University of Zurich^UZH

Design and Implementation of a Stateful Firewall to Mitigate Security Challenges in the QUIC Protocol

Dominik Sarman
Zürich, Switzerland
Student ID: 18-712-539

Supervisor: Thomas Grübl, Jan von der Assen
Date of Submission: February 15, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 14.02.2025

[illegible]

Signature of student

Kurzfassung

Das rasante Wachstum von Streaming-Diensten und ihre Dominanz im weltweiten Internetverkehr haben die Nachfrage nach effizienteren Echtzeit-Kommunikationsprotokollen erhöht. Als Reaktion darauf wurde QUIC 2012 von Google eingeführt und später 2021 von der IETF als verbessertes Protokoll der Transportschicht standardisiert. QUIC ermöglicht es HTTP/3, über UDP statt über das herkömmliche TCP zu laufen, und bietet so Verbesserungen in Bezug auf Geschwindigkeit und Sicherheit. Funktionen wie Connection Migrations gewährleisten eine kontinuierliche Kommunikation auch bei Änderungen der Netzwerkadresse. Dies erhöht die Benutzerfreundlichkeit, bringt jedoch gleichzeitig neue Sicherheitsherausforderungen für Firewalls mit sich.

In dieser Arbeit werden QUIC und seine Sicherheitsprobleme eingehend analysiert und ein Prototyp einer Stateful Firewall entwickelt, die in der Lage ist, QUIC-Traffic, einschliesslich während einer Connection Migration, verarbeiten zu können. Die Firewall kann bösartige von gutartigen Connection Migrations unterscheiden und kann potenziell schädliche Aktivitäten effektiv blockieren. Das Design und die Implementierung wurden in umfangreichen Tests evaluiert, womit nachgewiesen werden konnte, dass die Firewall in der Lage ist, QUIC-Traffic zu schützen, ohne dabei die Funktionalität von QUIC einzuschränken.

Abstract

The rapid growth of streaming services and their dominance in global Internet traffic have driven a demand for more efficient real-time communication protocols. In response, QUIC was introduced by Google in 2012 and later specified by the IETF in 2021 as an improved transport-layer protocol. QUIC allows HTTP/3 to run over UDP instead of relying on the traditional TCP, offering improvements in speed and security. Features such as connection migrations ensure continuous communication even through network address changes, increasing user experience but also introducing new security challenges for middleboxes.

This thesis presents an in-depth analysis of QUIC and its security challenges, alongside the development of a prototype stateful firewall designed of handling QUIC traffic during connection migrations. The firewall is able to identify malicious from benign connection migrations and can effectively block potentially harmful activity. Its design and implementation were evaluated through extensive testing, demonstrating its effectiveness in securing QUIC traffic while maintaining feature compatibility.

Acknowledgments

This master thesis would not have been possible without the guidance and constant support from my supervisor, Thomas Grübl. His expertise in the field and valuable advice helped me stay on track throughout the project. I would also like to thank Prof. Burkhard Stiller and the entire Communication Systems Group at the University of Zurich for providing me with the opportunity and resources to work on this project. A special thanks goes to Brian Trammell for his helpful feedback and the valuable insights he shared during our discussion, which greatly contributed to this thesis. I would also like to thank my friends and family for constantly supporting me during this time.

Contents

Chapter 1

Introduction

With the increasing popularity of streaming services, a substantial portion of global Internet traffic now consists of real-time media delivery. This shift has created a growing demand for communication protocols that can handle high performance and low-latency requirements. Traditional transport protocols like TCP, while reliable, struggle to meet these demands. In response to these challenges, QUIC, a transport-layer protocol, was first introduced by Google in 2012. QUIC allows HTTP/3 to run over UDP instead of relying on the traditional TCP, offering significant improvements in speed, security, and adaptability.

QUIC integrates TLS encryption directly into the transport layer, combining security and performance to reduce connection setup times. It introduces connection identifiers (Connection IDs) that allow endpoints to demultiplex and manage connections efficiently. These identifiers play a crucial role in supporting one of QUIC's most important features: connection migrations. This feature enables devices to seamlessly switch between networks— such as transitioning from WiFi to cellular — without requiring a new handshake or disrupting an ongoing session. However, this capability introduces unique challenges for network security infrastructure.

Most of QUIC's headers are encrypted, limiting the visibility of middleboxes, such as firewalls, that traditionally rely on inspecting packet headers to manage and secure traffic. Additionally, connection migrations allow IP addresses, ports, and even connection IDs to change dynamically during a session, further complicating the task of tracking and controlling connections. As a result, traditional firewalls face significant obstacles in processing and managing QUIC traffic effectively.

Given that QUIC is a relatively new protocol, most major firewall vendors do not yet support all features of QUIC at the time of writing this thesis. Their common recommendation is to block QUIC traffic entirely, forcing applications to fall back to HTTP/2 over TCP with TLS. While this approach restores compatibility with existing firewalls, it negates the performance and security advantages of QUIC, limiting its potential benefits.

This project aims to address this gap by conducting a comprehensive analysis of QUIC and its unique challenges for network security. Building upon this understanding, the work focuses on the design and implementation of a stateful firewall capable of handling QUIC traffic. The proposed solution is intended to ensure robust security while maintaining the protocol's advantages, such as connection migration and encryption, ultimately bridging the gap between QUIC and modern network security requirements.

1.1 Motivation

The primary motivation behind this work lies in addressing the security challenges introduced by the QUIC protocol. While QUIC offers significant advantages — such as reduced latency in connection establishment, improved privacy via encryption, and performance enhancements through features like connection migration — it also presents new obstacles for traditional network security mechanisms, such as firewalls. QUIC fully encrypts its payload and even part of its headers, limiting the visibility of middleboxes that rely on inspecting traffic for monitoring, routing, or security purposes. Consequently, conventional firewalls face significant difficulties in effectively processing and managing encrypted QUIC traffic, leaving networks vulnerable to new types of attacks.

A key feature of QUIC that introduces both benefits and security challenges are connection migrations. QUIC allows ongoing connections to seamlessly transition between networks, for instance, from a WiFi network to a cellular network, without requiring a new handshake and without interrupting the communication. While this feature enhances user experience and connectivity, particularly in mobile and dynamic environments, it opens up new attack possibilities. Traditional firewalls, which often rely on static connection tracking, are ill-equipped to handle these dynamic migrations, which can potentially be exploited by attackers.

In RFC 9000 [2], potential security considerations are mentioned. As outlined in Section 21.5 of the RFC, request forgery attacks occur when an attacker influences the victim's peer to issue requests towards a target, with the attacker's payload being executed by the victim's peer. Further specified in 21.5.3, the Destination Connection ID field of packets that the client sends to a preferred address can be used for request forgery. While the RFC specifies that clients must validate the preferred address before sending non-probing frames (Section 8), this measure alone may not be sufficient to prevent exploitation, especially in real-world deployments.

The goal of this work is to address these security challenges by designing a firewall specifically tailored to handle QUIC traffic. This firewall would not only be able to monitor QUIC traffic and parse QUIC headers but also ensure that connection migrations are correctly tracked and validated. The focus will be on preventing malicious or unauthorized connection migrations, without undermining the performance and benefits that QUIC provides.

In conclusion, this thesis seeks to contribute to the secure adoption of QUIC in real-world network environments by creating a firewall solution that can address both the privacy and performance needs of QUIC while ensuring robust security against new threats introduced by connection migration and potential request forgery attacks.

1.2 Description of Work

This work begins with a comprehensive analysis of QUIC, including its version-independent properties, security features, and the challenges it presents to traditional middleboxes like firewalls. The analysis also considers potential attack vectors, particularly those related to connection migrations. QUIC's migration feature, while valuable, can be misused by malicious actors to evade detection and bypass firewalls, which creates a need for an effective detection and mitigation strategy.

The main contribution of this work is the development of a QUIC-aware firewall. This firewall is designed to handle QUIC traffic by inspecting connection headers and detecting when a connection is being migrated, whether by a client or a server. The firewall's approach to managing QUIC traffic involves identifying and associating new connections with their previous ones, ensuring that legitimate connection migrations are allowed while malicious ones are blocked. A key aspect of this work is addressing the challenges of connection migration detection, particularly server-side migrations, which represent a significant attack vector. The firewall uses several techniques to validate connection migrations, including checking the public IP ranges of large cloud providers and performing WHOIS lookups to ensure that the same organization controls both the original and new IP addresses involved in a migration.

In addition to security, the work also focuses on the manageability challenges of QUIC-aware middleboxes. As QUIC is deployed in real-world networks, the ability to effectively monitor and enforce security policies is critical. The solution proposed here integrates robust traffic analysis with the flexibility to adapt to different network environments, ensuring that QUIC's advantages are maintained while mitigating potential risks.

This research aims to provide a deeper understanding of QUIC's security implications and propose practical solutions for middleboxes and firewalls. By developing a QUIC-aware firewall that can handle connection migrations and detect malicious activities, this work contributes to the safe, secure, and efficient deployment of QUIC in modern networks. Through this contribution, the research supports the broader goal of improving network security while enabling the full potential of next-generation protocols like QUIC.

1.3 Thesis Outline

This thesis is structured into the following chapters: Chapter 2 provides the necessary background information, including an overview of QUIC, the workings of stateful firewalls, and other foundational concepts. In Chapter 3, related work is reviewed, where current research and existing solutions for firewalls and QUIC traffic are discussed. Chapter 4 details the design of the firewall, explaining its architecture and the decisions that were made to ensure it can effectively handle QUIC traffic. Chapter 5 covers the implementation of the firewall, highlighting the technologies used and the challenges faced during development. The evaluation of the firewall is presented in Chapter 6, where the test setup, results, and a discussion of the firewall's performance and accuracy are included. Finally, Chapter 7 concludes the thesis, summarizing the work, addressing its limitations, and suggesting potential areas for future research.

Chapter 2

Background

This chapter provides background information to understand the context and technical aspects of this work. It begins with an introduction into QUIC, covering its key features, header formats, and the mechanics of connection migrations. This part is mostly based on the official specifications published by the Internet Engineering Task Force (IETF). The second part of the chapter explains the concepts of stateful firewalls, network address translation (NAT), and how firewalls traditionally handle UDP traffic, highlighting the challenges that arise when applying these concepts to QUIC.

2.1 Quick UDP Internet Connections

QUIC is a modern transport protocol introduced by Google in 2012 and standardized by the Internet Engineering Task Force (IETF) in 2021 [1], [2], [3], [4]. Operating as an encrypted, connection-oriented protocol built on top of UDP, QUIC serves as the foundation for HTTP/3 [5]. Today, around 25% of Internet traffic relies on QUIC [11], highlighting its importance in modern network communication. The protocol was designed to address the limitations of traditional transport protocols, such as TCP, by enhancing speed, reliability, and security.

One of QUIC's key features is its ability to streamline connection establishment. The protocol reduces the number of round-trip times (RTTs) required to establish a connection, as shown in Figure 2.2, significantly lowering latency and improving user experience. 0-RTT even allows applications to send data by a client before having received a response from the server.

In addition to reducing latency, QUIC enhances user privacy. All user data and parts of the QUIC header are encrypted, preventing third parties from analyzing traffic patterns or associating session identifiers with specific users. The protocol frequently rotates connection IDs, further complicating any attempt to track or profile users based on their online activity.

Another improvement of QUIC is its ability to handle multiple data streams within a single connection. Traditional protocols like TCP suffer from head-of-line blocking, where

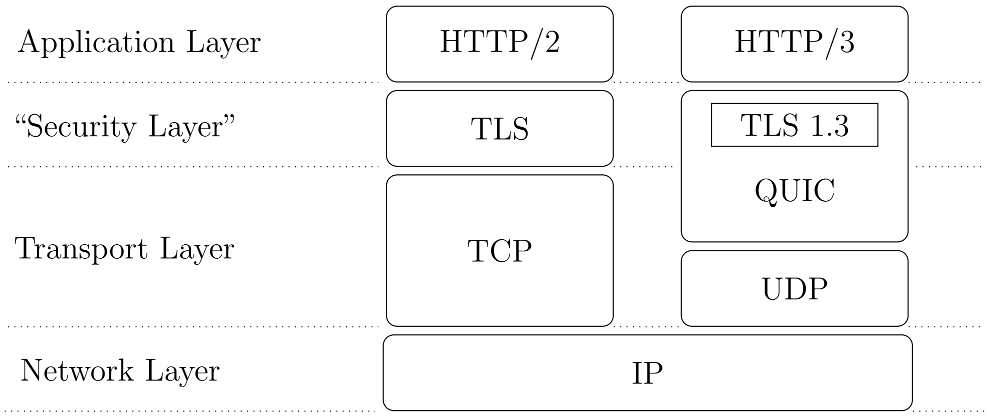


Figure 2.1: HTTP/2 vs HTTP/3

the loss of a single packet can delay the entire stream. QUIC eliminates this problem by enabling streams to be independently retransmitted. This capability improves performance, particularly in networks with high packet loss or inconsistent quality, and ensures a smoother user experience even in challenging conditions.

QUIC is built on top of UDP rather than TCP as depicted in Figure 2.1. UDP's connectionless and lightweight design makes it ideal for time-sensitive applications like gaming and streaming, which nowadays accounts for the largest portion of Internet traffic. However, UDP lacks the reliability features of TCP, such as packet ordering and congestion control. QUIC therefore implements these at the protocol level.

Despite its advantages, the adoption of QUIC poses challenges for enterprise environments. Many major firewall vendors do not yet fully support QUIC and recommend blocking QUIC traffic entirely or disabling it on managed devices through policy enforcement [55], [56], [57], [58], [59]. In such cases, HTTP traffic typically falls back to HTTP/2 over TCP and TLS.

Cloudflare's Quiche [60] is an example of an open-source implementation of QUIC. It was used extensively during this project to analyze the real-world behavior of QUIC traffic.

2.1.1 Connection IDs

In QUIC, each connection is identified by a set of connection identifiers, commonly referred to as connection IDs. These identifiers are selected independently by each endpoint, allowing both parties in a connection to define the connection IDs their peer should use [2]. The primary purpose of connection IDs is to maintain connection continuity, even when lower protocol layer, such as UDP or IP, experience changes in addressing. To ensure user privacy, connection IDs must not contain any information that could be exploited by external observers [2].

The length of a connection ID is variable and is not strictly defined in the RFC 8999 [1], which specifies version-independent properties of QUIC. In version 1 and version 2 of the protocol, connection IDs can range from zero length to a maximum of 160 bits, or 20

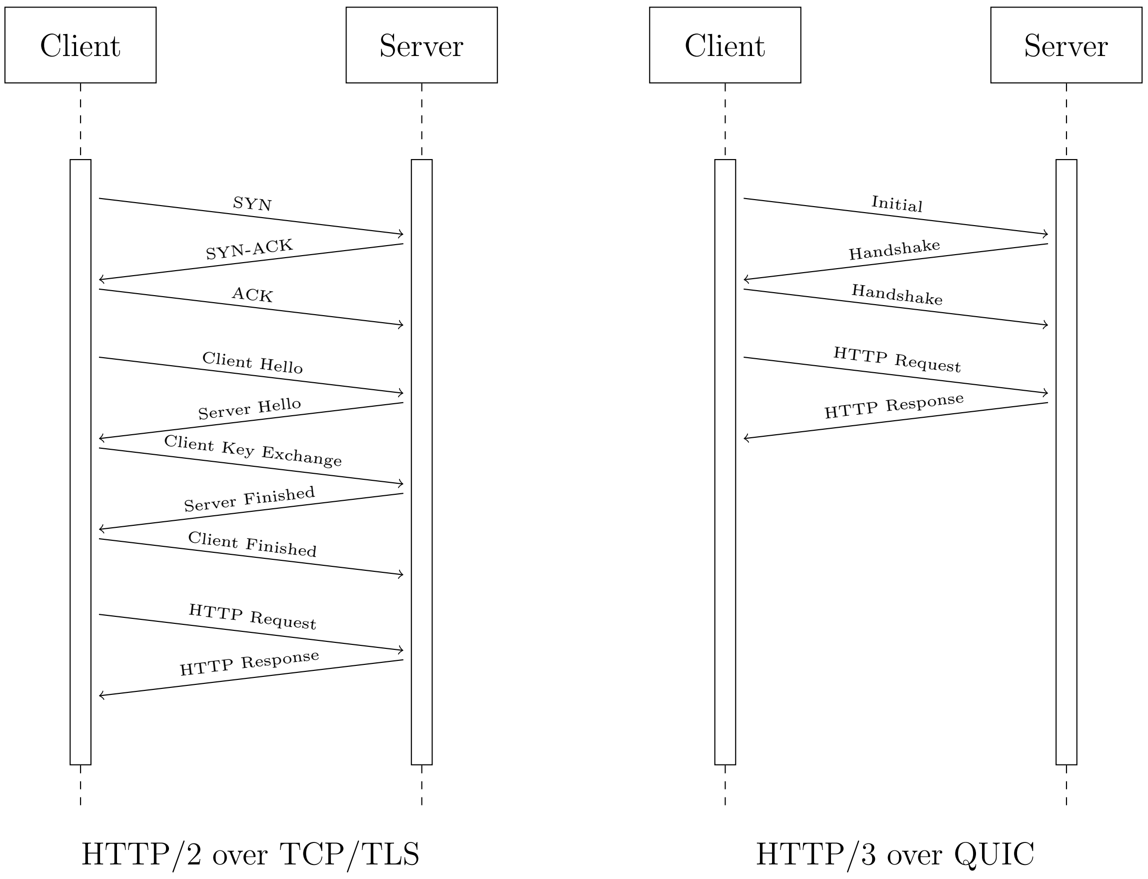


Figure 2.2: HTTP/2 vs HTTP/3 Handshake

bytes [2], [8]. However, RFC 8999 [1] states that a QUIC version can specify a connection ID length of up to 2040 bits or 255 bytes. All versions of QUIC should therefore be capable of handling longer connection IDs than 20 bytes to maintain compatibility with future QUIC versions. In practice, many implementations, including Cloudflare's Quiche, Google's Quiche, and Microsoft's MSQUIC, use connection IDs that are 20 bytes [60], [61], [62].

As described in section 19.15 of the RFC 9000 [2], a `NEW_CONNECTION_ID` frame is used by an endpoint to provide its peer with a new connection ID. This mechanism allows connection IDs to be updated or rotated during the lifetime of a connection. The negotiation of these updates occurs in an encrypted manner, ensuring that such changes remain invisible to middleboxes or external observers.

In a QUIC packet, the Destination Connection ID is selected by the packet's recipient and serves to facilitate consistent routing. Meanwhile, the Source Connection ID is used by the sender to specify the connection ID that its peer should use as the Destination Connection ID in return packets [2]. When a client sends an Initial packet to a server without having previously received any packets in return, the client populates the Destination Connection ID field with an unpredictable value.

2.1.2 QUIC Headers

QUIC employs two types of headers: long headers and short headers [1]. The choice of header depends on the stage of the connection and the type of packet being transmitted. Long headers are used during the initial stages of a connection, such as when establishing the handshake, while short headers are used for most communication after the handshake is complete.

Long Headers

Long headers play a critical role during the connection establishment phase and are used in packets that are exchanged before the 1-RTT encryption keys are available [2]. Once these keys are negotiated, communication transitions to short headers. The long header format is identifiable by its `Header Form` field, where the most significant bit `(0x80)` of the first byte is set to 1.

The long header of QUIC version 1 and version 2 includes the following fields [1], [2], [8]:

1. **Header Form:** The most significant bit (0x80) of byte 0 (the first byte) is set to 1 for long headers.
2. **Fixed Bit:** The next bit (0x40) of byte 0 is set to 1, unless the packet is a Version Negotiation packet. Packets containing a zero value for this bit are not valid packets and mustd be discarded.
3. **Long Packet Type:** The next two bits (those with a mask of 0x30) of byte 0 contain a packet type. Packet types are listed in Tables 2.1 and 2.2.
4. **Type-Specific Bits:** The semantics of the lower four bits (those with a mask of 0x0f) of byte 0 are determined by the packet type.
5. **Version:** A 32-bit field that follows the first byte. This field indicates the version of QUIC that is in use and determines how the rest of the protocol fields are interpreted.
6. **Destination Connection ID Length:** The byte following the version contains the length in bytes of the Destination Connection ID field that follows it. This length is encoded as an 8-bit unsigned integer. In QUIC version 1 and version 2, this value does not exceed 20 bytes [2], [8].
7. **Destination Connection ID:** The Destination Connection ID as described in the section above.
8. **Source Connection ID Length:** Contains the length in bytes of the Source Connection ID field that follows it.
9. **Source Connection ID:** The Source Connection ID as described in the section above.
10. **Type-Specific Payload:** The encrypted remainder of the packet.

Table 2.1: Long Header Packet Types in QUIC Version 1 [2].

Type Bits	Name
0x00	Initial
0x01	0-RTT
0x10	Handshake
0x11	Retry

Table 2.2: Long Header Packet Types in QUIC Version 2 [8].

Type Bits	Name
0x00	Retry
0x01	Initial
0x10	0-RTT
0x11	Handshake

Each packet type serves a specific purpose during connection setup [2]:

1. **Initial Packet:** Used by clients and servers to exchange the first cryptographic handshake messages and acknowledgments.
2. **0-RTT Packet:** Allows clients to send early data before the handshake is complete, enabling reduced latency for applications that support it.
3. **Handshake Packet:** Used to complete the cryptographic handshake and exchange keys securely.
4. **Retry Packet:** Carries an address validation token from the server, enabling the client to retry the connection.

Short Headers

Once the handshake is complete and 1-RTT keys are available, communication shifts to using short headers [2]. These headers are more compact and efficient, as they eliminate unnecessary fields that were critical during the handshake. Short headers are identified by their `Header Form` field, where the most significant bit `(0x80)` of the first byte is set to 0.

The short header of QUIC version 1 and version 2 includes the following fields [1], [2], [8]:

1. **Header Form:** The most significant bit (0x80) of byte 0 (the first byte) is set to 0 for short headers.
2. **Fixed Bit:** The next bit (0x40) of byte 0 is set to 1. Packets containing a zero value for this bit are not valid packets in this version.
3. **Spin Bit:** The third most significant bit (0x20) of byte 0 is the latency spin bit.
4. **Reserved Bits:** The next two bits (those with a mask of 0x18) of byte 0 are reserved. These bits are protected using header protection.
5. **Key Phase:** The next bit (0x04) of byte 0 indicates the key phase, which allows a recipient of a packet to identify the packet protection keys that are used to protect the packet.

