Identifying Logical Vulnerabilities in QUIC Implementations

Kaihua Wang*, Jianjun Chen*‡, Pinji Chen*, Jianwei Zhuge*, Jiaju Bai† , Haixin Duan*
*Tsinghua University †Beihang University
{wkh23, cpj24}@mails.tsinghua.edu.cn {jianjun, zhugejw, duanhx}@tsinghua.edu.cn baijiaju@buaa.edu.cn

***Abstract*—QUIC is a modern transport protocol increasingly adopted by major platforms and services, making its security and correctness critically important. However, the complexity of QUIC specification and implementations introduces opportunities for subtle and dangerous logic flaws. Existing QUIC testing tools primarily focus on memory-related vulnerabilities and are ill-equipped to detect logical vulnerabilities. Therefore, the discovery of logical vulnerabilities is currently still highly dependent on manual auditing.**

In this paper, we introduce MerCuriuzz, a novel black-box fuzzing framework designed to automatically uncover logical vulnerabilities in QUIC implementations. We evaluated MerCuriuzz against 16 widely used QUIC implementations and discovered 14 previously unknown logical vulnerabilities affecting popular implementations such as quiche, xquic, and aioquic. Those vulnerabilities can pose severe security risks, enabling attackers to exhaust server resources, crash services, or deny legitimate users access to the server. We categorize those vulnerabilities into six categories and propose mitigation strategies. We also responsibly disclosed our findings to the affected vendors, and 11 of them were confirmed and rewarded by the vendors, such as Cloudflare and Alibaba Cloud.

I. Introduction

The standardization of the QUIC protocol has progressed over nearly four years [65]. According to global website usage statistics from W3Techs [58], over 33% of websites now support HTTP/3—which is based on QUIC—up from 25% in 2022. This rapid growth highlights QUIC's emergence as a core component of modern internet infrastructure, with adoption expected to continue rising. Thanks to its integrated design that mandates TLS encryption and eliminates handshake latency [7], QUIC not only boosts communication efficiency but also enhances security. Furthermore, its ability to support faster data transfer and improved connection stability compared to traditional TCP has made it a preferred choice for content delivery networks (CDNs) and large-scale platforms. For instance, Akamai and Kuaishou have collaborated to optimize video streaming over QUIC [2], while major CDN providers like Cloudflare and Fastly, and platforms such as YouTube, TikTok, and Facebook, have widely adopted QUIC as their primary transport protocol.

Given this widespread deployment, vulnerabilities in QUIC implementations pose serious security risks to the Internet. While QUIC incorporates numerous security features by design, recent reports [18], [19] have revealed a growing class of logical vulnerabilities—subtle flaws arising not from memory corruption but from incorrect implementations logic. These vulnerabilities can be just as dangerous, enabling denial-of-service (DoS) attacks, state inconsistencies, and other unintended behaviors. A notable example is CVE-2024-22189 [45], which is similar to the HTTP/2 Rapid Reset vulnerability [28] and allowed attackers to overwhelm servers with minimal effort by exploiting a subtle logic flaw in connection migration. Unfortunately, state-of-the-art fuzzing and automated testing techniques have limited success in detecting such logic flaws in QUIC, due to the following three reasons.

First, there is a lack of effective and efficient mutation techniques for the QUIC protocol to generate high-value frame sequences that can cover a larger number of program paths. Triggering logical vulnerabilities in the QUIC protocol often requires sending many carefully crafted packets in a specific sequence, which leads to a huge mutation state space. Given that the number of states grows exponentially in the sequence quantity, existing techniques based on random mutation are infeasible to test the QUIC protocol in practice. Second, logical vulnerabilities often do not result in common error manifestations such as crashes or overflows, thus traditional fuzzing frameworks have difficulty detecting whether a vulnerability has been triggered. Third, QUIC services involve complex interactions, including port registration, service initialization, and frequent process restarts. These operations require resetting the service state before each fuzzing iteration, which significantly undermines the efficiency of the testing process.

To address these challenges, we propose `MerCuriuzz`, a novel testing method capable of automatically discovering logical vulnerabilities in QUIC implementations. To tackle the first challenge, we design a segmental mutation strategy that decomposes test case generation into three separate mutation stages, which maintains good mutation efficiency even when dealing with complex and long QUIC frame sequences. For the second challenge, we introduce a semantic and public resource consistency detector based on differential testing. This method allows us to detect traffic anomalies and potential

‡Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.231777
www.ndss-symposium.org

vulnerabilities in two differential QUIC services based on their traffic characteristics and usage of public resources. For the third challenge, we design and implement a snapshot manager based on Nyx snapshot technology [50], [51], which enables millisecond-level snapshot saving and recovery. This design eliminates the overhead associated with resetting the system state after each test case, including operations such as service initialization and connection establishment.

We implemented the MerCuriuzz prototype and evaluated it on 16 open source QUIC implementations recommended by the QUIC work group [17]. Consequently, we discovered 14 logical vulnerabilities across these implementations, affecting well-known projects such as quiche, xquic, and aioquic. We further categorize them into 6 typical classes and demonstrate their impact through controlled experiments on real-world implementations. We responsibly disclosed these findings to vendors and received vulnerability rewards from Cloudflare and Alibaba, along with five CVE IDs. Furthermore, we analyzed the common root causes of these logical vulnerabilities and proposed mitigation strategies to both the QUIC specification and implementation developers.

In summary, we make the following contributions:

- We proposed MerCuriuzz[1], a novel methodology for automatically discovering logical vulnerabilities in QUIC implementations, enabling efficient identification of such vulnerabilities.
- We implemented our approach and evaluated it on 16 mainstream QUIC implementations, discovering 14 logical vulnerabilities and 6 categories of attacks. These lead to various security issues, such as resource exhaustion and assertion failures, enabling denial-of-service attacks on major QUIC services.
- We responsibly disclosed our findings to affected vendors and received positive feedback. We also presented three mitigation recommendations to address such logical vulnerabilities of QUIC.

II. Background

A. QUIC Protocol

QUIC is a modern, multiplexed transport protocol developed by Google and standardized by the IETF as RFC 9000 [56]. It is designed to provide low-latency, secure, and reliable communication over UDP, integrating features traditionally split between TCP and TLS into a unified framework. QUIC has gained increasing popularity in recent years, with 14 QUIC-related RFC documents proposed successively, including HTTP/3 and DNS Over QUIC [8], [31], [56], [57]. Generally, the QUIC protocol can be logically decomposed into two phases: (1) QUIC connection establishment or migration, and (2) QUIC data transmission.

QUIC connection establishment. Initially, the QUIC protocol launches a handshake to establish a secure and authenticated connection between the client and the server. Unlike traditional TCP + TLS protocols, which require multiple round-trip times (RTTs) to complete the handshake and encryption setup, QUIC integrates the cryptographic handshake into the transport layer and completes the entire process in a single RTT under normal circumstances. This phase includes the following steps: (1) *Connection Initiation:* The client initiates the connection by sending a ClientHello message, encapsulated within a CRYPTO frame, alongside a randomly generated Connection ID (CID); (2) *Cryptographic Negotiation:* The server responds with a ServerHello message and cryptographic parameters, establishing encryption keys; (3) *0-RTT Data (optional):* If the client has previously connected to the server, it may send early application data using 0-RTT encryption keys derived from session resumption for reconnecting; (4) *1-RTT Completion:* The handshake concludes once the server verifies the client's finished message and both parties derive the final 1-RTT keys.

QUIC connection migration. One of the distinguishing features of the QUIC protocol is its native support for connection migration. Unlike traditional transport protocols, which bind the transport-layer connection to a four-tuple of IP addresses and ports, QUIC decouples connection identity from endpoint addresses by introducing CID. This design enables seamless migration of ongoing connections across network paths without requiring a full renegotiation, e.g., when a mobile device switches from Wi-Fi to cellular. In certain scenarios, QUIC allows the use of 0-RTT data during connection migration, particularly when session resumption is enabled. However, due to the risk of spoofing and path-based attacks, path validation is required. This process involves: (1) Path Challenge: Upon detecting a potential change in path, the peer sends a randomly generated challenge token using a PATH_CHALLENGE frame; (2) Path Response: The recipient must echo the challenge value in a PATH_RESPONSE frame; (3) ACK Confirm: If the peer receives the path response, it returns an ACK frame to confirm address ownership and reachability. Only after this validation succeeds, QUIC allows application data to be sent on the new path.

QUIC data transmission. Once the handshake is complete and secure communication is established, QUIC is changed to the data transmission phase. Instead of delivering a continuous byte stream, QUIC employs a frame-oriented abstraction, wherein each QUIC packet encapsulates one or more self-describing frames. These frames are broadly categorized into data frames and control frames. Data frames (STREAM frames) are used to carry application data, while control frames are utilized for managing connection state, flow control, congestion control, and transport-level signaling. For instance, the NEW_CONNECTION_ID control frame allows connection migration by issuing new CIDs. This flexible framing system allows QUIC to multiplex multiple types of information within the same packet, enabling more efficient utilization of the network. Additionally, QUIC offers TCP-equivalent reliability through a combination of mechanisms. First, each STREAM frame includes offset and length fields, allowing data to be reassembled in order and detect losses on a per-stream basis. Second, QUIC uses rich ACK frames that

[1]https://github.com/k4ra5u/MerCuriuzz

support selective acknowledgment of received packet ranges, enabling efficient retransmission strategies and minimizing head-of-line blocking. Third, QUIC maintains packet-level state, including packet numbers and transmission timestamps. Based on acknowledgments and timeouts, the sender can retransmit lost data frames, ensuring guaranteed delivery.

B. Protocol Logical Vulnerabilities

Protocol logical vulnerabilities have become an important area of research in the field of cybersecurity in recent years. These vulnerabilities differ significantly from traditional memory vulnerabilities in both nature and exploitation methods. Traditional memory vulnerabilities, such as buffer overflows and use-after-free (UAF) errors, are primarily due to low-level programming errors related to memory management. In contrast, protocol logical vulnerabilities arise from design flaws in the protocol logic, regardless of memory safety. Exploiting logical vulnerabilities does not require memory corruption, instead, attackers manipulate the system's intended logic to achieve unintended behavior.

Landscape of logical vulnerability. In recent years, the increasing complexity of network protocols and their widespread adoption [22] have driven a rapid growth in research on protocol logical vulnerabilities. Taking the QUIC protocol as an example, vulnerabilities in QUIC prior to 2021 were predominantly memory-related issues, such as stack corruption. However, after 2021, there was a significant rise in protocol logical vulnerabilities. This trend is caused by several factors. On one hand, the QUIC protocol has only been officially standardized for three years, and its complex protocol logic has not yet undergone extensive testing, with many QUIC features still under development. On the other hand, due to advancements in security tools, such as the increasing adoption and improvement of tools like Sanitizers, traditional memory vulnerabilities have become harder to exploit, prompting attackers to focus more on logical vulnerabilities. This shift in vulnerability landscape motivates us to systematically identify and analyze logical vulnerabilities in QUIC implementations.

III. Overview

A. Threat model

In our attack, the attacker first establishes a secure and legitimate connection with the victim's QUIC server, then sends a carefully crafted sequence of QUIC frames over this connection to induce the victim into executing flawed code logic. As a result, the victim enters an erroneous program state and fails to provide proper QUIC interactions for users, thereby achieving a DoS attack. To ensure the practicality of our attack in real-world scenarios, we assume the following basic properties.

