3 |
| Picoquic | NULL pointer dereference when getting the encryption keys. | **M**-4 |
| | Retry token tied to retry_source_connection_id. | **S**-7 |
| PQUIC | Invalid original_destination_connection_id. | **S**-8 |
| | Limitless active_connection_id_limit. | **S**-9 |
| | Retention of the unused encryption keys. | **S**-10 |
| | Incorrect way of emptying the re-transmission queue. | **L**-2 |
| | NULL pointer dereference when handling removed connection context. | **M**-5 |
| | Buffer overflow when processing frame type 0x30. | **M**-6 |
| | Infinite loop when processing frame type 0xFF. | **L**-3 |
| | Does not send HANDSHAKE_DONE after the handshake is confirmed (PSK configuration). | **S**-11 |
| Quiche (Cloudflare) | Client authentication bypass due to incorrect flag set in Quiche library. | **S**-12 |
| | Incorrect handling of **all** Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-13 |
| Quiche4j | Concurrent modification exception when discarding closed connections. | **M**-7 |
| | Limitless active_connection_id_limit. | **S**-14 |
| Quant | Incorrect handling of an initialPing message. | **S**-15 |
| | Incorrect handling of **all** Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-16 |
| Quiwi | Does not close the connection when the number of received NEW_CONNECTION_ID frames exceed the active_connection_id_limit. | **S**-17 |
| Quinn | Panic when unwrapping a None value when processing an unexpected frame type. | **M**-8 |
| | Process CRYPTO frame in 0-RTT packet. | **S**-18 |
| XQUIC (Alibaba) | Retention of the unused encryption keys. | **S**-19 |
| | Maintaining a number of active connection IDs that exceed the active_connection_id_limit. | **S**-20 |
| Aioquic, LSQUIC, Neqo, Quic-go, Quinn, Quiwi, S2n-quic (Amazon), XQUIC | Incorrect handling of the second and subsequent Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-21 to **S**-28 |
| Aioquic, Kwik, MsQuic, LSQuic, Quant, Quiche, Quic-go, Quiche4j, Quiwi, S2n-quic | Accept Handshake packet from an unmatched Destination Connection ID. | **S**-29 to **S**-38 |
| Lsquic, MsQuic, Neqo, Quiche4j, Quinn, XQUIC | Incorrect handling of packets without a frame. | **S**-39 to **S**-44 |

In total 44 specification violations (**S**), 8 memory-corruption bugs (**M**) and 3 logical flaws (**L**) were identified across 19 implementations.

the security or availability of servers. Detailed discussions of the case studies and a demonstration of valid behavior analysis on one of the reference models are included in the *Appendix*.

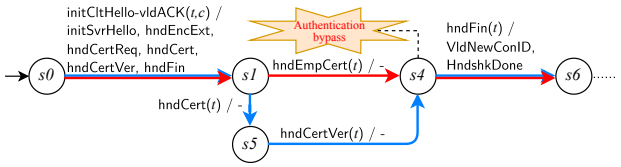



**Figure 5: Simplified learned model of a Quiche server with the ClientAuth configuration. Blue edges show a valid path to complete a QUIC handshake. Red edges demonstrate an invalid path that bypasses the client authentication. The complete model is in Figure 17, in the *Appendix*.**

**Results Summary.** We summarize all the observed faults in Table 2; a detailed table with extended discussions is in Table 7 within the *Appendix*. Each fault is categorized as one of the following: i) *specification bugs*. An implemented behavior violates the QUIC specification; ii) *memory-corruption bugs*. An input causing memory corruption and a server crash. iii) *logical flaws*. Incorrect logic implemented in code produces unexpected behavior.

## 5.1 Non-Compliance Issues

We discovered 44 specification violations—i.e. non-compliance issues. We present four case studies of significant issues and defer details and inputs for reproducing all of the issues to our GitHub repository [3].

S-18 **Client Authentication Bypass in Quiche.** The simplified Quiche learned model with **ClientAuth** configuration is shown in Figure 5. The valid path to establish a QUIC handshake is highlighted in blue color. Our analysis of this model revealed that the server bypasses the client authentication on the path: *s0*, *s1*, *s4*, *s6*, highlighted in red. This behavior was observed in all 3 handshake processes initiated with a different cipher suite(*AES_128*, *AES_256*, *ChaCha20*). In this path, after the exchange of transport parameters and cryptographic information between the client and server, the server sends a handshakeCertificateRequest message to authenticate the client. However, instead of responding with a valid certificate, the client sends a handshakeEmptyCertificate message to the server at *s1*. Subsequently, the client sends a handshakeFinished message to complete the handshake. The server processes the client's handshakeFinished message without verifying the client's certificate. Then, the server responds with ValidNewConnectionID and HandshakeDone messages to confirm the completion of a successful handshake. The summarized flow of this authentication bypass is illustrated in Figure 14 in the Appendix.

During our code analysis of Quiche, we discovered the developers had set the incorrect flag (`SSL_VERIFY_PEER`) for verifying the client certificate in the BoringSSL [2] configuration. To address this issue, the flag should be set to `SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_CERT`, which ensures that the handshake fails if an empty client certificate is used for authentication. However, according to BoringSSL documentation, this misconfiguration only affects the client authentication, i.e., an anonymous server will always fail to establish a connection with the client even without setting the `SSL_VERIFY_FAIL_IF_NO_CERT` flag.

*Impact* This allows an unauthenticated client to set up an anonymous connection with the server, allowing an on-path active attacker to perform man-in-the-middle attacks [23, 56]. Notably, the issue was fixed with a bug bounty awarded to our team).

S-19 **Retention of the unused encryption keys.** We discover that the XQUIC server will still respond to Initial packets after moving to the Handshake encryption level, as well as responding to Handshake packets after moving to the 1-RTT encryption level (when the handshake is confirmed). XQUIC does not follow the specification as stated in [62, Section 4.9], a QUIC server must discard the unused keys after moving to a new encryption level. For example, a server must discard its Initial key after it processes the first Handshake packet from the client so that the subsequent Initial packets will not be processed.

The non-conformance described above can lead to serious consequences. *For example*, an attacker can disrupt a connection. The Initial key does not provide confidentiality or integrity protection against attackers that can observe packets [40, Section 17.2.2]. Notably, the Initial key is determined by using HKDF-Extract with a default salt specified in [62, Section 5.2] and an input keying material (IKM) of the Destination Connection ID from the client's first Initial packet. An attacker can sniff a victim's (client) first Initial packet, obtain the Destination Connection ID and compute the victim's Initial key. Then, the attacker can use the victim's Initial key to send a spoofed initialConnectionClose message to the server. *A server that does not discard the Initial key may use this key for decryption, process the spoofed Initial packet and* ***close the connection with the victim*** (DoS attack).

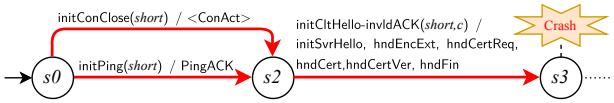
*Impact* The non-conformance behavior [62, Section 4.9] allows an off-path active attacker to disrupt a connection during the handshake in both security settings supported by XQUIC to mount a DoS attack. The attack requires a malicious actor able to sniff the first Initial packet sent by the victim (a QUIC client) on a network.

S-2 **Retention of the unused encryption keys.** Once a key is created for an encryption level, the Kwik server will continue decrypting and processing packets from that encryption level, even after moving to a new encryption level. Similar to XQUIC, this behavior is not conforming to [62, Section 4.9]. So, Kwik is also vulnerable to the spoofed initialConnectionClose attack explained earlier.

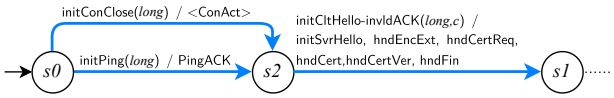
*Impact* An off-path active attacker can mount a DoS attack by disrupting a connection in all security settings supported by Kwik. The attack requires a malicious actor able to sniff the first Initial packet sent by the victim (a QUIC client). This vulnerability was assigned CVE-2024-22588 and patched by the Kwik developers.

Notably, Kwik presents a more critical issue than XQUIC. Beyond just retaining the unused encryption keys, Kwik does not actually track the current state of a connection. In other words, Kwik does not implement a proper state machine—as discussed in S-3 below.

S-3 **Implementation without a TLS state machine.** In testing, we found the Kwik implementation to reprocess a message that was already successfully processed before, such as an initialClientHello message. Notably, this message contains application protocol negotiation, transport parameters, and cryptographic information to perform key exchange. This reprocessing of an initialClientHello message will overwrite the existing connection's application protocol version, transport parameters, and encryption key.




**(a) Simplified model from Picoquic with ClientAuth configuration learned with $T = short$ parameter setting for inputs.**




**(b) Simplified model from Picoquic with ClientAuth configuration learned with $T = long$ parameter setting for inputs.**

**Figure 6: Differential analyzer reveals a deviation on Picoquic when different time parameters are used for inputs. The complete models are given in Figure 18 and 19 in the *Appendix*.**

This is a serious issue because, combined with the vulnerability we discussed earlier, attackers can reset or potentially hijack a victim's connection using a initialClientHello message. For example, at any state of a connection, an attacker with the victim's Initial key can send a spoofed initialClientHello message with transport parameters and cryptographic information that differs from the victim's to the Kwik server. The Kwik server will process the spoofed initialClientHello message and overwrite the existing transport parameters and encryption key that it has with the victim. Due to the desynchronization of transport parameters and encryption keys, the server no longer recognizes the victim and drops any packets coming from the victim. An attacker can then sniff the responses from the server, complete the overwritten handshake and use the connection to exchange data with the server.