6. **Packet Number Length:** The least significant two bits (those with a mask of 0x03) of byte 0 contain the length of the Packet Number field.

7. **Destination Connection ID:** Destination Connection ID as described in the section above. Unlike the long header, its length is not provided, making it unclear where exactly it ends.

8. **Packet Number:** The Packet Number field is 1 to 4 bytes long and is ecrypted using header protection.

9. **Packet Payload:** The encrypted remainder of the packet.

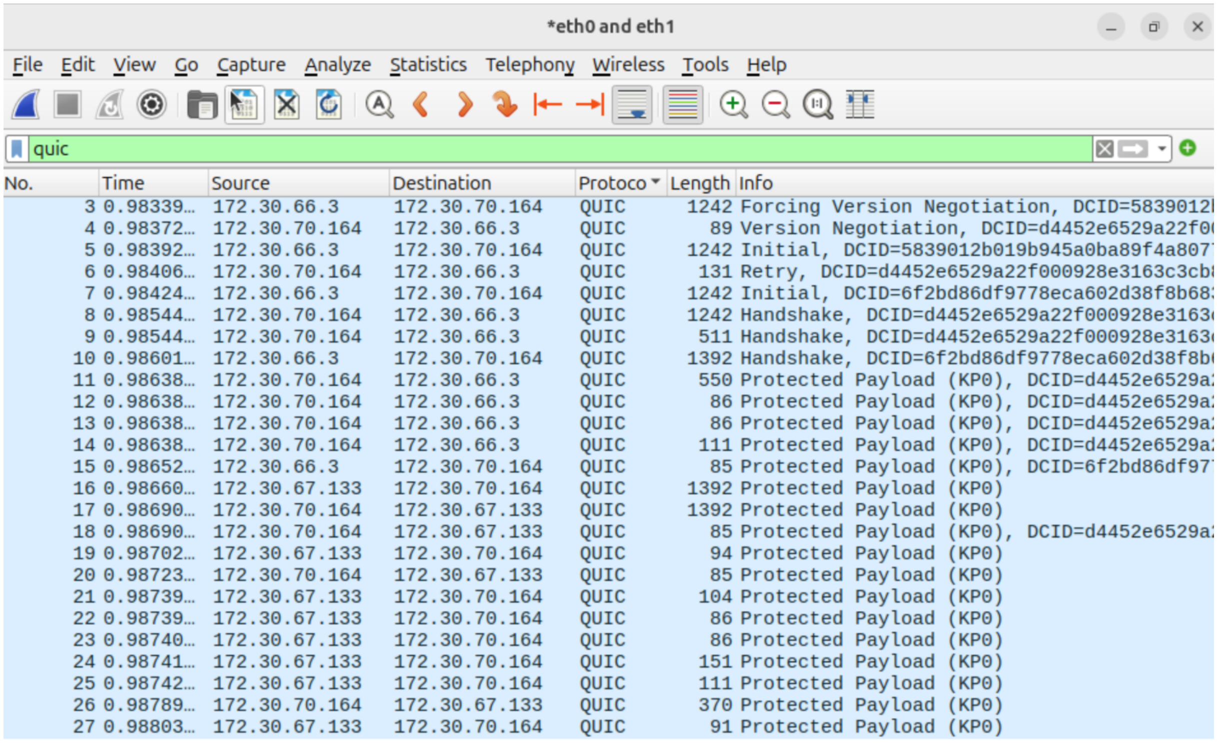
2.1.3 Connection Migrations

The use of connection IDs in QUIC provides a mechanism for connections to survive changes in endpoint addresses, such as those caused by migrating to a new network. A common example is when a mobile device switches from WiFi to cellular data as the user leaves their home. In traditional HTTP/2 over TCP, such a scenario would require a complete reestablishment of the connection. In contrast, QUIC allows for seamless migration, ensuring uninterrupted communication. An endpoint is prohibited from initiating a connection migration before the handshake has been completed [2].

To perform a connection migration, an endpoint must first perform a path validation and send probing packets to verify that the peer is reachable on the new path. Path validation is initiated by sending a `PATH_CHALLENGE` frame containing an unpredictable payload. The receiving endpoint must respond with a `PATH_RESPONSE` frame containing the same data as the challenge. If the `PATH_RESPONSE` payload does not match the `PATH_CHALLENGE` payload, the initiating endpoint may generate a connection error, terminating the migration attempt [2].

QUIC requires that datagrams containing `PATH_CHALLENGE` frames be expanded to a minimum size of 1200 bytes [2]. In most implementations, such as Cloudflare Quiche [60], datagrams are typically 1392 bytes, which includes headers for Ethernet, IP, and UDP. For IPv4 specifically, this leaves 1350 bytes for payload data.

During testing with Cloudflare Quiche, an unusual behavior was observed: after a successful connection migration, the server sends one final packet to the client using the new IP and port but retaining the client's old connection ID. This can be seen in Figure 2.3, where Wireshark successfully parses the Destination Connection ID of packet 18, but fails to do so for the surrounding packets. This behavior is not documented in the official RFCs, and its purpose is unclear. It is unknown whether this represents an intentional feature or a bug. Although this behavior is handled within this project's implementation, it is not essential for the migration process to function properly.