In terms of attacker capability, we assume two capabilities. First, the attacker can normally interact with the target QUIC server, which can use the correct QUIC version and application name to establish a connection. Second, the attacker can locally execute the victim program and observe its public resource usage metrics as provided by the operating system. This assumption is feasible, for example, many closed-source PCDN services provide local deployment solutions [42], [55].

In terms of the victim QUIC server, first, we assume that the server is properly deployed, bound to a valid IPv4 network address and port, and has unrestricted network connectivity, allowing it to interact with external clients without firewall or NAT constraints. Second, we do not impose any assumptions or limitations regarding the implementation language of the server. The server may be written in any programming language, as our approach is agnostic to the underlying technology stack.

B. Challenges

Although QUIC protocol logical vulnerabilities can have serious practical impacts, research on automatically discovering logical vulnerabilities in the QUIC protocol is still lacking. To develop a fuzzing framework tailored for the QUIC protocol that can efficiently detect state machine corruption type logical vulnerabilities, three fundamental challenges must be addressed:

Challenge 1: How to find frame sequences covering more program paths in a huge mutation state space? To cover as many program states as possible, the transmitted data frames must follow both the syntax rules (as defined in the QUIC RFC for data and control frames) and the semantic rules (based on the state machine of the QUIC server). We need to consider the dependencies between each frame in the sequence. If we use traditional field-based mutation methods, such as those in Bleem [29], to blindly mutate every field in the frame to be sent, the state space of mutated frames will increase exponentially with the length of the frame sequence. One available solution is to generate frame sequences in batches based on predefined rules, as in TLS-Anvil [30]. However, such rules depend on the developer's experience and can only detect known issues, failing to uncover unknown errors. Moreover, as a stateful protocol based on UDP, the behavior of a QUIC implementation can be influenced by factors such as the timing of the data sequence, packet ordering, packet loss, and retransmission, leading to unique states. Therefore, existing mutation strategies in protocol fuzzing cannot address the challenges of mutating QUIC frame sequences.

Our solution. To address this challenge, we designed the *segmental mutation* for QUIC protocol data. This is a multi-stage test case generation strategy that divides the mutation process into three targets: *frame content mutator*, *frame sequence generator*, and *interaction behavior mutator*. In the frame content mutator stage, we focus on the definitions and specifications of each frame as provided in the RFC documentation, performing mutations within syntactic constraints. In the frame sequence generator stage, we predefine multiple generation algorithms for each type of frame, expanding a single frame into a complete sequence. In the interaction behavior mutator stage, we simulate and mutate various characteristics of the connection, such as response delay, frame loss, retransmission, and out-of-order delivery.

Challenge 2: how to detect abnormalities in program behavior and state? When a program exhibits logic errors, there are generally two main manifestations. The first is the inconsistency between its response to packets and the Oracle standard. Since the RFC documentation does not define every detail of QUIC interaction behavior, we must evaluate the causes and types of inconsistencies and ignore those that are unrelated to vulnerability exploitation. The second manifestation is in the occupation of public resources. Logical vulnerabilities typically exhibit normal memory states, leading to a high rate of false negatives. Therefore, we must design a new observation and feedback mechanism to determine and verify whether the current interaction results in state or logic anomalies.

Our solution. To address this challenge, we developed *State Anomaly Detectors* that can detect whether logical vulnerabilities have been triggered and assess the expected value of the current interaction. Our Detectors consist of two main parts: *(1) Semantic Consistency Detector.* This is a differential testing module. By analyzing the returned frame sequences, it detects behavioral inconsistencies between implementations. To reduce false positives caused by minor variations, we ignore certain frames and contents that do not affect the state of the connection, and instead focus primarily on whether control frame logic and data frame content meet the expected conditions. *(2) Resource Consumption Detector.* First, we define three types of resources: computational resources, memory resources, and network handle resources. After obtaining the average values of resource consumption from baseline tests, we track the resource usage before and after each fuzzing session, paying attention to whether it exceeds the threshold generated by the baseline tests.

Challenge 3: how to mitigate the time overhead introduced by state resets and connection recovery? As a stateful communication protocol, a QUIC connection is forcefully closed whenever one endpoint encounters a state error or receives an unexpected packet, with the endpoint sending a Connection Close frame. This closure resets all accumulated connection states. Whether reconnecting or restoring the previous state, these processes incur significant time overhead. More importantly, they consume computational resources and reduce the accuracy of anomaly detection. Therefore, it is crucial to reduce time overhead from repetitive tasks like state resets and connection recovery, such as server restart and TLS handshake.

Our solution. To address this challenge, we designed a snapshot manager based on the fast snapshot restoration technique of Nyx [50]. Since Nyx requires a QEMU [44] virtual machine for the target program, we split the data collection component of the detector from its core logic and merged it with the QUIC communication module to form a data converter. This converter is primarily responsible for bridging the fuzzer and the QUIC server running inside the virtual machine. Each target implementation is encapsulated as a System Under Test (SUT), containing the target QUIC server, the data converter module, and the harness. Upon initial startup, the harness controls the converter to establish a connection with the QUIC server. Once the connection is established, a snapshot node is created. The relay handles communication with the QUIC server and reports test results back to the external fuzzer. After each fuzzing round, the harness restores the snapshot to the moment just after the connection is established, thereby saving the time otherwise required for service restart and connection setup.

IV. MerCuriuzz

We propose MerCuriuzz, a black box fuzzing framework for QUIC protocol. Unlike traditional black box fuzzers focusing on memory vulnerabilities, MerCuriuzz is capable of automatically discovering logical vulnerabilities in QUIC implementations through segmental mutation and state anomaly detectors. It also enables efficient state and connection resets using a snapshot manager. In the following sections, we provide a detailed description of the design and implementation of these components. In addition, to support our mutations and detectors, we design the Compressed Corpus for test case format, and construct the Monte Carlo seed tree as the scheduler. The implementation details of these components are described in Appendix.

A. Workflow

Figure 1 illustrates the workflow of MerCuriuzz. Overall, MerCuriuzz consists of four main phases: (1) receiving input testcases and outputting objectives and crashes; (2) managing and mutating the corpus; (3) managing snapshots and performing differential evaluation; (4) executing input tests under QEMU virtual machine snapshots. The complete workflow includes the following seven steps:

① We extract part of the QUIC traffic from the real world. Then, we abstract the packet data to construct the initial seed. ② The seed is inserted into the seed tree. The MCTS algorithm recommends the next seed to fuzz. ③ The selected seed is mutated using Segmental Mutation to generate a complete test case. ④ The snapshot manager in the Diff Executor is responsible for establishing virtual machine snapshots and injecting the test case into two virtual machines. After execution, the snapshot manager restores VMs to their initial snapshot state, ready for the next round of testing. ⑤ The QUIC Converter processes the input from the host and packages it into QUIC format packets to be sent to the target. The internal Observers collect reports on public resource usage and the interaction flow, which are transmitted back to the host through a preconfigured channel. ⑥ In the Evaluation stage, two detectors assess whether any abnormal state or behavior occurs. ⑦ If a test case is identified as abnormal, it is reported as an objective or crash. If no anomaly is found but the test case is marked interesting, it is returned to the seed and the process repeats from Step ②.

B. Segmental Mutation

The core idea of segmental mutation is to reduce the explosion of the state space caused by excessive mutability of seeds.

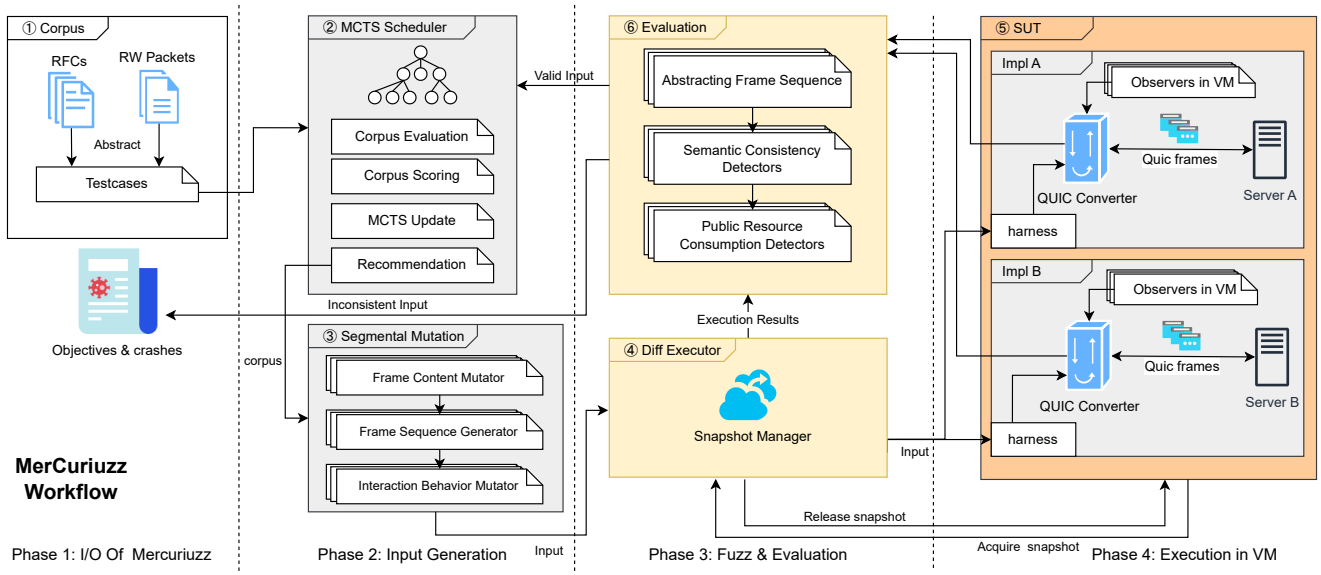

Fig. 1: Overview of MerCuriuzz.

We achieve this by extracting semantic constraints defined in the RFC for each QUIC frame and analyzing the dependencies among different frame types to prune and restrict mutation spaces that clearly violate protocol specifications. To support this, we design a custom test case format called Compressed Corpus (CC). In short, a CC consists of several base frames and a corresponding description document. The frame content mutator mainly performs modification operations on base frames. The frame sequence generator focuses on modifying the generation algorithm and the repeat count of base frames specified in the description document. The interaction behavior mutator alters the behavioral rules of the client defined in the document. By combining the base frames and the description document, we can reconstruct a complete frame sequence and interaction specification for fuzzing. Figure 2 illustrates the process by which we mutate the base frame sequence in the Compressed Corpus into a complete test case through the three mutators described above. Here we define the base frame sequence as a New Connection ID (NCI) frame.

Frame content mutator. Referring to the RFC specifications, we identify 34 distinct frame types, including both control and data frames defined in RFC 9000, as well as new frame types introduced by recent extensions such as the DATAGRAM frame in RFC 9221. We conduct a detailed analysis of the type characteristics and specific characteristics of all the fields within these frames. We categorize all field types into three major type characteristics: Content fields, which represent a sequence of raw bytes; Length fields, which indicate the length of the content field; Numerical fields, which cover all numerical values except for length indicators. Each field is subject to both general RFC constraints (e.g., no numerical field may exceed 2^{62}) and frame-specific constraints (e.g., in NCI frames, the seq_num must be greater than the maximum existing CID seq_num). We perform multiple rounds of frame content mutation; in each round, we randomly select one of the three field types and mutate a specific field while ensuring all constraints are met. As a result, frame content mutation ensures that all mutated frames remain compliant with the RFC and can be correctly accepted and processed by the server.