*Impact* A malicious off-path active actor can hijack a victim's active connection. Notably, this vulnerability has been fixed by the developers with CVE-2024-22590 assigned.

## 5.2 Memory-corruption bug: Server crashes

We discovered 8 memory-corruptions. We defer details and inputs for reproducing bugs to [3]. Here, we review M-4 uncovered with our *idea for discovering timing dependencies on protocol states*. We include three further case studies in Appendix C.1.

M-4 **Null Pointer Dereference in Picoquic.** Interestingly, our use of differential testing with time parameters was crucial to identifying the issue. Figure 6 depicts the Picoquic's **ClientAuth** simplified learned models using different time parameters. Our Differential Analyzer revealed a deviating state transition from *s2* in Figure 6a and Figure 6b when the same input, initialClientHello-invldACK was sent. The state transitions in both figures are responded to with the necessary messages to continue the handshake. However, when a client sends any further messages at *s3* in Figure 6a, the server does not respond. Data from our Crash Logger revealed a segmentation fault occurred on each occasion the specific input sequence was received by the server at *s3* in Figure 6a.

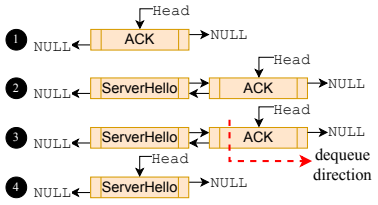



**Figure 7: Changes in Picoquic's re-transmission queue stored as a double-linked list, leading to a segmentation fault.**

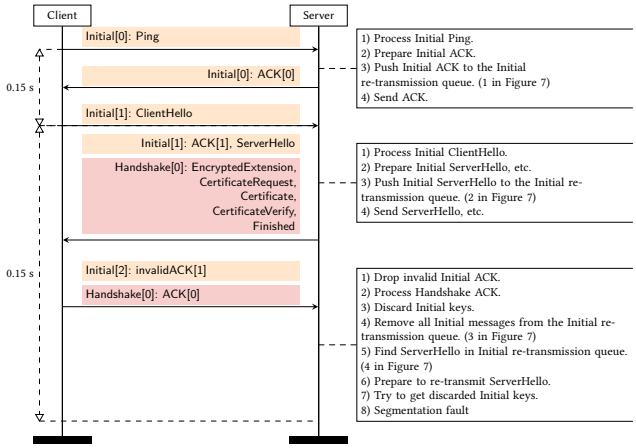



**Figure 8: A message sequence chart showing message flow required to trigger a segmentation fault in Picoquic with the ClientAuth configuration.**

We use `rr` [4] to investigate further and record Picoquic's execution while sending the crashing input sequence. We debug the root cause of this crash by replaying the recorded program execution. The segmentation fault error occurs because the server tries to re-transmit an initialServerHello message and attempts to dereference a null pointer when retrieving the required encryption key.

This segmentation fault error in Picoquic depends on several events within the server's operation. When the server acknowledges the initialPing, it pushes its first message (PingACK) to the head node of its Initial re-transmission queue in ❶ shown in Figure 7. Notably, the server uses a double-linked list as its re-transmission queue. In response to the client's initialClientHello-invalidACK, the server sends its second message (initialServerHello). The message is added to the previous node of the head node, as illustrated in ❷ in Figure 7. Subsequently, the server drops the invalid Initial ACK that acknowledges the initialServerHello message and processes the Handshake ACK that acknowledges the first Handshake message it sent. As a result, the PingACK and initialServerHello remain in the Initial re-transmission queue. After processing the Handshake ACK, the server removes its Initial keys as described in [62, Section 4.9.1]. Subsequently, the server attempts to remove all the messages in the Initial re-transmission queue to prevent any transmission of Initial messages.

Interestingly, the server's method of emptying the Initial re-transmission queue does not follow the order in which the messages were added. The server removes the head node and all its sub-sequence messages stored in its next node as shown in ❸ in Figure 7. Consequently, only the oldest message is removed, while the remaining messages in the re-transmission queue remain as shown in ❹. Later, when a re-transmission callback occurs, the server attempts to encrypt the initialServerHello message by dereferencing the null pointer that previously stored the discarded Initial keys. This is the source of the segmentation fault error. The summarized flow of QUIC handshake protocol messages leading to the segmentation fault is illustrated in Figure 8 in the *Appendix*.

Notably, when the first input, initialPing, is set to *long* timeout, the server removes the PingACK from the Initial re-transmission queue before processing the Handshake ACK. As a result, the initialServerHello message becomes the head node in the queue. When the server discards the Initial keys, it also removes the first message in the queue, which is the initialServerHello message. This proactive step prevents the server from encountering a segmentation fault and allows it to complete the handshake successfully.

Interestingly, the complete **ClientAuth** model learned with inputs ($T$ = *short*) has 11 states and shows the adverse behavior described above, while the complete **ClientAuth** model learned with inputs ($T$ = *long*) has only 9 states with no anomalies (complete learned models can be found at [3]). As evidenced, the difference in models demonstrates testing with time-parameterized inputs elicits new behavior of a network protocol implementation, leading to new states, state transitions, and potential new bug discoveries.

*Impact* The vulnerability allows an attacker to perform a simple DoS attack with a single client during connection establishment.

## 5.3 Logical Flaw: Unexpected Behavior

Logical flaws produce unexpected behavior due to incorrect program logic. We discovered 3 logical flaws but defer details and inputs for reproducing all issues to our GitHub repository [3]. We review L-1 discovered by QUICTester here and include two more in Appendix C.2.

**L-1 Incorrect handling of re-transmissions.** During connection establishment, Lsquic server with **PSK** configuration will reach the close state every time it attempts to re-transmit the last unacknowledged message.

*Impact* A client unable to acknowledge the server's handshake messages in time will always fail to establish a connection with the server.

## 5.4 Specification Ambiguity: Exposing a New DoS Attack

When comparing the optimized models across 19 server implementations, we identified a scenario under which an ambiguity related to the connection management aspects in the QUIC specification can lead to implementations exhibiting different behaviors; some more detrimental than others.

The specification states:

> `The first packet sent by a client always includes a CRYPTO frame that contains the start or all of the first cryptographic handshake message.` (Section 17.2.2, RFC 9000).

However, the specification does not explicitly state how a server should/must handle a first packet received without a CRYPTO frame nor what first packets a server must process. Further, as described in the specification:

> After processing the first Initial packet, each endpoint sets the Destination Connection ID field in subsequent packets it sends to the value of the Source Connection ID field that it received. (Section 7.2, RFC 9000),

A server will also need to "remember" the Connection IDs (or connection context) sent from clients after processing the first packet, *even if the client has no intention of initiating a connection with the server.* When a client sends an initialPing as the first packet to assess reachability of a server, we observed:

- A few implementations elect not to acknowledge pings from clients that do not have an existing open connection with the server. These servers elect to drop the packet, or respond with a connectionClose message.
- However, many implementations accepting initialPing packets from clients *without* an existing connection, create a connection based on the ping and respond with an acknowledgment. In this behavior, the server is forced to "remember" the connection ID (leading to creating a *connection context*) *even if the client has no intention of initiating a connection with the server.* Hence, irrespective of the client continuing the handshake, the server has expended resources in creating a connection context. These resources are not de-allocated until a timeout occurs.
- Only one implementation (MsQuic) responds to the ping without creating a connection context.

To investigate the impact of implementation choices, we experiment by initiating 50,000 QUIC clients. Each client sends one packet, initialPing, to a target QUIC server with **Basic** configuration. Since the specification states the first packet always contains a CRYPTO frame and servers can elect to drop the initialPing without a CRYPTO frame, we tested with initialPing packets without a CRYPTO frame. We tested the 19 QUIC servers we studied. During the experiment, we made an interesting observation—a significant increase in memory usage (from 500 MB to 3 GB) for 10 server implementations, denoted as *Category 1* in Figure 9. This is a direct consequence of the servers always creating a connection context for each incoming *first* packet, even when a client does not intend to establish a connection (initialPing that excludes a CRYPTO frame).

Interestingly, we were able to ***crash the Kwik server* (M-1) *after it exceeded the operating system's maximum number of memory mappings*** for a single process (100,000) (the server created 2 threads for each incoming initialPing). We disclosed our findings to each affected QUIC implementation developer.

We propose the specification to allow responding to clients not seeking to establish a connection to support liveness testing via initialPing without a CRYPTO frame where a connection context is not created and amending the specification to state a first packet to initiate a connection *MUST* include a CRYPTO frame.

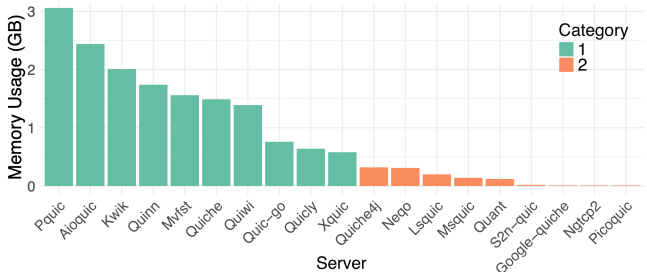



**Figure 9: Memory usage recorded for 19 QUIC servers tested with 50,000 clients sending an Initial packet with a PING frame without a CRYPTO frame.**

## 6 RELATED WORK

**Passive State Machine Inference.** These approaches [17, 33] employ template data, e.g. pcap files to generate an FSM. However, the approach cannot detect server failures (crashes) and actively explore *unobserved* states of the system. Hence, we consider an active learning approach.