*eth0 and eth1
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
quic
```

```
No.  Time       Source         Destination    Protoco Length Info
   3 0.98339…  172.30.66.3    172.30.70.164  QUIC     1242 Forcing Version Negotiation, DCID=5839012l
   4 0.98372…  172.30.70.164  172.30.66.3    QUIC       89 Version Negotiation, DCID=d4452e6529a22f0
   5 0.98392…  172.30.66.3    172.30.70.164  QUIC     1242 Initial, DCID=5839012b019b945a0ba89f4a807
   6 0.98406…  172.30.70.164  172.30.66.3    QUIC      131 Retry, DCID=d4452e6529a22f000928e3163c3cb
   7 0.98424…  172.30.66.3    172.30.70.164  QUIC     1242 Initial, DCID=6f2bd86df9778eca602d38f8b68
   8 0.98544…  172.30.70.164  172.30.66.3    QUIC     1242 Handshake, DCID=d4452e6529a22f000928e3163
   9 0.98544…  172.30.70.164  172.30.66.3    QUIC      511 Handshake, DCID=d4452e6529a22f000928e3163
  10 0.98601…  172.30.66.3    172.30.70.164  QUIC     1392 Handshake, DCID=6f2bd86df9778eca602d38f8b
  11 0.98638…  172.30.70.164  172.30.66.3    QUIC      550 Protected Payload (KP0), DCID=d4452e6529a
  12 0.98638…  172.30.70.164  172.30.66.3    QUIC       86 Protected Payload (KP0), DCID=d4452e6529a
  13 0.98638…  172.30.70.164  172.30.66.3    QUIC       86 Protected Payload (KP0), DCID=d4452e6529a
  14 0.98638…  172.30.70.164  172.30.66.3    QUIC      111 Protected Payload (KP0), DCID=d4452e6529a
  15 0.98652…  172.30.66.3    172.30.70.164  QUIC       85 Protected Payload (KP0), DCID=6f2bd86df97
  16 0.98660…  172.30.67.133  172.30.70.164  QUIC     1392 Protected Payload (KP0)
  17 0.98690…  172.30.70.164  172.30.67.133  QUIC     1392 Protected Payload (KP0)
  18 0.98690…  172.30.70.164  172.30.67.133  QUIC       85 Protected Payload (KP0), DCID=d4452e6529a
  19 0.98702…  172.30.67.133  172.30.70.164  QUIC       94 Protected Payload (KP0)
  20 0.98723…  172.30.70.164  172.30.67.133  QUIC       85 Protected Payload (KP0)
  21 0.98739…  172.30.67.133  172.30.70.164  QUIC      104 Protected Payload (KP0)
  22 0.98739…  172.30.67.133  172.30.70.164  QUIC       86 Protected Payload (KP0)
  23 0.98740…  172.30.67.133  172.30.70.164  QUIC       86 Protected Payload (KP0)
  24 0.98741…  172.30.67.133  172.30.70.164  QUIC      151 Protected Payload (KP0)
  25 0.98742…  172.30.67.133  172.30.70.164  QUIC      111 Protected Payload (KP0)
  26 0.98789…  172.30.70.164  172.30.67.133  QUIC      370 Protected Payload (KP0)
  27 0.98803…  172.30.67.133  172.30.70.164  QUIC       91 Protected Payload (KP0)
```

Figure 2.3: Quiche Connection Migration

Client-Side Connection Migrations

Connection migrations can be initiated by the client to change its IP address and/or port, for example when moving from WiFi to cellular data. In this process, the client sends a `PATH_CHALLENGE` to the server on the new path [2]. The server, upon receiving the challenge, responds with a `PATH_RESPONSE` containing the same payload. This exchange verifies the reachability and functionality of the new path before the migration is considered complete. Once validated, the client and server can continue communication seamlessly using the new IP and port, with no disruption to the established connection.

Server-Side Connection Migrations

Server-side connection migrations allow the server to change its IP address and/or port. The server can request migration by setting the `preferred_address` transport parameter [2]. This transport parameter includes addresses and ports for both IPv4 and IPv6, as well as additional details to facilitate the migration process.

The client, upon receiving the `preferred_address` transport parameter, initiates path validation by sending a `PATH_CHALLENGE` to the server's preferred address. The server responds with a `PATH_RESPONSE` to confirm its availability on the new path. Until the server receives a non-probing packet from the client on its preferred address and validates the path, it must continue sending non-probing packets from its original address. Once the server completes path validation and processes a non-probing packet from the client on the preferred address, it transitions fully to the new address and sends non-probing packets exclusively from that address [2].

At present, most QUIC implementations do not support server-side connection migrations yet. A possible next step are server-initiated connection migrations, where the server is allowed to send a `PATH_CHALLENGE` to the client to initiate a migration. This is neither specified nor implemented in current RFCs, but discussions about incorporating this feature into future versions of QUIC are ongoing.

2.1.4 Security Considerations

QUIC introduces several features that enhance privacy and security but also create challenges for network monitoring and enforcement. The packet payload along with parts of the header are fully encrypted. This limits the amount of metadata available to middleboxes, complicating traffic filtering and enforcement policies. When combined with encrypted DNS protocols such as DNS-over-HTTPS or DNS-over-QUIC, network administrators are left with minimal visibility into connection destinations, making traditional security measures like monitoring and filtering harder to apply.

QUIC's dynamic nature also poses challenges for tracking connections over time. Parameters such as connection IDs can change at any time, and connection migrations allow endpoints to switch IP addresses and ports seamlessly [2]. These features, while improving resilience and mobility, make it difficult to maintain persistent tracking of connections, as identifiers and endpoints are not static.

Network security appliances and firewalls, including those from major vendors like Palo Alto, Fortinet, Sophos, and Zscaler, currently lack support for all QUIC features [55], [56], [57], [58], [59]. These vendors recommend blocking QUIC traffic by default and forcing a fallback to HTTP/2 over TCP with TLS. While this approach provides a temporary solution, it undermines the performance and efficiency benefits of QUIC.

Server-side connection migrations, which are a focus of this project, introduce additional security considerations. In theory, a malicious server could exploit this feature to hijack an existing connection by requesting a connection migration to itself.

2.2 Stateful Firewall

A firewall is an important component of network security, designed to monitor and control incoming and outgoing network traffic based on predefined security rules. Firewalls can act as a barrier between a trusted internal network and untrusted external networks, such as the Internet, to prevent unauthorized access and mitigate security threats.

A stateful firewall goes beyond simple packet filtering by maintaining information about the state of active connections. This connection state tracking enables the firewall to make more informed decisions about whether to allow or block packets [48].

Dynamic packet filtering is a key feature of stateful firewalls. Rather than relying solely on static rules, the firewall dynamically adapts its filtering behavior based on the state of connections. For example, when a connection is initiated, the firewall records its details, such as source and destination IP addresses, ports, and protocol type. This allows the firewall to permit only packets that belong to established or expected connections while blocking others.

To achieve this, stateful firewalls maintain a table of known connections, often referred to as a state table or connection table. This table tracks active connections and their associated parameters, such as sequence numbers, timeouts, and connection states such as `ESTABLISHED` or `CLOSED`. When a new packet arrives, the firewall checks the state table to determine whether the packet is part of an existing connection or is a valid new connection request. If the packet matches an entry in the state table, it is allowed to pass. Otherwise, it is subjected to further verification based on the firewall's rules.

In modern networks, stateful firewalls are widely used to provide robust security for applications and services. They are often combined with other security features, such as deep packet inspection and intrusion prevention systems, to deliver comprehensive protection against complex threats.

2.2.1 State Table

The state table is a critical component of a stateful firewall, as it contains detailed information about active connections. This table allows the firewall to track ongoing sessions and make decisions about whether incoming or outgoing packets are part of legitimate, established connections or potential threats [45], [46], [48].

Typically, each entry in the state table is represented as a 5-tuple, which uniquely identifies a connection. The 5-tuple includes the following elements [45], [46]:

1. **Protocol:** The transport protocol used by the connection, such as TCP, UDP, or ICMP.
2. **Local IP Address:** The IP address of the local, internal computer initiating or participating in the connection.
3. **Local Port:** The port number on the local computer associated with the connection.
4. **Remote IP Address:** The IP address of the remote computer with which the connection is established.
5. **Remote Port:** The port number on the remote computer associated with the connection.

In addition to these fields, the state table may also include other metadata, such as the connection state, sequence numbers for TCP connections, time-to-live (TTL) or timeout values, and flags indicating whether the connection is inbound or outbound.

When a new packet arrives at the firewall, it is compared against the entries in the state table. If a match is found, the packet is allowed to pass, provided it aligns with the expected behavior of the connection. If no match exists, the firewall applies its rules to decide whether to create a new state table entry or block the packet.

The state table is dynamic and updates continuously as connections are established, maintained, and terminated. For instance, when a TCP handshake occurs, the state table records the initial SYN packet and updates the connection's state as the handshake progresses. Similarly, when a connection is closed, the corresponding entry is removed from the state table to free up resources.

Efficient management of the state table is crucial, especially in high-traffic environments. Firewalls must balance maintaining accurate connection states with minimizing memory and computational overhead. Modern implementations often employ techniques such as aging and timeouts to remove stale entries and ensure optimal performance.

2.2.2 Network Address Translation

Network Address Translation (NAT) is a method used to map one set of IP addresses to another, allowing for efficient use of IP address space and providing an additional layer of security [45], [46], [48]. NAT operates by modifying the IP address and, optionally, the port number in the header of IP packets as they traverse a router or firewall. There are two primary types of NAT: Source NAT and Destination NAT.

Source NAT

Source NAT (SNAT) is primarily used to modify the source IP address of outgoing packets. This is commonly applied in scenarios where devices within a private network need to access the Internet or another external network. Since private IP addresses cannot be routed on the public Internet, SNAT rewrites the source IP address of packets with the public IP address of the NAT device [45], [48], [51].

For example, when a device in a local network sends a packet to an external server, the NAT device replaces the device's private IP address with its own public IP address before forwarding the packet. The NAT device also creates an entry in its state table to track this connection. When a response arrives, the NAT device uses the state table to map the public IP address back to the original private IP address and forwards the packet to the correct device within the local network.

Source NAT is widely used in enterprise networks, home routers, and cloud infrastructure to enable multiple devices to share a single public IP address. It also provides a layer of anonymity, as the external server only sees the public IP address of the NAT device rather than the internal private IP addresses.

Destination NAT

Destination NAT (DNAT) is used to modify the destination IP address of incoming packets. This is typically employed in scenarios where external clients need to access specific servers or services within a private network, such as hosting a webserver behind a firewall [51].

When a packet arrives at the NAT device with a public IP address as its destination, DNAT rewrites the destination IP address to the private IP address of the corresponding server within the internal network. For example, an organization hosting a webserver behind a firewall may use DNAT to translate requests sent to the public IP address of the firewall into the private IP address of the web server. Similar to SNAT, the NAT device maintains a state table to track these connections and ensure that responses from the internal server are correctly routed back to the external client.

DNAT is often used in conjunction with port forwarding to map specific ports on the public IP address to corresponding ports on private servers. This allows organizations to host multiple services behind a single public IP address by assigning unique ports to each service.

Both SNAT and DNAT are essential to modern network architecture, enabling efficient use of IP address space while facilitating secure and seamless communication between private and public networks.

2.2.3 Firewalls and UDP

Unlike TCP, which is a connection-oriented protocol with defined stages for establishing and terminating connections, such as `SYN`, `SYN-ACK`, and `ACK` for initiation, and `FIN` for closure, UDP operates fundamentally different. UDP is a connectionless protocol, meaning it does not establish a formal handshake or session with the recipient before transmitting data. Similarly, there are no sequence numbers to track the order or reliability of packets, making UDP simpler and faster but inherently stateless.

The lack of state in UDP leads to unique challenges for firewalls, which are designed to track connection states to enforce security policies. With TCP, firewalls can monitor the state of a connection through its lifecycle, from initiation to termination, and adjust filtering rules accordingly [7]. UDP, however, has no such lifecycle. This forces firewalls to rely on heuristics and time-based mechanisms to manage UDP traffic.

To manage UDP flows effectively, firewalls impose timeouts to close idle UDP ”connections” after a certain period of inactivity. Without these timeouts, stale UDP flows would remain in the state table indefinitely, consuming resources and potentially being exploited for malicious purposes. A typical timeout value for idle UDP connections is two minutes, as recommended in many network security practices [7].

This timeout mechanism strikes a balance between security and usability, but it is not without drawbacks. For applications requiring long-lived or periodic UDP communication, such as video streaming or gaming, the timeout can prematurely terminate sessions unless periodic keepalive messages are sent to refresh the connection in the firewall's state table. This highlights the importance of understanding application behavior and configuring firewalls to accommodate UDP traffic without compromising security.

Chapter 3

Related Work

In this chapter, recent scientific literature about QUIC is discussed. The literature is organized into four main sections. The first section provides a general overview of the QUIC protocol, including its specifications and other general papers that cover its fundamental features. The second section focuses on papers that discuss connection migrations, exploring the possibilities and challenges associated with this feature. The third section looks into the security aspects of QUIC, highlighting potential vulnerabilities and solutions. Finally, the last section covers research on stateful firewalls in general and their handling of UDP traffic.

3.1 QUIC

QUIC was initially developed by Google and later adopted and standardized by the Internet Engineering Task Force (IETF). The IETF's specification for the first version of QUIC was published in a series of RFCs in 2021, namely RFC 8999 [1], RFC 9000 [2], RFC 9001 [3], and RFC 9002 [4].

RFC 8999 [1] establishes the version-independent properties of QUIC, providing a foundation for compability of future iterations. RFC 9000 [2] provides a comprehensive description of QUIC Version 1, emphasizing its design as a secure, multiplexed transport protocol operating over UDP, with a focus on low latency and high performance. To secure QUIC connections, RFC 9001 [3] details the integration of TLS, enabling robust encryption and authentication during connection establishment. Additionally, RFC 9002 [4] specifies loss detection and congestion control mechanisms tailored to QUIC, highlighting its performance-oriented approach to reliability and network efficiency.

The protocol's applicability and deployment considerations are further addressed in RFC 9308 [6] and RFC 9312 [7], which explore QUIC's practical use cases and manageability, respectively. RFC 9114 [5] extends the protocol's adoption by defining HTTP/3, the latest iteration of HTTP designed to run over QUIC. Lastly, RFC 9369 [8] introduces QUIC Version 2, which is identical to Version 1 but aims to combat ossification vectors and exercise the version negotiation framework.

Table 3.1: Literature: General QUIC

Research Work	**Year**	**Summary**
RFC 8999 [1]	2021	Defines version-independent properties of QUIC for compatibility.
RFC 9000 [2]	2021	Specification of QUIC Version 1.
RFC 9001 [3]	2021	Integration of TLS 1.3 to ensure secure connections.
RFC 9002 [4]	2021	Explains QUIC's loss detection and congestion control mechanisms.
RFC 9114 [5]	2022	Specifies HTTP/3, the application layer running on QUIC.
RFC 9308 [6]	2022	Discusses use cases and deployment considerations for QUIC.
RFC 9312 [7]	2022	Explores challenges in managing QUIC traffic in networks.
RFC 9369 [8]	2023	Specification of QUIC Version 2.
The QUIC Transport Protocol: Design and Internet-Scale Deployment [9]	2017	Google's overview of QUIC's design and large-scale deployment.
Quick UDP Internet Connections: Multiplexed stream transport over UDP [10]	2012	Early presentation of the initial concept of QUIC.
A QUIC way to bypass your firewall [11]	2023	High-level description of QUIC and it's implications for corporate networks
QUIC (Quick UDP Internet Connections) - A Quick Study [12]	2020	A detailed overview of QUIC.
QUIC: Better for what and for whom? [13]	2017	Evaluates QUIC's benefits in diverse networking scenarios.
Multipath QUIC: Design and Evaluation [14]	2017	Proposes and evaluates MPQUIC, a multipath extension of QUIC.
Implementation and analysis of QUIC for MQTT [15]	2019	Implements and analyzes a QUIC extension for MQTT, a popular IoT application layer protocol.
QUIC - Quick UDP Internet Connections [16]	2016	Overview of QUIC's most important features.
The Performance and Future of QUIC Protocol in the Modern Internet, Network and Communication Technologies [17]	2021	Compares and discusses the performance of QUIC and TCP/TLS.
Evaluating QUIC performance over web, cloud storage, and video workloads [18]	2022	Comparing performance of different QUIC versions and TCP/TLS
Web censorship measurements of HTTP/3 over QUIC [19]	2021	Using HTTP/3 to circumvent censorship measurements by different countries

Beyond the formal specifications, various studies have examined QUIC's design, deployment, and implications. Langley et al. [9] provide a foundational overview of the protocol's design principles and its development at Google. Roskind [10] presents an early conceptualization of QUIC as a multiplexed transport over UDP, laying the groundwork for its eventual standardization. Glisovic [11], Kumar et al. [12] and Gratzer et al. [16] provide a detailed overview of QUIC's features and potential caveats.

Other studies, such as those by De Coninck and Bonaventure [14] and Kumar and Dezfouli [15], explore innovative uses of QUIC, such as enabling multipath communication and optimizing IoT protocols like MQTT. Cook et al. [13], Wang [17] and Shreedhar et al. [18] contribute by evaluating QUIC's performance in diverse contexts and identifying areas for improvement.

Elmenhorst et al. [19] analyzed the use of QUIC in web censorship contexts, demonstrating how HTTP/3 can be used to circumvent website blocking in countries such as China, Iran, India, and Kazakhstan.

These specifications and papers lay the foundation for understanding QUIC and HTTP/3, providing a solid base for further research and the development of the firewall. They offer a detailed exploration of QUIC's architecture and its various features.

3.2 Connection Migrations

Connection migrations, which allows an ongoing QUIC session to switch IP addresses or network interfaces without interrupting communication, has been the focus of various studies exploring to use its potential for performance enhancement, privacy, and network efficiency.

Govil et al. [20] introduced MIMIQ, an approach that leverages connection migrations in QUIC to mask IP addresses for privacy preservation. By strategically migrating connections, this method obfuscates the user's identity and enhances resistance to surveillance and tracking, while maintaining secure and seamless communication. Similarly, Wang et al. [22] explored how strategic traffic splitting could improve user privacy. Their study emphasizes the potential to distribute traffic across multiple paths, mitigating network-level traffic analysis.

Puliafito et al. [21] examined the use of server-side QUIC connection migration to support microservice deployments. By enabling connections to move between servers without service interruption, their approach improves flexibility and performance in edge-based microservice architectures. Haoran [23] proposed a load-balancing solution using QUIC connection migrations, enabling seamless state transfers between servers to optimize resource utilization and ensure low-latency communication.

Yan and Yang [24] conducted a case study on mobile WiFi hotspots, investigating whether QUIC's connection migrations can effectively reduce latency caused by cellular network suspensions in multi-carrier WiFi relays. The study identifies a deadlock issue that negates the benefits of passive migrations.

Table 3.2: Literature: Connection Migrations

Research Work	Year	Summary
MIMIQ: Masking IPs with Migration in QUIC [20]	2020	Increasing privacy by changing client's IP address frequently
Server-side QUIC connection migration to support microservice deployment at the edge [21]	2022	Extension of QUIC to support server-side connection migrations for microservice containers
Leveraging strategic connection migration-powered traffic splitting for privacy [22]	2022	Enhancing privacy with CoMPS, that splits traffic mid-session across network paths to mitigate network-level fingerprinting
The Design and Evaluation of a Seamless Approach to Migrate the State of QUIC Connections for Load Balancing Purposes [23]	2021	Implementing and Evaluating Load Balancing using picoquic
When QUIC's Connection Migration Meets Middleboxes [24]	2021	Identification of a deadlock when migrating connections with a multi-carrier Wi-Fi relay to counter latency caused by cellular network suspension
An Analysis of QUIC Connection Migration in the Wild [25]	2024	Scanning webservers world-wide to check whether connection migrations are implemented or not

From a broader perspective, Buchet and Pelsser [25] analyzed the current state of connection migrations in real-world QUIC implementations. Through Internet-wide scans, they tested whether current deployments supported connection migrations and found that most do not yet support this feature in production.

Together, these works display the potential of QUIC's connection migration. Whether for privacy enhancement, load balancing, edge computing, or overcoming network obstacles, this feature enables innovative networking solutions.

3.3 Security Challenges

Several studies in the past few years have explored the security benefits of QUIC, highlighting it's built-in encryption. At the same time, several have identified potential vulnerabilities and discussed possible mitigations to enhance the protocol's security.

Lychev et al. [26] conducted one of the earliest comprehensive analyses of QUIC's security and performance. Their work offered provable security guarantees while benchmarking its performance, demonstrating how QUIC balances robustness and speed in comparison to traditional protocols. Chatzoglou et al. [27] revisited the protocol's vulnerabilities

Table 3.3: Literature: Security Considerations

Research Work	**Year**	**Summary**
How Secure and Quick is QUIC? [26]	2015	Exploring attacks and weaknesses of QUIC seemingly introduced mechanisms to reduce latency.
Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study [27]	2023	Identifies several vulnerabilities to overwhelm the server resource.
A QUIC(K) Way Through Your Firewall? [28]	2021	QUIC exposes traditional stateful firewalls to UDP hole punching bypass attacks.
Security and Service Vulnerabilities with HTTP/3 [29]	2024	Connection migrations can be used to launch Denial of Service attacks and disrupts middlebox services like load-balancers, rate-limiters, and intrusion detection /prevention systems.
Secure Middlebox-Assisted QUIC [30]	2023	Selectively exposing information to enable middleboxes while preserving its privacy, integrity, and authenticity.
Security and Performance Evaluations of QUIC Protocol [31]	2020	Evaluation and analysis of QUIC implementations finding issues in data loss recovery and forward error correction.
Intrusion detection on Quic Traffic: A machine learning approach [32]	2022	Machine learning approach based on fingerprinting to detect malicious C2 QUIC traffic.
Rescuing QUIC Flows From Countermeasures Against UDP Flooding Attacks [33]	2024	QUIC can falsly be blocked by existing countermeasures against UDP flooding attacks and approaches to circumvent that.
Machine Learning for QUIC Traffic Flood Detection [34]	2024	Machine learning algorithm to distinguish between normal traffic and potential HTTP/3 flood.
Security Review and Performance Analysis of QUIC and TCP Protocols [35]	2022	QUIC outperforms TCP in page load times, throughput and security applications, but requires more processing power.

in a detailed review of QUIC-related attacks on different implementations. Gbur and Tschorsch [28] show that QUIC exposes traditional stateful firewalls to UDP hole punching attacks.

HTTP/3 security is further explored by Kulkarni [29], who demonstrated how connection migrations can be used to launch denial of service (DoS) attacks. By filling the state table of network devices with numerous unused connections, attackers can overwhelm these systems.

Soni and Rajput [31] evaluated QUIC's trade-offs, presenting detailed analyses of how its security features affect overall performance. Their findings highlight the efficiency of QUIC's built-in encryption mechanisms compared to external solutions. In addition, Oran et al. [35] conducted a comparative study of QUIC and TCP, offering insights into their respective strengths in terms of both security and performance.

Meanwhile, Kosek et al. [30] enhance QUIC by selectively exposing information to middleboxes, while preserving its privacy, integrity, and authenticity.

QUIC's growing adoption has also prompted research into new defense mechanisms. Al-Bakhat and Almuhammadi [32] developed a machine learning-based intrusion detection system tailored to QUIC traffic. Kadi et al. [34] extended this approach, applying machine learning techniques to detect and mitigate QUIC traffic flooding attacks. Lee et al. [33] proposed solutions for addressing the issue of QUIC being falsly blocked by existing countermeasures against UDP flooding attacks.

These studies hightlight QUIC's security advantages and challenges. As QUIC continues to gain adoption, addressing its security challenges will remain crucial to ensure QUIC's long-term success.

3.4 Stateful Firewalls

Stateful firewalls are a crucial component of modern network security, enabling the inspection and management of traffic flows based on their state and context. Unlike stateless packet filters, stateful firewalls maintain information about active connections, allowing them to enforce security policies across multiple packets. Research into stateful firewalls has explored their underlying models, evolution, and the challenges they face in the context of encrypted traffic.

Gouda and Liu [36] proposed one of the first formal models of stateful firewalls, defining their structure and properties. Their work offered a theoretical foundation for understanding how stateful inspection mechanisms operate and identified key characteristics that differentiate them from stateless approaches. Wool [37] complemented this by providing an accessible overview of stateful firewalls and their capabilities.

With the increase of encrypted traffic and complex application-layer protocols, stateful firewalls face new challenges. Kühlewind et al. [39] examined the difficulties network operators encounter when managing encrypted traffic, which limits the ability of stateful

Table 3.4: Literature: Stateful Firewalls

Research Work	**Year**	**Summary**
A model of stateful firewalls and its properties [36]	2005	Proposing a firewall model with a stateful and a stateless section.
Packet filtering and stateful firewalls [37]	2006	Discussing general firewall features.
Evolution of Firewalls: Toward Securer Network Using Next Generation Firewall [38]	2022	Reviewing the weaknesses of traditional firewalls and the features and adoption of Next-Gen firewalls.
Challenges in network management of encrypted traffic [39]	2018	Discussing challenges for middleboxes and providing recommendations for future protocols and use cases.
Traffic classification through simple statistical fingerprinting [40]	2007	Classification mechanism based on simple properties: packet size, inter-arrival time and arrival order.
Handling Stateful Firewall Anomalies [41]	2012	Providing solutions to analyze and handle stateful firewall anomalies and misconfiguration.
Improved Session Table Architecture for Denial of Stateful Firewall Attacks [42]	2018	Presenting a stateful session table architecture for a splay tree firewall.
Management of stateful firewall misconfiguration [43]	2013	Implementing an automatic audit tool that verfifies stateful firewall configuration files.
Advanced algorithms for fast and scalable deep packet inspection [44]	2006	Presenting an alternative representation of patterns used by middleboxes for pattern matching.
how TCP/IP works in a modern network [45]	2017	General information about the TCP/IP Stack.
TCP/IP network administration [46]	2002	General information about handling TCP/IP.
Transport layer proxy for stateful UDP packet filtering [47]	2002	Proposing a transport layer proxy that performs authentication, packet filtering, and more.
Stateful inspection firewalls [48]	2004	Juniper whitepaper about their firewall design.

firewalls to inspect payloads. Liang and Kim [38] highlighted the evolution of firewalls toward next-generation solutions, integrating deep packet inspection (DPI) and advanced analytics to overcome such limitations.

Techniques for improving firewall performance and security have also been extensively studied. Trabelsi et al. [42] proposed an enhanced session table architecture to mitigate DoS attacks. Kumar et al. [44] introduced an alternative representation of patterns used by middleboxes to match malicous patterns faster.

The practical implications of stateful firewall misconfigurations have been analyzed by Garcia-Alfaro et al. [43], who developed a systematic framework for detecting and correcting configuration errors. Similarly, Cuppens et al. [41] provide solutions to analyze and handle stateful firewall anomalies and misconfiguration.

In terms of traffic classification, Crotti et al. [40] introduced statistical fingerprinting techniques to classify traffic flows, which stateful firewalls can leverage for more precise rule enforcement. Chang and Fung [47] proposed transport-layer proxies for UDP packet filtering, extending stateful inspection capabilities to stateless protocols like UDP.

Furthermore, foundational works like those of Goralski [45] and Hunt [46] provide broader insights into how TCP/IP networks operate, contextualizing the role of stateful firewalls within modern network architectures. These works underscore the importance of understanding the transport layer, which is essential for effective firewall deployment.

Finally, Roeckl [48] provided a detailed technical explanation of stateful inspection in firewalls, illustrating how session tracking is implemented in enterprise-grade solutions.

Together, this research emphasizes the importance of stateful firewalls in protecting modern networks. As the share of encrypted traffic grows and attacks become more complex, ongoing improvements in inspection methods, traffic classification, and performance are important for maintaining strong network defenses.

3.5 Research Gap

Despite the a lot of research on QUIC, connection migrations, and stateful firewalls, a gap remains in integrating these areas. QUIC has been thoroughly analyzed, with significant efforts dedicated to understanding its potential, including advanced use cases of connection migration for privacy, load balancing, and performance optimization. Similarly, QUIC's security has been examined, with studies identifying vulnerabilities and proposing countermeasures. On the other hand, stateful firewalls are well-established and have evolved to handle modern traffic challenges, including encrypted communications and high-performance requirements.

However, no work has explicitly focused on designing or implementing stateful firewalls for QUIC. The unique characteristics of QUIC pose challenges to traditional firewall architectures. Current stateful firewall solutions are not equipped to handle these features effectively, leading to a disconnect between QUIC's capabilities and firewall functionality. Addressing this gap by developing stateful firewalls tailored for QUIC is essential for unlocking the protocol's potential without compromising network security.

Chapter 4

Design

This chapter presents the overall design of the firewall, describing its architecture, core components, and functionality. It begins with an overview of the system, outlining its requirements and scope. The following sections then break down the steps that a packet takes as it passes through the firewall. This design serves as the foundation for the implementation, which is described in detail in Chapter 5.

4.1 Overview

This project focuses on the design and implementation of a stateful firewall tailored specifically to QUIC traffic. The firewall operates within a typical corporate network setup, where an internal network relies on a firewall as the gateway to external networks such as the internet, as depicted in Figure 4.1. To simplify the implementation, only Source Network Address Translation (SNAT) is implemented, meaning that only outgoing traffic can establish connections. There is no Destination NAT or port forwarding, making the firewall unsuitable for hosting web servers or similar services behind it. This decision was made to avoid general firewall concerns and instead focus on QUIC-specific functionality.

To further simplify the design, the implementation focuses exclusively on IPv4, avoiding the added complexity of supporting both IPv4 and IPv6. Handling IPv6 would require the system to address differences in address lengths and formats, as well as unique features like link-local addresses and stateless address autoconfiguration. NAT behavior also differs significantly, with IPv6 often relying on prefix-based translation rather than port mapping [49], [50], [52]. By restricting the implementation to IPv4, the design remains straightforward, ensuring a clean and manageable structure for packet processing and connection tracking. This decision was made to focus on QUIC-specific features.

The firewall operates as a passive observer, without engaging in deep packet inspection. It does not decrypt the encrypted payloads of QUIC packets, which would require the modification of endpoints such as the distribution of custom certificates. Instead, it analyzes packet headers to manage and control connections. This approach ensures simplicity while maintaining the ability to monitor and handle QUIC traffic effectively.

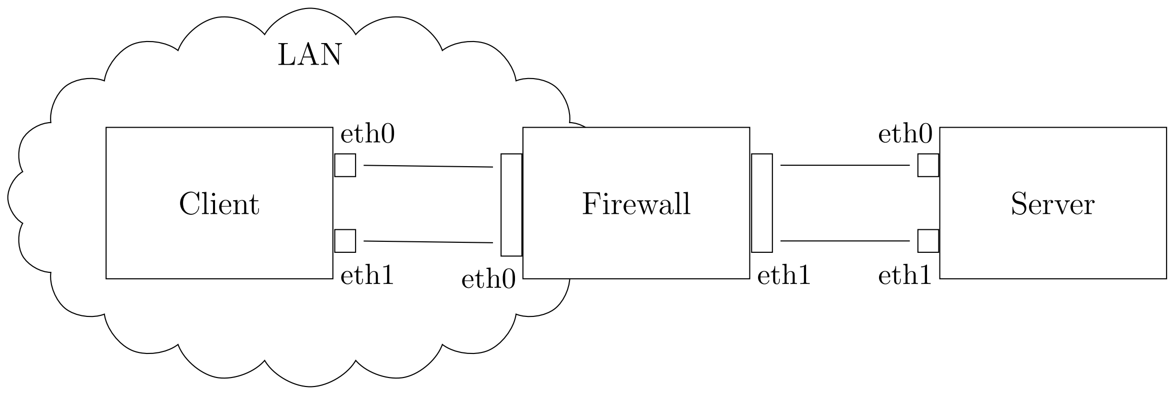


Figure 4.1: Network Setup

4.2 Packets

As a passive observer, only limited information is available to the firewall, including the IP headers, UDP headers and QUIC headers. These include properties such as the source and destination IP addresses, source and destination ports, QUIC packet type, and source and destination connection IDs. These fields are the basis of the state table.

The structure of a QUIC header is determined by its type, which can be either a long header or a short header. Long headers are used in the early stages of a connection and include packet types defined in RFC 9000 [2] such as `Initial`, `Handshake`, `Retry`, and `0-RTT`. Short headers, on the other hand, are used for regular traffic during an established connection. While the RFC specifies only one type of short header packet, `1-RTT`, it would be helpful to distinguish between different kinds of short header traffic to address specific scenarios. Regular traffic therefore is categorized as `EncryptedPayload`, similar to how Wireshark classifies this traffic. Additionally, packets used in connection migration should be identified, namely `PathChallenge` and `PathResponse`.

Another critical aspect of packet analysis is determining the direction of traffic. Knowing whether a packet is outgoing or incoming is essential for several reasons. Outgoing traffic, originating from trusted internal devices, is the only traffic permitted to initiate new connections. Additionally, the direction is important for the correct use of SNAT. For outgoing traffic, the source IP and port must be modified; for incoming traffic, the destination IP and port must be modified. The firewall determines the direction by identifying which interface — the internal or external one — captured the packet.

Finally, the firewall tracks the state of each packet to guide its handling. Packets can fall into one of three states: `Allowed`, `Invalid`, or `Dropped`. A packet is marked as `Invalid` if parsing fails due to a malformed header, indicating that it does not conform to expected standards. `Dropped` packets are correctly constructed packets that are not permitted, such as `EncryptedPayload` without an active connection or packets that are associated with blocked connections.

4.3 State Table

Stateful firewalls rely on maintaining a state table to track active connections. Typically, this involves a 5-tuple consisting of the protocol, source IP, source port, destination IP, and destination port. However, since this firewall exclusively handles QUIC traffic, the protocol field can be omitted, reducing the tuple to four parameters. In return, QUIC has two additional parameters, which can be included in the state table: the source connection ID and the destination connection ID.

The main challenge with QUIC is its flexibility, as any of these parameters can change during the lifetime of a connection. This requires the firewall to carefully manage when and how these changes are permitted to ensure the integrity of the connection.

The state table also tracks the current state of each connection, indicating whether it is `Open`, `Closed`, or `Blocked`. Because the firewall in this project only handles QUIC traffic, additional QUIC specific states can be added to show the lifecycle of a connection. Instead of `Open` states like `New`, `Handshake`, `Established`, and `PathProbe` can be added.

Another challenge is that UDP does not provide explicit connection termination signals. Unlike TCP, which uses a FIN packet to indicate the end of a connection, UDP connections are typically considered closed when no traffic is observed for a specified period. RFC 9312 [7] recommends the UDP-typical value of 2 minutes for such a timeout.

4.4 Packet Flow

When a packet passes through the firewall, it undergoes different steps, shown in Figure 4.2, to analyze the headers and to determine the validity of the packet. The following sections describe these stages in detail.

4.4.1 Listening on network interface and capturing packets

As a gateway between an internal network and an external network, the firewall needs two network interfaces, one in each respective network. The internal interface uses a static IP address, such as `10.0.0.1` in the test setup. On all internal clients, this IP is then configured as their gateway. In contrast, the external interface is configured to obtain a public IP address dynamically via DHCP.

To handle continuous traffic flow, the system operates with two concurrent threads, one assigned to each interface. This enables simultaneous processing of internal and external traffic, ensuring timely handling of packets. Each thread captures incoming packets and processes them individually, while accessing the same state table.

Because forwarded traffic arrives on one interface and is sent out by the firewall on the other, the firewall must ignore this outgoing traffic, otherwise the packet would be processed twice. The crate `pcap` [65], which is used to create the interface listener, can

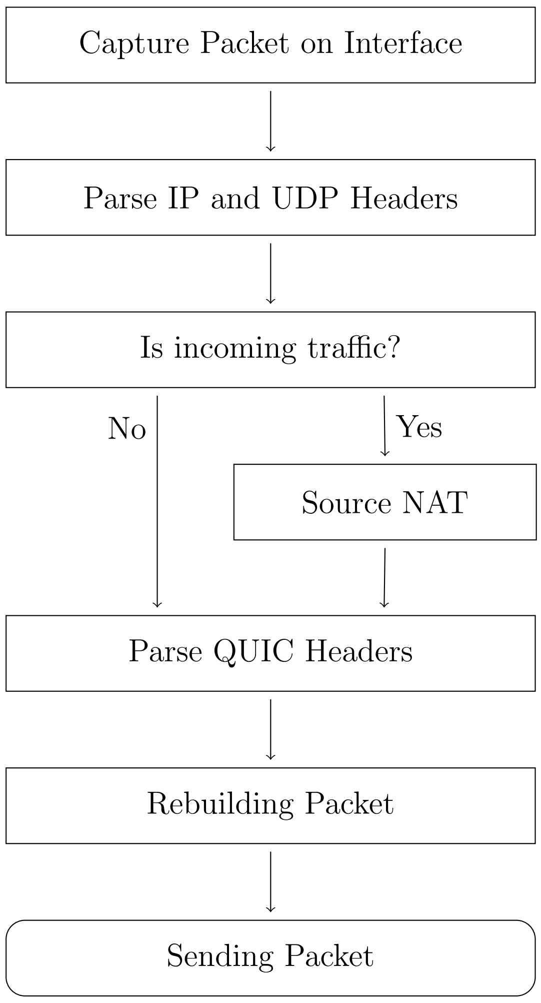


Figure 4.2: Packet Flow

configure the direction of a packet from the point of view of an interface. The `Direction::In` is required for both interfaces, both internal and external, which means that only arriving packets and not sent packets are processed on either interface.

To further simplify the design of the firewall, only QUIC traffic is processed. This is also done with a `pcap` filter, which only allows UDP traffic on a source or destination port 443.

4.4.2 Parsing of IP and UDP Headers

The first step in processing packets is parsing the IP and UDP headers. For this purpose, the `etherparse` crate [66] is utilized to parse these headers effectively. The IP header contains the source and destination IP addresses, while the UDP header contains the source and destination port. This ensures that each packet's essential metadata is extracted for further processing.

To uniquely identify each packet, an incremental counter is used. Since multiple threads may be processing packets simultaneously, ensuring uniqueness across threads is critical.

A global counter is shared between the threads, where each thread reads the current value, assigns it to the packet, and then increments the counter. This mechanism guarantees that every packet is assigned a unique ID, even in a multi-threaded environment.

When handling a large number of packets, simply printing errors to the console becomes impractical. To better manage invalid packets, an empty packet structure with default values is instantiated at the start of processing. As the packet is being parsed, this structure is filled with the parsed values. This approach makes it easier to pinpoint the specific stage where the parsing failed. Additionally, an `error` field is included in the packet structure to store any encountered error directly, improving traceability and simplifying debugging.

4.4.3 Source Network Address Translation

This next step is only relevant for incoming traffic, meaning traffic originating from the external network and destined for the internal network. For outgoing traffic, source NAT is applied later, during the packet reconstruction phase (discussed in Chapter 4.4.6).

Figure 4.3 illustrates how SNAT is implemented. When a packet is sent from an internal client, it initially contains the client's internal IP address as the source IP and typically uses a high source port, such as 50000 in this example. The destination of the packet is the external server's IP address and port. Since the internal IP address is not visible to the external server, the firewall must modify the packet's source IP to its own public IP address. This ensures that the external server knows where to send the response to.

When the server sends a response, the packet's destination IP address is the firewall's public IP address. To deliver this response to the correct internal client, the firewall must translate the destination IP back to the client's original internal IP address. This is done by looking up the connection in the state table using what is now the destination port, 50000 in this case.

An issue arises when multiple internal clients use the same source port. For instance, if Client 1 and Client 2 both use source port 50000, the firewall would not know which client should receive the response if it only modified the source IP address. To prevent this, the firewall also modifies the source port, ensuring that each source port is unique. Typically, the firewall uses the same source port as the client, but if this port is already assigned, a new one is selected. The new source port, called `firewall_port` in the implementation, is also stored in the state table.

In the implementation, when an incoming packet arrives, the destination IP address is the public IP of the external interface, and it needs to be translated to the actual recipient's private IP address. Using the destination port of the incoming packet, the state table can be searched to retrieve the correct private destination IP and port. These values are then used to overwrite the destination ip and port in the packet struct.

For outgoing packets, this step is skipped entirely since source NAT is applied later in the process.

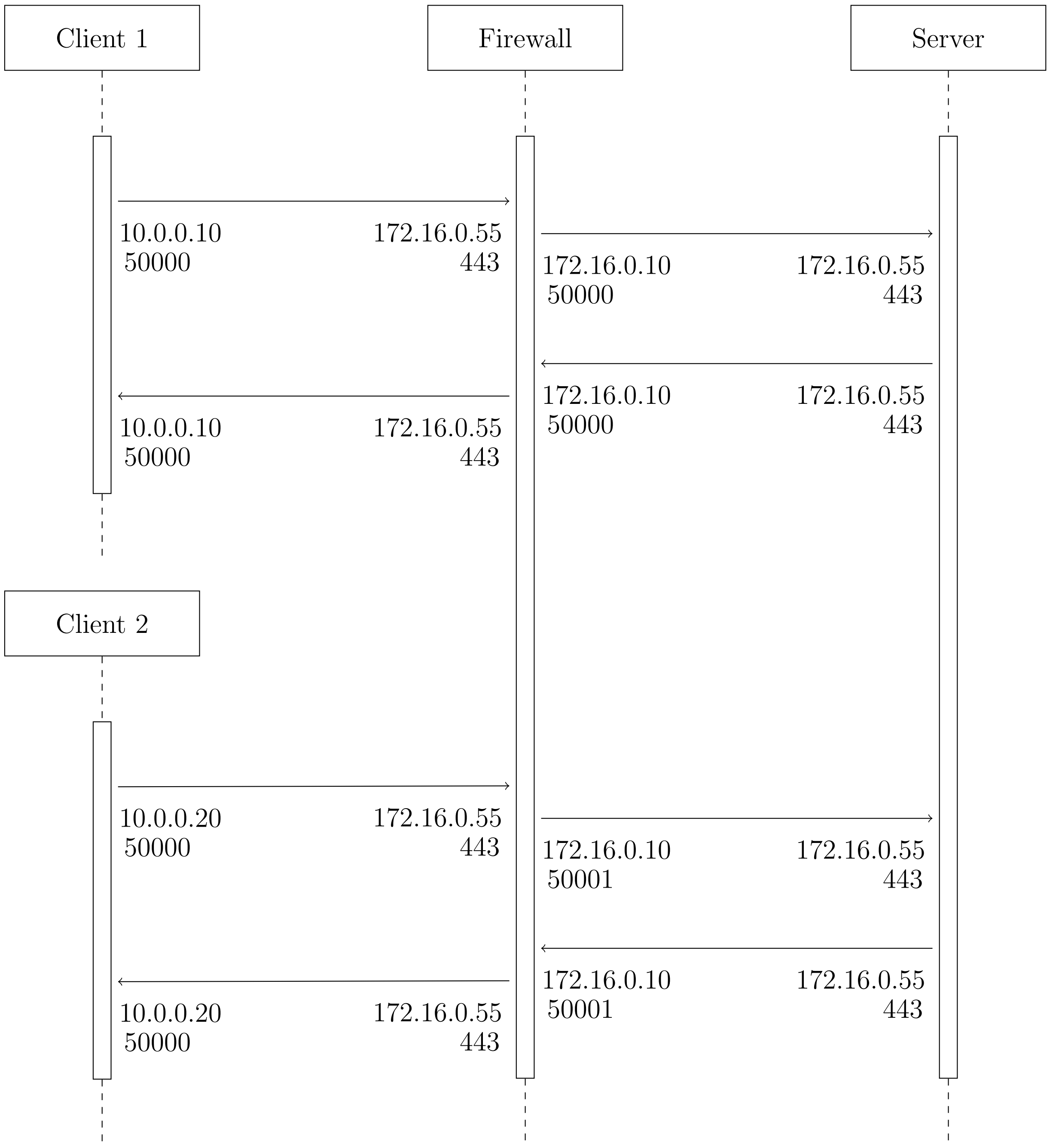


Figure 4.3: Source Network Address Translation

4.4.4 Parsing of QUIC Headers

Parsing QUIC headers requires different approaches for different types of headers. As discussed in Chapter 2.1.2, the RFC 9000 [2] defines two types of headers — long and short headers — each serving different purposes and requiring specific handling. The first bit of a QUIC packet determines the header type: a value of 1 indicates a long header, while a value of 0 indicates a a short header.

Long headers include both the source and destination connection IDs, along with their respective lengths. This makes it straightforward to determine the correct source and

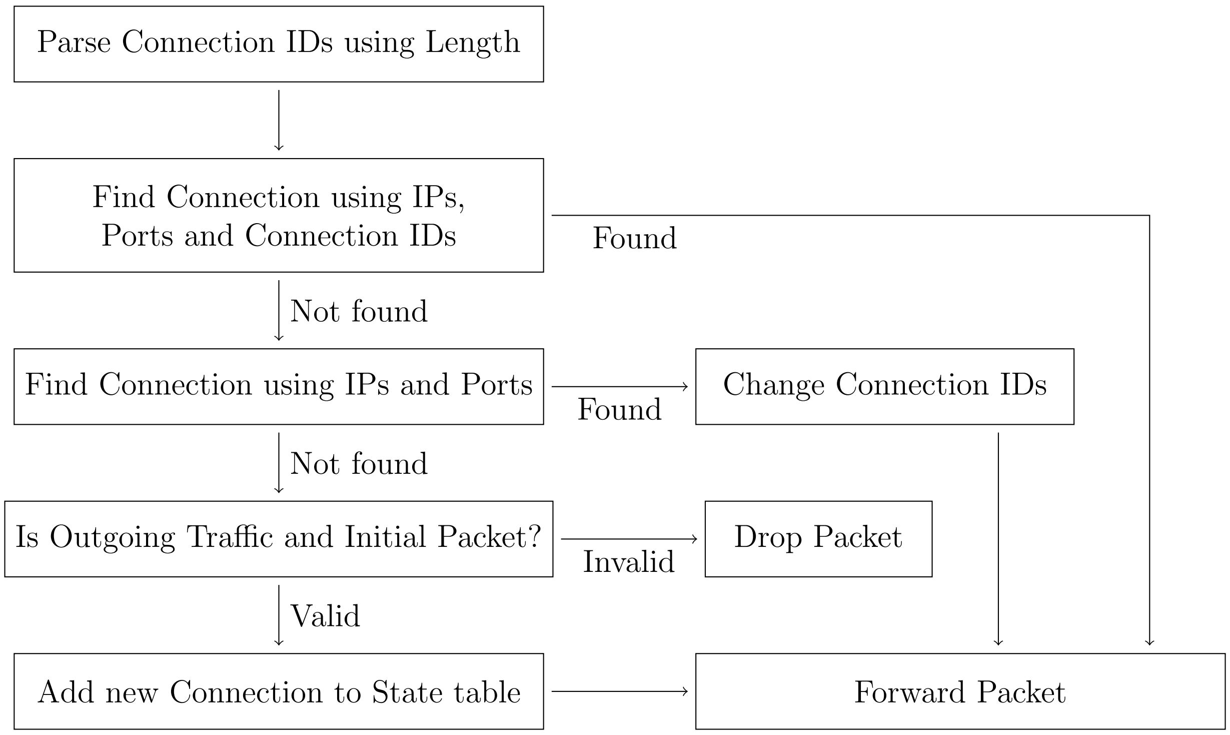

Parse Connection IDs using Length
Find Connection using IPs, Ports and Connection IDs
Found
Not found
Find Connection using IPs and Ports
Found
Change Connection IDs
Not found
Is Outgoing Traffic and Initial Packet?
Invalid
Drop Packet
Valid
Add new Connection to State table
Forward Packet

Figure 4.4: Long Header Parsing

destination connection IDs. Furthermore, long headers specify the packet type, which can be either `Initial`, `Handshake`, `Retry`, or `Zero-RTT`. These packet types are important for understanding and managing the current state of a QUIC connection.

Handling long headers involves three main scenarios. The first scenario occurs when a packet is the very first packet in a new connection, meaning there is no existing entry in the state table. Only outgoing traffic is allowed of creating new connections. In the current project setup, which does not include DNAT, connection setup from an external device is not practical because there is no clear way to determine which client should receive the traffic. As a result, the firewall only creates a new state table entry if the packet is outgoing and of type `Initial`.

The second scenario occurs when a packet introduces new connection IDs, such as during a `Retry`. In this case, the system updates the state table to reflect the new connection IDs, while both source and destination IPs and ports remain valid.

The final scenario involves packets that are part of an existing handshake. Here, the system validates the packet against the state table using IPs, ports, and connection IDs to confirm its role in the ongoing connection.

The firewall checks these scenarios in reverse order, as depicted in Figure 4.4. It starts by looking for an existing connection in the state table using the IP addresses, ports and connection IDs. If no match is found, but a connection with the same IP addresses and ports can be found, the connection IDs of the connection are updated. Finally, if no existing connection is found, a new state table entry is added.

The parsing process for short headers, shown in Figure 4.5, is more complex due to the absence of explicit connection ID lengths and the source connection ID. Short headers are used for regular traffic in established connections or during connection migrations. To identify the corresponding connection, the system matches packets against the state table using IPs and ports, then verifies that the packet payload starts with the connection ID that is stored in the state table. If no match is found, the packet is flagged as either invalid or as part of a connection migration.

When the packet is not part of an existing connection, it might be part of a connection migration. This process begins with a `PATH_CHALLENGE`, where a full-size packet (typically 1350 bytes of payload, or 1392 bytes including headers) is sent with a new destination connection ID. In a client-side connection migration, the client may use a new source IP and port, while in a server-side connection migration, the server may use a new destination IP and port. The server responds with a `PATH_RESPONSE`, also a full-size packet, containing the same QUIC payload, but using a new destination connection ID (source connection ID from the point of the firewall). When using Cloudflare's quiche [60], a final packet is sent using the old source connection ID but already using the new IPs and ports. This last packet does not need to be full-size.

Again, checking for these scenarios is done in reverse order. The system first determines whether the packet is full-size. If the packet is not full-size, it might be a final packet. In this case, the firewall checks for two existing connections. First, it verifies whether

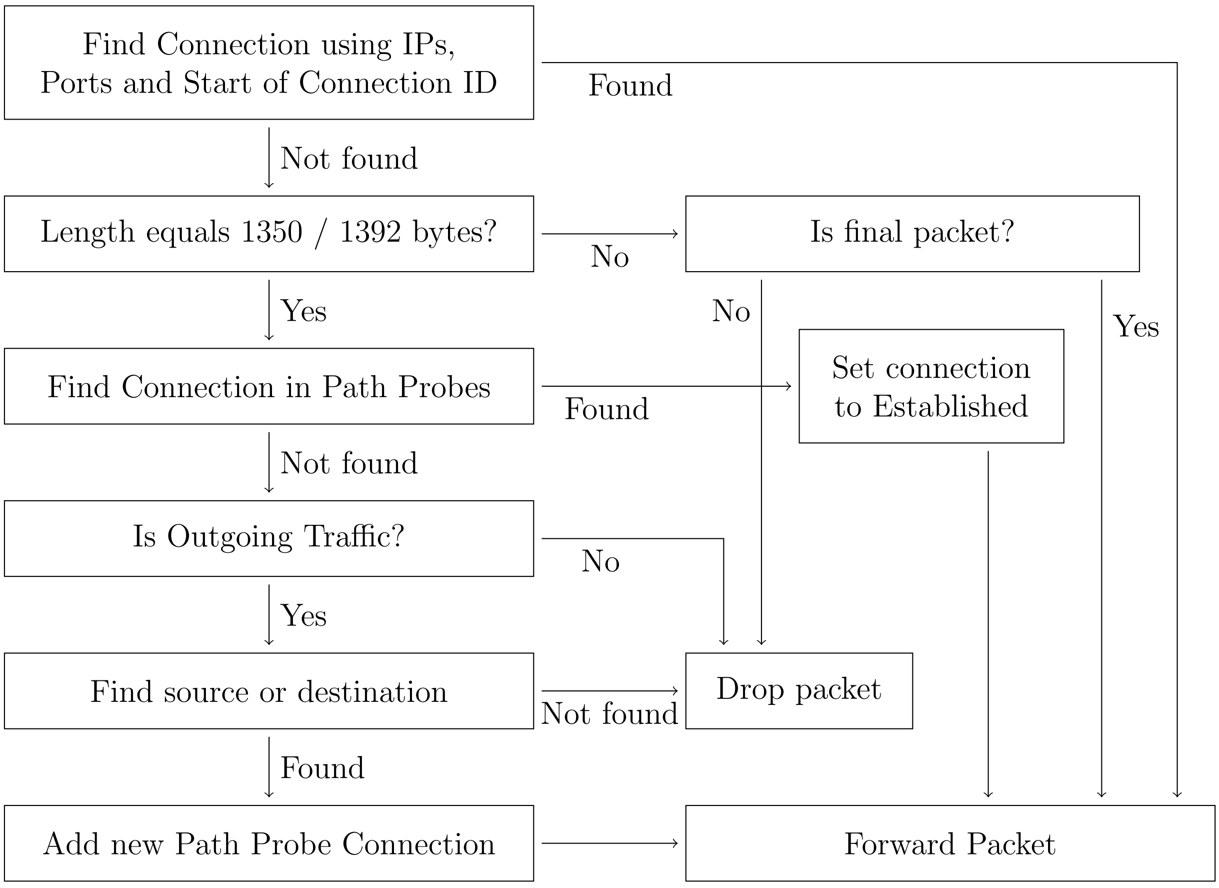

Find Connection using IPs, Ports and Start of Connection ID
Found
Not found
Length equals 1350 / 1392 bytes?
No
Is final packet?
Yes
No
Yes
Find Connection in Path Probes
Found
Set connection to Established
Not found
Is Outgoing Traffic?
No
Yes
Find source or destination
Not found
Drop packet
Found
Add new Path Probe Connection
Forward Packet

Figure 4.5: Short Header Parsing

a connection exists based on the packet's IPs and ports. This connection should have seen only a limited number of packets, as this final packet is sent immediately after the connection migration. Second, the firewall checks for a connection with the same source connection ID, corresponding to the previous connection. If both matches are found, the packet is forwarded; otherwise, it is dropped.

For full-size packets, incoming traffic could be either a `PATH_CHALLENGE` or `PATH_RESPONSE`. First, the firewall checks if the packet could be a `PATH_RESPONSE`. This is the case if the traffic is `Incoming`, meaning the sender is an external server, and a corresponding connection with a state of `PathProbe` can be found in the state table using the packet's IPs and ports. If such a match exists, the firewall updates the state table by adding the destination connection ID from the `PATH_RESPONSE` as the source connection ID and changes the state of the connection to `Established`.

If the packet is `Outgoing`, it is likely a `PATH_CHALLENGE`. In this case, the firewall adds a new entry to the state table. Because a `PATH_CHALLENGE` is using a short header, it does not include the source connection ID or the length of the destination connection ID. The firewall therefore leaves the source connection ID blank and fills it when the `PATH_RESPONSE` arrives. For the destination connection ID, the firewall makes an assumption about its value and stores it in the state table. Since QUIC version 1 and most implementations use a connection ID length of up to 20 bytes, the firewall relies on this length to determine and extract the connection ID. Over the next few packets, the firewall verifies this connection ID by comparing it with subsequent packets to ensure a complete or partial match.

Before adding the new state table entry, the firewall checks for existing connections using either the source IP and port (for server-side migrations) or the destination IP and port (for client-side migrations). If no match is found, meaning neither the source nor the destination is already present in the state table, the migration is considered invalid, and the packet is dropped.

4.4.5 Detecting Suspicious Connection Migrations

Detecting suspicious connection migrations in the firewall setup relies on a combination of real-time and asynchronous checks to ensure security without compromising system performance. The design allows the identification and response to potential misuse of QUIC's connection migration feature.

Client-side connection migrations involve changes to the source IP and port while maintaining the same destination IP and port. For such cases, the state table is checked to ensure there is at least one active connection matching the destination IP and port. If no such connection exists, the migration is flagged as invalid and blocked. Similarly, server-side migrations, where the source IP and port remain constant but the destination changes, require the state table to have an entry matching the source IP and port. Failing this, the migration is deemed invalid. These checks are computationally lightweight and are executed in real-time whenever a connection migration occurs.

More resource-intensive checks are performed asynchronously to verify the validity of migrations further. When potential matches for a connection migration are found in the state table, the identifiers of the new connection, along with those of the possible matches, are sent to a separate thread for deeper validation. The associated packet is forwarded immediately, allowing for some initial data exchange before the migration is finally validated. This trade-off had to be done in order to ensure performance.

As server-side migrations are the primary attack vector in this context, additional asynchronous checks are focused on these scenarios to enable deeper validation. Public IP ranges from major cloud providers like Microsoft [78], Google [76], and Amazon [77] are verified to determine whether the destination IP from the original connection and any potential match fall within the same range. A connection migration originating from an IP in one provider's range is only allowed to migrate to another IP within the same range. To prevent attackers from exploiting this by creating servers in the respective cloud provider services such as GCP [76] or Azure [79], public IPs associated with these cloud services are treated as separate entities, distinct from those associated with their general services such as google.com and microsoft.com.

If no match is found in the cloud provider IP checks, the firewall resorts to a whois lookup to verify whether the same organization is responsible for the IP addresses in question. If the lookup reveals a mismatch in ownership, the connection migration is blocked.

There are limitations to this design. Connection migrations that bypass the firewall entirely cannot be detected. For instance, client-side migrations from an internal to an external IP, such as when a device switches from a corporate Wi-Fi network to cellular data, are not visible to the firewall. In such cases, the connection times out after two minutes, similar to other stale connections. Client-side migrations from an external to an internal IP are outright blocked by the firewall. Any blocked migration that results in a failed path challenge prompts QUIC to initiate a new connection using a full handshake, providing a secure fallback mechanism.

4.4.6 Rebuilding and Sending Packet

Rebuilding and sending packets is the final step in handling network traffic after parsing and processing. This is where the second part of SNAT occurs.

For outgoing traffic, the original source IP and port of the packet, representing the internal client, are replaced with the firewall's external IP address and a unique port assigned by the firewall. This transformation ensures that external recipients see the firewall as the source of the traffic, not the internal client. The destination IP and port remain unchanged, as these represent the intended target outside the firewall.

Incoming traffic, on the other hand, requires reverse translation to direct the packet to the correct internal client. The source IP and port, representing the external sender, are left untouched. However, the destination IP and port, which currently reflect the firewall's address, are replaced with the original client's IP and port. This ensures that the packet reaches its intended recipient within the internal network.

Throughout this process, the integrity of the UDP payload, including any QUIC headers, remains intact. The payload is attached to the rebuilt packet without alteration to preserve all application-layer information.

The final stage involves transmitting the rebuilt packet using a transport channel. This channel serves as the conduit for sending packets back into the network after they have been processed. Once the packet is sent, the system prepares for the next packet in the traffic stream, ensuring continuous and efficient handling of network communication. The focus on maintaining both security and functionality underpins the design of this packet rebuilding and forwarding process.

4.4.7 iptables

iptables [63] is a utility in Linux that allows administrators to configure and manage the system's built-in packet filtering framework, Netfilter. It is used to define rules for network traffic, specifying what packets should be allowed, blocked, or modified. iptables can forward packets and can be used to build simple firewalls or gateways. It can filter packets based on attributes such as source or destination IP address, port numbers, or protocols.

In this project, iptables was used to forward DNS, ICMP, and NTP traffic, allowing the application to focus only on QUIC traffic. DNS is used to resolve domain names to IP addresses, ICMP facilitates network diagnostics such as ping requests, and NTP ensures time synchronization so that Wireshark logs remain consistent.

Chapter 5

Implementation

This chapter explains the technical details of the implementation. It covers the project's structure, the Rust crates used, and key components such as the `Packet` struct and state table `Connection` struct. The following sections describe the same steps a packet takes as it passes through the firewall, as outlined in Chapter 4. The project's source code is available on Github [80], and an installation guide for setting up the project can be found in Appendix A.

5.1 Overview

The project setup consists of three machines: a server, a client, and a firewall. These machines are implemented as virtual machines running in Hyper-V on a Windows Pro host. The network configuration uses two virtual switches. One switch allows communication with the internet, while the other is a private network restricted to communication between the virtual machines.

Each machine is configured with two network interfaces. The firewall connects the two networks, acting as a gateway between the client and the server. This setup allows the firewall to inspect, forward, and block packets as needed. The client and server both have two network interfaces in order to simulate connection migrations by switching between those interfaces during operation.

The firewall was written in Rust. Rust was chosen for its strong emphasis on performance and memory safety, as well as its ability to handle low-level system operations efficiently. This combination makes Rust particularly suitable for implementing a network firewall that requires high reliability and robustness.

5.1.1 Project Structure

As visible in Figure 5.1, the Git repository is structured into four main components: the firewall, iptables, quiche, and Traffic Faker. Each component plays a specific role in the project.