Frame sequence generator. Building on the frame content mutator, we introduce the frame sequence generator. We collect all inter frame constraints described in the RFC documents. Starting from the base frames within the seed, the generator applies the selected algorithm to construct successor frames, forming a basic frame sequence. We predefine various algorithms, such as fixed step increments of numerical fields or overlapping / gaped modifications of offset / length fields, to ensure that different frame types comply with protocol constraints. By iterating over all base frames, the frame sequence generator assembles a complete initial frame sequence. Although this approach may exclude some valid mutation spaces, it significantly improves the quality of generated test cases by reducing the exponential growth of the mutation space to a linear scale. In Figure2, we describe a sequence length of n, so the frame sequence generator produces n NCI frames numbered 1 to n.

Interaction behavior mutator. In addition to mutating frame content and sequence, the transmission order and packet-loss scenarios of QUIC frames are equally critical due to UDP's unreliable datagram nature. QUIC servers may experience out-of-order frame reception, loss of critical frames, or receipt of multiple retransmitted frames; peers may similarly see delayed or missing replies. To model these scenarios, we design an interaction behavior mutator. Mutation strategies include partial frame dropping, repeated frame sending, and reordering the initial frame sequence based on specific algorithms. We predefine various remapping algorithms, such as

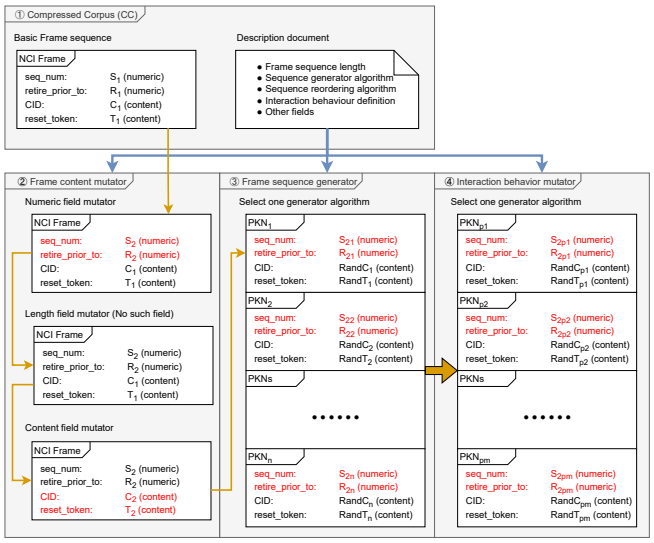

Fig. 2: Mutation workflow diagram. The basic frame sequence is mutated along the orange lines to generate complete test cases, while the document is sent to each mutator along the blue lines, specifying the mutation strategy.

randomly deleting or shuffling one frame every k frames (e.g., in a 100-frame sequence, randomly reordering or deleting one frame every five frames). This introduces variability without prematurely terminating the connection. In Figure 2, we map the original sequence 1–n to a permuted sequence p_1 to p_m. Additionally, we impose rate limits on client-to-server reply frames; in multi-round interactions, delaying, canceling, or crafting incorrect replies can also lead to abnormal server states.

Benefit of our segmental mutation. Remarkably, our segmental mutation approach significantly reduces the mutation state space. For instance, if the initial corpus contains 100 `New Connection ID (NCI)` frames. Each frame includes fields such as the `seq_num`, `retire_prior_to`, the CID content, and the `reset_token`. Mutating all these fields could theoretically yield 2^{400} combinations. However, valid mutations require at least two following constraints:

$$SeqOfNewCid \geq SeqOfLatestCid$$

$$SeqOfRetireCid \in SeqOfActiveCids$$

`SeqOfLatestCid` is the largest `seq_num` in the currently stored list of CIDs. `SeqOfRetireCid` is the `retire_prior_to` field as well as the max identifier of the CID we prepare to retire; this identifier must be present in the current available CID list. Moreover, the values of the CID and reset token do not affect the program's execution state. Therefore, the mutation space for these 100 CIDs can be pruned to focus only on behaviors such as whether to mutate `SeqOfNewCid` or `SeqOfRetireCid`, reducing the mutation space to 2^2. This allows more computational resources to be allocated to determining how to construct these CID numbers.

C. State Anomaly Detector

Our state anomaly detector consists of two fundamental parts: (1) a semantic consistency detector and (2) a public resource consumption detector.

1) Semantic Consistency Detector: To efficiently identify the inconsistencies, it is impractical to conduct an exhaustive comparison of every frame sequence across all implementations. So, we abstract the frame sequences to extract relevant semantic information and focus on monitoring specific indicators. After each round of interaction between the MerCuriuzz and a QUIC implementation, these indicators are independently recorded and passed to evaluate the difference. Specifically, the process involves the following steps:

Abstracting frame sequence. To detect reply consistency, we send data to all tested endpoints at the same rate and in the same order during each fuzzing iteration. As previously mentioned, directly comparing the content of each reply frame may lead to excessive false positives due to implementation differences among servers, resulting in inefficiency. To address this, it is crucial to understand the root cause of inconsistencies. First, we abstract all response frame data and extract their key information. This key information is then compared against the requirements specified in the RFC and other response sequences to identify the causes of inconsistencies. Specifically, we consider both the content of sent frames and response frames. While sent frames are mostly identical, during multi-round interactions between the client and server, it is essential to understand the content of the sent frames to determine and compare the validity of the response frames. To achieve this, we extract the following features from the frame sequences: (1) frame sequence numbers, (2) frame types, and (3) the fields and content within the frames. In the abstracted frame sequences, each element is represented as a (sent, received) tuple. To simplify the model, we exclude frames actively sent by the server, such as ping frames and frames used to establish and initialize connections, as these frames are not directly controllable by the fuzzer.

Detecting the disconnection and checking the causes behind. When the server sends a `ConnectionClose` frame, it typically indicates that the server has detected a critical state error, potentially difficult to handle or indicative of a vulnerability. Similarly, other mandatory control frames such as `ApplicationClose` or `ResetStream` signify failures in different layers of the QUIC protocol state. Consequently, these control frames provide clear indicators of abnormal behavior. If, during differential testing, some servers throw such control frames while others continue to operate normally, we classify the input as potentially containing a vulnerability. To avoid excessive false positives caused by hard-coded server logic that might produce these control frames at different times or with mismatched sequence numbers, we do not consider their positions within the frame sequence.

Examining control frame type and quantity. If the program enters an abnormal state, it may manifest as an inability to generate response frames, a noticeably slower response rate,

or ignoring subsequent frames after detecting a state error. To identify such anomalies, we collect all response frames from each server and compare the types and counts of control frames, as well as response rates and contents. If any server's control frame messages differ significantly from the rest, it implies that the server's program state is in error.

Inspecting data frame payload. In addition to examining control frames that indicate state changes, we also verify the correctness of QUIC data frames. These include payload-carrying frames, such as `STREAM` and `CRYPTO` frames, as well as `ACK` frames used to acknowledge data from the peer. For payload-carrying frames, we provide identical backend resources to all servers and thus expect uniform response data. Upon receiving these data frames, we parse them according to the agreed-upon format (e.g., HTTP/3) and analyze fields that must match (e.g., page content, return header status codes). As the most commonly used server response frames, `ACK` frames are non-retractable. By parsing `ACK` frames, we can accurately determine which data the server has received and processed. Hence, we specifically designed an ACK range checker to record all `ACK` frames generated by each QUIC implementation in a single fuzz run and compute the full set of inputs each server has processed. If any implementation shows inconsistent range information (e.g., repeated acknowledgments or divergent ranges), the ACK range checker logs these discrepancies.

2) Public Resource Consumption Detector: **Setting public resource consumption baseline.** In order to identify abnormal resource consumption, it is essential to first characterize the expected baseline of normal resource usage. Since different implementations are written in various languages, we define and measure a public resource consumption usage baseline for each implementation. Under established connection conditions, the client transmits `path_challenge` frames padded to 1200 bytes at a rate of 100 Mbps to the server and waits for a `path_response`, calculating the average memory and CPU usage. We chose `path_challenge` frames because the server must fully read the frame data and construct a padded response frame. Once processed, these `path_challenge` frames are discarded, without allocating or occupying additional runtime memory.

Component of public resource consumption detector. Our detector encompasses three components: (1) CPU Resource Usage Detector. We employ a CPU usage monitor, assigning a fixed CPU thread for each server. By observing whether CPU usage surpasses a defined threshold during fuzzing, we determine if the server is experiencing excessive public resource consumption. (2) Memory Resource Usage Detector. In a manner similar to CPU resource monitoring, at the end of each fuzz iteration, we read the `/proc/<pid>/maps` file to obtain the current memory usage. (3) Network Resource Usage Detector. Considering that network bandwidth usage can vary substantially based on fuzzing content and configurations, this detector focuses on network handle usage and service availability. Specifically, it checks whether the server retains too many network handles without promptly releasing them, and whether the service remains capable of receiving complete QUIC requests and completing the key exchange process after each fuzz iteration.

D. Snapshot Manager

To avoid the performance overhead caused by frequent QUIC service restarts and state recoveries during fuzzing, we introduce a snapshot manager based on Nyx. As illustrated in Figure 1, the snapshot manager coordinates the delivery of mutated inputs to two target implementations. However, traditional Nyx only supports coverage-guided greybox fuzzing, which cannot accurately record and restore snapshots when dealing with black box applications. To address this limitation, we construct a dedicated System Under Test (SUT). We utilize shared memory to establish input and output channels between the snapshot manager and the SUT, enabling complex data collection and message forwarding within the SUT itself. Additionally, we implemented a harness running inside the virtual machine. When the harness detects that the converter has successfully established a connection with the target implementation, it notifies the snapshot manager to create a restore point for the entire SUT. This mechanism eliminates the need for restarting the service and resetting the connection in each fuzzing iteration.

SUT design. The original Nyx-fuzz framework provides only three communication channels for interaction between the fuzzer and the in-VM target: input delivery from the fuzzer to the target, synchronization, and coverage feedback transmission from the target back to the fuzzer. However, inputs based on memory pipes and file-based formats cannot be directly consumed by the target. Moreover, since the target runs entirely inside the virtual machine, deploying observers on the host side to monitor the target's usage of shared system resources is infeasible. To address these limitations, we made the following modifications to the snapshot workflow.

First, we extended the Nyx codebase by introducing a shared memory region res_shm that is accessible and modifiable by both the snapshot manager and the virtual machine. We then designed and implemented a component named `quic_converter`, which is responsible for establishing and maintaining a QUIC connection with the target after startup. Upon receiving compressed test cases from the snapshot manager, `quic_converter` decompresses them into a complete frame sequence and interacts with the target according to the specifications in the associated descriptor. To enable traffic monitoring and resource usage analysis, we decoupled the Observer and Evaluator components within the state anomaly detector. The observer, embedded in `quic_converter`, monitors system resource usage and QUIC traffic during execution, and submits this data via res_shm to the evaluator for further assessment after each test case is completed. Finally, we implemented a harness that runs at the beginning of each VM boot-up. The harness uses Nyx's internal APIs to register all communication channels inside the VM, invokes the target program, and delivers inputs from the snapshot manager to the

target. It is also responsible for informing Nyx when to create or restore a snapshot for the SUT.

Rapid status recovery. To ensure that the SUT can immediately parse and transmit data to the target after each snapshot restoration, we configure the system snapshot created by the harness to cover both the `quic_converter` and the target. During the initialization phase, the harness launches the target and waits for it to start listening on its designated port. Once the target is ready, it launches the `quic_converter`. When a successful QUIC connection is established between the `quic_converter` and the target, the `quic_converter` sends a signal to the harness, which then instructs the snapshot manager to save a snapshot. After each test case execution, once the interaction between the `quic_converter` and the target is completed, a second signal is sent to the harness. The harness then triggers the snapshot manager to restore the previously saved state. At this point, the SUT is rolled back to the moment immediately after the connection was established, thereby eliminating the overhead of both state and connection reinitialization on the target. This design enables efficient and continuous fuzzing.