**Active State Machine Inference.** Past studies used the approach to successfully test network protocols [6, 19, 20, 27, 28, 30, 35, 48, 61]; a concurrent study employed the methods to successfully discover specification non-conformities of Bluetooth Low Energy [44] implementations. Notably, in [44], a divide-and-conquer approach speeds up learning, which we do not consider for QUIC due to the possibility of the latter state transitions being influenced by the first inputs. A concurrent study also proposed an automated method [29] to detect state machine bugs from learned models. The approach requires the construction of a Deterministic Finite Automaton first to describe the correct state transitions for the protocol handshake. In contrast, similar to the approach in [6, 27, 30, 44], our chosen method simplifies the analysis by directly performing cross-model checking to extract the deviations. Further, we also provide differential testing with curated reference models to reduce the burden of identifying non-compliant behaviors (see Figure 10).

**Network Protocol Fuzzing.** We acknowledge efforts in implementing mutation-based [7, 8, 10, 11, 46, 47, 51, 58] and generation-based fuzzers [24, 41] for discovering memory-related bugs in protocols. A recent study [50] leveraged large language models to enable structure-aware mutation on the non-security protocol specifications (RFCs). In contrast, our work focuses on employing automata learning as a black-box tool for uncovering specification violations in protocol implementations.

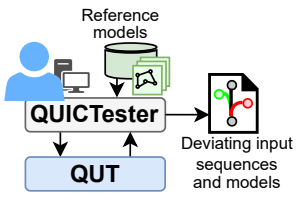



**Figure 10: Practitioners can use QUICTester with our curated reference models to automate future testing of target QUIC implementations (QUT).**

**Table 3: Comparison of automated testing studies on QUIC (Both Compliance checkers & Fuzzers).**

| Tools | RFC 9000/ 9001 | Open-source | *QUIC* Impls. tested | #Faults | Active learning | Timing related behavior | Invalid test cases | Black box | State model opt. | Automated (w/o formal model) | Under spec. detection |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Non-compliance checking** | | | | | | | | | | | |
| **QUICTester (Ours)** | ✓ | ✓ | 19 | 55 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| EPIQ'2021[18]‡ | ✗ | ✓ | 7 | 9 | ✗ | ✗ | ✓ | ✓ | - | ✗ | ✓ |
| SIGCOMM'2021[26] | ✗ | ✓ | 3 | 3 | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ |
| SIGCOMM'2019[49]‡ | ✗ | ✓ | 4 | 27 | ✗ | ✗ | ✓ | ✓ | - | ✗ | ✓ |
| *Unpublished* (2019) | ✗ | ✓ | 1 | 0 | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| IMC'2017[42]⋆ | ✗ | ✓ | 1 | 1 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| **Other** testing studies (fuzzing, fault injection, traffic analysis) | | | | | | | | | | | |
| QUIC-Fuzz [8] (ESORICS'2025)† | ✓ | ✓ | 6 | 10 | - | ✗ | ✓ | ✗ | - | - | - |
| Fuzztruction-net [11] (CCS'2024)◇ | ✓ | ✓ | 2 | 5 | - | ✓ | ✓ | ✗ | - | - | - |
| Bleem [46] (USENIX'2023)† | ✓ | ✗ | 1 | 1 | - | ✗ | ✓ | ✓ | - | - | - |
| DPIFuzz [55] (ACSAC'2020)★ | ✗ | ✓ | 5 | 4 | - | ✗ | ✓ | ✓ | - | - | ✓ |
| BooFuzz (*Communit/Industry*)★ | ✓ | ✓ | - | - | - | ✗ | ✓ | ✓ | - | - | - |

‡: Formal verification; ◇: Fault-injection fuzzer; †: Mutation-based fuzzer.; ★ : Generation-based fuzzer; ⋆: Instruments a QUIC implementation to extract its execution trace.

## 7 DISCUSSION

**Automated Compliance Testing With Reference FSMs.** Through our differential analysis method (Temporal and Cross Model, illustrated in Figure 3a), we identified and curated 11 models conforming to QUIC RFC 9000/9001 (see Appendix D). The reference models capture compliant FSM variations across all 5 security configurations. Importantly, the models allow developers to employ QUICTester to effectively, efficiently and automatically identify non-conformance behaviors or faults in the future as illustrated in Figure 10.

**Threats to validity.** We uncovered 55 faults and confirmed all of the deviations detected by Differential Analyzer are due to either: i) specification bugs, ii) memory-corruption bugs, or iii) logical flaws as listed in Table 2. QUICTester did not detect any false positives; identifying a deviation that does not lead to a fault. Further, we have made efforts to ensure the series of membership queries produced based on our dictionary of symbols comprehensively covers all protocol variations. However, a non-zero probability exists, despite our best efforts, that we have inadvertently missed a symbolization and hence, a potential uncovering of a state or transition. Further, it is important to recognize the test oracles (diverse QUIC implementations) are inherently unfaithful because they can all suffer from the same logical vulnerability. Thus, although highly unlikely, it is possible that a noncompliance remains, yet undiscovered despite the diverse range of implementations and settings we employed in our testing.

Notably, the noncompliance checker is not purposed for detecting memory-corruption bugs effectively because active automata learning focuses on the logical structure and behavior of the state machine rather than the underlying data manipulation. For instance, it does not perform mutations on packet fields that could trigger buffer overflows or similar vulnerabilities.

**Correctness.** As highlighted in Section 4, we have manually validated all 186 models and confirmed that all the behaviors illustrated in the models (across 19 implementations) are reproducible. In addition, we provide a list of inputs for each deviating behavior in our code repository for reproducibility [3]. As an interesting anecdote, we spent approximately 1860 mins (approximately 10 mins per model for 186 models) in our efforts to evaluate the veracity of the state machine comparison method discussed in Section 3.3.

We also considered if the optimizations are sound and produce FSMs that accept the same strings as the un-optimized FSMs. To this end, we applied the differential analysis method on un-optimize FSMs, for the set of models we evaluated, the task consumes approximately 5.4 hours. Subsequently, we compared the unique deviations extracted with those deviations extracted from optimized FSMs (taking approximately 2.2 hours). We found the unique deviations from the un-optimised and optimised models to be the same.

**Completeness.** In this study, we focused on comprehensively analysing the security components of QUIC (the handshake stages). Hence, QUICTester is not currently capable of detecting deviations in other components of QUIC. These include connection migration of a client on an active connection with a server or the state of open streams for application data exchange. But, QUICTester is modular and extensible to test these components, it will require defining new inputs and output symbols and modifying the existing Test Harness. We leave these avenues for further development.

**Learning Time-Dependent Behaviour.** As explained in Section 5.2, learning with time-parametrised inputs can uncover previously unobserved behaviours, lead to the discovery of new states, state transitions, and potential new bug discoveries such as **M**-4. However, the average time required for learning increases with different time settings—29.6 hours for *short*, 38.9 hours for *long*, and 76.5 hours for both *short* and *long*. Therefore, exploring optimization strategies to reduce runtime while maintaining the bug discovery effectiveness is a valuable avenue for future work.

**Comparison with existing tools.** Prior to the QUIC specifications [40, 62] being finalized, [18, 26, 42, 49, 54, 55] have made significant efforts in testing QUIC implementations. However, most of the QUIC-specific tools are only built for testing the IETF-draft or Google-QUIC, are no longer actively maintained, and have limited scope for security testing (e.g. do not consider invalid input while testing). Therefore, these tools lack suitability for testing the ratified specification. To the best of our knowledge, our work is *first* to develop a black-box noncompliance checker to test *all 5 different security settings*, including client address validation and

client authentication of the ratified QUIC specification. We summarize our comparison in Table 3 and, *for completeness*, include mutation-based fuzzers and a generation-based fuzzers that have tested QUIC implementations.

## 8 CONCLUSIONS AND FUTURE WORK

In this study, we presented the first, programming language agnostic, comprehensive noncompliance tester for the security critical connection establishment components of the QUIC protocol. QUICTester is validated with 5 CVEs assigned from developers, 55 faults discovered and confirmed by developers, a bug bounty and uncovering of a specification ambiguity. Although we have made significant in-roads (our Optimizer and Differential Analyzer) to reduce the manual effort required to analyze the generated models, exploring automatic model analysis techniques leveraging LLMs could further assist in model analysis to reduce dependence on domain expertise. Further, extending QUICTester to evaluate non-security components of QUIC are avenues for future work.

## REFERENCES


[1] [n. d.]. Aioquic. https://aioquic.readthedocs.io/en/latest/. Accessed: 10 October 2022.

[2] [n. d.]. BoringSSL. https://boringssl.googlesource.com/boringssl/. Accessed: 7 June 2022.

[3] [n. d.]. Bug Description with input sequence to reproduce the faults. https://anonymous.4open.science/r/QUICTester-7EBC/results/README.md. Accessed: 2 August 2024.

[4] [n. d.]. rr: lightweight recording & deterministic debugging. https://rr-project.org/. Accessed: 16 January 2023.

[5] [n. d.]. Usage statistics of HTTP/3 for websites. https://w3techs.com/technologies/details/ce-http3. Accessed: 7 June 2023.

[6] Bernhard K Aichernig, Edi Muškardin, and Andrea Pferscher. 2021. Learning-based fuzzing of IoT message brokers. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*. 47–58.

[7] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: high-throughput fuzzing of network applications. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 340–351.

[8] Kian Kai Ang and Damith C. Ranasinghe. 2025. QUIC-Fuzz: An Effective Greybox Fuzzer For The QUIC Protocol. In *European Symposium on Research in Computer Security (ESORICS)*.

[9] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.

[10] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *USENIX Security Symposium (USENIX Security)*. 3255–3272.

[11] Nils Bars, Moritz Schloegel, Nico Schiller, Lukas Bernhard, and Thorsten Holz. 2024. No Peer, no Cry: Network Application Fuzzing via Fault Injection. (2024).