```
quicfirewall/
├── firewall/
│   ├── ip_ranges/
│   │   ├── cleaned/
│   │   │   └── ...
│   │   └── original/
│   │       └── ...
│   ├── src/
│   │   ├── config.rs
│   │   ├── ipranges.rs
│   │   ├── listen.rs
│   │   ├── main.rs
│   │   ├── packet.rs
│   │   ├── serializer.rs
│   │   ├── statetable.rs
│   │   └── whois.rs
│   └── Cargo.toml
├── iptables/
│   └── firewall.sh
├── quiche/
│   └── ...
└── trafficfaker/
    ├── src/
    │   ├── client.rs
    │   ├── config.rs
    │   ├── main.rs
    │   └── server.rs
    └── Cargo.toml
```

Figure 5.1: Github Project Directory

The firewall contains all the code related to packet filtering and connection tracking, forming the core of this project. It is responsible for enforcing network rules and handling QUIC connection migrations.

The iptables folder includes a Bash script that configures iptables to manage specific types of traffic, such as DNS, ICMP, and NTP. These protocols are handled directly by iptables rather than the firewall application. More details on this configuration can be found in Chapter 5.3.

The quiche component includes a modified version of Cloudflare's Quiche library [60]. The original Quiche repository is used as a base, but the example implementation has been modified to support client-side connection migrations involving changes to the source IP. This is further explained in Chapter 5.4.

Traffic Faker is a tool designed to generate test traffic to evaluate the firewall's performance. It can operate in both client and server modes, allowing controlled traffic patterns to be simulated for testing purposes. Further details on Traffic Faker and its role in testing are provided in Chapter 5.5.

5.1.2 Rust Crates

The following Rust crates were used to build the firewall, each providing specific functionalities that contribute to its overall usability and performance. The key crates handle packet capture and parsing, while others provide utilities for data processing and user input handling.

- **pcap:** This crate provides a Rust interface to the libpcap library, enabling the capture and inspection of network packets directly from the network interface. It allows filtering of packets and offers access to raw packet data for detailed analysis [65].
- **etherparse:** A packet parsing library that allows for decoding of Ethernet, IP, and UDP Headers. It simplifies packet inspection and modification, making it easier to work with structured network data [66].
- **pnet:** A cross-platform network programming library in Rust that enables low-level packet manipulation and transmission. It complements pcap and etherparse by providing tools for creating, sending, and receiving custom packets [67].
- **ipnetwork:** Provides utilities for handling IP addresses and network prefixes, including parsing, validation, and manipulation of CIDR notations [68].
- **itertools:** Extends Rust's iterator functionality with additional combinators, methods, and features, streamlining complex data processing tasks [69].
- **regex:** A crate for compiling and using regular expressions, enabling efficient pattern matching and text searching capabilities [70].
- **serde:** A serialization and deserialization framework used to convert data structures to and from various formats like JSON [71].
- **serde_json:** A specific implementation of the Serde framework for working with JSON data, supporting parsing, serialization, and deserialization of JSON strings [72].
- **clap:** A command-line argument parser that facilitates the creation of user-friendly CLI interfaces, supporting argument validation and help message generation [73].
- **rand:** A random number generation library that provides functionality for generating random values and sampling from distributions [74].
- **ctrlc:** A utility for handling interrupt signals such as Ctrl+C, enabling graceful shutdown of the application when terminated by the user [75].

5.2 Firewall

The following sections provide a detailed description of the firewall. To improve clarity, some code snippets have been reordered and simplified for better visualization. The full source code is available on GitHub [80].

5.2.1 Structs

The firewall is built around two primary Rust structs, depicted in Figure 5.2: `Packet` and `Connection`. The state table holds multiple `Connection` instances.

5.2.2 Packet

The `Packet` struct contains several fields that store information for each captured packet. These fields are used to uniquely identify, classify, and process packets within the firewall. Below is a breakdown of the key fields and their purpose:

- **id:** An atomic, thread-independent, unique incrementing identifier. This ensures that each packet has a distinct ID across all threads.
- **ts:** A timestamp indicating when the packet was captured. It is stored as `[seconds, microseconds]` since the Unix epoch.
- **len:** The total length of the packet in bytes. This represents the complete size of the packet. It is distinct from caplen, which is the length of the captured portion of the packet available in the buffer. The pcap crate, by default, captures up to 65535 bytes. For typical QUIC traffic, which is often around 1392 bytes, this limitation is not significant.
- **src_ip:** The source IP address, representing the sender's IPv4 address. To simplify the design, only IPv4 is supported. For outgoing traffic, this is the client's actual IP address before SNAT is applied.
- **dst_ip:** The destination IP address, representing the receiver's IPv4 address. Similar to `src_ip`, only IPv4 is supported. For incoming traffic, this is the receiver's actual IP address after SNAT is applied.
- **src_port:** The source port number, representing the sender's port. For outgoing traffic, this is the port before SNAT is applied.
- **dst_port:** The destination port number, representing the port to which the packet is sent. For incoming traffic, this is the port after SNAT is applied.
- **src_conn_id:** The QUIC connection ID of the sender, used to identify the connection on the sender's side.

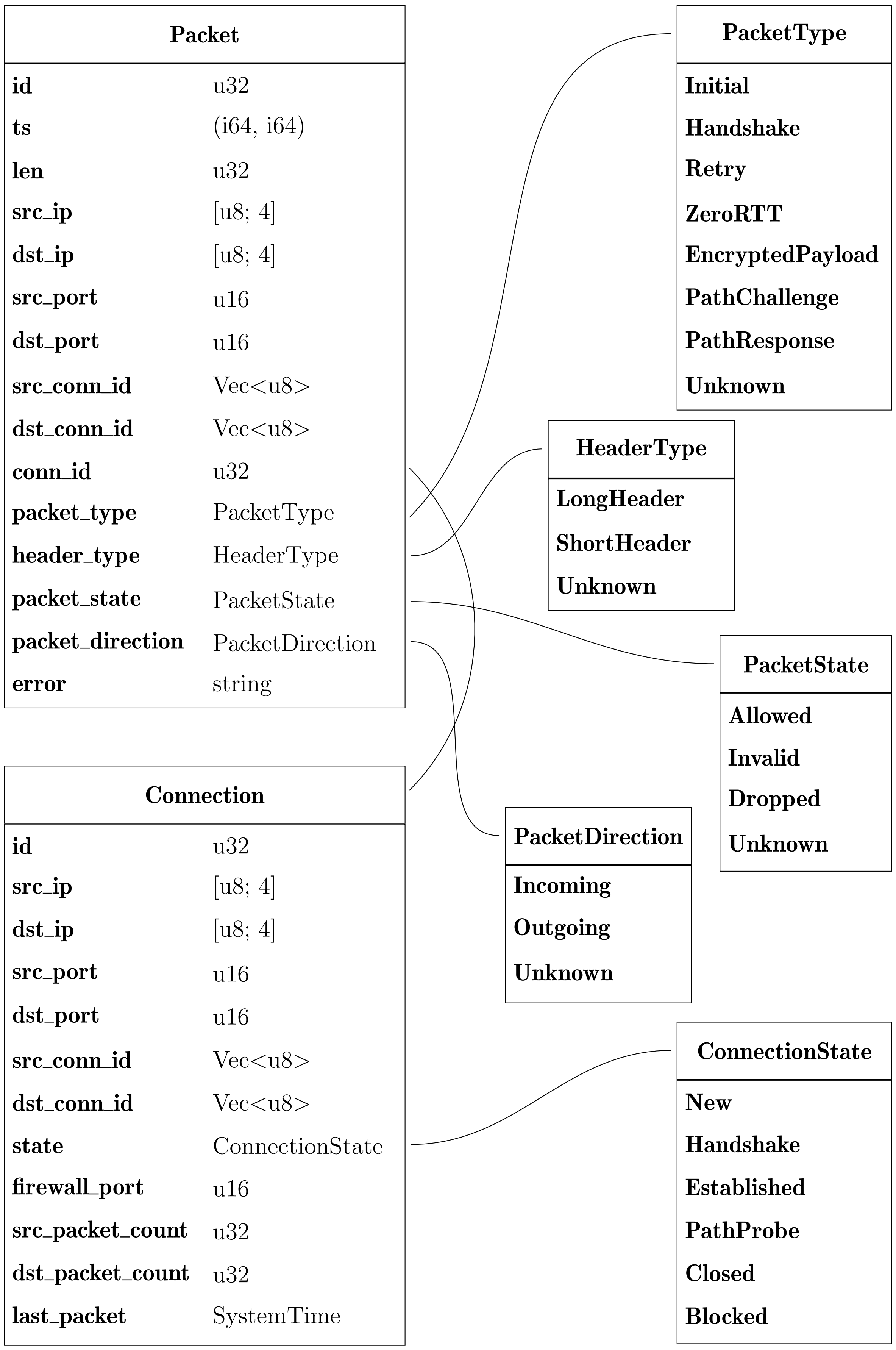


Figure 5.2: Firewall Structs

- **dst_conn_id:** The QUIC connection ID of the receiver, used to identify the connection on the receiver's side.
- **conn_id:** The ID associated with the respective connection. This serves as a foreign key, establishing a one-to-many relationship where a connection can contain multiple packets.
- **packet_type:** An enum representing the packet type as defined in the RFC. Possible values are `Initial`, `Handshake`, `Retry`, `ZeroRTT`, `EncryptedPayload` as well as `PathChallenge` and `PathResponse`. An `Unknown` state is used as a placeholder during parsing.
- **header_type:** An enum indicating whether the packet's header is a `LongHeader` or a `ShortHeader`. An `Unknown` state is used as a placeholder during parsing.
- **packet_state:** An enum specifying the packet's state, which can be `Allowed`, `Invalid`, or `Dropped`. An `Unknown` state is used as a placeholder during parsing.
- **packet_direction:** An enum indicating the packet's direction. It can be `Outgoing`, meaning it travels from the internal network to the external one, or `Incoming`, meaning it travels from the external network to the internal one. An `Unknown` state is used when listening on only one interface.
- **error:** A field capturing any errors associated with invalid or dropped packets. This provides diagnostic information for troubleshooting or analysis.

5.2.3 State Table

The state table consist of multiple `Connection` instances, each representing a connection tracked by the firewall. Below is an overview of the fields in the `Connection` struct:

- **id:** An atomic, thread-independent, unique incrementing identifier. This ensures that each packet has a distinct ID across all threads.
- **src_ip:** The source IP address, representing the internal client's IPv4 address. Unlike in the packet structure, `src_ip` here always refers to he client's IP address within the internal network.
- **dst_ip:** The destination IP address, representing the external server's IPv4 address. Similar to `src_ip`, `dst_ip` always refers to the server's IP address in the external network.
- **src_port:** The source port number, representing the internal clients's port.
- **dst_port:** The destination port number, representing the external servers' port.
- **src_conn id:** The QUIC connection ID of the internal client, used to identify the connection on the client's side.

- **dst_conn id:** The QUIC connection ID of the external server, used to identify the connection on the server's side.
- **state:** An enum indicating the state of a connection. Possible states include `New`, `Handshake`, `Established`, `PathProbe`, `Closed`, and `Blocked`.
- **firewall port:** The unique port assigned by the firewall to handle SNAT for outgoing packets. It is typically the same as `src_port`, but if that port already exists, a new port is selected.
- **src_packet_count:** The number of packets sent by the internal client. This counter is used to determine whether the `src_conn_id` can be modified, particularly after a migration when the new connection IDs are not fully known yet. The `src_conn_id` will only be modified if fewer than 5 packets have been sent by the client.
- **dst_packet_count:** The number of packets sent by the external server. This counter is used to determine whether the `dst_conn_id` can be modified, particularly after a migration when the new connection IDs are not fully known yet. The `dst_conn_id` will only be modified if fewer than 5 packets have been sent by the server.
- **last_packet:** The timestamp of the last packet sent in the connection. This is used to close stale connections

5.2.4 Capturing Packets

The firewall operates with two network interfaces, each handled by a dedicated thread. These threads listen for incoming packets using the `pcap` crate. The relevant code is shown in Listing 5.1.

First, the firewall retrieves a list of all available network interfaces and filters them to identify the one it should monitor. It then extracts the IP address of these interfaces, which is later used for SNAT. The firewall then creates a capture instance and opens it.

As described in Chapter 4.4.1, traffic direction is filtered using `Direction::In`, ensuring that only incoming packets are captured. Outgoing packets, which have already been processed and forwarded, are not captured again. This prevents the firewall from processing its own outbound traffic.

Additionally, a filter is applied to capture only QUIC traffic. This is achieved by selecting packets with either a destination UDP port of 443 (for outgoing QUIC packets) or a source UDP port of 443 (for incoming QUIC responses).

The firewall then enters an infinite loop, where `cap.next_packet()` blocks the thread until a packet is captured. Once a packet is received, it is checked for validity before processing. At this stage, the packet is simply an array of bytes, a `[u8]` in Rust, along with a pcap header containing the packet length and a timestamp indicating when it was captured.