V. Evaluation

A. Experiment Setup

Targets and experiment platforms. To evaluate MerCuriuzz, we tested the 16 actively maintained QUIC projects [1], [3], [6], [11], [16], [20], [25], [27], [35], [36], [40], [41], [43], [46], [47] recommended by the QUIC Working Group [17]. These projects include well-known open-source implementations from companies such as Cloudflare, Google, and Alibaba, as well as QUIC implementations in various programming languages, such as quic-go in Golang and aioquic in Python. For version selection, we uniformly chose the latest versions released before January 31, 2025. Detailed project and code lines are listed in Table III of Appendix C. For the test platform, we used a server equipped with two E5-2673V4 CPUs (40 cores, 80 threads, supporting the Intel PT technology required by NYX snapshots) and 128 GB of memory, running Ubuntu 24.04. Each QUIC service was allocated one dedicated CPU core and 4 GB of memory.

Details of experiment implementation. *Regarding the implementation of all MerCuriuzz features*, we wrote 18,000 lines of Rust code and 1,000 lines of C code to realize all the above designs. To improve development efficiency, we used LibAFL as the development framework and embedded these components into it. *Regarding the configuration of each QUIC test target*, all implementations except nginx were run in single-threaded mode to ensure accurate measurement of compute and network resources. Nginx cannot be configured in single-threaded mode because QUIC connection handling requires a dedicated worker thread. Thus, we configured it to use only one worker thread. *Regarding QUIC version and ALPN selection*, we used Version 1 and h3. The quinn platform supports only Draft 29 and hq (HTTP/0.9 over QUIC Draft 29), a relatively uncommon version currently supported by only seven implementations (s2n-quic, picoquic, neqo, lsquic, msquic, aioquic, and cloudflare). Therefore, we performed differential testing of quinn only against these. *Regarding application-layer resource configuration*, to ensure stable application-layer output, we used a single GET request for `/index.html` over HTTP/3. *Regarding selecting differential targets*, we did not designate any single implementation as an oracle, instead, we paired the 16 implementations in all possible combinations, generating 112 test pairs. We ran 24-hour fuzzing on each test pair and recorded all corpus entries that led to observed inconsistencies.

B. Results and Findings

MerCuriuzz generated a total of 10M test cases across all test targets. We collected all test reports and corpus information. First, we performed three rounds of traffic replay to eliminate false positives caused by public resource fluctuations. Then, we removed duplicate data based on traffic features and MerCuriuzz's anomaly judgments. Each report selected by the above steps was manually analyzed. In total, we discovered 14 new logic vulnerabilities in QUIC implementations affecting prominent cloud service providers such as Cloudflare and Alibaba Cloud. We have responsibly disclosed our findings to the affected vendors. Up to now, we have received five CVE IDs, two commitments to assign CVE IDs, and three Alibaba vulnerability database IDs. In addition, four vulnerabilities earned bug bounties. Table I summarizes the information on these vulnerabilities. We conducted an in-depth analysis of the experimental results and identified 6 novel interesting categories of logic vulnerabilities in QUIC. Below, we elaborate on how MerCuriuzz uncovered these vulnerabilities and explain the underlying principles behind them.

Category 1: Crypto Flood. Crypto frames in the QUIC protocol are used to transmit TLS handshake messages during connection establishment. When a message is long, crypto frames are segmented for transmission. Similar to QUIC stream structures, crypto frames provide simple in-order delivery of these messages, achieved via the offset and length fields in each crypto frame, which mark the current encrypted data length and its position. STREAM frames constrain peer-sent data size using the FIN bit and the MAX_STREAM_DATA transport parameter; unlike STREAM structures, crypto has no such mechanisms, allowing unlimited use of crypto frames to send data. Additionally, section 7.5 of the RFC mandates buffering at least 4096 bytes of out-of-order crypto frames but does not specify how to handle excessive out-of-order data. As shown in Figure 3(a), during fuzz testing, our frame sequence mutator generated approximately 500 crypto frame variants with different offset/length ranges; some implementations, such as aioquic, picoquic, and xquic, cached all these crypto frames in memory. Our memory resource detector captured the corresponding memory growth, thereby identifying this flaw.

Category 2: Path_challenge Flood. In connection migration, both endpoints use path_challenge (PC) and PATH_RESPONSE (PR) to verify a new path's reachability. When the server sends PR1 in response to the first PC frame PC1 but has not received an ACK for PR1 before the next

TABLE I: The 14 security defects discovered by MerCuriuzz

Bugs ID	Project Name	Language	Description	Impact	Status	Assign CVE	DSM	DSCD	DPCD	RAD
M01	Aioquic	Python	CRYPTO Flood	Memory Exhausting	fixed	●		✓		
M02	Picoquic	C	CRYPTO Flood	Memory Exhausting	fixed	●		✓		
M03	Aioquic	Python	PC Flood	Memory Exhausting	fixed	●			✓	
M04	Aioquic	Python	ACK Confusion	CPU Exhausting	fixed	●	✓		✓	
M05	Cloudflare	Rust	Connection Hold-on Flood	Silent DoS	fixed	-	✓	✓		
M06	H2o	C	ACK Confusion	Immediate Crash	reported	◑	✓	✓	✓	✓
M07	H2o	C	Double Unregistered CID	Immediate Crash	reported	◑	✓	✓	✓	✓
M08	Xquic	C	CRYPTO Flood	Memory Exhausting	fixed	○		✓		
M09	Xquic	C	PC Flood	Memory Exhausting	fixed	○		✓	✓	
M10	Xquic	C	NCI Flood	Memory Exhausting	fixed	○		✓	✓	
M11	Lsquic	C	Memory leak during TLS handshake	Memory Exhausting	reported	-	✓	✓		
M12	Aioquic	Python	Double Unregistered CID	CPU Exhausting	reported	-	✓		✓	
M13	Aioquic	Python	Close The Stream Twice	CPU Exhausting	reported	-	✓		✓	
M14	neqo	Rust	Assert Error	Immediate Crash	fixed	●	✓	✓	✓	✓

◑ M06 and M07 are fixing, and were promised assigned CVE IDs.
○ M09-M11 assigned Alibaba Vulnerability Library IDs, no CVE IDs.
DSM: Disable segmental mutation.
DSCD: Disable Semantic Consistency Detector.
DPCD: Disable Public Resource Consumption Detector.
RAD: Replace all anomaly detectors with LibAFL's default.

PC frame PC2 arrives, it must buffer both PC1 and PC2 until it either receives the corresponding ACK or determines that PR1 failed and the new path is unreachable. If the rate of clearing completed path challenges is slower than the arrival of new challenges, memory accumulation occurs. Our frame sequence mutator generated a large number of PC frames, and our interaction behavior mutator limited the fuzzer's send rate, revealing memory buildup in aioquic and xquic. We use Figure 3(b) to show how Path Challenge Flood works. Notably, the memory-consumption logic differs: aioquic leaks memory by failing to free buffers allocated for PC frames, whereas xquic accumulates too many PR frames unsent, causing the send queue to grow rapidly.

Category 3: New_connection_id Flood. New Connection ID (NCI) frames are used to request new CIDs, which identify new paths during connection migration. Generally, a QUIC server allows no more than four coexisting CIDs, so NCI frames use the retire_prior_to field to retire old CIDs and send Retire Connection ID (RC) frames. Under normal processing, after the client sends an NCI, the server sends an RC frame, and the client acknowledges it with an ACK. The problem arises in the server's processing of NCI frames and the acknowledgment of RC frames. Since handling an NCI frame involves multiple memory operations such as CID registration and storage, the server may respond to NCI frames with some delay. Additionally, if the client delays the acknowledgment of an RC frame, the server may assume it was lost and add it to the retransmission queue. Similar to previous cases, if the rate at which the server receives NCI frames exceeds its processing capacity, or if RC frames are added to the send queue faster than they are removed, the send queue may grow without bound. We illustrate the impact of this rate mismatch in Figure 3(c). In our test framework, the frame sequence mutator generated a large number of valid NCI frames, each retiring a previously registered CID to maintain the current CID count. During testing, we found that xquic can cache excessive RC frames, risking memory buildup.

Category 4: Connection Hold-On Flood. Unlike the previous three vulnerabilities involving direct unbounded memory consumption, we discovered a more subtle logic flaw while fuzzing Cloudflare Quiche's quiche-server (hereafter, the server). During testing, we observed a strange phenomenon: CPU and memory resources appeared normal, yet network resources reached the timeout limit and connections could not be established. After carefully reviewing the source code and network traffic, we summarized the vulnerability trigger process as shown in Figure 3(d), as follows: The server stores all client connections throughout their life cycles in a hash map; whenever QUIC data is received, it polls these clients to determine which client the data belongs to and processes them in order. At this time, a local variable total_write is defined to indicate how much data has been sent to the current client; however, if total_write is zero (meaning all streams are marked as closed), the developer erroneously uses break to exit the loop over all clients instead of skipping the current client. Suppose user Alice is loading an HTTP/3 page via the quiche-server. Bob opens numerous connections to occupy the front positions in the hash map and uses STOP_SENDING frames to close the streams on those connections; the quiche-server then ends the polling over clients prematurely and cannot send any data to Alice. This covert denial-of-service attack cannot be detected simply by monitoring process state or CPU and memory usage, and even traditional fuzzers cannot find it, because the attacker and victim are not on the same connection. MerCuriuzz reestablishes connections when network resource exhaustion is detected and evaluates their availability, thus reliably uncovering this type of vulnerability.

Category 5: Double Unregistered CID. We mentioned above how the server handles NCI frames. We discovered that this process can cause another logical vulnerability. As shown in Figure 3(e), since the number of stored CIDs is

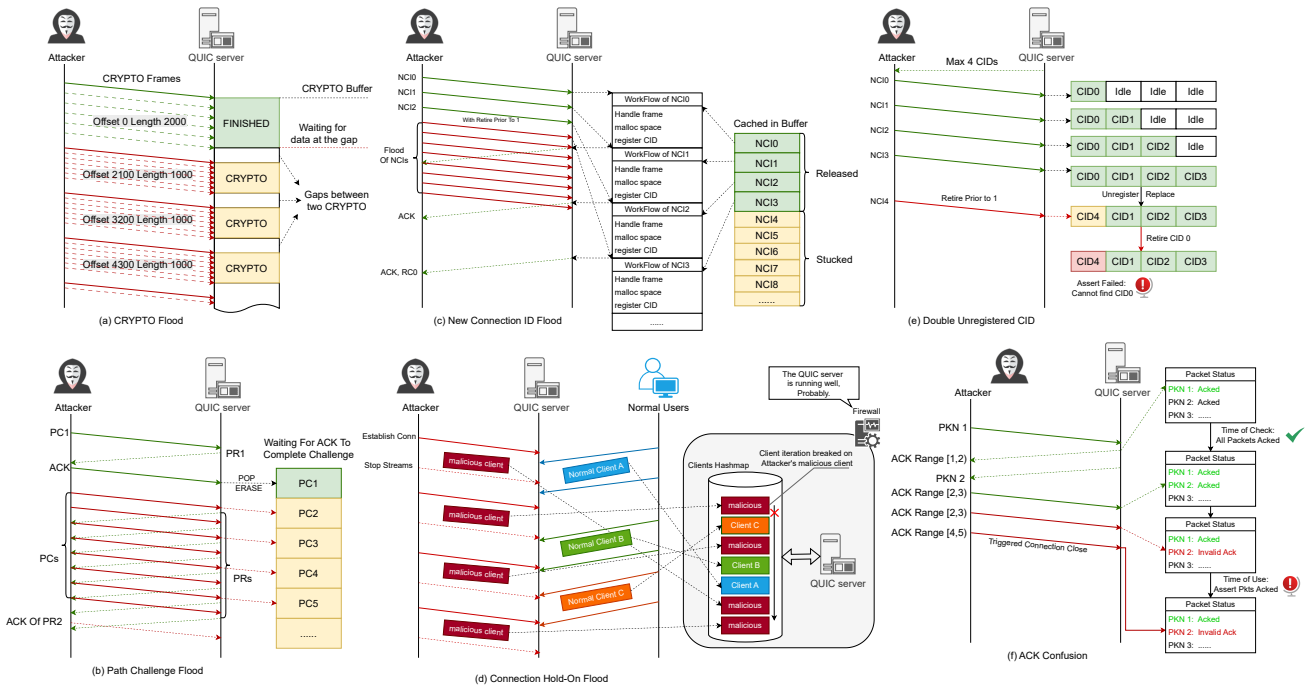

Fig. 3: During the fuzzing process, we identified 6 categories of interesting logical vulnerabilities. To illustrate their attack models, we present corresponding sequence diagrams.The vertical dimension of the diagram (the Y-axis) represents the flow of time, with events occurring sequentially from top to bottom.