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security and Privacy (S&P)*. 535–552. https://doi.org/10.1109/SP.2015.39

[13] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[14] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*. 114–129. https://doi.org/10.1109/SP.2014.15

[15] Yufan Chen, Arjun Arunasalam, and Z Berkay Celik. 2023. Can large language models provide security & privacy advice? measuring the ability of llms to refute misconceptions. In *Annual Computer Security Applications Conference (ACSAC)*. 366–378.

[16] Tsun S. Chow. 1978. Testing software design modeled by finite-state machines. *IEEE transactions on software engineering* 3 (1978), 178–187.

[17] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 110–125.

[18] Christophe Crochet, Tom Rousseaux, Maxime Piraux, Jean-François Sambon, and Axel Legay. 2021. Verifying QUIC implementations using Ivy. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*. 35–41.

[19] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. 2018. Inferring OpenVPN state machines using protocol state fuzzing. In *IEEE European Symposium On Security And Privacy Workshops (EuroS&PW)*. 11–19.

[20] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium (USENIX Security)*. 193–206.

[21] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 423–435.

[22] Tim Dierks and Eric Rescorla. 2008. The transport layer security (TLS) protocol version 1.2. RFC 5246.

[23] Tim Dierks and Eric Rescorla. 2008. The transport layer security (TLS) protocol version 1.2. RFC 5246.

[24] Michael Eddington. [n. d.]. Peach fuzzing platform. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.. Accessed: 2 August 2024.

[25] Fátima Fernández, Mihail Zverev, Pablo Garrido, José R Juárez, Josu Bilbao, and Ramón Agüero. 2020. And QUIC meets IoT: performance assessment of MQTT over QUIC. In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*.

[26] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 762–774.

[27] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification (CAV)*.

[28] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri De Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS implementations using protocol state fuzzing. In *USENIX Security Symposium (USENIX Security)*. 2523–2540.

[29] Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2023. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations.. In *Network and Distributed System Security (NDSS)*.

[30] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. 2017. Model learning and model checking of SSH implementations. In *ACM SIGSOFT International Symposium on Model Checking of Software (SPIN)*. 142–151.

[31] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. 2010. Learning of event-recording automata. *Theoretical Computer Science* 411, 47 (2010), 4029–4054.

[32] Kaiyu Hou, Sen Lin, Yan Chen, and Vinod Yegneswaran. 2022. QFaaS: accelerating and securing serverless cloud networks with QUIC. In *Symposium on Cloud Computing (SoCC)*. 240–256.

[33] Yating Hsu, Guoqiang Shu, and David Lee. 2008. A model-based approach to security flaw detection of network protocol implementations. In *IEEE International Conference on Network Protocols*. IEEE, 114–123.

[34] Christian Huitema, Sara Dickinson, and Allison Mankin. 2022. DNS over Dedicated QUIC Connections. RFC 9250. https://doi.org/10.17487/RFC9250

[35] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. 2021. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4G LTE cellular devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1082–1099.

[36] IoT Analytics GmbH. 2023. State of IoT 2023: Number of connected IoT devices growing 16% to 16.7 billion globally. https://iot-analytics.com/number-connected-iot-devices.

[37] Malte Isberner. 2015. Foundations of active automata learning: an algorithmic perspective. (2015).

[38] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT algorithm: a redundancy-free approach to active automata learning. In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer, 307–322.

[39] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The open-source learnLib: a framework for active automata learning. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*. Springer, 487–495.

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

[41] jtpereyda. [n. d.]. BooFuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[42] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the Internet Measurement Conference*. 290–303.

[43] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[44] Imtiaz Karim, Abdullah Al Ishtiaq, Syed Rafiul Hussain, and Elisa Bertino. 2023. BLEDiff: Scalable and Property-Agnostic Noncompliance Checking for BLE Implementations. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 3209–3227.
[45] Puneet Kumar and Behnam Dezfouli. 2019. Implementation and analysis of QUIC for MQTT. *Computer Networks* 150 (2019), 28–45.
[46] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *USENIX Security Symposium (USENIX Security)*. 4481–4498.
[47] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. 2022. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR)*.
[48] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. 2018. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security (ESORICS)*. 325–345.
[49] Kenneth L McMillan and Lenore D Zuck. 2019. Formal specification and testing of QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. 227–240.
[50] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Network and Distributed System Security (NDSS)*.
[51] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 460–465.
[52] Jon Postel. 1981. Transmission control protocol. RFC 793.
[53] Harald Raffelt, Maik Merten, Bernhard Steffen, and Tiziana Margaria. 2009. Dynamic testing via automata learning. *International Journal on Software Tools for Technology Transfer (STTT)* 11, 4 (2009), 307–324. https://doi.org/10.1007/s10009-009-0120-7
[54] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. 2019. State machine inference of QUIC. *ArXiv* (2019).
[55] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: a differential fuzzing framework to detect DPI elusion strategies for QUIC. In *Annual Computer Security Applications Conference (ACSAC)*. 332–344.
[56] Eric Rescorla. 2018. The transport layer security (TLS) protocol version 1.3. RFC 8446.
[57] Eric Rescorla and Brian Korver. 2003. *Guidelines for writing RFC text on security considerations*. Technical Report.
[58] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519591
[59] Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1492–1504.
[60] Simeng Sun, Yang Liu, Dan Iter, Chenguang Zhu, and Mohit Iyyer. 2023. How does in-context learning help prompt tuning? *arXiv preprint arXiv:2302.11521* (2023).
[61] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. 2017. Model-based testing IoT communication via active automata learning. In *IEEE International conference on software testing, verification and validation (ICST)*. 276–287.
[62] Martin Thomson and Sean Turner. 2021. Using TLS to secure QUIC. RFC 9001.
[63] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan 2017), 86–95. https://doi.org/10.1145/2967606
[64] Neil Walkinshaw and Kirill Bogdanov. 2013. Automated comparison of state-based software models in terms of their language and structure. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 1–37.
[65] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal fuzzing with large language models. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 1–13.

# APPENDIX

## A SYMBOLIZATION WITH LLMS

Recently, language models (LLMs) have been investigated as potential tools for supporting fuzzing [21, 43, 50, 65]. In compliance testing with active learning, we explore the utility of LLMs to automate the manual task of examining a protocol specification to generate the input and output symbols for the QUIC Learner. We perform an evaluation on OpenAI's GPT-4o (the most advanced model at the time of writing) using the prompt engineered in Figure 11. Inspired by [50], to ensure the LLM model always returns the symbols in a consistent format, we employ *in-context few-shot learning* [13, 60], a prompt engineering technique that helps the LLM model to understand the desired pattern of the output based on given examples. In this evaluation, we ask the LLM to return *only* valid messages that are exchanged during a QUIC handshake. The symbols from the responses can be grouped into 2 categories: i) valid symbols—valid messages in the desired format; and ii) invalid symbols—either invalid messages (incorrect packet and frame combination) or symbols that do not align with the format we requested. We sampled 50 responses from the LLM and combined them into one symbol set as shown in Figure 12. We measure the number of misses and hallucination rates. Number of misses refers to the generation of missing information based on the ground truth, and Hallucination refers to the generation of untruthful information [15].

From our evaluation, we see that the answers given by the LLM is not guaranteed and consistent. The ground truth, extracted from manually inspecting the specification, consists of 25 symbols; however, the LLM only generates 8.6 symbols on average (3 symbols are given as part of the prompt). This results in a 69% missing rate on average as shown in Table 4. In addition, as shown in Table 5 the symbols sampled across 50 queries have an average of 18.6% hallucination rate. Therefore, we do not rely on symbol generation using an LLM model because it requires multiple queries to get all the correct symbols and, subsequently, an expert with protocol-specific knowledge to examine the symbols to eliminate undesirable invalid symbols.

**Prompt**

Instruction: According to RFC 9000 (QUIC specification), list all the packet and frame type combinations that can exist during a QUIC handshake.

Desired Format:

**Shot-1**:
For the QUIC protocol, the Initial packet with CRYPTO frame carrying a Client Hello TLS message will be:
Initial_Client_Hello

**Shot-2**:
For the QUIC protocol, the Handshake packet with a PING frame will be:
Handshake_Ping

**Shot-3**:
For the QUIC protocol, the 1-RTT packet with CRYPTO frame carrying a Finished TLS message frame will be:
Handshake_Finished

**Figure 11: An example prompt to automatically symbolize all valid messages during a QUIC handshake.**

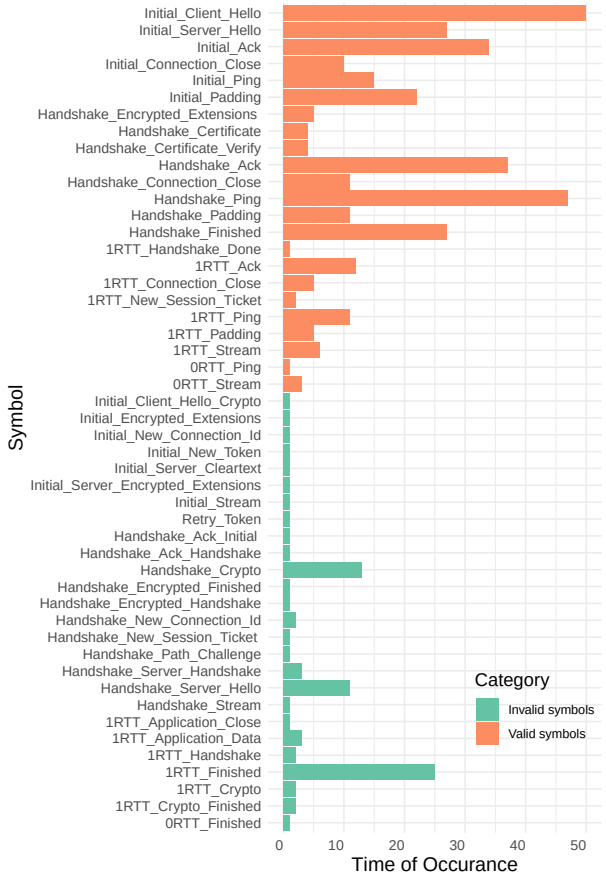



**Figure 12: Symbols in 50 answers sampled from OpenAI GPT-4o model using the prompt shown in Figure 11.**

## A.1 Extended Material on Automatically Learning a Behavior Model and Symbolization (Learner)

We described the general role of a learner in Section 2—recall, the Learner generates input sequences from a set of input symbols. We curated and defined symbols for protocol parameters and messages, parameterized by time, for QUIC. These are summarized in Table 8 in the Appendix. Notably, to increase the readability of the models, these symbols are represented using their acronyms. Specifically, for input test cases, we included symbols:

- To construct *all* valid QUIC messages in a handshake, including Ping and ConnectionClose messages, to increase the learning coverage as these messages can exist in both Initial and Handshake packets [40, Section 17].
- To allow constructing *invalid* messages violating the specification. These messages carry one of the following: i) incorrect header fields (e.g. Connection ID); ii) not permitted frame structures (e.g. frame without any content); and iii) frame with incorrect content (e.g. Invalid Certificate).