```rust
# main.rs

use pcap::{Capture, Device, Direction};

let devices = Device::list().unwrap();
let device = devices.into_iter().find(|d| d.name == interface).unwrap();
let firewall_ip = get_device_ip(device.clone());

let mut cap = Capture::from_device(device).unwrap()
    .immediate_mode(true)
    .timeout(10)
    .open().unwrap();

cap.direction(Direction::In);
cap.filter("udp dst port 443 or udp src port 443", true).unwrap();

while true {
    if let Some(packet) = cap.next_packet().ok(){
        ...
    }
}
```

Listing 5.1: Initializing pcap Capture

5.2.5 Parsing of IP and UDP Headers

The next step in processing captured packets is parsing the IP and UDP headers, visible in Listing 5.2. This begins with the creation of an "empty" packet using the `Packet` struct. At this stage, a unique ID is assigned to the packet by fetching an atomic ID, ensuring uniqueness across both threads.

The firewall then uses the `etherparse` crate to parse the IP and UDP headers. The function `from_ethernet_slice()` is called to decode the packet into different headers. If successful, it returns a struct containing network headers (such as IP headers), transport headers (such as UDP headers), and the remaining payload. Rust's `match` statements are used to safely handle unexpected values. Since `from_ethernet_slice()` returns a `Result` type, it can either be `Ok(v)` (indicating successful parsing) or `Err(e)` (indicating an error). In the success case, the parsed value `v` is extracted and assigned to the variable `parsed`. If an error occurs, the packet is marked as `Invalid`, an error message is written into the `error` field, and the function exits.

Next, the firewall extracts the IP header from the `parsed` struct. Again, `match` statements are used, but in this case, they handle an `Option` type, which can either be `Some(v)` (indicating that an IP header is present) or `None` (indicating that no IP header was found). If the packet contains a `NetHeaders::Ipv4`, the firewall updates the `Packet` struct with the source and destination IP addresses. If the header is of a different type, such as `NetHeaders::Ipv6`, or if no IP header is present, the packet is marked as `Invalid`, an error message is logged, and the function exits.

A similar process is used to extract the UDP headers. The firewall ensures that the transport header is of type `TransportHeader::Udp`, and if so, it assigns the source and destination ports to the `Packet` struct. If a different transport type, such as `TransportHeader::Tcp`, or `None` is found, the packet is marked as `Invalid`, and the function exits.

```
# packet.rs

use etherparse::{PacketHeaders, NetHeaders, TransportHeader};
static PACKET_ID: AtomicU32 = AtomicU32::new(1);

let id = PACKET_ID.fetch_add(1, Ordering::SeqCst);

let mut packet = Packet {
    id,
    ts: (header.ts.tv_sec.into(), header.ts.tv_usec.into()),
    len: header.len,
    src_ip: [0; 4],
    dst_ip: [0; 4],
    src_port: 0,
    dst_port: 0,
    src_conn_id: Vec::new(),
    dst_conn_id: Vec::new(),
    conn_id: 0,
    packet_type: PacketType::Unknown,
    header_type: HeaderType::Unknown,
    packet_state: PacketState::Unknown,
    packet_direction,
    error: String::new()
};

let parsed = match PacketHeaders::from_ethernet_slice(&data){
    Ok(v) => v,
    _ => {
        packet.packet_state = PacketState::Invalid;
        packet.error = "Cannot Parse Packet".to_string();
        return (packet, None, Vec::new());
    }
};

match &parsed.net {
    Some(NetHeaders::Ipv4(ip, _)) => {
        packet.src_ip = ip.source;
        packet.dst_ip = ip.destination;
    },
    _ => {
        packet.packet_state = PacketState::Invalid;
        packet.error = "Missing IP Header".to_string();
        return (packet, None, Vec::new());
    }
}

match &parsed.transport {
    Some(TransportHeader::Udp(udp)) => {
        packet.src_port = udp.source_port;
        packet.dst_port = udp.destination_port;
    },
    _ => {
        packet.packet_state = PacketState::Invalid;
        packet.error = "Missing UDP Header".to_string();
        return (packet, None, Vec::new());
    }
}
```

Listing 5.2: Parsing IP and UDP Headers

5.2.6 Source NAT

The next step is Source Network Address Translation (SNAT), outlined in Listing 5.3. As described in Chapter 4.4.3, this step applies only to incoming traffic, not outgoing. After parsing the IP and UDP headers, the destination IP and port of an incoming packet are initially set to the firewall's public IP and a unique firewall port assigned by the firewall when the connection was added to the state table.

To determine the actual destination, the firewall looks up the connection in the state table using the destination port. This lookup only considers active connections, ignoring `Closed` or `Blocked` ones. In the code, the function returns the source IP and source port from the state table. This makes sense because, from the firewall's perspective, the source IP refers to the internal client. However, from the packet's perspective, this internal client is actually the destination. Therefore, the firewall updates the packet's destination IP and port with the corresponding source IP and port retrieved from the state table.

For outgoing traffic, where no SNAT is required, the `source_nat` function simply returns the same source IP and port without modification.

```
# packet.rs

match Self::source_nat(&packet, &state_table, firewall_ip) {
    Ok((dst_ip, dst_port)) => {
        packet.dst_ip = dst_ip;
        packet.dst_port = dst_port;
    },
    _ => {
        packet.packet_state = PacketState::Invalid;
        packet.error = "Destination not found".to_string();
        return (packet, None, Vec::new());
    }
}

fn source_nat(packet: &Packet, st: &StateTable, firewall_ip: [u8; 4])
    -> Result<([u8; 4], u16), String>{
        if packet.dst_ip == ip {
            match st.find_conn_by_firewall_port(packet.dst_port){
                Ok((src_ip, src_port)) => return Ok((src_ip, src_port)),
                Err(e) => return Err(e)
            }
        } else {
            return Ok((packet.dst_ip, packet.dst_port));
        }
    }
}

# statetable.rs

fn find_conn_by_firewall_port(&self, dst_port: u16)
    -> Result <([u8; 4], u16), String>{
    for conn in self.connections.values(){
        if conn.state != ConnectionState::Closed
        && conn.state != ConnectionState::Blocked
        && conn.firewall_port == dst_port {
            return Ok((conn.src_ip, conn.src_port));
        }
    }
    return Err("Destination not found".to_string());
}
```

Listing 5.3: Source Network Address Translation

5.2.7 Parsing of QUIC Headers

Long Header

The QUIC headers are parsed next, depicted in Listing 5.4. A QUIC header is classified as a long header if the first bit is set to 1. This can be determined by checking whether the first byte is greater than 127. If the value exceeds this threshold, the packet contains a long header; otherwise, it is a short header.

The packet type is then extracted. As described in Chapter 2.1.2, bits 3 and 4 of the first byte indicate whether a packet is of type `Initial`, `ZeroRTT`, `Handshake` or `Retry`. The firewall uses an enum called `PacketType` to represent these variants. The next four bytes contain the QUIC version. Since QUIC Version 2 uses different packet types, the firewall uses the version to determine the specific packet type.

The fifth byte specifies the length of the destination connection ID. The firewall then reads the destination connection ID using as many bytes as indicated by this length. The following byte defines the length of the source connection ID, and the firewall extracts the corresponding number of bytes as the source connection ID.

The `Packet` struct is then populated with the extracted connection IDs, along with the `PacketType` and `HeaderType`.

```
# packet.rs

if payload[0] > 127 {
    let type_bits = (payload[0] & 0b0011_0000) >> 4;
    let version = &payload[1..5];
    let packet_type = match version {
        [170, 51, 67, 207] => match type_bits {
            0b00 => PacketType::Retry,
                0b01 => PacketType::Initial,
                0b10 => PacketType::ZeroRTT,
                0b11 => PacketType::Handshake
            },
            _ => match type_bits {
                0b00 => PacketType::Initial,
                0b01 => PacketType::ZeroRTT,
                0b10 => PacketType::Handshake,
                0b11 => PacketType::Retry
            }
        };

        let dst_conn_id_len = payload[5] as usize;
        let dst_conn_id = &payload[6..6 + dst_conn_id_len];

        let src_conn_id_start = 6 + dst_conn_id_len;
        let src_conn_id_len = payload[src_conn_id_start] as usize;
        let src_conn_id = &payload[src_conn_id_start + 1..src_conn_id_start
            + 1 + src_conn_id_len];

        packet.src_conn_id = src_conn_id.to_vec();
        packet.dst_conn_id = dst_conn_id.to_vec();
        packet.packet_type = packet_type;
        packet.header_type = HeaderType::LongHeader;

        if let Ok(conn) = state_table.find_conn_by_ip_port_and_id(&packet){
            conn.state = ConnectionState::Handshake;
            conn.last_packet = SystemTime::now();
            return Ok((packet.clone(), conn.clone()));
        }

        if let Ok(conn) = state_table.find_conn_by_ip_and_port(&packet){
            conn.state = ConnectionState::Handshake;
            conn.last_packet = SystemTime::now();
            return Ok((packet.clone(), conn.clone()));
        }

        return match state_table.add_connection(
            &packet,
            ConnectionState::New
        ){
            Ok(conn) => {
                return Ok((packet.clone(), conn.clone()));
            },
            _ => Err("Cannot parse QUIC Headers".to_string())
        }
}
```

Listing 5.4: Long Header Parsing

As described in Chapter 4.4.4, packets with long headers can fall into three scenarios. The first scenario involves the packet being the first packet of a new connection, which requires a new entry in the state table. The second scenario occurs when new connection IDs are introduced, such as during a `Retry`. In this case, the IPs and ports can be found in the state table, but the connection IDs differ. The final scenario involves regular packets that are part of an existing handshake, meaning the IPs, ports, and connection IDs should already be found in the state table.

The firewall checks these scenarios in reverse order. The first check is to see whether a connection can be found in the state table using the IPs, ports, and connection IDs. If such a connection is found, the firewall updates the connection's state to `Handshake`. The timestamp of the last observed packet for this connection is also updated, which is needed for removing stale connections (as described in Section 5.2.10).

If no connection is found with the IPs, ports, and connection IDs, the firewall performs a second check using only the IPs and ports. If a connection is found based on them, the function `find_conn_by_ip_and_port` modifies the connection IDs accordingly. The update behavior depends on whether the packet is incoming or outgoing. For outgoing traffic, the packet's source connection ID will be the source connection ID stored in the state table. For incoming traffic, the packet's source connection ID is set to the destination connection ID in the state table.

If no matching connection is found, the firewall adds a new connection to the state table, presented in Listing 5.5. New connections are only allowed to be added for `Outgoing` traffic and `Initial` packets.

Similar to the packet IDs, a unique connection ID is obtained from an atomic counter to ensure uniqueness across threads. The firewall assigns a unique firewall port to the connection and ensures that no other existing connection is already using the same port. If the port is already in use, the firewall increments the port number until an unused one is found. Finally, the new connection is added to the state table.