fixed after negotiation, when the CID count is saturated, the server must first unregister older CIDs before adding new ones. A QUIC server must first process the retire_prior_to field to unregister all matching CIDs, and then replace them with new CIDs. However, the server also needs to notify the client of the retired CIDs via RC frames. If it does not temporarily store the CIDs that have been unregistered, it fails to find the CID and encounters an error. In our testing, we found that h2o uses a similar logic. It performs two unregister operations. On the second unregister, the old CID has already been replaced, causing a crash error.

Category 6: ACK Confusion. ACK frames indicate to the peer that packets have been received. To improve acknowledgment efficiency, an ACK can include multiple ranges to acknowledge packets. For example, acknowledging packets numbered 1, 2, 4, 5, and 7 may use an ACK frame with the ranges [1,3), [4,6), and [7,8). As shown in Figure 3(f). During fuzzing, our mutator generated ACK frames with overlapping ranges. All packets in these ranges existed and had been legitimately sent by the server. Most QUIC implementations ignore such duplicate ACK frames; stricter implementations consider overlapping ranges invalid and terminate the connection. We found that h2o, when processing these ACKs, neither ignores them nor terminates the connection. Instead, it repeatedly updates the acknowledgment status of the same packet, causing an already acknowledged packet to have its state modified again at a later time, which can lead to a Time of Check to Time of Use (TOCTOU) issue when the server triggers Connection Close.

C. Real-World Attack Impact

Impact 1: immediate service crash. Some vulnerability types directly cause the server process to crash. Even when watchdog tools monitor the server's status, an attacker can force the server to crash and restart frequently with a minimal payload, leading to a DoS attack. For example, in M07, an attacker sends four NCI frames to h2o to fill the CID list, then one additional NCI frame to replace an old CID, triggering an assertion failure and crash in h2o with under 1 KB of traffic. In M06, the TOCTOU issue does not trigger reliably and requires the attacker to repeatedly send specially crafted ACK frames. In our tests, an average of 200 ACK frames causes one h2o crash, enabling an attack with 4KB of traffic in approximately 0.5 s.

Impact 2: exhausting server memory. During CRYPTO, PC, and NCI flood attacks, the server continuously caches either the attacker's sent frames or data generated in the process of responding, eventually allocating more memory than the physical limit allows and being forcibly terminated by the operating system. Throughout this period, the attacker must continuously send data to the server. We also conducted experiments to validate the impact under practical conditions, assuming both the attacker and victim use a 1000 Mbps network bandwidth and the server is allocated 4 GB of memory. In the case of crypto flood, we tested aioquic and xquic. All crypto frame contents sent by the attacker were

received and cached by the server, resulting in memory usage growing at a steady rate of 1 GB/min and triggering forced process termination after 4 minutes. In PC flood, the server only stores 8 bytes of challenge data per frame and gradually releases old challenges, so the memory growth rate is slower. However, for aioquic, the memory usage still increased at 0.4 GB/min, triggering a DoS attack within 10 minutes. For NCI flood, we tested xquic under attack and achieved a memory growth rate of 0.6GB/min, reaching a DoS condition within 7 minutes.

Impact 3: silent DoS. Some vulnerabilities, such as Connection hold on flood, do not manifest through explicit failures and are categorized as Silent DoS. To assess the impact of such attacks, we designed the following experiment. We allocated one CPU core and 4 GB of memory to the Cloudflare quiche server. A 4 KB HTML page was provided for the victim to request via GET. During the test, the victim sent a GET request to the server every second and recorded the response time. Meanwhile, the attacker injected malicious clients into the server's hash map using the aforementioned attack strategy, gradually increasing the sending rate. When no attack was taking place, the victim's average response time was 10 ms. As the attacker's rate increased to 1 client per second, the victim occasionally failed to establish a connection, though refreshing the page would recover the session. At 3 clients per second, the victim rarely succeeded in establishing a connection, and the average response time increased to over 300 ms. At 10 clients per second, the victim was unable to receive any response from the server at all, resulting in a severe DoS effect.

Impact 4: exhausting CPU resource. For other aioquic vulnerabilities such as M04, M12, and M13, the server does not crash entirely but instead causes the subprocess of the current connection to crash. Aioquic promptly collects and reports the error, then restarts the subprocess. However, this recovery process consumes significant CPU and memory I/O resources.

To evaluate the impact of these attacks, we used the same server and victim configuration as in the Silent DoS experiment. The attacker sent malicious requests to aioquic at a bandwidth of 10 Mbps. Initially, the average response time for client requests was 12 ms. As aioquic repeatedly triggered exceptions and restarted subprocesses, CPU utilization quickly reached 100%, accompanied by a large volume of error output. The time it takes for the victim to access the page gradually increases, and after one minute of sustained attack, the average response time exceeded 1 second.

D. Comparison Experiment

1) Comparison with SOTA Fuzzing Framework: We surveyed three SOTA black box protocol fuzzing tools: BooFuzz, DPIFuzz, and Bleem, and selected two of the currently open source tools for evaluation. DPIFuzz is currently the latest differential fuzzing framework for QUIC, and BooFuzz is a widely tested black-box fuzzing framework. BooFuzz does not support TLS handshake or encryption mechanisms of the QUIC protocol. To enable QUIC fuzzing, we implemented a middleware using cloudflare-quiche. This middleware replaces BooFuzz in establishing connections with the target QUIC server. We then supply templates for various QUIC frames; BooFuzz generates plaintext frames based on these templates, and the middleware automatically parses and interacts with the server, returning results to BooFuzz to complete one test round. Bleem is a black box protocol fuzzing tool that operates as a man-in-the-middle (MITM), collecting frame sequences from both the client and server during their interaction. It automatically learns frame formats and performs mutations on the frame sequences. However, since Bleem is not yet open-sourced, we do not include it in the Comparison experiment in this evaluation.

Comparison of mutation and detection capabilities. Table II lists the main differences in vulnerability detection capabilities among BooFuzz, DPIFuzz, Bleem, and MerCuriuzz. In terms of mutation capabilities, all three tools perform random and unconstrained mutations at the content level. Among them, only Bleem supports mutations at the frame sequence level, and none of them support mutations of interaction behaviors. MerCuriuzz, through its segmental mutation strategy, is capable of performing mutations at all three levels. In terms of detection capabilities, BooFuzz, Bleem and MerCuriuzz use port occupancy and process status to determine crashes; DPIFuzz only analyzes traffic and does not automatically detect crashes, and developers must manually maintain server state and check for failures. For correctness, BooFuzz and Bleem do not verify correct output; DPIFuzz only checks the correctness of QUIC stream data, and MerCuriuzz checks both data frames and control frames. For consistency detection, BooFuzz and Bleem lack differential testing support; DPIFuzz only detects byte-level inconsistencies; MerCuriuzz detects semantic-level inconsistencies. Finally, only MerCuriuzz can detect and report abnormal resource consumption.

TABLE II: Comparison of Mutation and Detection Capabilities Among fuzzers

Fuzzer	Mutation-level			Detection			
	Content	Sequence	Interaction	Crashes	Correctness	Consistency	Resource Consumption
BooFuzz	◑	○	○	●	○	○	○
DPIFuzz	◑	○	○	◑	◑	◑	○
BLEEM	◑	●	○	●	○	○	○
MerCuriuzz	●	●	●	●	●	●	●

● Capable ◑ Partially Capable to mutate or detect ○ Not Capable.

Comparison of code coverage. For the evaluation of test-case generation and comparison of actual testing results, we introduced coverage and the number of detected vulnerabilities as objective metrics. Although none of the three fuzzers is coverage-guided, coverage information allows us to gain insight into the diversity of generated test cases and the likelihood of detecting vulnerabilities. In the fuzzing process, we selected four QUIC applications implemented in C and C++ and collected coverage via code instrumentation. We ran each application with each of the three fuzzers for 24 hours; the coverage data are shown in Figure 4.

Overall, the coverage of BooFuzz growth was relatively slow and reached saturation prematurely on all four implementations. This is because BooFuzz randomly populates each field of QUIC frames, resulting in an excessively large

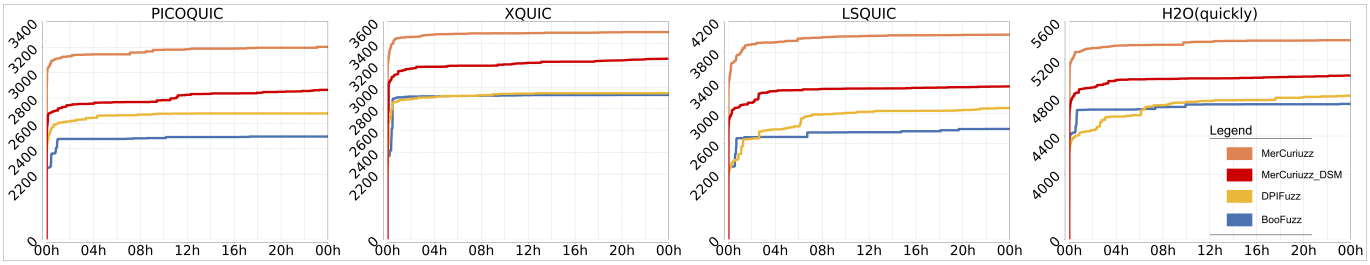

Fig. 4: Coverage comparison of MerCuriuzz, MerCuriuzz_DSM, DPIFuzz, and BooFuzz across four applications

mutation state space. It therefore struggles to find test cases that trigger more program paths. MerCuriuzz overcomes this challenge by performing a frame sequence mutator. DPIFuzz outperforms BooFuzz in coverage: after a rapid initial increase, it continues to grow slowly, but the overall improvement is limited. This behavior stems from its internal design: DPIFuzz provides detailed generation and mutation algorithms for STREAM frames according to the RFC, ensuring each STREAM frame can be received without causing state anomalies, which yields high early coverage of the server's STREAM-frame processing logic. For all other frame types, DPIFuzz uses simple random generation and thus faces the same large state-space issue. Furthermore, we used DPIFuzz and BooFuzz to perform 24-hour black-box fuzzing on 16 QUIC applications. Neither tool detected any known vulnerabilities or triggered anomalies or crashes.

