FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	VPN Based on QUIC Protocol
Student:	Jakub Kubík
Supervisor:	Ing. Jiří Dostál, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

The objective of this thesis is to implement a usable VPN protocol based on QUIC - a transport layer protocol made for low-latency data transfer with mandatory encryption. Most commonly used VPN protocols (OpenVPN, IPsec) have a range of shortcomings - performance (single-threaded endpoints, high latency), code-base bloat (very large codebases, difficult to maintain and audit), difficult configuration, etc. The VPN implementation in this thesis should be lightweight with a relatively small codebase and provide satisfactory performance and security compared with other commonly-used protocols.

1) Research existing VPN protocols, and compare their features, performance, and shortcomings.
2) Using a QUIC library of your choice, design and implement a viable VPN protocol with the following features:
2.a) Establishing a tunnel using basic authentication (username and password).
2.b) Routing traffic through the tunnel using TUN/TAP interfaces.
3) Conduct performance tests and compare the implementation with OpenVPN and WireGuard (features, performance, security).

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 12 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

VPN Based on QUIC Protocol

Jakub Kubík

Department of Information Security
Supervisor: Ing. Jiří Dostál, Ph.D.

May 15, 2024

Acknowledgements

I would like to thank my supervisor, Ing. Jiří Dostál Ph.D., for the guidance, knowledge and help he provided during the creation of this thesis. I would also like to thank the open-source community around Quinn, which have been of great help during the development of Quincy at many steps of the way. Last but not least I would like to thank my family, friends and my partner for all the emotional support, which have made creating this thesis a much more pleasant endeavour.

Declaration

Citation of this thesis

Kubík, Jakub. *VPN Based on QUIC Protocol.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstract

This Bachelor's thesis explores the viability of the QUIC protocol in the space of Virtual Private Networks (VPN). It documents the process of designing and implementing a new VPN based on the QUIC protocol and compares it with established VPN implementation, such as OpenVPN and WireGuard, using a series of tests aimed at measuring throughput, latency, and stability. The resulting implementation fulfills all requirements presented by the thesis assignment and achieves good results in performance tests, providing higher throughput and lower latency than OpenVPN and nearing the throughput and latency of WireGuard, proving the viability the QUIC protocol in the VPN space.

Keywords VPN, QUIC, TLS, network traffic tunnelling, secure communication

Abstrakt

Tato bakalářská práce zkoumá použitelnost protokolu QUIC v prostředí virtuálních privátních sítí (VPN). Dokumentuje proces návrhu a implementace nové VPN založené na protokolu QUIC a porovnává ji s existujícími VPN jako OpenVPN a WireGuard pomocí série testů zaměřených na propustnost, odezvu a stabilitu spojení. Výsledná implementace splňuje všechny požadavky ze zadání a dosahuje dobrých výsledků ve výkonostních testech, poskytujíc větší propustnost a nižší odezvu než OpenVPN a přibližujíc se propustnosti a odezvě protokolu WireGuard, čímž potvrzuje použitelnost protokolu QUIC v prostředí VPN.

Klíčová slova VPN, QUIC, TLS, tunelování síťového provozu, bezpečná komunikace

Contents

List of Figures

List of Listings

Introduction

In the rapidly evolving landscape of online security and privacy, the importance of Virtual Private Networks (VPNs) cannot be overstated. The widespread use of digital communication has led to a greater focus on ensuring the confidentiality, integrity, and authenticity of data transmitted over the Internet. Transport Layer Security (TLS) and QUIC are the most commonly used protocols to secure Hypertext Transfer Protocol (HTTP) Web traffic and, as such, serve as a cornerstone of online security and privacy. Although these protocols excel in securing web traffic, their application in VPNs has opened new frontiers to ensure security and privacy in a wider range of online activities.

The space of VPN protocols has been relatively stagnant in the last decade, with innovations few and far between. WireGuard, a notable exception, has gained widespread adoption due to its simplicity, performance, and user-friendly design. Many commercial VPN services have quickly adopted it after its first stable release, leading to additional innovation and research. Despite the availability of various VPN protocols that adequately address security concerns, a critical issue persists: Many of these protocols are relatively easy to detect by sophisticated networking equipment. This often leads to unwarranted monitoring and blocking of user traffic, especially in contexts where security and privacy are of utmost concern.