```
# statetable.rs

static CONNECTION_ID: AtomicU32 = AtomicU32::new(1);

fn get_firewall_port(&mut self, mut port: u16) -> u16 {
    while self.connections.values().any(|conn| conn.firewall_port ==
    port){
        port += 1;
    }
    port
}

fn add_connection(&mut self, packet: &Packet, state: ConnectionState)
    -> Result<Connection, &str> {
    if packet.packet_direction == PacketDirection::Incoming
    || packet.packet_type != PacketType::Initial {
        return Err("State table entry creation not allowed");
    }
    let id = CONNECTION_ID.fetch_add(1, Ordering::SeqCst);
    let firewall_port = self.get_firewall_port(packet.src_port);
    let connection = Connection {
        id,
        src_ip: packet.src_ip,
        dst_ip: packet.dst_ip,
        src_port: packet.src_port,
        dst_port: packet.dst_port,
        src_conn_id: packet.src_conn_id.clone(),
        dst_conn_id: packet.dst_conn_id.clone(),
        state,
        firewall_port,
        src_packet_count: 1,
        dst_packet_count: 0,
        last_packet: SystemTime::now()
    };
    self.connections.insert(id, connection.clone());
    return Ok(connection)
}
```

Listing 5.5: Adding new State Table Connection

Short Header

For short headers, multiple scenarios exist, as described in Chapter 4.4.4 and shown in Listing 5.6. A packet can either be part of regular traffic or involved in a connection migration. The firewall first attempts to find a matching connection using the IPs, ports, and the beginning of the payload as either the source or destination connection ID. Since the length of the connection ID is not known for short headers, the firewall only determines its position at the start of the payload. If a matching connection is found, the connection state is updated to `Established`, and the `last_packet` field is updated, following the same approach as with long headers. At this point, both the source and destination connection IDs can be accurately determined by referencing the state table.

If the packet does not belong to an existing connection, it may be related to a connection migration. In such cases, the packet can represent one of three types: `PATH_CHALLENGE`, `PATH_RESPONSE`, or a final packet.

A `PATH_CHALLENGE` and a `PATH_RESPONSE` are always full-size packets, each containing 1392 bytes. The firewall first verifies whether the packet length matches this size. If the packet is smaller, it may still be a final packet. To confirm this, the firewall searches for two matching connections, one where the IPs and ports match and another where the destination connection ID corresponds to a source connection ID in the state table. If both connections can be found, the packet is identified as a final packet.

If the packet is 1392 bytes long and is `Outgoing` traffic, it may be a `PATH_CHALLENGE`. In this case, a new entry is added to the state table with a connection state of `PathProbe`. Since the short header does not include the source connection ID, only the destination connection ID is set in the state table entry, while the source connection ID remains unset.

For `Incoming` traffic, the packet may be a `PATH_RESPONSE`. The firewall checks whether a matching connection exists in the state table with a state of `PathProbe`. Since this is the second packet in the connection migration sequence and the first `Incoming` packet, the firewall verifies this by checking the `src_packet_count` and `dst_packet_count` fields. If a match is found, the connection state is updated to `Established`. Because this is `Incoming` traffic, the destination connection ID in the short header corresponds to the source connection ID in the state table. When the state table entry was initially created, the source connection ID was left blank, but at this stage, it can be properly assigned.

As with previous checks, the firewall first evaluates whether the packet is a PATH_RESPONSE before checking for a PATH_CHALLENGE.

```
# packet.rs

if payload[0] < 127 {
    let conn_id_slice = if payload.len() > MAX_CONN_ID_LENGTH {
        &payload[1..1 + MAX_CONN_ID_LENGTH]
    } else {
        &payload[1..]
    };

    packet.packet_type = PacketType::EncryptedPayload;
    packet.header_type = HeaderType::ShortHeader;

    if let Ok(conn) = state_table.find_conn_by_partial_id(
        &packet,
        conn_id_slice
    ){
        conn.state = ConnectionState::Established;
        conn.last_packet = SystemTime::now();
        packet.src_conn_id = conn.src_conn_id.to_vec();
        packet.dst_conn_id = conn.dst_conn_id.to_vec();
        return Ok((packet.clone(), conn.clone()));
    }

    if packet.len != 1392 {
        if let Ok(conn) = state_table.find_conn_by_final_packet(
            &packet,
            conn_id_slice,
            queue
        ){
            conn.state = ConnectionState::Established;
            conn.last_packet = SystemTime::now();
            if packet.packet_direction == PacketDirection::Incoming {
                packet.src_conn_id = conn.dst_conn_id.to_vec();
                packet.dst_conn_id = conn.src_conn_id.to_vec();
            } else {
                packet.src_conn_id = conn.src_conn_id.to_vec();
                packet.dst_conn_id = conn.dst_conn_id.to_vec();
            }
            return Ok((packet.clone(), conn.clone()));
        }
        return Err("Packet not found and no Path Probe".to_string());
    }

    if let Ok(conn) = state_table.find_conn_in_path_probes(&packet){
        conn.state = ConnectionState::Established;
        conn.last_packet = SystemTime::now();
        conn.src_conn_id = conn_id_slice.to_vec();
        packet.src_conn_id = conn_id_slice.to_vec();
        packet.dst_conn_id = conn.dst_conn_id.to_vec();
        packet.packet_type = PacketType::PathResponse;
        return Ok((packet.clone(), conn.clone()));
    }

    let matches = match state_table.find_src_or_dst(&packet){
        Ok(matches) => matches,
        _ => return Err("Suspicious Connection Migration".to_string())
    };

    packet.dst_conn_id = conn_id_slice.to_vec();
    packet.packet_type = PacketType::PathChallenge;

    return match state_table.add_connection(
        &packet,
        ConnectionState::PathProbe
    ){
        Ok(conn) => {
            if matches.len() > 0 {
                let mut queue = queue.lock().unwrap();
                queue.push_back((conn.id, matches));
            }
            return Ok((packet.clone(), conn.clone()));
        },
        _ => Err("Cannot parse QUIC Headers".to_string())
    }
}
```

Listing 5.6: Short Header Parsing

5.2.8 Detecting Malicious Connection Migrations

This firewall uses three main methods to detect malicious connection migrations. The first approach is to determine whether the source or destination of the connection already exists in the state table. Another method involves verifying whether the IP addresses are within the public IP ranges of three major cloud providers. If neither check provides conclusive results, a final verification step is performed using a WHOIS lookup to determine if the IPs are managed by the same organization.

Check for Source or Destination Existence

As explained in Chapter 4.4.5, the first check involves checking whether either the source or destination already exists in the state table. In client-side connection migrations, the firewall expects the destination IP and port to be present, while in server-side connection migrations, the source IP and port should be present. This is the only check performed synchronously and is executed immediately before creating a state table entry for a `PATH_CHALLENGE`.

IP Ranges

The next level of validation focuses on server-side connection migrations. Due to their impact on performance, these checks are delegated to a separate thread, ensuring that packet processing continues without delays. The firewall performs these checks at two points: first, when a `PATH_CHALLENGE` packet arrives and a new state table entry is created, and second, when a final packet is detected. The process begins by determining whether the involved IPs fall within the public ranges of cloud providers such as Google, Microsoft, and Amazon. These IP ranges are publicly available in JSON format. They are downloaded and stored in the `ip_ranges/original` directory. On the firewall's first run, these files are parsed using the `serde` crate, converting them into uniform arrays of IP ranges while discarding unnecessary data. The cleaned data is stored in `ip_ranges/cleaned`. They are only regenerated if these files are no longer present. The firewall then compares the previous and new destination IP by checking against all stored IP ranges using the `ipnetwork` crate, shown in Listing 5.7.

There are three main outcomes possible: If both IPs belong to the same cloud provider, the connection migration is considered valid, and no further verification is required. If neither IP is found within the known ranges, the firewall remains uncertain and proceeds to perform a WHOIS lookup. If the IPs belong to different cloud providers or only one IP is present in an IP range, the migration is considered malicious, the connection state is set to `Blocked`, and no further packets from the connection are processed.

```rust
# ipranges.rs

use ipnetwork::IpNetwork;

pub fn compare_ip_ranges(&self, old_ip: [u8; 4], new_ip: [u8; 4])
    -> Option<bool> {
    if old_ip == new_ip {
        return Some(true);
    }
    let old_ip = IpAddr::V4(Ipv4Addr::from(old_ip));
    let new_ip = IpAddr::V4(Ipv4Addr::from(new_ip));

    let mut old_company: Option<String> = None;
    let mut new_company: Option<String> = None;

    for (company, ip_ranges) in &self.ip_ranges {
        for ip_range in ip_ranges {
            if let Ok(network) = ip_range.parse::<IpNetwork>(){
                if network.contains(old_ip){
                    old_company = Some(company.clone());
                }
                if network.contains(new_ip){
                    new_company = Some(company.clone());
                }
            }
        }
    }

    match (old_company, new_company){
        (Some(old_company), Some(new_company)) => {
            if old_company == new_company {
                Some(true)
            } else {
                Some(false)
            }
        },
        (None, None) => None,
        _ => Some(false)
    }
}
```

Listing 5.7: Compare IP Ranges

WhoIS

When the IP range check is inconclusive, the firewall attempts to determine ownership of the IPs through a WHOIS lookup, depicted in Listing 5.8. This process queries `whois.iana.org`, which may redirect to another WHOIS server responsible for the specific IP block. The firewall follows these referrals up to five levels deep, searching for a valid `orgName` or `netname`. If ownership details do not match or the maximum referral depth is reached without a valid result, an error is triggered, and the connection is blocked.

```
# whois.rs

fn compare_whois(src_ip: [u8; 4], dst_ip: [u8; 4]) -> bool{
    if src_ip == dst_ip {
        return true
    }
    let src_result = whois_lookup(src_ip);
    let dst_result = whois_lookup(dst_ip);
    match (src_result, dst_result) {
        (Ok(src_org), Ok(dst_org)) => src_org == dst_org,
        _ => false
    }
}

fn whois_lookup(ip: [u8; 4]) -> Result<String, &str>{
    let mut whois_server = "whois.iana.org".to_string();
    let referral_limit = 5;
    let mut referral_count = 0;

    let ip = ip.iter().join(".");

    loop {
        match query_whois_server(&whois_server, &ip) {
            Some(response) => {
                if let Some(org_name) = extract_org_name(&response){
                    return Ok(org_name);
                }
                if let Some(referral_server) = extract_referral_server(
                    &response
                ){
                    whois_server = referral_server;
                    referral_count += 1;
                    if referral_count >= referral_limit {
                        return Err("Max referral limit reached");
                    }
                } else {
                    return Err("No further referral server found");
                }
            },
            None => {
                return Err("Cannot resolve ip");
            }
        }
    }
}
```

Listing 5.8: WhoIs Lookup

5.2.9 Rebuilding and Sending Packet

After processing, the firewall rebuilds the packet using the `PacketBuilder` from the `etherparse` crate. Depending on whether the packet is part of incoming or outgoing traffic, SNAT is applied accordingly.

For outgoing traffic, shown in Listing 5.9, the source IP is replaced with the `firewall_ip` obtained in Chapter 4.4.6. Additionally, the source port is replaced with the unique firewall port assigned when the connection was first added to the state table.

For incoming traffic, displayed in Listing 5.10, the destination IP and port are replaced with the values stored in the state table, ensuring that the packet is correctly routed to its intended recipient.

```
let builder = PacketBuilder::ipv4(
    firewall_ip,
    connection.dst_ip,
    64
).udp(
    connection.firewall_port,
    connection.dst_port
);
```

Listing 5.9: Outgoing Traffic.

```
let builder = PacketBuilder::ipv4(
    connection.dst_ip,
    connection.src_ip,
    64
).udp(
    connection.dst_port,
    connection.src_port
);
```

Listing 5.10: Incoming Traffic.

To transmit the packet, the firewall utilizes the `pnet` crate, a low-level networking API for Rust. A transport channel is instantiated once and reused across the thread for all packets. The IP and UDP headers built using the `PacketBuilder` are then combined with the remaining UDP payload. Finally, the complete packet is sent through the transport channel.

```
# main.rs

use etherparse::PacketBuilder;
use pnet::packet::{ip::IpNextHeaderProtocols, ipv4::Ipv4Packet};
use pnet::transport::{transport_channel, TransportChannelType::Layer3};

let (mut tx, _) = transport_channel(2048, Layer3(
    IpNextHeaderProtocols::Udp
)).unwrap();

let builder = PacketBuilder::ipv4(
    firewall_ip,
    connection.dst_ip,
    64
).udp(
    connection.firewall_port,
    connection.dst_port
);

let mut result = Vec::<u8>::with_capacity(builder.size(payload.len()));
builder.write(&mut result, &payload).unwrap();

tx.send_to(
    Ipv4Packet::new(&result).unwrap(),
    IpAddr::V4(Ipv4Addr::from(connection.dst_ip))
).unwrap();
```

Listing 5.11: Rebuilding and sending of Packet

5.2.10 Timout Thread

Unlike TCP, UDP connections do not include an explicit connection close signal. Instead, they rely on timeout mechanisms to determine when a connection should be closed. According to the QUIC specification, a timeout of 2 minutes is recommended.

To handle stale connections, the firewall runs a dedicated thread that loops indefinitely, periodically removing expired UDP connections, outlined in Listing 5.12. Every second, the `remove_expired_connection` function is executed. It retrieves the current timestamp and iterates through all active connections, checking whether the time elapsed since the `last_packet` exceeds 120 seconds. If this threshold is met, the connection state is updated to `Closed`.

```
# main.rs

while true {
    thread::sleep(Duration::from_secs(1));
    state_table.remove_expired_connections();
}

# statetable.rs

pub fn remove_expired_connections(&mut self){
    let now = SystemTime::now();
    for conn in self.connections.values_mut(){
        if conn.state != ConnectionState::Closed
        && conn.state != ConnectionState::Blocked {
            if let Ok(duration) = now.duration_since(conn.last_packet){
                    if duration.as_secs() > 120{
                        conn.state = ConnectionState::Closed;
                        println!("Connection closed due to time out");
                    }
                }
            }
        }
}
```

Listing 5.12: Removing Stale Connections

5.3 iptables

To configure iptables, the `iptables/firewall.sh` script shown in Listing 5.13 can be executed. This script requires `root` privileges, as Netfilter typically requires elevated permissions.

The configuration begins by resetting all existing iptables rules. The `-X` option deletes all user-defined chains, leaving only the built-in chains, while `-F` flushes all rules. Because no table is specified using `-t`, these operations are done on the `filter` table. The same process is then applied to the `nat` table.

The next action is to drop all traffic. Iptables is configured with a `default deny` policy, meaning by default, everything is blocked. Certain traffic is later manually allowed.

Next, iptables is then configured to allow the forwarding of DNS, ICMP and NTP traffic. While DNS operates on port 53 over both UDP and TCP, and NTP operates on port 123 over UDP, ICMP does not use ports.

In a standard iptables setup, allowing all `RELATED` and `ESTABLISHED` connections would typically be sufficient, without the need to specify individual protocols and ports. However, this approach does not work in this project because iptables would interfere with the firewall's operations. When a packet is forwarded by the firewall, iptables automatically adds it to its own state table, even though it was not processed by iptables. This results in both iptables and the firewall handling each packet separately, causing packets to be forwarded twice.

Within the `nat` table, SNAT is enabled using `MASQUERADE` in the `POSTROUTING` chain. The `POSTROUTING` chain processes packets after they have been handled by the firewall, and `MASQUERADE` modifies the sender address to the local IP address.

Rules are then defined to specify which traffic is allowed to create new connections. The `-s` option defines the source network, specifying which IPs are allowed to send the packet, while the `-d` option defines the destination network, specifying where the packets are allowed to be sent to. Furthermore, the `-i` option can be used to specify the interface on which the packet was received, while the `-o` option defines the outgoing interface. This ensures that spoofed external packets pretending to be internal ones are not processed.

Finally, IP forwarding is enabled by setting `ip_forward` to 1 in `/proc`. The `/proc` directory is a virtual filesystem that represents the current state of the Linux kernel. Since this change is applied directly to `/proc` and not written into a startup script, it is not persistent and will be lost upon reboot.

```
# firewall.sh
IPTABLES=/sbin/iptables

$IPTABLES -X
$IPTABLES -F
$IPTABLES -t nat -X
$IPTABLES -t nat -F

$IPTABLES -P FORWARD DROP
$IPTABLES -P INPUT DROP
$IPTABLES -P OUTPUT DROP

$IPTABLES -A FORWARD -p tcp --sport 53 -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p udp --sport 53 -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p tcp --dport 53 -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p udp --dport 53 -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p udp --sport 123 -m conntrack
    --ctstate=NRELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p udp --dport 123 -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT
$IPTABLES -A FORWARD -p icmp -m conntrack
    --ctstate=RELATED,ESTABLISHED -j ACCEPT

$IPTABLES -t nat -A POSTROUTING -o $EXT -j MASQUERADE

$IPTABLES -A FORWARD -p icmp -i $INT -o $EXT -s $LOCAL -d $ANY
    -m conntrack --ctstate=NEW -j ACCEPT
$IPTABLES -A FORWARD -p tcp -i $INT -o $EXT -s $LOCAL -d $ANY
    --dport 53 -m conntrack --ctstate=NEW -j ACCEPT
$IPTABLES -A FORWARD -p udp -i $INT -o $EXT -s $LOCAL -d $ANY
    --dport 53 -m conntrack --ctstate=NEW -j ACCEPT
$IPTABLES -A FORWARD -p udp -i $INT -o $EXT -s $LOCAL -d $ANY
    --dport 123 -m conntrack --ctstate=NEW -j ACCEPT

echo 1 > /proc/sys/net/ipv4/ip_forward
```

Listing 5.13: iptables

5.4 Modifications to Cloudflare Quiche

Cloudflare's Quiche [60] is an open-source QUIC and HTTP/3 implementation written in Rust. It includes various features, including client-side connection migrations. In this project, Quiche was utilized to generate and analyze real-world QUIC traffic. However, the example provided in the official Quiche repository only allowed migrations that involved a change in the client port, without modifying the IP address.

To overcome this limitation, minor modifications, depicted in Listing 5.14, were made to enable migration to a new client IP address.

```
# apps/src/args.rs
## before
let source_port = args.get_str("--source-port");
let source_port = source_port.parse::<u16>().unwrap();

## after
let source_port = args.get_str("--source-port");
let source_port = source_port.parse::<u16>().unwrap();

let source_ip = if args.get_bool("--source-ip") {
    args.get_str("--source-ip").to_string()
} else {
    "0.0.0.0".to_string()
};

let new_ip = if args.get_bool("--new-ip") {
    args.get_str("--new-ip").to_string()
} else {
    "0.0.0.0".to_string()
};

# apps/src/client.rs
## before
let bind_addr = match peer_addr {
    std::net::SocketAddr::V4(_) => format!("0.0.0.0:{}",
        args.source_port
    )
};


## after
let bind_addr_migr = match peer_addr {
    std::net::SocketAddr::V4(_) => format!("{}:{}",
        &args.new_ip,
        args.source_port
    )
};
```

Listing 5.14: Modifications to Cloudflare Quiche's Example

Two new command-line arguments were introduced, shown in Listing 5.15: `--source-ip` and `--new-ip`. The `--source-ip` argument specifies the IP address and corresponding network interface from which the packet should be sent. The `--new-ip` argument designates the IP address to which the client should migrate to.

Instead of binding only a `source_port` to `0.0.0.0`, which tells the system to use any available IP, the binding was modified to explicitly use the IP address provided via `--source-ip`. If no IP is supplied, the default behavior remains binding to `0.0.0.0`.