2) Ablation Experiments: To evaluate our contributions, we conducted ablation experiments and designed three fuzzer variants, each disabling or replacing one of MerCuriuzz three key components:

Assessing the segmentation-based mutator. Disable segmental mutation (MerCuriuzz_DSM). Among its three modules, we turned off the frame sequence mutator and interaction behavior mutator. We did not replace the segmentation-based mutator with AFL's default mutator; instead, we retained the frame content mutator to avoid breaking frame structure. We tested the four QUIC applications used in the comparative experiments and recorded coverage changes over 24 hours, as shown in Figure 4. Although this variant outperformed BooFuzz and DPIFuzz in some cases, it still lagged noticeably behind full MerCuriuzz. Among the 14 defects, MerCuriuzz_DSM detected only 7 (Table I). Under the DSM mode, vulnerabilities such as crypto flood and NCI flood went undetected, because merely sending packets in order and immediately processing all responses does not trigger them. Only when the fuzzer actively drops some normally sent frames and delays responses can these bugs be triggered and caught by the Semantic Consistency Detector.

Assessing the anomaly detectors. We evaluated three variants. Disable Semantic Consistency Detector (MerCuriuzz_DSCD), Disable Public Resource Consumption Detector (MerCuriuzz_DPCD), and Replace all anomaly detectors with LibAFL's default (MerCuriuzz_RAD). We measured each variant's ability to detect vulnerabilities. We evaluated the vulnerability detection capabilities of each variant, as shown in Table I. Under the DSCD mode, a total of 9 vulnerabilities were detected. However, some vulnerabilities in PC Flood and aioquic could be easily missed if only the Public Resource Consumption Detector is used. Under the DPCD mode, 8 vulnerabilities were detected. It fails to effectively detect vulnerabilities like CRYPTO Flood and Connection Hold on Flood, which primarily rely on public resource consumption and are not easily identifiable through traffic analysis. The results show that the two detectors focus on different aspects of detection and are highly complementary, with identifying vulnerabilities that the other misses. Under the RAD mode, all vulnerabilities except for M06, M07 and M14, which cause service crashes, were difficult to detect.

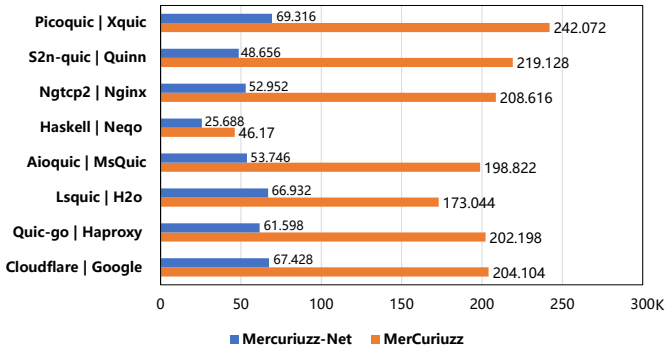

Fig. 5: Comparison of the number of executions during 24 hours of Fuzzing. Efficiency increased by up to 350% in the best case; average improvement (excluding Haskell — Neqo) reached 225%.

Assessing the snapshot manager. Replace snapshots with service-restart resets (MerCuriuzz_RS). We ran 24-hour tests on all target implementations and counted the total number of differential test executions; results are in Figure 5. Most tests performed well. In the S2n-quic vs. quinn comparison, MerCuriuzz achieved a 350% efficiency improvement over the non-snapshot version. Excluding Haskell and Neqo, the average improvement was 225%, demonstrating that NYX snapshots effectively eliminate SUT state reset and connection-establishment overhead. Because snapshot performance depends heavily on CPU and memory bandwidth, faster compute and storage media could further boost MerCuriuzz's

speed. We also observed poor performance on Haskell and Neqo due to the Haskell implementation's inefficiency, yet snapshots still yielded an 80% speedup.

E. Analysis of False Positive and Duplicate Rates

Improvements to reduce duplicate reports. We employ two complementary methods to identify and filter duplicate test reports. (1) Compare the root causes of reports. Whenever the State Anomaly Detector flags an exception, such as a mismatched CC frame in differential testing or discrepancies in data frames, MerCuriuzz emits an anomaly brief detailing the type of anomaly and its trigger. A single vulnerability may trigger multiple detectors (e.g., a CRYPTO FLOOD attack may produce CC, DATA, ACK mismatches, and abnormal memory usage). We merge these briefs into a unique fingerprint per testcase; matching fingerprints reveals overlaps with previous reports. (2) Eliminate public resource fluctuations. Even inside a VM, CPU, memory, and I/O may fluctuate due to other processes, the host, or MerCuriuzz itself. We replay each testcase's traffic and verify that identical briefs recur. Only reports with unique fingerprints and identical briefs under replay are accepted. Together, these methods remove most duplicates and greatly reduce the duplicate report rate.

Measurement and analysis. We evaluated the filter on 24 hour fuzzing runs of implementations with known logic flaws. We logged total reports, deduplication effectiveness, and causes of remaining duplicates or false positives. Aioquic as the implementation in which we discovered the greatest number of logical flaws, , our run executed 99,411 independent testcases. Applying fingerprint matching yielded 92 unique inconsistency reports. After replay verification, 16 reports remained that pinpointed aioquic defects, covering all discovered vulnerabilities. Of these, 50% were duplicates (8/16), and 18.75% were false positives (3/16). Manual review confirmed each classification.

Duplicate reports typically stemmed from benign, irrelevant traffic altering the fingerprint, which can be quickly discarded during manual review. False positives fell into three categories: (1) public resource fluctuations. Sustained high CPU or I/O delays may still trigger spurious anomaly briefs. (2) Stricter scrutiny and restrictions on traffic. Some test targets enforce constraints not defined in the RFC (or enforce them more strictly). For instance, lsquic upon receiving DATAGRAM frames it immediately emits a CC Frame, whereas other implementations accept them without error, this cause differential anomalies without real flaws;(3) Deviation in the prediction of vulnerabilities. For some logical vulnerabilities, vulnerabilities will only occur when public resources are continuously occupied or the program responds with a long delay. Due to efficiency requirements, we avoid fuzzing efficiency being too slow by limiting the maximum frame sequence size of a single testcase. As a result, an individual testcase cannot always trigger the full vulnerability, and we rely on manual confirmation to determine whether a genuine logic flaw exists. A typical example is quic-go's mitigation for the PC Flood attack: it requires 256 in-flight PATH_RESPONSE frames to trigger, but most fuzzed cases fall short of this threshold, leading to false positives flagged by the Semantic Consistency Detector.

VI. Discussion

A. Mitigation

Insights on QUIC and RFC Specifications. Through analyzing the vulnerabilities uncovered by MerCuriuzz, we observed that many QUIC implementations faithfully follow the mandatory and recommended requirements defined in RFC documents. However, in areas where the RFC does not provide explicit guidance, implementations still diverge. These differences can lead to exploitable vulnerabilities, especially when implementations make differing assumptions. For example, the RFC's specification for the crypto frame mandates a minimum buffering capacity but does not define an upper limit. Similarly, in the case of PC and NCI frames, the RFC requires endpoints to respond to every path_challenge and to send a retire_connection_id frame for every retired CID. However, it does not place any constraints on the maximum number of PC frames a connection can receive or the maximum number of CIDs that can be retired. We recommend supplementing the QUIC RFC with definitions for these upper bounds to prevent excessive resource consumption and ensure consistent behavior across implementations. For instance, A maximum of 512 KB of out-of-order crypto frame data should be buffered. This limit is unlikely to interfere with normal certificate transmission and key exchange. No more than 256 path_challenge frames should be accepted before a connection migration is completed. This provides ample allowance for endpoints to probe and establish a low-RTT path, after which the count can reset. A maximum of 256 CID retirements should be allowed prior to completing a migration. In typical scenarios, switching CIDs primarily serves to indicate a new path, and this limit is sufficient to cover most network environments.

Insights on QUIC implementations. We recommend adding audits for variable length arrays in high-level programming languages. During our vulnerability analysis, we observed an interesting phenomenon: memory consumption–related logical vulnerabilities are more likely to occur in high-level programming languages (such as Python and Rust). Unlike C, these languages often support variable length structures with built-in iteration and query methods, such as list and dict in Python, and vector in Rust. Developers do not need to manually manage the memory usage of these structures, and iterators eliminate the need to count the number of elements. As a result, attackers are provided with the opportunity to allocate arbitrary memory within these variable-length structures. Therefore, QUIC service providers and software developers can mitigate vulnerabilities caused by public resource consumption by enforcing stricter element number limits on these special structures.

Sustained fuzzing. This work tests 16 open-source tools among all current QUIC implementations. Due to limitations of the experimental environment and time, we did not test other closed-source QUIC implementations or additional

QUIC tools. Therefore, we suggest that developers integrate the findings of this work into their development workflows and continuously fuzz QUIC projects to discover more implementation flaws and logical vulnerabilities. Although logical vulnerabilities are important and have a wide impact, their early manifestations are easy to overlook, and they have not received sufficient attention so far. Hence, the entire community needs to work together and adopt a systematic security perspective to mitigate this problem.

B. Limitation

In terms of vulnerability discovery, we do not necessarily uncover all security issues in QUIC, this is an inherent limitation of fuzzing, which cannot exhaust all possibilities. In the future, combining symbolic execution with logical vulnerability discovery may enable more precise logical vulnerability testing frameworks. In terms of the tool's scope, we have not yet adapted MerCuriuzz to protocols beyond QUIC; however, such porting is feasible, as discussed in next subsection. In terms of vulnerability determination and confirmation, our current workflow still requires manual intervention for validation and root cause analysis. We plan to address this by integrating automated state-machine construction. Therefore, constructing a state machine to assist differential testing, as well as automating the mechanisms for report verification and replay, will be a key focus of our future research.

C. Porting to Other Protocols

Although MerCuriuzz is principally designed for the QUIC protocol, similar logic vulnerabilities may exist in other protocols. In general, porting MerCuriuzz to new protocols is feasible, and we are actively working toward this goal.

Specifically, the following are the parts that can be reused or less modified. The public resource detector mainly detects system resource usage, and the snapshot manager is responsible for managing the VM and transferring data to the fuzzer, and thus can be directly reused. The frame sequence generator and the interaction behavior mutator in the sectional mutation use predefined algorithms to expand the base frame sequence and therefore require only minor modifications. The following are the parts that need to be re-adapted. The frame content mutator and semantic consistency detector modules are mainly designed for the QUIC protocol, so when porting to other protocols, in order to improve the quality of the testcases, we need to extract the constraints on length, content, and numeric fields in the frames based on the definition of the frames as defined in the RFC document and redefine semantic inconsistencies based on the protocol characteristics. In addition, we use converters to abstract protocol interactions into interfaces, so developers only need to write the corresponding protocol interaction programs, which greatly improves the porting efficiency.

In terms of predictions, MerCuriuzz favors protocols with streams and contexts, such as HTTP/2 and HTTP/3, which we believe are highly portable and expect to see more adaptations in the future. For protocols that do not have a context, such as HTTP/1 or SMTP, the frame sequence generator and the interaction behavior mutator will no longer be available, and for protocols that do not have a flow structure, such as TLS, the modules of the segmentation mutation and state anomaly detector will not be fully functional, as they do not involve flow and congestion control.

VII. Related Work

In this section, we review prior work on logical vulnerabilities and fuzzing techniques for the QUIC protocol.