**Table 4: Symbol missing rate in 50 LLM queries compared to the ground truth extracted from RFC9000 by a domain expert.**

| Symbol (Ground Truth) | Number of Occurances | Number Missed | Missing Rate(%) |
|---|---|---|---|
| Initial_Client_Hello | 50 | 0 | 0% |
| Initial_Server_Hello | 27 | 23 | 46% |
| Initial_Ack | 34 | 16 | 32% |
| Initial_Connection_Close | 10 | 40 | 80% |
| Initial_Ping | 15 | 35 | 70% |
| Initial_Padding | 22 | 28 | 56% |
| Handshake_Encrypted_Extensions | 5 | 45 | 90% |
| Handshake_Certificate | 4 | 46 | 92% |
| Handshake_Certificate_Verify | 4 | 46 | 92% |
| Handshake_Ack | 37 | 13 | 26% |
| Handshake_Connection_Close | 11 | 39 | 78% |
| Handshake_Ping | 47 | 3 | 6% |
| Handshake_Padding | 11 | 39 | 78% |
| Handshake_Finished | 27 | 23 | 46% |
| 1RTT_Handshake_Done | 1 | 49 | 98% |
| 1RTT_Ack | 12 | 38 | 76% |
| 1RTT_Connection_Close | 5 | 45 | 90% |
| 1RTT_New_Session_Ticket | 2 | 48 | 96% |
| 1RTT_Ping | 11 | 39 | 78% |
| 1RTT_Padding | 5 | 45 | 90% |
| 1RTT_Stream | 6 | 44 | 88% |
| 0RTT_Ping | 1 | 49 | 98% |
| 0RTT_Stream | 3 | 47 | 94% |
| Retry | **0** | 50 | 100% |
| 1RTT_New_Connection_Id | **0** | 50 | 100% |
| | | **Average** | **69%** |

**Table 5: The hallucination rate of the symbols sampled across 50 queries.**

| Number of Queries | Number of Symbols Sampled | Number of Correct Symbols | Number of Invalid Symbols | Hallucination Rate(%) |
|---|---|---|---|---|
| 50 | 430 | 350 | 80 | 18.6 |

While, for QUIC protocol responses or output symbols, we include symbols:

- To construct *all* valid QUIC client response messages.
- To elicit the status (connection availability) of the QUT which would otherwise be hidden from the Learner. This addition is important to detect DoS attacks (such as **M**-18 in Table 2) and uncover non-compliant behaviors of the QUT; process a message where it should not, for example, initialConnectionClose after employing the Handshake encryption key [62, Section 4.9].

So, the crafted input symbols include *all* valid QUIC client messages described in Section 2.1. We also added Ping and ConnectionClose messages to increase the learning coverage as these messages can exist in both Initial and Handshake packets [40, Section 17]. Additionally, we include *12* invalid QUIC messages that violate the specifications [40, 56, 62]: i) initialClientHello-invldACK; ii) handshakeEmptyCertificate; iii) handshakeInvalidCertificate; iv) InvalidNewConnectionID; v) initialNoFrame; vi) initialUnexpectedFrameType; vii) handshakeNoFrame; viii) handshakeUnexpectedFrameType; ix) 0rttNoFrame; x) 0rttUnexpectedFrameType; xi) 0rttFinished; and xii) 0rttACK. We also crafted two input symbols for Test Harness configuration settings for the Learner to select during learning: i) [RemovePaddingFromInitialPackets]; and ii) [ChangeDestination-ConnectionID-Original]. A detailed description of each input is included in Table 8.

The output symbols are message types a QUIC protocol will respond with, these include the messages described in Section 2.1,

PingACK and ConnectionClose. Importantly, we include two output symbols to elicit the status of the QUT which would otherwise be hidden from the Learner. We check the QUT status by sending a ping message after sending a ConnectionClose message carried by either an Initial or a Handshake packet. The QUT is considered alive if it acknowledges the ping. The symbols are enclosed in angle brackets. <ConnectionClose> indicates the QUT has closed the connection, and <ConnectionActive> denotes that the connection with the QUT is still active. This addition is important for testing the SUT state for conditions where it should not process a message, for example, initialConnectionClose or handshakeConnectionClose messages as stated in [62, Section 4.9]. A detailed description of each output is included in Table 8.

# B DEMONSTRATING A VALID BEHAVIOR ANALYSIS

In Figure 13, *s0* denotes the state the server is fully initialized and waiting for incoming connections. The client first sends an initialClientHello-validACK and receives a retry from the server for client address validation, then transitions to *s1*. With the [IncludeRetryToken] input at *s1*, the client is configured to start including the received RetryToken in all its following Initial packets. From *s2* to *s3*, the client sends an initialClientHello-validACK that includes the received RetryToken. The Ngtcp2 server verifies the RetryToken and responds with initialServerHello, handshakeEncryptedExtension, handshakeCertificateRequest, handshakeCertificate, handshakeCertificateVerify, handshakeFinished and ValidNewConnectionID messages. Importantly, the handshakeCertificateRequest message indicates to the client it must present a certificate for authentication to proceed.

Subsequently, the client sends handshakeCertificate to the server at *s3*, followed by handshakeCertificateVerify. At this point, the server verifies the client certificate, and the handshake transitions to *s6*. The client sends handshakeFinished and the server validates the message to ensure that the previous handshake messages have not been modified. The handshake transitions to *s7* after the server responds with ValidNewConnectionID and HandshakeDone messages. Now the QUIC handshake is considered confirmed. The handshake ends at *s8* after the client completes the exchange of the ValidNewConnectionID message that carries several new connection IDs that can be used for the established connection. Notably, the handshake can still proceed despite the client sending initialClientHello-invalidACK at *s2*, which contains an invalid Initial ACK to acknowledge the server's initialServerHello message. This is because the server recognizes and drops the invalid Initial ACK. But, it continues with the handshake process when it receives a valid Handshake ACK from the client, acknowledging the Handshake packets correctly. *This is an example of the expected handshake flow according to the specification.*

# C ADDITIONAL CASE STUDIES

We provide additional case studies in this section, covering more memory corruption bugs and logical flaws discovered by QUICTESTER.

## C.1 Memory-corruption bugs: Server crashes

We discovered 8 memory-corruption bugs. The inputs for reproducing each bug as well as a detailed description of each bug are available at our open-source QUICTESTER code repository on GitHub [3]. Here, in addition to Section 5.2, we detail three additional memory-corruption bugs that can result in DoS attacks.

M-8 **Null Pointer Dereference in Quinn.** When testing Quinn, our crash logger detected crashes when handling hndUnxpFrType. These crashes arise from Quinn *panicking* when it attempts to unwrap a None value after matching an unexpected frame to the Type enum.

*Impact* This allows an attacker to perform a DoS attack using a malformed packet (hndUnxpFrType). Notably, the bug exists in both the server and client implementations since they share the same library. We responsibly reported this vulnerability to the Quinn developers. This vulnerability was assigned CVE-2023-42805 with high severity and patched.

M-2 **Null Pointer Dereference in Neqo.** Upon looking at the Learner logs, it was found that the Neqo server crashed with an assertion error in cases where the selected input sequence contains an initialConnectionClose message that precedes an initialClientHello message. This assertion is also guaranteed to occur when the input sequence includes the initialConnectionClose message but lacks the initialClientHello message. Based on our findings, it appears that the server attempts to respond with a ConnectionClose message. However, it cannot obtain the connection's primary path when creating the message. This happens because the server will only set a primary path for that connection when it receives and processes an initialClientHello message. This finding demonstrated that the Learner logs are helpful in detecting memory-corruption bugs that are not directly shown in the learned models.

*Impact* This vulnerability allows an attacker to launch a denial of service (DoS) attack on Neqo servers by sending a single initialConnectionClose input at the start of a connection establishment.