```
## before
cargo run --bin quiche-client -- https://<server-ip>:<port e.g. 4433>
    --no-verify --enable-active-migration --perform-migration

## after
cargo run --bin quiche-client -- https://<server-ip>:<port e.g. 4433>
    --no-verify --enable-active-migration --perform-migration
    --source-ip <source ip> --new-ip <ip to migrate to>
```

Listing 5.15: Quiche Usage

5.5 Traffic Faker

To evaluate the firewall, simulate different scenarios and test different functionalities, a custom tool named Traffic Faker was developed.

The primary goal of the Traffic Faker is to send artificial QUIC-like traffic mith minimal complexity. Only the IP headers, UDP headers, and partially constructed QUIC headers are generated. The encrypted portions of the header and payload replaced with randomly generated bytes. No handshake, key exchange, or actual data transfer occurs. This approach was taken for two reasons. First, the firewall is only analyzing specific parts of the packet, so simulating full QUIC behavior is unnecessary and would only increase complexity. Second, for the performance evaluation, reducing client and server overhead ensures that the focus remains on the firewall's, not the endpoints' performance.

Traffic Faker consists of a client and a server. The client is the active component, initiating the connection. It generates both the source and destination connection IDs and constructs the QUIC header. The remaining packet content, with a randomly determined length, consists of random bytes. To simulate more realistic traffic patterns, the client and server should not simply alternate sending packets. Instead, the client should sometimes send multiple packets, and at other times, the server should respond with multiple packets. To achieve this, the client generates a random, but weighted, value between 0 and 5, with 0 being the most likely. This value, referred to as the `return flag`, indicates how many packets the client expects from the server in response. The client embeds this value in a predefined position within the random packet payload.

The server passively waits for incoming packets and responds to the client. It parses the QUIC headers, swaps the connection IDs, and reads the return flag to determine how many packets the client expects. The server then sends the requested number of packets to the client.

The client waits until it receives the expected number of response packets from the server before continuing to send more packets.

Connection Migrations are more complex. The client must specify not only how many packets it expects in return, but also the length of the new destination connection ID, the new source connection ID, and its length. In actual QUIC traffic, this information is negotiated during the handshake or exchanged in encrypted form. For simplicity, Traffic

Faker writes this information unencrypted at a specific location in the packet payload. Since the firewall does not analyze the packet payload, this has no implications on its functionality.

The server extracts the new source connection ID from this location and sets it as the destination connection ID on the `PATH_RESPONSE`. If the return flag is set to 2, it also sends a final packet to complete the migration process.

Traffic Faker allows for two main types of evaluation: performance and delay measurement. For performance evaluation, the client sends a large number of packets to generate high load on the firewall. By counting the total number of packets, including packets sent from the server, and measuring the time taken, the capacity of the firewall in packets per second can be determined. For delay measurements, the client records the current time, sends a packet, and measures the elapsed time upon receiving the response to calculate the round-trip time.

The client can be started using the commands shown in Listing 5.16. The `-i` flag specifies the interface from which traffic should be sent, and the `-d` flag specifies the destination IP for the packets. If multiple interfaces or destination IPs are provided, a connection migration will be performed. Server-side connection migrations will only succeeed if all relevant interfaces are also specified on the server, as visible in Listing 5.17. The `-p` flag sets the target number of packets to send. While the exact number may vary due to random packet generation, the program will stop once the total packets sent exceeds this threshold.

```
cargo build && sudo ./target/debug/traffic_faker client -i <Interface>
    -d <Server-IP> -p <Number of Packets>

## Examples
# Client-Side Connection Migration
cargo build && sudo ./target/debug/traffic_faker client -i eth0 eth1
    -d 172.16.0.10 -p 10000

# Server-Side Connection Migration
cargo build && sudo ./target/debug/traffic_faker client -i eth0
    -d 172.16.0.10 172.16.0.11 -p 10000

# Both Client- and Server-Side Connection Migration
cargo build && sudo ./target/debug/traffic_faker client -i eth0 eth1
    -d 172.16.0.10 172.16.0.11 -p 10000
```

Listing 5.16: Traffic Faker Client Usage

The commands in Listing 5.17 can be used to run Traffic Faker as the server. The `-i` flag specifies the interfaces the server should listen on. Multiple interfaces can be supplied to support server-side connection migrations.

```
cargo build && sudo ./target/debug/traffic_faker server -i <Interface>

## Example
cargo build && sudo ./target/debug/traffic_faker server -i eth0 eth1
```

Listing 5.17: Traffic Faker Server Usage

Chapter 6

Evaluation

This chapter presents the evaluation of the firewall. It begins by describing the experimental setup and includes the tools used for testing. The following section presents the results, highlighting key findings from the evaluation. Finally, this chapter concludes with a discussion of these results and potential improvements to the firewall.

6.1 Experimental Setup

To evaluate the firewall, a controlled test environment was set up that mirrors the network configuration described in the design and implementation chapters. The test setup consists of three Ubuntu Desktop virtual machines, each with 3 GB of RAM. These machines are configured as follows: one acting as the firewall, another as the client, and the third as the server.

Both the client and the server machines are equipped with two network interfaces to facilitate testing of both client-side and server-side connection migrations. This setup allows the evaluation of the firewall's ability to detect and block malicious connection migrations where the client might transition from one network interface, and therefore one IP-address, to another, or where the server switches between different interfaces.

The firewall itself is positioned as a middlebox between the client and the server, monitoring and controlling all traffic passing through it. This provides an opportunity to examine the firewall's performance and effectiveness in identifying and handling connection migrations, as well as to test how well it integrates with the QUIC protocol and handles encrypted traffic.

6.1.1 Cloudflare quiche

Cloudflare's quiche was used as a real-world QUIC implementation to test the effectiveness of the firewall. Quiche is an open-source QUIC implementation written in Rust, and it serves as a high-performance and flexible option for handling QUIC traffic.

Quiche provides an example application in their GitHub repository that demonstrates basic functionality and usage. However, this example only supports connection migrations to other client-ports. Therefore, this example was extended to support changes in client IP addresses as well. This extension allowed for more comprehensive testing of client-side connection migrations. More details can be found in Chapter 5.4.

However, one limitation of quiche at the time of writing is that server-side migrations, so the use of preferred addresses, are still under development. Despite this, the client-side migration functionality provided a foundation for evaluating the firewall's ability to manage real-world connection migrations and block potentially malicious activities.

6.1.2 Traffic Faker

In order to test server-side connection migrations and simulate more complex traffic scenarios, a custom tool named Traffic Faker was developed.

Traffic Faker operates with a simple client-server model. The client generates packets with random payloads and includes a flag within the payload specifying the number of packets the server should send in response. Upon receiving these packets, the server reads this flag and sends the requested number of packets back. The client waits for all requested responses before continuing. This mechanism enables the systematic generation of traffic while allowing simulation of both client-side and server-side connection migrations.

The combination of quiche and Traffic Faker allowed for an extensive evaluation of the firewall's performance under varying load conditions, realistic traffic patterns, and in both client- and server-side connection migrations.

6.1.3 Exporting Firewall Data

In order to facilitate the analysis and tracking of the firewall's operations, the application needed a way to export data. The `ctrlc` crate was utilized to handle a graceful shutdown of the application. When a termination signal is received, this crate ensures that the `pcap` listeners on the network interfaces are closed and stop blocking the main thread. The crate `serde` was then used to serialize and export all data in the state table and the packet table into JSON format. These tables contain detailed records of all packets and connections that the firewall has processed. By exporting this data, it becomes possible to conduct post-operation analysis, making it easier to debug and assess the firewall's performance.

6.1.4 Collecting System Information

To monitor the system's resource usage during the firewall's operation, the built-in Linux tool `ps` [64] was used. This tool is capable of gathering system information such as CPU usage and memory consumption. The tool was configured to retrieve this data for the specific firewall process every second for the duration of the test. The collected system information was then exported into a `txt` file for later analysis.

6.2 Results

This section presents the results of the evaluation, focusing on the firewall's performance, resource consumption and accuracy. Performance is measured in terms of throughput and the delay introduced by the firewall. Additionally, CPU and memory usage are analyzed to assess resource efficiency, along with the accuracy in tracking and handling connection migrations.

6.2.1 Performance

Performance was evaluated in two ways: first, by generating a large amount of traffic using the Traffic Faker tool and measuring how many packets per second the firewall can process, and second, by measuring the round-trip time of a single packet as it passed through the firewall. Both methods were tested across three different configurations: direct communication between the client and server (without any middlebox), traffic routed through iptables (the default Linux tool), and traffic routed through the custom Rust-based firewall.

In the first test, a client and server were used to generate 10'000, 100'000, and 1 million packets. The total time to complete the packet generation was measured, and the number of packets processed per second was calculated by dividing the total number of packets by the time taken. For all three configurations — direct communication, iptables, and the firewall — no significant differences were observed between processing 10'000, 100'000, or 1 million packets. As shown in Table 6.1, direct communication, where packets do not pass through a middlebox, naturally had a higher throughput than when traffic passed through either iptables or the firewall. Specifically, direct communication achieved a throughput of 3'777 packets per second, whereas iptables and the firewall both handled approximately 3'388 to 3'488 packets per second.

These small differences in throughput suggest that the performance of the firewall and iptables may not be the bottleneck, but rather, the network itself. This observation was further supported when additional client machines were introduced and run in parallel. The firewall's throughput significantly increased with three clients, reaching over 9'200 packets per second, compared to just 3,500 packets per second with a single client.

Table 6.1: Performance Measurements across Three Different Scenarios

Number of Test Clients	**direct, no middlebox**	**iptables**	**firewall**
1 client	3777 packets/sec	3388 packets/sec	3488 packets/sec
2 clients	7046 packets/sec	6671 packets/sec	6766 packets/sec
3 clients	9996 packets/sec	9467 packets/sec	9206 packets/sec

A second method for evaluating performance involved measuring the round-trip time (RTT) of a packet to assess the delay introduced by the middlebox. This was tested again in three configurations: direct communication, iptables, and the firewall. The RTT was also measured for different types of packets, including initial packets, handshake packets, encrypted payloads, and both client-side and server-side connection migrations. For each packet type, 1'000 individual packets were measured, and the results were averaged.

As visible in Table 6.2, the data showed that initial packets took the longest to process in all three configurations. This is especially true for the middleboxes (iptables and the firewall), as new entries need to be created in the state table. Handshake packets and encrypted payloads were processed in approximately half the time of initial packets, indicating that the middleboxes can process these types of packets more efficiently. Connection migrations, however, took a similar amount of time to initial packets, which is expected since new connections must also be added to the state table during a migration.

The firewall was found to be between 25% and 50% slower than direct communication and between 5% and 30% slower than iptables, depending on the packet type and configuration. It is important to note that the measured delays can vary significantly due to a number of factors, including the state of the operating system, other services using the network, and the activities of the host machine at the time of testing. The measurements for direct communication ranged from 350 microseconds to 1 millisecond for the same packet types.

Given these variations, it is difficult to draw definitive conclusions about exact performance differences. However, it can concluded that while the firewall does introduce some additional latency, the impact is relatively moderate, and the firewall operates within a similar range of performance to iptables. Therefore, the firewall's performance, though slightly slower than direct communication and iptables, is acceptable and does not significantly hinder overall traffic handling.

Table 6.2: Delay Introduced by the Firewall for each Packet Type

Packet Type	**direct, no middlebox**	**iptables**	**firewall**
Initial	677.63 μs	710.73 μs	788.74 μs
Handshake	628.57 μs	705.09 μs	757.66 μs
Encrypted Payload	655.88 μs	700.89 μs	688.93 μs
Client-side Migration	762.92 μs	831.71 μs	936.88 μs
Server-side Migration	625.43 μs	726.83 μs	933.78 μs

6.2.2 Resource Consumption

Resource consumption was evaluated by monitoring CPU and memory usage of the firewall process using the built-in Linux tool `ps`. These figures were recorded every second to track how the firewall's resource usage varied under different levels of load.

The first scenario tested the firewall running without any traffic. This provided a baseline measurement of resource consumption when the firewall was idle and not processing any packets.

In the second scenario, the firewall was subjected to a low load by having one client send 1 million packets, resulting in the firewall processing roughly 3'500 packets per second. This made it possible to observe the resource usage under moderate traffic conditions.

The other scenarios involved a higher load, where up to three clients sent traffic simultaneously, resulting in the firewall processing 9'200 packets per second. This scenario tested the firewall's performance under heavy traffic and allowed for an assessment of its resource usage under stress.

By comparing resource consumption across these scenarios, the efficiency of the firewall and its ability to maintain stable performance under increasing amounts of traffic were evaluated.

CPU Usage

As shown in Table 6.3, under idle load, where no packets were processed by the firewall, the CPU usage averaged at 1.13%. As the traffic load increased, the CPU usage also rose. Under low load, when one client was sending traffic at a rate of 3'500 packets per second, the CPU usage increased to 11%. Under heavy load, with three clients sending packets simultaneously at a combined rate of 9'500 packets per second, the CPU usage peaked at 29%. These results show a clear correlation between CPU usage and traffic volume. Importantly, it can be assumed that the firewall is capable of handling more than 9'500 packets per second, as the system was not fully maxed out during the tests. The increase in performance with additional clients suggests that the firewall's capacity can scale further without reaching its limits under the current testing conditions.

Table 6.3: CPU Usage under Varying Load Conditions

CPU	**Usage (%)**
Idle	1.13
Low load (1 client)	11.41
Medium load (2 clients)	22.13
Heavy load (3 clients)	29.55

Memory Usage

Memory consumption was also monitored across different load levels. As presented in Table 6.4, under idle load, with no packets processed, the firewall consumed an average of 0.8% of the system's total memory. When operating under low load, where one client sent 3'500 packets per second, memory usage increased to 8%. Under heavy load, with three clients sending a total of 9'500 packets per second, memory usage peaked at 15%. These figures indicate that the firewall's memory consumption grows as the traffic load increases, but it remains relatively low even under heavy load. The small increase in memory usage suggests that the firewall efficiently handles traffic processing without putting excessive strain on the system's memory resources.

Table 6.4: Memory Usage under Different Load Conditions

Memory	**Usage (%)**	**Virtual Memory**	**Physical Memory**
Idle	0.8	420.04 MB	30.93 MB
Low load (1 client)	5.54	531.76 MB	176.62 MB
Medium load (2 clients)	8.68	603.59 MB	272.11 MB
Heavy load (3 clients)	15.36	806.78 MB	475.66 MB

6.2.3 Accuracy

The firewall demonstrated high accuracy in classifying and parsing QUIC traffic from both Cloudflare Quiche and the custom Traffic Faker tool. Client-side connection migrations, generated from both sources, were consistently recognized and handled properly. The firewall correctly identified and processed pseudo server-side connection migrations, which were simulated using Traffic Faker.

For testing purposes, specific IP addresses, similar to those of Google, Microsoft, and AWS, can be added to the firewall's IP range list. During testing, when a connection attempted to migrate to one of these IP addresses, the firewall correctly identified the change and blocked the migration as expected. This confirms that the firewall can accurately monitor and control connection migrations based on IP addresses, ensuring it functions as intended for both client-side and simulated server-side migrations.

In summary, the firewall's performance in detecting and handling various types of connection migrations has proven to be reliable, demonstrating its capacity to accurately manage real-world QUIC traffic scenarios.

6.3 Discussion

The goal of this project was to develop a firewall capable of handling QUIC traffic, including connection migrations, and to evaluate its performance in real-world scenarios. Throughout the implementation and testing phases, several insights were gained about the capabilities and limitations of the firewall, as well as the challenges of working with a protocol like QUIC. This section will discuss the results, focusing on the effectiveness of the firewall, its limitations, and potential improvements. Additionally, it will address broader considerations for security solutions in corporate and consumer environments, and the practical constraints of implementing such a system in a variety of contexts.

One important takeaway from this project is that QUIC traffic can indeed be tracked, even during connection migrations. The firewall developed as part of this project demonstrated the ability to handle and process QUIC traffic at a rate comparable to established tools like iptables, confirming that it can support real-world traffic.

However, while it is able to track and manage connection migrations effectively, it remains limited in certain scenarios. For example, client-side connection migrations from an external IP address to an internal one are not supported. This means that the firewall is not designed to handle cases where a client initially uses an external IP address (e.g., using cellular network) and then switches to an internal network (e.g., when transitioning to a WiFi network within an office). This gap in functionality highlights the challenges of fully supporting all types of connection migration scenarios, particularly in environments where devices change network interfaces as part of a seamless user experience.

For enterprise environments, a more robust and secure solution might involve deploying Deep Packet Inspection (DPI) or host-based solutions. DPI works by decrypting the entire traffic, allowing the firewall to analyze and filter packets more comprehensively. This approach, however, requires more control over the client devices, as it typically involves the installation of certificates on the client to ensure the traffic between the client and the firewall is decrypted. While this approach is feasible in a corporate setup where devices are managed and controlled, it is not a practical solution for consumer environments. Furthermore, the implementation of DPI raises concerns around privacy, as intercepting and decrypting traffic could lead to potential breaches of confidentiality if not done correctly.

Another alternative for enterprise use cases would be a host-based firewall solution, which involves installing software directly on the client devices to monitor and block traffic before it even leaves the machine. This approach removes the need for a middlebox solution but requires that each device is running the necessary software, adding complexity and requiring a level of control over the devices that may not always be feasible.

It is important to note that the firewall developed in this project is a prototype, and while it is effective in managing QUIC traffic under specific conditions, it is not designed to handle other protocols such as TCP (and thus HTTP/2), DNS, or ICMP. Expanding the firewall's capabilities to include support for these protocols would be an important step for making it more versatile and capable of handling a broader range of network traffic.

In conclusion, while the firewall prototype developed for this project shows promising capabilities for monitoring and controlling QUIC traffic, it is clear that further work is needed to handle more complex use cases, especially those involving connection migrations across different types of network boundaries. Moreover, a more complete solution would need to address other protocols and network services beyond QUIC. The work done in this project lays a foundation for future developments in this area but is not yet ready for wide-scale deployment in a variety of environments.

Chapter 7

Summary and Conclusions

This chapter is divided into three sections. The first section summarizes the thesis and discusses the main results and conclusions drawn from the work conducted. The limitations are covered in Section 7.2. The final section gives an outlook on what can be done in future work.

7.1 Conclusions

This thesis had a deeper look into the QUIC protocol, focusing on its unique features, security considerations, and challenges it introduces for traditional network monitoring. The main goal was to explore how QUIC traffic could effectively be managed by a firewall, particularly during connection migrations.

A key aspect of QUIC's design is its reliance on UDP rather than TCP, offering significant advantages such as faster connection establishment and full encryption. However, this shift presents new challenges for middleboxes. Unlike TCP, UDP is a connectionless protocol, with no built-in handshake or connection termination on the transport layer. This makes it difficult for network devices to track sessions reliably. QUIC further complicates matters by allowing key connection parameters, such as IP addresses, ports and connection IDs, to change dynamically during connection migrations, making connection tracking even more complex.

The contribution of this thesis is the design and development of a stateful firewall prototype implemented in Rust. This firewall is capable of parsing and analyzing QUIC traffic and is able to track connections during connection migrations. It can identify potentially malicious connection migrations based on three factors and can block them effectively.

To validate the firewall's performance and reliability, an extensive evaluation was conducted using Cloudflare Quiche, an open-source implementation of QUIC and HTTP/3, alongside a custom-built tool called Traffic Faker. This tool was used to simulate high volumes of synthetic QUIC traffic to evaluate how the firewall handles different scenarios and stress conditions. The results showed that the firewall could process thousands of packets per second, with performance levels only moderately slower than iptables. Similarly, the delay introduced for individual packets was also only marginally higher compared to iptables.

Resource usage during testing showed that the firewall runs efficiently, consuming approximately 30% of CPU and 15% of available memory on a small Ubuntu virtual machine with 3GB of RAM.

As QUIC continues to evolve, with protocol specifications still subject to change, future iterations of this firewall may need to be adapted to accommodate new features and updates. Nevertheless, this thesis provides a foundation for future research and practical implementations.

7.2 Limitations

This project is a prototype and is designed to only handle very specific use cases. As a result, it may not cover all potential scenarios that could arise in real-world usage. The firewall is specifically built for QUIC and does not extend to other protocols. It is tailored to handle particular types of attacks related to QUIC connection migrations, and does not cover other forms of attacks, whether targeting QUIC, UDP, or other parts of the protocol.

Another limitation is that the dataset generated by Traffic Faker during the evaluation is mostly artificial. The traffic patterns do not fully reflect the complexity and variability of actual real-world traffic.

Furthermore, this thesis does not present any documentation of an actual malicious connection migration. The absence of a concrete attack case study means that certain assumptions about how malicious actors might exploit connection migrations may not align perfectly with actual threats.

Another important consideration is the evolving nature of the QUIC protocol itself. Since QUIC specifications are still relatively new and under active development, future modifications to the protocol could introduce changes that may affect how connection migrations are handled. Assumptions made in this project are based on the current specifications and may need to be revised as the protocol matures.

Finally, most current QUIC implementations do not fully support all features yet, such as server-side connection migrations. This project made several assumptions about how these features might function once implemented, but there is still uncertainty about their final design and behavior. When these features become available, the firewall's design may need to be re-evaluated and adjusted accordingly.

7.3 Future Work

Future versions and iterations of the QUIC protocol will introduce new features and improvements, and there might be changes that could impact how connection migrations are handled. One important direction of future research is exploring actual server-side connection migrations, once this is fully implemented in real-world implementations. Additionally, server-initiated connection migrations, where the server sends the `PATH_CHALLENGE` instead of the client, is another potential feature that might be implemented in the future.

Another area of future research is the development of more sophisticated methods for detecting and blocking malicious connection migrations. This could involve the integration of machine learning models to analyze traffic patterns or DPI to analyze the actual contents of the packet.

Future research could also explore how connection migrations impact the server. Implementing DNAT to support hosted web servers behind the firewall might require different strategies to handle connection migrations.

This project only handles client-side connection migrations within the internal network and server-side connection migrations in the external network. Connection migrations across these boundaries, such as when a client moves from an external cellular network to an internal WiFi network, were not considered and would require reestablishing a connection. New approaches would be needed to maintain seamless connectivity in these scenarios.

With the industry moving towards host-based solutions, exploring possibilities and challenges of these approaches is another topic for future research.

Finally, concrete attack scenarios could be explored to better understand how malicious actors might exploit weaknesses in the QUIC protocol. Simulating and analyzing these attacks would provide valuable insights into how attackers operate and help in designing more effective countermeasures.

Bibliography

[1] M. Thomson: Version-Independent Properties of QUIC, Internet Requests for Comments, IETF, RFC 8999, https://datatracker.ietf.org/doc/html/rfc8999, May 2021.

[2] J. Iyengar and M. Thomson: QUIC: A UDP-Based Multiplexed and Secure Transport, Internet Requests for Comments, IETF, RFC 9000, https://datatracker.ietf.org/doc/html/rfc9000, May 2021.

[3] M. Thomson and S. Turner: Using TLS to Secure QUIC, Internet Requests for Comments, IETF, RFC 9001, https://datatracker.ietf.org/doc/html/rfc9001, May 2021.

[4] J. Iyengar and I. Swett: QUIC Loss Detection and Congestion Control, Internet Requests for Comments, IETF, RFC 9002, https://datatracker.ietf.org/doc/html/rfc9002, May 2021.

[5] M. Bishop: HTTP/3, Internet Requests for Comments, IETF, RFC 9114, https://datatracker.ietf.org/doc/html/rfc9114, June 2022.

[6] M. Kühlewind and B. Trammell: Applicability of the QUIC Transport Protocol, Internet Requests for Comments, IETF, RFC 9308, https://datatracker.ietf.org/doc/html/rfc9308, September 2022.

[7] M. Kühlewind and B. Trammell: Manageability of the QUIC Transport Protocol, Internet Requests for Comments, IETF, RFC 9312, https://datatracker.ietf.org/doc/html/rfc9312, September 2022.

[8] M. Duke: QUIC Version 2, Internet Requests for Comments, IETF, RFC 9369, https://datatracker.ietf.org/doc/html/rfc9369, May 2023.

[9] A. Langley et al.: The QUIC Transport Protocol: Design and Internet-Scale Deployment, Association for Computing Machinery, https://doi.org/10.1145/3098822.3098842, 2017.

[10] J. Roskind: Quick UDP internet connections: Multiplexed stream transport over UDP, Online, https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf, 2012, Accessed 06. February 2025.

[11] B. Glisovic: A QUIC way to bypass your firewall, Online, https://documents.swisscom.com/product/filestore/lib/cbb66c05-4db7-432e-a95b-d6d9523a1c0f/ly2_quic_whitepaper_en_v3.pdf?idxme=pex-search, April 2023, Accessed 06. February 2025.

[12] P. Kumar: QUIC (Quick UDP Internet Connections) - A Quick Study, arXiv preprint arXiv:2010.0305, https://arxiv.org/abs/2010.03059, 2020.

[13] S. Cook, B. Mathieu, P. Truong and I. Hamchaoui: QUIC: Better for what and for whom?, 2017 IEEE International Conference on Communications (ICC), https://doi.org/10.1109/ICC.2017.7997281, 2017.

[14] Q. De Coninck and O. Bonaventure: Multipath QUIC: Design and Evaluation, Association for Computing Machinery, https://doi.org/10.1145/3143361.3143370, 2017.

[15] P. Kumar and B. Dezfouli: Implementation and analysis of QUIC for MQTT, Computer Networks, https://doi.org/10.1016/j.comnet.2018.12.012, 2019.

[16] F. Gratzer, S. Gallenmüller and Q. Scheitle: QUIC - Quick UDP Internet Connections, Future Internet and Innovative Internet Technologies and Mobile Communications, https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-09-1/NET-2016-09-1_06.pdf, 2016.

[17] J. Wang: The Performance and Future of QUIC Protocol in the Modern Internet, Network and Communication Technologies, http://ebooks.manu2sent.com/id/eprint/1034/, 2021.

[18] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai: Evaluating QUIC performance over web, cloud storage, and video workloads, IEEE Transactions on Network and Service Management, https://doi.org/10.1109/TNSM.2021.3134562, 2022.

[19] K. Elmenhorst, B. Schütz, N. Aschenbruck and S. Basso: Web censorship measurements of HTTP/3 over QUIC, Proceedings of the 21st ACM Internet Measurement Conference, https://doi.org/, November 2021.

[20] Y. Govil, L. Wang and J. Rexford: MIMIQ: Masking IPs with Migration in QUIC, USENIX Association, https://www.usenix.org/conference/foci20/presentation/govil, 2020.

[21] C. Puliafito, L. Conforti, A. Virdis, and E. Mingozzi: Server-side QUIC connection migration to support microservice deployment at the edge, Pervasive and mobile computing, https://doi.org/10.1016/j.pmcj.2022.101580, 2022.

[22] M. Wang, A. Kulshrestha, L. Wang and P. Mittal: Leveraging strategic connection migration-powered traffic splitting for privacy, arXiv preprint arXiv:2205.03326, https://doi.org/10.48550/arXiv.2205.03326, 2022.

[23] Y. Haoran: The Design and Evaluation of a Seamless Approach to Migrate the State of QUIC Connections for Load Balancing Purposes, Online, https://www.diva-portal.org/smash/record.jsf?dswid=-1242&pid=diva2%3A1539441, 2021, Accessed 06. February 2025.

[24] Y. Yan and Z. Yang: When QUIC's Connection Migration Meets Middleboxes A case study on mobile Wi-Fi hotspot, 2021 IEEE Global Communications Conference (GLOBECOM), `https://doi.org/10.1109/GLOBECOM46510.2021.9685048`, December 2021.

[25] A. Buchet and C. Pelsser: An Analysis of QUIC Connection Migration in the Wild, arXiv preprint arXiv:2410.06066, `https://arxiv.org/abs/2410.06066`, 2024.

[26] R. Lychev, S. Jero, A. Boldyreva and C. Nita-Rotaru: How Secure and Quick is QUIC? Provable Security and Performance Analyses, 2015 IEEE Symposium on Security and Privacy, `https://doi.org/10.1109/SP.2015.21`, 2015.

[27] E. Chatzoglou, V. Kouliaridis, G. Karopoulos et al.: Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study, International Journal of Information Security 22.2, `https://doi.org/10.1007/s10207-022-00630-6`, April 2023.

[28] K. Gbur and F. Tschorsch: A QUIC(K) Way Through Your Firewall?, arXiv preprint arXiv:2107.05939, `https://doi.org/10.48550/arXiv.2107.05939`, 2021.

[29] S. G. Kulkarni: Security and Service Vulnerabilities with HTTP/3, 2024 16th International Conference on COMmunication Systems & NETworkS (COMSNETS), `https://doi.org/10.1109/COMSNETS59351.2024.10427406`, January 2024.

[30] M. Kosek, B. Spies and J. Ott: Secure Middlebox-Assisted QUIC, 2023 IFIP Networking Conference (IFIP Networking), `https://doi.org/10.23919/IFIPNetworking57963.2023.10186363`, 2023.

[31] M. Soni and B. S. Rajput: Security and Performance Evaluations of QUIC Protocol, Springer Singapore, `https://doi.org/10.1007/978-981-15-4474-3_51`, June 2020.

[32] L. Al-Bakhat and S. Almuhammadi: Intrusion detection on Quic Traffic: A machine learning approach, 2022 7th International Conference on Data Science and Machine Learning Applications (CDMA), `https://doi.org/10.1109/CDMA54072.2022.00037`, March 2022.

[33] J. Lee, M. Kim, W. Song, Y. Kim, and D. Kim: Rescuing QUIC Flows From Countermeasures Against UDP Flooding Attacks, Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, `https://doi.org/10.1145/3605098.3635885`, April 2024.

[34] A. Kadi, L. Khoukhi, J. Viinikka and P. E. Fabre: Machine Learning for QUIC Traffic Flood Detection, 2024 Global Information Infrastructure and Networking Symposium (GIIS), `https://doi.org/10.1109/GIIS59465.2024.10449925`, 2024.

[35] S. Oran, A. Koçak and M. Alkan: Security Review and Performance Analysis of QUIC and TCP Protocols, 2022 15th International Conference on Information Security and Cryptography (ISCTURKEY), `https://doi.org/10.1109/ISCTURKEY56345.2022.9931821`, October 2022.

[36] M. Gouda and A. X. Liu: A model of stateful firewalls and its properties, 2005 International Conference on Dependable Systems and Networks (DSN'05), https://doi.org/10.1109/DSN.2005.9, 2005.

[37] A. Wool: Packet filtering and stateful firewalls, Handbook of Information Security, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1e3cdf72351748e00e45d8fcbd157e1c66cd8fca, 2006.

[38] J. Liang and Y. Kim: Evolution of Firewalls: Toward Securer Network Using Next Generation Firewall, 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC), https://doi.org/10.1109/CCWC54503.2022.9720435, 2022.

[39] M. Kühlewind, B. Trammell, T. Bühler, G. Fairhurst and V. Gurbani: Challenges in network management of encrypted traffic, arXiv preprint arXiv:1810.09272, https://arxiv.org/abs/1810.09272, 2018.

[40] M. Crotti, M. Dusi, F. Gringoli and L. Salgarelli: Traffic classification through simple statistical fingerprinting, Association for Computing Machinery, https://doi.org/10.1145/1198255.1198257, January 2007.

[41] F. Cuppens, N. Cuppens-Boulahia, J. Garcia-Alfaro, T. Moataz and X. Rimasson: Handling Stateful Firewall Anomalies, IFIP International Information Security Conference, https://doi.org/10.1007/978-3-642-30436-1_15, June 2012.

[42] Z. Trabelsi, S. Zeidan, K. Shuaib and K. Salah: Improved Session Table Architecture for Denial of Stateful Firewall Attacks, IEEE Access, https://doi.org/10.1109/ACCESS.2018.2850345, 2018.

[43] J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, S. Martinez and J. Cabot: Management of stateful firewall misconfiguration, Computers & Security, https://doi.org/10.1016/j.cose.2013.01.004, 2013.

[44] S. Kumar, J. Turner and J. Williams: Advanced algorithms for fast and scalable deep packet inspection, Association for Computing Machinery, https://doi.org/10.1145/1185347.1185359, 2006.

[45] W. Goralski: The illustrated network: how TCP/IP works in a modern network, Morgan Kaufmann, 2017.

[46] C. Hunt: TCP/IP network administration. Vol. 2, O'Reilly Media, Inc., 2002.

[47] R. K. C. Chang and K. P. Fung: Transport layer proxy for stateful UDP packet filtering, Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications, https://doi.org/10.1109/ISCC.2002.1021735, July 2002.

[48] C. Roeckl: Stateful inspection firewalls, Juniper Networks White Paper, http://www.abchost.cz/download/204-4/juniper-%EE%80%80stateful%EE%80%81-inspection-firewall.pdf, 2004.

[49] J. Davies: Understanding ipv6, Pearson Education, 2012.

[50] S. Hagen: IPv6 essentials, O'Reilly Media, Inc., 2006.

[51] N. Brightwood: SNAT vs DNAT, Online, https://netseccloud.com/snat-vs-dnat, May 2024, Accessed 06. February 2025.

[52] S. Hogg: You Thought There Was No NAT for IPv6, But NAT Still Exists, Online, https://blogs.infoblox.com/ipv6-coe/you-thought-there-was-no-nat-for-ipv6-but-nat-still-exists/, December 2021, Accessed 06. February 2025.

[53] R. T. El-Maghraby, N. M. Abd Elazim and A. M. Bahaa-Eldin: A survey on deep packet inspection, 2017 12th International Conference on Computer Engineering and Systems (ICCES), https://doi.org/10.1109/ICCES.2017.8275301, 2017.

[54] C. Lu, B. Liu, Y. Zhang, Z. Li, F. Zhang, H. Duan, Y. Liu, J. Q. Chen, J. Liang, Z. Zhang, S. Hao and M. Yang: From WHOIS to WHOWAS: A Large-Scale Measurement Study of Domain Registration Privacy under the GDPR, NDSS, https://personal.utdallas.edu/~shao/papers/lu_ndss21.pdf, February 2021.

[55] E. Liebetrau: How Google's QUIC Protocol Impacts Network Security and Reporting, Online, https://www.fastvue.co/fastvue/blog/googles-quic-protocols-security-and-reporting-implications/, June 2018, Accessed 06. February 2025.

[56] Zscaler: Managing the QUIC Protocol, Online, https://help.zscaler.com/zia/managing-quic-protocol, Accessed 06. February 2025.

[57] Sophos: Prevent QUIC protocol from bypassing firewall scanning, Online, https://support.sophos.com/support/s/article/KBA-000005027, July 2024, Accessed 06. February 2025.

[58] FortiGate: Technical Tip: Disabling / Blocking QUIC Protocol to force Google Chrome to use TLS, Online, https://community.fortinet.com/t5/FortiGate/Technical-Tip-Disabling-Blocking-QUIC-Protocol-to-force-Google/ta-p/191657?externalId=FD36529, April 2015, Accessed 06. February 2025.

[59] Palo Alto: Internet Gateway Best Practice Security Policy, Online, https://docs.paloaltonetworks.com/best-practices/internet-gateway-best-practices/best-practice-internet-gateway-security-policy/define-the-initial-internet-gateway-security-policy/step-3-create-the-application-block-rules, January 2024, Accessed 06. February 2025.

[60] Cloudflare Quiche, Online, https://github.com/cloudflare/quiche, Accessed 06. February 2025.

[61] Google Quiche, Online, https://github.com/google/quiche, Accessed 06. February 2025.

[62] MSQUIC, Online, https://github.com/microsoft/msquic, Accessed 06. February 2025.

[63] Iptables Documentation, Online, https://help.ubuntu.com/community/IptablesHowTo, Accessed 06. February 2025.

[64] ps Documentation, Online, `https://manpages.ubuntu.com/manpages/xenial/man1/ps.1.html`, Accessed 06. February 2025.

[65] Crate pcap Documentation, Online, `https://docs.rs/pcap/latest/pcap/`, Accessed 06. February 2025.

[66] Crate etherparse Documentation, Online, `https://docs.rs/etherparse/latest/etherparse/`, Accessed 06. February 2025.

[67] Crate pnet Documentation, Online, `https://docs.rs/pnet/latest/pnet/`, Accessed 06. February 2025.

[68] Crate ipnetwork Documentation, Online, `https://docs.rs/ipnetwork/latest/ipnetwork/`, Accessed 06. February 2025.

[69] Crate itertools Documentation, Online, `https://docs.rs/itertools/latest/itertools/`, Accessed 06. February 2025.

[70] Crate regex Documentation, Online, `https://docs.rs/regex/latest/regex/`, Accessed 06. February 2025.

[71] Crate serde Documentation, Online, `https://docs.rs/serde/latest/serde/`, Accessed 06. February 2025.

[72] Crate serde_json Documentation, Online, `https://docs.rs/serde_json/latest/serde_json/`, Accessed 06. February 2025.

[73] Crate clap Documentation, Online, `https://docs.rs/clap/latest/clap/`, Accessed 06. February 2025.

[74] Crate rand Documentation, Online, `https://docs.rs/rand/latest/rand/`, Accessed 06. February 2025.

[75] Crate ctrlc Documentation, Online, `https://docs.rs/ctrlc/latest/ctrlc/`, Accessed 06. February 2025.

[76] Google Public IP Ranges, Online, `https://support.google.com/a/answer/10026322?hl=en`, Accessed 06. February 2025.

[77] AWS Public IP Ranges, Online, `https://docs.aws.amazon.com/vpc/latest/userguide/aws-ip-ranges.html`, Accessed 06. February 2025.

[78] Microsoft Public IP Ranges, Online, `https://learn.microsoft.com/en-us/microsoft-365/enterprise/urls-and-ip-address-ranges?view=o365-worldwide`, Accessed 06. February 2025.

[79] Microsoft Azure Public IP Ranges, Online, `https://www.microsoft.com/en-us/download/details.aspx?id=56519`, Accessed 06. February 2025.

[80] D. Sarman: QUIC-Firewall, Online, `https://github.com/dosar1/QUIC-Firewall`, 09. February 2025.

Abbreviations

API	Application Programming Interface
ACK	Acknowledgement
CPU	Central Processing Unit
DNAT	Destination Network Address Translation
DNS	Domain Name System
DoS	Denial of Service
DPI	Deep Packet Inspection
GB	Gigabyte
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
LAN	Local Area Network
NAT	Network Address Translation
NTP	Network Time Protocol
MB	Megabyte
QUIC	QUIC UDP Internet Protocol
RAM	Random Access Memory
RFC	Request for Comment
RTT	Round Trip Time
SSL	Secure Sockets Layer
SNAT	Source Network Address Translation
SYN	Synchronize
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
UDP	User Datagram Protocol
VM	Virtual Machine

List of Figures

List of Tables

Listings

Appendix A

Installation Guide

This project setup was developed on a Windows Pro Machine using Hyper-V as the hypervisor. The installation steps may vary for other operating systems systems and other hypervisors.

A.1 Installation

Below are the steps to set up the required environment and to deploy the firewall:

1. **Ensure that a Hypervisor is installed:** Install and setup a hypervisor such as Hyper-V, VMWare, or VirtualBox. This guide assumes the use of Hyper-V.

2. **Create a Private Virtual Switch:** Configure a new virtual switch within your hypervisor that does not allow communication with external networks. This switch will act as our internal LAN. Ensure there is an existing virtual switch, often called Default Switch, which is connected to the internet. If it does not exist, create one as well. In the end, you should have two virtual switches: one connected to the internet and another one exclusively for communication between virtual machines.

3. **Set up and install three Ubuntu Virtual Machines:** Create three virtual machines, allocate at least 3GB of RAM each, and connect all of them to the Default Switch initially. Download the latest Version of Ubuntu and install it on all three machines. It does not matter whether you choose the Desktop or Server version.

4. **Install Rust:** Install Rust on all three virtual machiens by following the official Rust installation guide on `https://www.rust-lang.org/tools/install`. At the time of writing this thesis, Rust can be installed using the following commands:

   ```
   curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
   sudo apt install build-essential cmake  -y
   ```

   For the pcap crate to work, the libary libpcap-dev needs to be installed as well:

   ```
   sudo apt install libpcap-dev -y
   ```

5. **Clone Git Repository:** Ensure Git is installed on the system. If it is not, install it using your package manager. Once it is installed, clone the repository by running the following command:

```
sudo apt install git  -y
git clone https://github.com/dosar1/QUIC-Firewall.git
```

6. **Add Network Interfaces:** Add an additional network interface to each of the three virtual machines so that each VM has two interfaces. Connect them as follows:

   - **Client:** Connect both interfaces to the newly created private switch.
   - **Server:** Connect both interfaces to the Default Switch.
   - **Firewall:** Connect the first interface (e.g. `eth0`) to the Default Switch, and the second one (e.g. `eth1`) to the private switch.

7. **Configure Network Interfaces:** The network configuration for the client and server can be done via the GUI. However, for the firewall, it is recommended to modify the configuration file directly in `/etc/netplan`.

   - **Client:** Assign static IPs within a private IP range for both interfaces, e.g., `10.0.0.10` and `10.0.0.11`.
   - **Server:** Set both network interfaces to use DHCP.
   - **Firewall:** Open the configuration file as follows:

     ```
     sudo nano /etc/netplan/50-cloud-init.yaml
     ```

     Add the following configuration. In this case, `eth0` is the external, public-facing interface connected to the Default Switch, and `eth1` is connected to the private switch.

     ```
     network:
         version: 2
         ethernets:
             eth0:
                 dhcp4: true
             eth1:
                 addresses:
                 - 10.0.0.1/24
     ```

   Run the following command to check your configuration and apply the settings:

   ```
   sudo netplan generate && sudo netplan apply && ip a
   ```

8. **Optional: Set ARP Cache on the Firewall:** In some cases, it may be necessary or useful to manually configure the ARP cache. Because the client is initially connected to the Default Switch, the firewall may incorrectly send packets to the wrong interface due to ARP conflicts. To resolve this, you can use the following commands:

   - To list all entries in the arp cache:

     ```
     sudo ip neigh
     ```

- To delete all stale entries:

```
sudo ip -s -s neigh flush all
```

- To manually add a permanent ARP entry:

```
sudo ip neigh add <IP> lladdr <MAC-Address> dev <Interface>
```

- To manually delete an ARP entry:

```
sudo ip neigh del <IP> dev <Interface>
```

A.2 Run the Project

A.2.1 Firewall

In the project root, run the following command to enable iptables to handle and forward DNS, NTP and ICMP traffic:

```
sudo ./iptables/firewall.sh
```

To build and run the firewall itself, use the following commands:

```
cd firewall
cargo build && sudo ./target/debug/firewall --firewall
```

By default, the internal interface is `eth0` and the external interface is `eth1`. To specify custom interfaces, provide them in the following order:

```
cargo build && sudo ./target/debug/firewall --firewall <Internal
    Interface> <External Interface>
cargo build && sudo ./target/debug/firewall --firewall eth0 eth1
```

A.2.2 Client

You can use the following commands to use Cloudflare quiche as the client:

```
cd quiche
cargo run --bin quiche-client -- https://cloudflare-quic.com/
cargo run --bin quiche-client -- https://<Server-IP>:<Port e.g. 4433>
    --no-verify
cargo run --bin quiche-client -- https://<Server-IP>:<Port e.g. 4433>
    --no-verify --enable-active-migration --perform-migration
cargo run --bin quiche-client -- https://<Server-IP>:<Port e.g. 4433>
    --no-verify --enable-active-migration --perform-migration
    --source-ip <Source IP> --new-ip <IP to migrate to>
```

You can run the following commands to use the traffic faker as the client, where `-i` specifies the interface from which to send traffic from and `-d` specifies the destination IP to which packets should be sent to. If multiple interfaces or destinations are provided, a connection migration will be preformed. The `-p` flag indicates the number of packets to be sent to the server. While the exact number may vary due to random packet generation, the program will exit once the number of packets sent exceeds this specified threshold.

```
cd traffic_faker
cargo build && sudo ./target/debug/traffic_faker client -i <Interface>
    -d <Server-IP> -p <Number of Packets>
```

This is an example of a client-side connection migration with 10'000 packets sent:

```
cargo build && sudo ./target/debug/traffic_faker client -i eth0 eth1
    -d 172.16.0.10 -p 10000
```

This is an example of a server-side connection migration with 10'000 packets sent:

```
cargo build && sudo ./target/debug/traffic_faker client -i eth0
    -d 172.16.0.10 172.16.0.11 -p 10000
```

A.2.3 Server

You can use the following commands to use Cloudflare quiche as the server:

```
cd quiche
cargo run --bin quiche-server -- --listen 0.0.0.0:4433 --root html
    --cert apps/src/bin/cert.crt --key apps/src/bin/cert.key
    --enable-active-migration
```

You can run the following commands to use the traffic faker as the server, where `-i` is the interface the server listens to.

```
cd traffic_faker
cargo build && sudo ./target/debug/traffic_faker server -i <Interface>
```

This is an example of a server listening on the interfaces `eth0` and `eth1` to allow for server-side connection migrations:

```
cargo build && sudo ./target/debug/traffic_faker server -i eth0 eth1
```

Appendix B

Submitted Documents

The following documents were handed in:

1. The Latex source code and PDF.
2. The abstract in both German and English in separate txt-Files.
3. The intermediate presentation of this thesis.
4. Links to the GitHub repository.