Logical vulnerabilities in QUIC. Early studies such as *QuicSand* [38] examined QUIC's defenses against resource-exhaustion attacks. Cao et al. [9] and Cui et al. [12] further analyzed large-scale DoS vectors like *0-RTT* and *1-RTT* attacks. Developers have also uncovered issues during implementation—for example, Seemann's blog on quic-go [52]. However, these efforts rely on manual analysis of QUIC features and code audits, lacking automated methods to detect logical flaws in existing implementations.

Automated testing frameworks. No prior work focuses on logic-vulnerability testing for QUIC. McMillan et al. [33] perform formal analysis and fuzzing on the QUIC handshake. DPIFuzz [49] disguises test traffic to evade DPI but does not target logic bugs. BLEEM [29] supports general-purpose protocol fuzzing, detecting only memory errors via crash observation. In fuzzing work targeting other protocols [26], [37], [53], [59]–[62], [64], [66], they leverage specific application behaviors or semantic inconsistencies to detect implementation discrepancies. In contrast, MerCuriuzz is the first automated fuzzer that uncovers QUIC logic flaws from both semantic inconsistencies and resource-consumption anomalies.

State machine testing for logical bugs. State machines have been widely used to detect illegal transition paths in protocols like TLS. De Ruiter et al. [13] build a TLS state machine via black-box testing; Stone et al. [32] extend it to grey-box; Fiterau-Brostean et al. [14] integrate test-case generation and evaluation. Rasool et al. [48] constructed a QUIC state machine and ported it to other protocols. State machines have proven effective for discovering and validating logic vulnerabilities. Jero et al. [21] found TCP congestion-control flaws by analyzing its state machine and crafting abstract attacks. Our testcases require repeatedly triggering the same state through state machine self loops, yet this still poses a challenge due to state space explosion. This motivates our future work to integrate automata into MerCuriuzz.

Other QUIC attacks. Numerous works target QUIC's broader attack surface. McMillan [63], Zhang [63], and Chatzoglou [10] expose TLS-handshake flaws. Damian [34] demonstrates UDP loop attacks that reflect stateless resets, causing feedback loops. Gbur et al. [15] present *QuicForge*, forging QUIC traffic as DNS. These studies underscore QUIC's evolving security challenges and the need for automated testing tools.

VIII. Conclusion

This paper presents *MerCuriuzz*, a novel automated tool designed to systematically test and uncover logical vulnerabilities

in QUIC protocol implementations. We propose the Segmental Mutation strategy for generating structured test inputs. Additionally, to effectively determine whether a logic bug has been triggered, we adopt a differential testing approach by designing and implementing two types of detectors: the Semantic Consistency Detector and the Public Resource Consumption Detector. We design the Snapshot Manager to efficiently restore the state of the SUT. We applied MerCuriuzz to a comprehensive set of 16 actively maintained QUIC implementations recommended by the IETF QUIC working group. In doing so, we discovered 14 previously unknown logical vulnerabilities. All findings were responsibly disclosed to the affected vendors, and we received acknowledgment and bounty rewards from teams including Cloudflare, Alibaba Cloud, quickly, aioquic, and picoquic. We hope this work raises awareness of resource-exhaustion logic bugs and encourages the broader community to proactively defend against such subtle but impactful threats.

IX. Ethics Considerations

Ethical considerations. We take the utmost care of potential ethical issues. (1) *In terms of experiment*, we deployed all QUIC services and fuzzing frameworks in a controlled internal network. All our experiments were conducted locally, which does not affect any other servers or users; (2) *In terms of disclosure process*, companies encourage security tests through bug bounty programs, and we carefully follow disclosure guidelines when disclosing issues to vendors.

Responsible disclosure. We have reported all vulnerabilities to the affected vendors. Up to now:

- **Alibaba Cloud** accepted the report of XQUIC, classified the vulnerability as high severity, and awarded 7,200 CNY as a bounty.
- **Cloudflare** accepted the report and fixed the bug. Considering the vulnerable HTTP/3 server to be only the demo implementation recommended by Cloudflare Quiche, rather than a mandatory production standard, they only awarded 200 USD as a bounty.
- **h2o and quicly** acknowledged our report and have agreed to assign CVE identifiers for the reported vulnerabilities.
- **aioquic and picoquic** are interested in the vulnerabilities. They invited us to assist in fixing the vulnerabilities. We raised three issues [5], [23], [24] with aioquic, one issue with picoquic [4], and submitted CVE requests to `cve.mitre.org`, receiving four CVE identifiers: CVE-2024-51410, 51411, 51413, and 51414.
- **lsquic** confirmed the vulnerability, but the developers attributed the root cause to BoringSSL's memory management issues [54] and refused to fix the issue or assign a CVE.
- **neqo** confirmed the vulnerability [39], and helped us to assign the CVE identifier: CVE-2025-6703.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd for their insightful feedback that helped improve the quality of the paper. We are grateful to Keran Mu and Qi Wang for their peer review and helpful suggestions. This work was supported by the National Natural Science Foundation of China (grant #62272265). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the funding agencies.

References

[1] aiortc, "GitHub - aiortc/aioquic: QUIC and HTTP/3 implementation in Python — github.com," https://github.com/aiortc/aioquic, 2025.

[2] Akamai, "Akamai and kuaishou implement quic to improve video experience," https://www.akamai.com/newsroom/press-release, 2022.

[3] alibaba, "GitHub - alibaba/xquic: XQUIC Library released by Alibaba is a cross-platform implementation of QUIC and HTTP/3 protocol. — github.com," https://github.com/alibaba/xquic, 2025.

[4] AsakuraMizu, "Security vulnerability due to unbounded storage of TLS stream · Issue 1745 · private-octopus/picoquic — github.com," https://github.com/private-octopus/picoquic/issues/1745, 2024.

[5] AsakuraMizu and k4ra5u, "[SECURITY] aioquic may store an unlimited number of remote path challenges · Issue 544 · aiortc/aioquic — github.com," https://github.com/aiortc/aioquic/issues/544, 2024.

[6] aws, "GitHub - aws/s2n-quic: An implementation of the IETF QUIC protocol — github.com," https://github.com/aws/s2n-quic, 2025.

[7] S. Bauer, P. Sattler, J. Zirngibl, C. Schwarzenberg, and G. Carle, "Evaluating the benefits: Quantifying the effects of tcp options, quic, and cdns on throughput," in *Proceedings of the 2023 Applied Networking Research Workshop*, ser. ANRW '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 27–33. [Online]. Available: https://doi.org/10.1145/3606464.3606474

[8] M. Bishop, "RFC 9114: HTTP/3 — datatracker.ietf.org," https://datatracker.ietf.org/doc/rfc9114, 2022.

[9] X. Cao, S. Zhao, and Y. Zhang, "0-rtt attack and defense of quic protocol," in *2019 IEEE Globecom Workshops (GC Wkshps)*, 2019, pp. 1–6.

[10] E. Chatzoglou, V. Kouliaridis, G. Karopoulos, and G. Kambourakis, "Revisiting quic attacks: A comprehensive review on quic security and a hands-on study," *International Journal of Information Security*, vol. 22, no. 2, pp. 347–365, 2023.

[11] Cloudflare, "GitHub - cloudflare/quiche: Savoury implementation of the QUIC transport protocol and HTTP/3 — github.com," https://github.com/cloudflare/quiche/, 2025.

[12] B. Cui, Z. Li, and F. Yu, "Manipulated client initial attack and defense of quic," in *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, 2022, pp. 611–618.

[13] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[14] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in *NDSS*, 2023.

[15] K. Y. Gbur and F. Tschorsch, "Quicforge: Client-side request forgery in quic." in *NDSS*, 2023.

[16] google, "quiche - Git at Google — quiche.googlesource.com," https://quiche.googlesource.com/quiche/, 2025.

[17] Q. W. Group, "Implementations — github.com," https://github.com/quicwg/base-drafts/wiki/Implementations, 2021.

[18] h2o, "cve.org," https://www.cve.org/CVERecord?id=CVE-2021-43848, 2021.

[19] ——, "cve.org," https://www.cve.org/CVERecord?id=CVE-2023-50247, 2023.

[20] ——, "GitHub - h2o/h2o: H2O - the optimized HTTP/1, HTTP/2, HTTP/3 server — github.com," https://github.com/h2o/h2o, 2025.

[21] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, "Automated attack discovery in tcp congestion control using a model-guided approach," in *Proceedings of the 2018 Applied Networking Research Workshop*, ser. ANRW '18. New York, NY, USA:

Association for Computing Machinery, 2018, p. 95. [Online]. Available: https://doi.org/10.1145/3232755.3232769

[22] Y. A. Joarder and C. Fung, "A survey on the security issues of quic," in *2022 6th Cyber Security in Networking Conference (CSNet)*, 2022, pp. 1–8.

[23] k4ra5u, "[SECURITY] Accepting and storing an unlimited number of CRYPTO frames within a single connection · Issue 501 · aiortc/aioquic — github.com," https://github.com/aiortc/aioquic/issues/501, 2024.

[24] ——, "[SECURITY] Excessive ranges in a single ACK frame causes an exception in python programs · Issue 549 · aiortc/aioquic — github.com," https://github.com/aiortc/aioquic/issues/549, 2024.

[25] kazu yamamoto, "GitHub - kazu-yamamoto/quic: IETF QUIC library in Haskell — github.com," https://github.com/kazu-yamamoto/quic, 2025.

[26] Y. Liang, J. Chen, R. Guo, K. Shen, H. Jiang, M. Hou, Y. Yu, and H. Duan, "Internet's Invisible Enemy: Detecting and Measuring Web Cache Poisoning in the Wild," in *31th ACM Conference on Computer and Communications Security*, 2024.

[27] litespeedtech, "GitHub - litespeedtech/lsquic: LiteSpeed QUIC and HTTP/3 Library — github.com," https://github.com/litespeedtech/lsquic, 2025.

[28] J. D. Lucas Pardue, "HTTP/2 Rapid Reset: deconstructing the record-breaking attack — blog.cloudflare.com," https://blog.cloudflare.com/technical-breakdown-http2-rapid-reset-ddos-attack/, 2023.

[29] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4481–4498. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/luo-zhengxiong

[30] M. Maehren, P. Nieting, S. Hebrok, R. Merget, J. Somorovsky, and J. Schwenk, "TLS-Anvil: Adapting combinatorial testing for TLS libraries," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 215–232. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/maehren

[31] A. Mankin, "RFC 9250: DNS over Dedicated QUIC Connections — datatracker.ietf.org," https://datatracker.ietf.org/doc/rfc9250, 2022.

[32] C. McMahon Stone, S. L. Thomas, M. Vanhoef, J. Henderson, N. Bailluet, and T. Chothia, "The closer you look, the more you learn: A grey-box approach to protocol state machine learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2265–2278. [Online]. Available: https://doi.org/10.1145/3548606.3559365

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

[34] D. Menscher, "Blog: Preventing Cross-Service UDP Loops in QUIC — bughunters.google.com," https://bughunters.google.com/blog/5960150648750080/, 2024.

[35] Microsoft, "GitHub - microsoft/msquic: Cross-platform, C implementation of the IETF QUIC protocol, exposed to C, C++ and Rust. — github.com," https://github.com/microsoft/msquic, 2025.

[36] Mozilla, "GitHub - mozilla/neqo: Neqo, the Mozilla Firefox implementation of QUIC in Rust — github.com," https://github.com/mozilla/neqo, 2025.

[37] K. Mu, J. Chen, J. Zhuge, Q. Li, H. Duan, and N. Feamster, "The Silent Danger in HTTP: Identifying HTTP Desync Vulnerabilities with Gray-box Testing," in *34th USENIX Conference on Security Symposium*, 2025.