M-3 **Limited connections due to a hardcoded value.** The Neqo server in all configurations can only accept at most 32,767 connections, including closed connections. After the 32,767$^{th}$ connection, the server crashes with an assertion error when the variable of `PRDescIdentity` data type is storing a value that is equal to `int_16_max()` at the beginning of the `PD_GetUniqueIdentity()` function. This function creates a unique identity for each connection, and each identity is assigned a unique identity number. The unique identity number starts from 0 and then increases by 1 for every new unique identity created. To ensure the identity is unique, the server uses a variable to track the most recent unique identity number. In the `PD_GetUniqueIdentity()` function, if the current unique identity number is equal to int_16_max(), it will stop the unique identity creation and raise an assertion error. As described in Mozilla's documentation, the data type, `PRIntn`/`PRDescIdentity`, that is used to store the unique identity number is guaranteed to be at least 16 bits, but the architecture that runs the server can define it to be wider, such as 32 bits or 64 bits. However, due to the hardcoded comparison value, int_16_max() in the assertion statement,

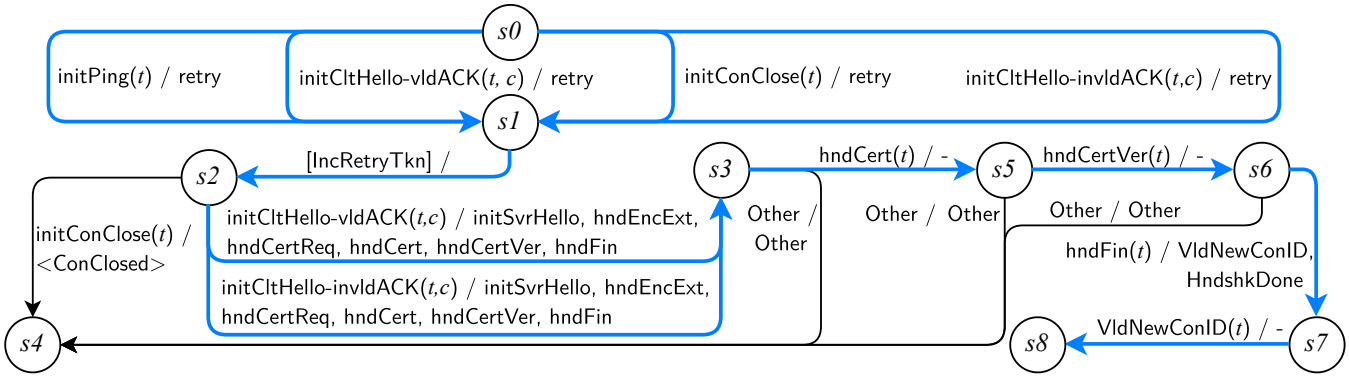



**Figure 13: Optimized learned model of a Ngtcp2 server with the RetryClientAuth configuration, generated from a Learner with the symbols required for RetryClientAuth to generate a simplified model for illustration. Here, blue edges are the valid path to complete a RetryClientAuth QUIC handshake.**

our architecture that defined `PRDescIdentity` to 32 bits which can actually create up to 2,147,483,647 unique identities, will still crash after the 32767$^{th}$ connection. This may affect the performance of the Neqo server because, with the hard-coded value in the assertion, it is guaranteed to crash on the 32,768$^{th}$ connection, no matter the data type used by the architecture.

*Impact* This allows an attacker to perform a DoS attack by establishing more than 32,767 QUIC connections with the server. Both **M-2** and **M-3** were fixed by Neqo developers. Notably, the developers stated these vulnerabilities only affect the server-side implementation as they currently focus on client implementation.

## C.2 Logical Flaws: Unexpected Behaviors

We discovered 3 logical flaws. The inputs for reproducing each bug as well as a detailed description of each bug are available at our open-source QUICTESTER code repository on GitHub [3]. Here, we detail two more logical flaws.

**L-2 Incorrect method of emptying the re-transmission queue.** Because Pquic is built on top of the older Picoquic library, it shared the issue **M-4** discussed in Section 5.2 with Picoquic. However, unlike Picoquic, which attempts to access a `NULL` pointer to obtain the encryption key for re-transmission, Pquic will always have access to the encryption keys because it never discards them—see **S-10**.

*Impact* This behaviour causes the server to re-transmit acknowledged messages to the peer, unnecessarily increasing network traffic and reducing the utility of network bandwidth.

**L-3 Infinite loop when processing frame type** `0xFF`. When the PQUIC server processes a packet carrying a `0xFF` frame type, the server always gets stuck in a loop that attempts to match `0xFF`, an invalid frame type, with the expected frame type. This issue is specific to the `0xFF` frame type and does not happen with other invalid frame types.

*Impact* Because PQUIC is running on a single thread, getting stuck in an infinite loop causes the PQUIC server to become unavailable to serve any client until the server administrator manually restarts

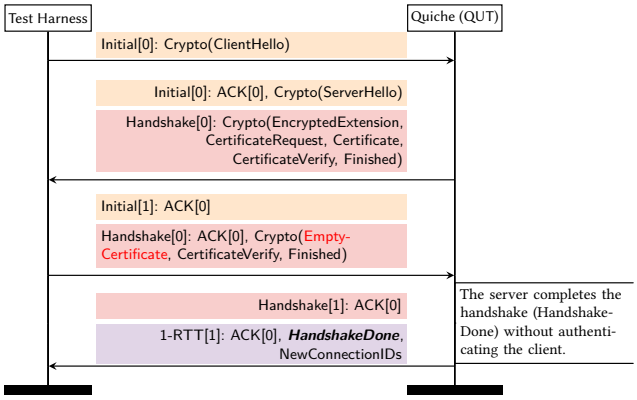



**Figure 14: Client authentication bypass in Quiche. The invalid (EmptyCertificate) message is shown in red text.**

it. This allows an attacker to perform a DoS attack on the server using the message described above.

# D REFERENCE MODELS

As explained in Section 7, we have curated 11 reference models from our experiment. These reference models are FSMs with no specification violations. Our library has at least one reference model for each security configuration we tested that developers can use with QUICTESTER. We have curated 11 reference models (3 for Basic, 2 for Retry, 2 for ClientAuth, 1 for RetryClientAuth and 3 for PSK). The main differences between the reference models in the same security configuration but from different vendors are due to the variations in the interpretation and implementation of the response to the initialPing message sent by a client as the first message. For example, Ngtcp2 server drops the first initialPing message from clients, while the Quicly server responds to the initialPing. As we discussed in Section 5.4, the current specification does not explicitly state the expected behavior of a server to a first packet received without a CRYPTO frame. In our reference models, we

consider both implementations as adhering to the specification. All the reference models can be found on our public GitHub repository https://github.com/QUICtester.

## E QUICTESTER IMPLEMENTATION EFFORT

We summarize our implementation effort in Table 6.

**Table 6: Extensions made to implement QUICTester.**

| Component | Library | Lines of Code |
|---|---|---|
| Learner | LearnLib | 627 |
| Mapper | Aioquic | 3560 |
| Optimizer | - | 416 |
| Differential Analyzer | LTSDiff | 719 |

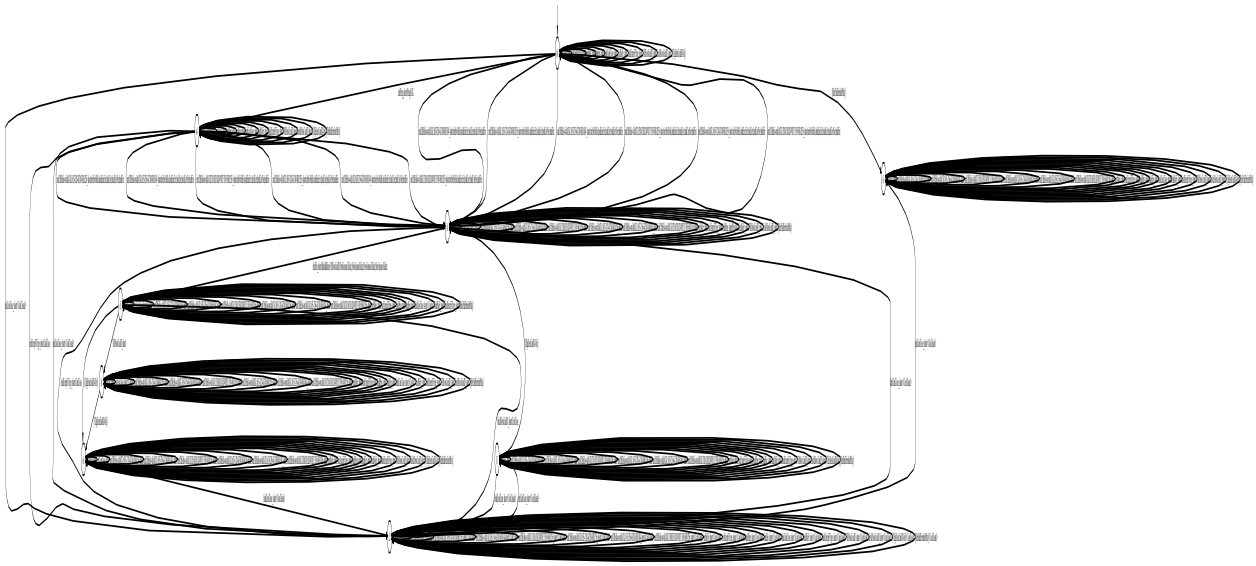


**Figure 15: The learned model of Quinn—notably the QUIC implementation with the valid FSM—in the most simple, Basic configuration before optimization. This illustration, whilst not fully legible, is provided to show an example of the complexity created in even the most basic security configuration for a learned model before optimization. The model after using our Optimizer is shown in Figure 16.**