[38] M. Nawrocki, R. Hiesgen, T. C. Schmidt, and M. Wählisch, "Quicsand: quantifying quic reconnaissance scans and dos flooding events," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21. ACM, Nov. 2021, p. 283–291. [Online]. Available: http://dx.doi.org/10.1145/3487552.3487840

[39] neqo, "transport/fc.rs: panic attempting to send MAX_DATA with value larger max varint — github.com," https://github.com/mozilla/neqo/security/advisories/GHSA-jfv6-x22w-grhf, 2025.

[40] nginx, "GitHub - nginx/nginx: The official NGINX Open Source repository. — github.com," https://github.com/nginx/nginx, 2025.

[41] ngtcp2, "GitHub - ngtcp2/ngtcp2: ngtcp2 project is an effort to implement IETF QUIC protocol — github.com," https://github.com/ngtcp2/ngtcp2, 2025.

[42] onethingcloud, "onethingcloud.com," https://www.onethingcloud.com/, 2025.

[43] private octopus, "GitHub - private-octopus/picoquic: Minimal implementation of the QUIC protocol — github.com," https://github.com/private-octopus/picoquic, 2025.

[44] qemu, "QEMU — qemu.org," https://www.qemu.org/, 2025.

[45] quic go, "cve.org," https://www.cve.org/CVERecord?id=CVE-2024-22189, 2024.

[46] ——, "GitHub - quic-go/quic-go: A QUIC implementation in pure Go — github.com," https://github.com/quic-go/quic-go, 2025.

[47] quinn rs, "GitHub - quinn-rs/quinn: Async-friendly QUIC implementation in Rust — github.com," https://github.com/quinn-rs/quinn, 2025.

[48] A. Rasool, G. Alpár, and J. de Ruiter, "State machine inference of quic," 2019. [Online]. Available: https://arxiv.org/abs/1903.04384

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

[50] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[51] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22, 2022. [Online]. Available: https://doi.org/10.1145/3492321.3519591

[52] M. Seemann, "Hello, I am Marten Seemann. — seemann.io," https://seemann.io/, 2025.

[53] K. Shen, J. Lu, Y. Yang, J. Chen, M. Zhang, H. Duan, J. Zhang, and X. Zheng, "HDiff: A Semi-automatic Framework for Discovering Semantic Gap Attack in HTTP Implementations," in *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2022.

[54] Spwpun, "a memory leak issue · Issue 460 · litespeedtech/lsquic — github.com," https://github.com/litespeedtech/lsquic/issues/460, 2023.

[55] tempload, "tiptime.cn," https://www.tiptime.cn/, 2025.

[56] M. Thomson, "RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport — datatracker.ietf.org," https://datatracker.ietf.org/doc/rfc9000, 2021.

[57] S. Turner, "RFC 9001: Using TLS to Secure QUIC — datatracker.ietf.org," https://datatracker.ietf.org/doc/rfc9001, 2021.

[58] W3Techs, "Usage Statistics of HTTP/3 for Websites, April 2025 — w3techs.com," https://w3techs.com/technologies/details/ce-http3, 2025.

[59] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications," in *2024 IEEE Symposium on Security and Privacy*, 2024.

[60] Q. Wang, J. Chen, Z. Jiang, R. Guo, X. Liu, C. Zhang, and H. Duan, "Break the Wall from Bottom: Automated Discovery of Protocol-Level Evasion Vulnerabilities in Web Application Firewalls," in *2024 IEEE Symposium on Security and Privacy*, 2024.

[61] Y. You, J. Chen, Q. Wang, and H. Duan, "My ZIP isn't your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers," in *34th USENIX Conference on Security Symposium*, 2025.

[62] J. Zhang, J. Chen, Q. Wang, H. Zhang, C. Wang, J. Zhuge, and H. Duan, "Inbox Invasion: Exploiting MIME Ambiguities to Evade Email Attachment Detectors," in *31th ACM Conference on Computer and Communications Security*, 2024.

[63] J. Zhang, L. Yang, X. Gao, G. Tang, J. Zhang, and Q. Wang, "Formal analysis of quic handshake protocol using symbolic model checking," *IEEE Access*, vol. 9, pp. 14 836–14 848, 2021.

[64] L. Zheng, X. Li, C. Wang, R. Guo, H. Duan, J. Chen, and K. Shen, "ReqsMiner: Automated Discovery of CDN Forwarding Request Inconsistencies with Differential Fuzzing," in *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[65] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, "It's over 9000: analyzing early quic deployments with the standardization on the horizon," in *Proceedings of the 21st ACM*

Internet Measurement Conference, ser. IMC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 261–275. [Online]. Available: https://doi.org/10.1145/3487552.3487826

[66] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou

APPENDIX A
COMPRESSED CORPUS

To support the aforementioned mutation strategies, we designed a Compressed Corpus (CC) structure to represent the traffic of a single interaction from the client's perspective. We define a frame loop set $S = \{\langle f_1, f_2, \ldots, f_m \rangle, len\}$, which represents an ordered combination of initial frames f_1 to f_m, repeated len times. In addition to multiple frame loop sets $T = \{S_1, \ldots, S_n\}$, a CC also contains the following attributes:
Send-to-receive ratio ($sr : rr$): Defines the ratio at which sr frames are sent before receiving rr frames from the server. By adjusting the $sr : rr$ ratio, it is possible to simulate network congestion scenarios with high precision.

Packet loss and reordering Algorithm (PA): Selects an algorithm to modify the current sequence, simulating characteristics such as unstable packet loss and out-of-order delivery.

Random seed (RS): To ensure consistency and reproducibility in differential testing between both sides, the current random seed is also included in the corpus.

During the frame content mutation phase, one or more initial frames from S_i are selected for field-level mutation, modifying elements of the set and other attributes in CC. In the sequence mutation phase, we use the random seed to select a frame sequence mutation algorithm for each frame in S, arrange them in the current order, and expand each set in T to form a complete frame sequence. In the interaction behavior mutation phase, we use the reordering and packet loss algorithms to construct the final frame sequence.

By applying a segmented mutation algorithm to expand and mutate a CC, we balance the constraints necessary to avoid state explosion while maintaining mutation diversity, thereby improving mutation efficiency.

APPENDIX B
CORPUS UPDATE AND RECOMMENDATION

Monte Carlo Seed Tree. To guide fuzzing more effectively in a black-box setting and to evaluate seeds in a systematic way, we not only rely on differential detection and public resource consumption detectors to determine whether a vulnerability is triggered, but also use these detectors to quantify our level of interest in each seed. Besides marking certain seeds as "interesting," we must also exclude those deemed "uninteresting." Hence, we design and maintain a Monte Carlo seed tree to update and recommend seeds in the corpus.

Specifically, for constructing and maintaining this seed tree, we begin from a virtual root node and connect each initial seed downward. Whenever the mutator selects a seed represented by a node in the tree and mutates it, the leaf node corresponding to that seed is expanded. Once the test concludes, if the evaluation result indicates the seed is "interesting," we compute its Upper Confidence Bound (UCB) and propagate that value back to the root node. If the result is "uninteresting," we delete that node. This iterative process incrementally builds and refines the seed tree.

Our approach differs from the standard Monte Carlo algorithm in that, theoretically, each parent node can have infinitely many child nodes (since the mutation state space is unbounded). Consequently, our selection algorithm must decide whether to mutate a node at level i of the seed tree, rather than exclusively picking leaf nodes. Moreover, to prevent the seed tree from growing excessively large and dispersing fuzzing energy too thinly (which would effectively degrade to a traversal of all seeds), we introduce a time parameter into the UCB formula. This gives newly added subtrees more energy for testing.

APPENDIX C
QUIC IMPLEMENTATIONS

Below are the 16 recommended implementations selected from the QUIC workgroup, along with statistics on the programming languages used and the code size of these implementations.

TABLE III: Tested QUIC Projects and Their Versions

Number	Project Name	Language	code lines
1	Cloudflare QUIC	rust	57K
2	Google QUIC	C++	317K
3	Quic-go	Golang	75K
4	Aioquic	python	20K
5	Haproxy	C	295K
6	H2o(quickly)	C	516K
7	Lsquic	C	111K
8	MsQuic	C	188K
9	Haskell quic	haskell	20K
10	Neqo	Rust	66K
11	Ngtcp2	C	75K
12	Nginx	C	193K
13	Picoquic	C	104K
14	Quinn	Rust	27K
15	S2n-quic	Rust	194K
16	Xquic	C	77K

APPENDIX D
ARTIFACT APPENDIX

A. Description & Requirements

This tool is developed as a fork of the LibAFL v0.13.0 framework and integrates several other open-source components to implement its features. For QUIC interaction, it uses Cloudflare's quiche component from cloudflare. Snapshot support is enabled through libnyx, packer, and QEMU-Nyx modules of Nyx-fuzz project. The tool targets mainstream QUIC implementations and is primarily written in Rust, with portions in C and Python.

1) How to access: The MerCuriuzz main program is available on GitHub and can be accessed via Mercuriuzz. Its auxiliary components have been functionally adapted based on the original projects for use with MerCuriuzz, including: LibAFL, Cloudflare-quiche. libnyx, QEMU-Nyx, packer. We have also uploaded our artifacts to a DOI-providing platform: https://doi.org/10.5281/zenodo.17015304.

2) Hardware dependencies: Running the minimal implementation requires at least 4GB of memory and 8 logical CPU cores (CPU IDs 0–7). Running the snapshot-based implementation also requires a processor that supports Intel Processor Trace (Intel PT) (supported by most Intel CPUs; AMD CPUs have been tested) and KVM virtualization support (if the host is itself a virtual machine, nested virtualization must be enabled).

3) Software dependencies: All Linux-kernel-based operating systems are supported; Ubuntu versions 22.04 or 24.04 are recommended.

4) Benchmarks: None.

B. Artifact Installation & Configuration

From the initial setup to running a complete minimal instance involves several steps:

First install the required runtime libraries and components, including build tools and the libraries needed to execute the code. These are described in the "Environment" section of the README. As test targets, we chose two implementations: lsquic, and neqo. The build and testing processes are detailed in the "Target Installation" section of the README. Finally, building and running the MerCuriuzz project itself is covered in the "Building" section of the README, with all the detailed steps provided there.

Figure 1 illustrates the workflow of MerCuriuzz. Overall, MerCuriuzz consists of four main phases: (1) receiving input testcases and outputting objectives and crashes; (2) managing and mutating the corpus; (3) managing snapshots and performing differential evaluation; (4) executing input tests under QEMU virtual machine snapshots.

Regarding the code paths for each module, we take the MerCuriuzz directory as the root. The implementation of the MCTS scheduler is located in `libafl-modules/src/schedulers`, while Segmental Mutation is implemented in `libafl-modules/src/mutators`. The executor functionality resides in `libafl-modules/src/executors`, providing both a minimized implementation based on local area networks and a full implementation based on snapshots. All evaluation features are distributed under `libafl-modules/src/` across the observers and feedbacks directories. The former collects data, while the latter evaluates this data. The snapshot component is decoupled from the MerCuriuzz project, requiring manual configuration of virtual machines for each QUIC implementation, along with adding necessary runtime libraries to the filesystem. Virtual machine initialization and snapshot functionalities are controlled using vendors/packer/packer/linux_x86_64-userspace/src/harness.c.

The main program is located in the fuzzers directory. For the minimized implementation, use `network_quic_fuzz`; for snapshot-based experiments, use nyx_quic_fuzz. The main program connects all components and interfaces with either QUIC implementations or virtual machine processes to enable continuous fuzzing execution.