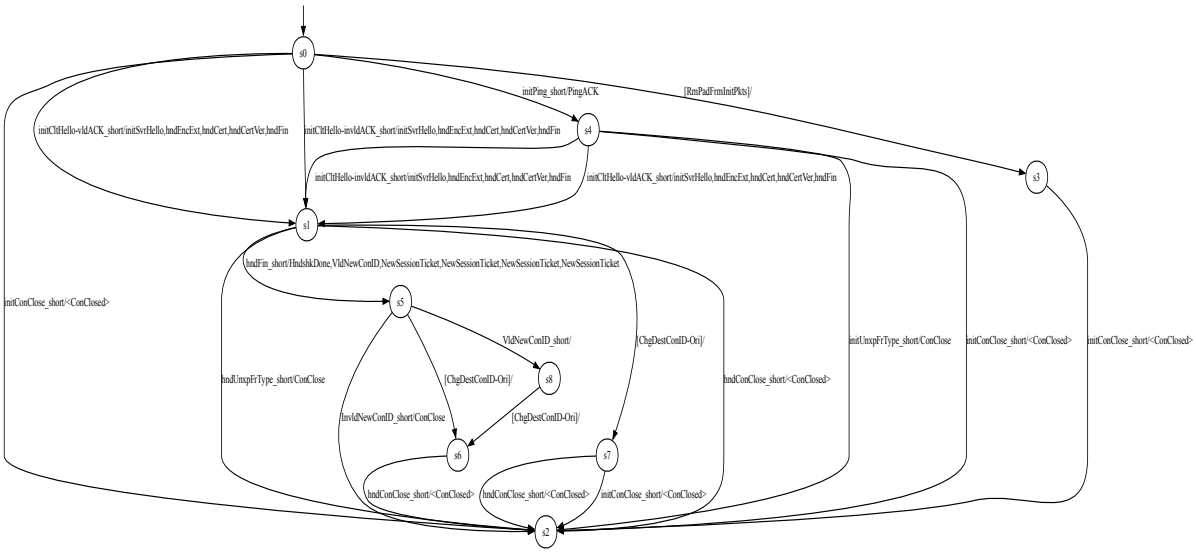



**Figure 16: The learned model of Quinn Basic after employing our Optimizer (compared with Figure 15 generated prior to simplification, there are far fewer edges, greatly improving *interpretability* and the task of model analysis).**

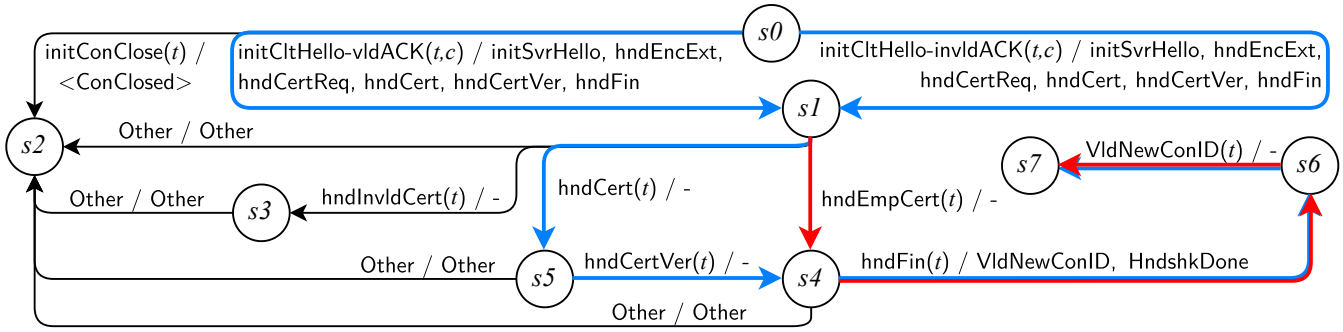



**Figure 17: Optimized learned model of a Quiche server with the ClientAuth configuration. Blue edges show a valid path to complete a QUIC handshake. Red edges demonstrate an invalid path that bypasses the client authentication, but still completes the handshake without errors.**

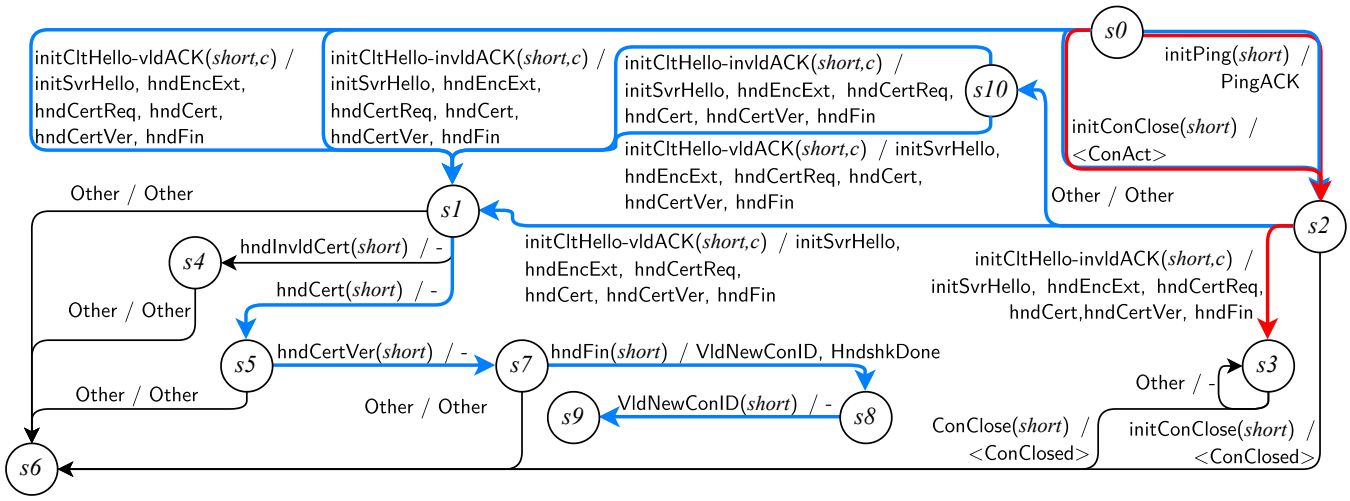



**Figure 18: An optimized model from Picoquic with ClientAuth configuration learned with $t = short$ parameter setting for inputs. Differental analysis with Figure 19 using $t = long$ reveals a software bug exploit.**

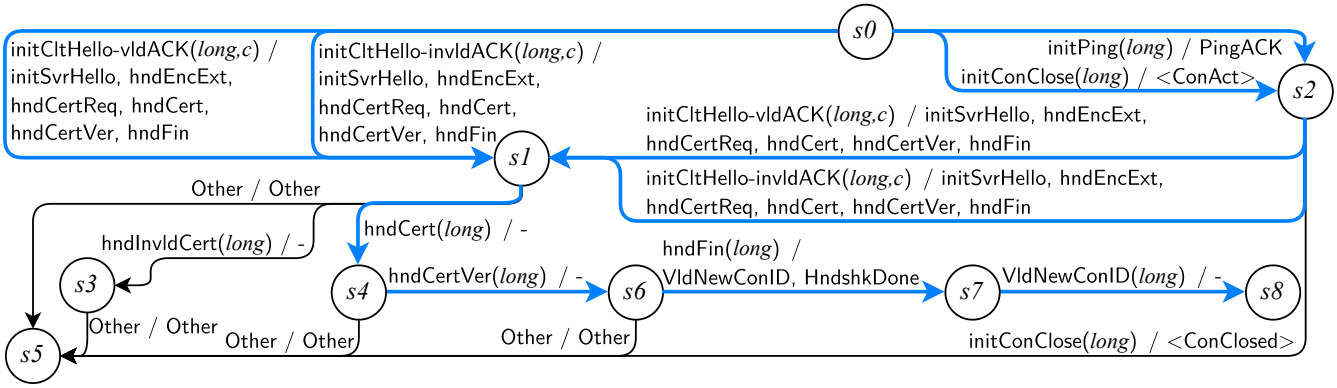



**Figure 19: An optimized model from Picoquic with ClientAuth configuration learned with $t = long$ parameter setting for inputs.**

**Table 7: Overview of identified faults. In total, 44 specification violations, 8 memory-corruption bugs and 3 logical flaws were identified in 19 QUIC implementations. *Specification bugs* (S): An implemented behavior violates the QUIC specification. *Memory-corruption bugs* (M): An input causing a memory corruption and a server crash. *Logical flaws* (L): Incorrect logic implemented in code produces unexpected behavior.**

| Server | Fault Description | Type-ID | Disclosed | Most Recent Response to Disclosure |
|---|---|---|---|---|
| Aioquic | Incorrect handling of packets with unexpected frame type. | **S**-1 | ✓ | Fixed. |
| Kwik | CVE-2024-22588: Retention of the unused encryption keys. | **S**-2 | ✓ | Fixed. |
| | CVE-2024-22590: Implementation without a TLS state machine. | **S**-3 | ✓ | Fixed. |
| | Process CRYPTO frame in a 0-RTT packet. | **S**-4 | ✓ | Acknowledged findings. |
| | Exceeds the operating system's maximum number of memory mappings for a single process (100,000) when receiving PING frame from 50,000 clients. | **M**-1 | ✓ | Acknowledged findings. |
| Lsquic (Lite Speed) | CVE-2024-25678: Retention of the unused encryption keys (PSK configuration in 1b113d19). | **S**-5 | ✓ | Fixed. |
| | Incorrect handling of re-transmission, leaving a half-opening connection on the client side (PSK configuration in v4.0.2). | **L**-1 | ✓ | Fixed. |
| MsQuic (Microsoft) | Does not issue its initial_source_connection_id at the correct connection state. This finding is part of the connection management ambiguity discussed in Section 5.4. | **S**-6 | ✓ | Developers acknowledge the findings and plan to propose an amendment to the *QUIC specification* to address the ambiguity. |
| Neqo (Mozilla) | NULL pointer dereference when getting the primary path. | **M**-2 | ✓ | Fixed. |
| | Limited connections due to a hardcoded value. | **M**-3 | ✓ | Fixed. |
| Picoquic | NULL pointer dereference when getting the encryption keys. | **M**-4 | ✓ | Fixed. |
| | Retry token tied to retry_source_connection_id. | **S**-7 | ✓ | Developer stated the code was written specifically to add an additional constraint to ensure the client follows the specification. Fix: Propose an *amendment to the QUIC specification* to address the ambiguity. |
| PQUIC | Invalid original_destination_connection_id. | **S**-8 | ✓ | Fixed. |
| | Limitless active_connection_id_limit. | **S**-9 | ✓ | Fixed. |
| | CVE-2024-25679: Retention of the unused encryption keys. | **S**-10 | ✓ | Fixed. |
| | Incorrect way of emptying the re-transmission queue. | **L**-2 | ✓ | Fixed. |
| | NULL pointer dereference when handling removed connection context. | **M**-5 | ✓ | Acknowledged findings. |
| | Buffer overflow when processing frame type 0x30. | **M**-6 | ✓ | Acknowledged findings. |
| | Infinite loop when processing frame type 0xFF. | **L**-3 | ✓ | Acknowledged findings. |
| | Does not send HANDSHAKE_DONE after the handshake is confirmed (PSK configuration). | **S**-11 | ✓ | Acknowledged findings. |
| Quiche (Cloudflare) | Client authentication bypass due to incorrect flag set in Quiche library. | **S**-12 | ✓ | Fixed with bug bounty awarded. |
| | Incorrect handling of **all** Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-13 | ✓ | Acknowledged findings. |
| Quiche4j | Concurrent modification exception when discarding closed connections. | **M**-7 | ✓ | Acknowledged findings. |
| | Limitless active_connection_id_limit. | **S**-14 | ✓ | Acknowledged findings. |
| Quant | Incorrect handling of an initialPing message. | **S**-15 | ✓ | Acknowledged findings. Not fixed because the GitHub Repository is no longer under active maintenance. |
| | Incorrect handling of **all** Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-16 | ✓ | |
| Quiwi | Does not close the connection when the number of received NEW_CONNECTION_ID frames exceed the active_connection_id_limit. | **S**-17 | ✓ | Acknowledged findings. |
| Quinn | CVE-2023-42805: Panic when unwrapping a None value when processing an unexpected frame type. | **M**-8 | ✓ | Fixed. |
| | Process CRYPTO frame in 0-RTT packet. | **S**-18 | ✓ | Fixed. |
| XQUIC (Alibaba) | Retention of the unused encryption keys. | **S**-19 | ✓ | Acknowledged findings. Potential security vulnerability. **Unresolved for over 90 days**. See here. |
| | Maintaining a number of active connection IDs that exceed the active_connection_id_limit. | **S**-20 | ✓ | Fixed. |
| Aioquic, LSQUIC, Neqo, Quic-go, Quinn, Quiwi, S2n-quic (Amazon), XQUIC | Incorrect handling of the second and subsequent Initial packets carried in a UDP datagram with a payload size smaller than 1200 bytes. | **S**-21 to **S**-28 | ✓ | Aioquic, LSQUIC, S2n-quic: Fixed. However, Neqo, Quic-go and Quinn teams acknowledged the protocol violation. Fix: Developers propose amending the QUIC specification to provide futher clarifications. Others: Acknowledged findings. |
| Aioquic, Kwik, MsQuic, LSQuic, Quant, Quiche, Quic-go, Quiche4j, Quiwi, S2n-quic | Accept Handshake packet from an unmatched Destination Connection ID. | **S**-29 to **S**-38 | ✓ | Aioquic, Kwik, Lsquic, S2n-quic: Fixed. Others: Acknowledged findings. |
| Lsquic, MsQuic, Neqo, Quiche4j, Quinn, XQUIC | Incorrect handling of packets without a frame. | **S**-39 to **S**-44 | ✓ | Lsquic, Quinn, XQUIC : Fixed. Others: Acknowledged findings. |

**Table 8: Symbolized QUIC messages, configuration settings and QUT status (connection active or closed) used by the Learner and the Mapper (here, *we only mention the symbols used in the paper*). The variable $t$ represents the possible timeout and the variable $c$ represents the possible cipher suite that the Learner can select. The Configuration settings (options for the Test Harness selected by the Learner) are within square brackets. The output symbols that show the hidden status of the QUT are within angle braces. In the learned models, we represent the symbols using their acronym form for brevity.**

| Input Symbol | Acronym | Description |
|---|---|---|
| initialPing($t$) | initPing($t$) | An Initial packet with a PING frame. |
| initialConnectionClose($t$) | initConClose($t$) | An Initial packet with a CONNECTION_CLOSE frame. |
| initialNoFrame($t$) | initNoFr($t$) | An Initial packet without a frame. |
| initUnexpectedFrameType($t$) | initUnxpFrType($t$) | An Initial packet with 0xFF frame type. |
| initialClientHello-validACK($t$,$c$) | initCltHello-vldACK($t$,$c$) | An Initial packet with a CRYPTO frame carrying Client Hello message. This input will respond to the Server Hello message with an Initial packet with an ACK frame with PADDING frames. |
| initialClientHello-invalidACK($t$,$c$) | initCltHello-invldACK($t$,$c$) | An Initial packet with a CRYPTO frame carrying Client Hello message. This input will respond to the Server Hello message with an Initial packet with an ACK frame with no PADDING frames. $c \in \{AES\_128, AES\_256, ChaCha20\}$ |
| 0rttPing($t$) | 0rttPing($t$) | A 0-RTT packet with a PING frame. |
| 0rttConnectionClose($t$) | 0rttConClose($t$) | A 0-RTT packet with a CONNECTION_CLOSE frame. |
| 0rttNoFrame($t$) | 0rttNoFr($t$) | A 0-RTT packet without a frame. |
| 0rttUnexpectedFrameType($t$) | 0rttUnxpFrType($t$) | A 0-RTT packet with 0xFF frame type. |
| 0rttFinished($t$) | 0rttFin($t$) | A 0-RTT packet with a CRYPTO frame carrying Finished message. |
| 0rttACK($t$) | 0rttACK($t$) | A 0-RTT packet with an invalid ACK frame. |
| handshakePing($t$) | hndPing($t$) | A Handshake packet with a PING frame. |
| handshakeConnectionClose($t$) | hndConClose($t$) | A Handshake packet with a CONNECTION_CLOSE frame. |
| handshakeNoFrame($t$) | hndNoFr($t$) | A Handshake packet without a frame. |
| handshakeUnexpectedFrameType($t$) | hndUnxpFrType($t$) | A Handshake packet with 0xFF frame type. |
| handshakeEmptyCertificate ($t$) | hndEmpCert($t$) | A Handshake packet with a CRYPTO frame type carrying an empty list of certificates. |
| handshakeInvalidCertificate($t$) | hndInvldCert($t$) | A Handshake packet with a CRYPTO frame type carrying a certificate that is not signed by the certificate authority used for verification. |
| handshakeCertificate($t$) | hndCert($t$) | A Handshake packet with a CRYPTO frame type carrying a certificate that is signed by the certificate authority used for verification. |
| handshakeCertificateVerify($t$) | hndCertVer($t$) | A Handshake packet with a CRYPTO frame type carrying Certificate Verify message. |
| handshakeFinished($t$) | hndFin($t$) | A Handshake packet with a CRYPTO frame type carrying Finished message. |
| ValidNewConnectionID($t$) | VldNewConID($t$) | A 1-RTT packet with a number of NEW_CONNECTION_ID frames that follows the number of Connection IDs that the QUT can support. |
| InvalidNewConnectionID($t$) | InvldNewConID($t$) | A 1-RTT packet with a number of NEW_CONNECTION_ID frames that exceed the number of Connection IDs that the QUT can support. |
| [IncludeRetryToken] | [IncRetryTkn] | Instructs the Mapper to include the Retry Token in its following Initial packets. |
| [RemovePaddingFromInitialPackets] | [RmPadFrmInitPkts] | Instructs the Mapper to remove PADDING frames from its following Initial packets. |
| [ChangeDestinationConnectionID-Original] | [ChgDestConID-Ori] | Instructs the Mapper to change the Destination Connection ID of its following packets to original_destination_connection_id. |
| **Output Symbol** | **Acronym** | **Description** |
| retry | retry | A Retry packet that carries a Retry Token |
| initialServerHello | initSvrHello | A Server Hello message encapsulated in a CRYPTO frame of an Initial packet. |
| handshakeEncryptedExtensions | hndEncExt | An Encrypted Extensions message encapsulated in a CRYPTO frame of a Handshake packet. |
| handshakeCertificateRequest | hndCertReq | A Certificate Request message encapsulated in a CRYPTO frame of a Handshake packet. |
| handshakeCertificate | hndCert | A Certificate message encapsulated in a CRYPTO frame of a Handshake packet. |
| handshakeCertificateVerify | hndCertVer | A Certificate Verify message encapsulated in a CRYPTO frame of a Handshake packet. |
| handshakeFinished | hndFin | A Finished message encapsulated in a CRYPTO frame of a Handshake packet. |
| HandshakeDone | HndshkDone | A HANDSHAKE_DONE frame in a 1-RTT packet. |
| NewToken | NewTkn | A NEW_TOKEN frame in a 1-RTT packet. |
| ValidNewConnectionID | VldNewConID | A NEW_CONNECTION_ID frame in a 1-RTT packet. |
| PingACK | PingACK | A PING ACK frame in either Initial, Handshake, or 1-RTT packets. |
| ConnectionClose | ConClose | A CONNECTION_CLOSE frame in either Initial, Handshake, or 1-RTT packets. |
| <ConnectionActive> | <ConAct> | Indicates the QUT has closed the connection. |
| <ConnectionClosed> | <ConClosed> | Indicates the connection with the QUT is still active. |