Due to the existence and increasing prevalence of such environments, the exploration of new VPN protocols that provide better security and privacy becomes necessary. The quest for protocols that not only meet strict security standards, but also circumvent detection and blocking by modern networking infrastructures, is ongoing. Addressing these challenges is crucial to ensure the unimpeded use of VPNs in a diverse range of scenarios, from safeguarding sensitive corporate communications to enabling individuals to access data on the Internet in an unrestricted and uncensored manner.

Chapter 1

State-of-the-art

The goal of this chapter is to provide an introduction to network protocols relevant to this thesis and to explore the realm of existing VPNs. By doing so, it builds the foundation that will be used in the subsequent chapters.

1.1 Network protocols

This section summarises the important information about all network protocols needed to understand the rest of the thesis, including TCP and UDP, both of which are heavily utilised in multiple VPN protocols, TLS used by OpenVPN and QUIC used by the implementation in this thesis.

TCP

The Transmission Control Protocol (TCP) is one of the oldest and most commonly used protocols still serving today's Internet. It allows applications and devices to reliably exchange data, encapsulated in *packets*, over a potentially unreliable network [1].

TCP is a connection-orientated protocol, meaning that a connection is established before any data is transferred. There is a concept of a *client* and a *server*, with the client initiating a connection, while the server listens for incoming connections on a *socket*, which is a combination of an IP address and a port, which in a way identify a running network application, such as a web client or server.

When a connection is established, TCP divides the data to be sent into *packets* of an appropriate size, which are then sent to the other party, while ensuring data integrity by sending acknowledgements of received packets. This is useful for standard operation, but can cause issues when used by VPNs as the backing protocol for the created tunnel. This is due to the deterioration in latency and, transitively, throughput, caused by multiple layers of acknowledgements being sent in both directions.

UDP

The User Datagram Protocol (UDP) is another commonly used protocol that serves as a counterpart to TCP. It allows applications and devices to send data in an unreliable manner, encapsulated in *datagrams*, over a network with little additional overhead.

Unlike TCP, UDP is a connectionless protocol that works in a transaction-orientated manner, with the transaction consisting of a single *datagram*. The protocol does not provide any protection with respect to data integrity and was created to contain as few protocol mechanisms as possible [2]. This makes it especially useful in cases where latency and low protocol overhead are of great importance, such as VPN protocols. This is why many VPN protocols, such as OpenVPN and WireGuard, use it as the underlying transport protocol[1].

TLS

The Transport Layer Security (TLS) protocol is the de facto standard of secure communication over the Internet. It enables endpoints to communicate in a secure manner over a network, protecting both the confidentiality and integrity of the transferred data [3]. It is based on the Public Key Infrastructure (PKI), which leverages asymmetric encryption for peer endpoint authentication and encryption, allowing endpoints to securely share data, such as symmetric encryption keys used for later communication, without the need to pre-share an encryption key over a potentially unsecured channel beforehand.

TLS provides applications with a wide array of different *cipher suites*, which are a combination of an asymmetric encryption-based key exchange and signing algorithm, usually based on Rivest–Shamir–Adleman (RSA) or Eliptic Curve Cryptography (ECC), and an authenticated symmetric encryption algorithm, usually Advanced Encryption Standard (AES) with Secure Hashing Algorithms (SHA) or ChaCha20 with Poly1305 (for more information about these cipher suites, please refer to [4]). This allows endpoints to choose a cipher suite which fits them best (e.g. some ciphers might be hardware accelerated, which might be relevant due to their impact on lower power consumption or higher throughput), allowing for greater versatility of the protocol.

TLS is a successor of the Secure Sockets Layer (SSL) protocol, which served a very similar purpose but eventually became too outdated and insecure, and has seen 4 major versions with numerous improvements. The latest version, TLS 1.3, greatly increases the security of the protocol, deprecating numerous insecure cipher suites and configuration options [5].

The described versatility and its widespread use are the reason why this protocol, and SSL before it, served as the underlying protocols used for authentication, encryption, and transport for numerous VPNs, the most notable of which is OpenVPN.

QUIC

The QUIC protocol is a novel protocol, originally developed by Google and later standardised by the Internet Engineering Task Force (IETF), aiming to replace TLS-over-TCP for HTTP/3 and other connection-orientated application protocols. QUIC uses UDP as the underlying transport protocol and TLS for authenticated encryption of transferred data. It brings much needed

[1]OpenVPN can also use TCP as the underlying transport protocol, however, due to throughput and latency issues, the UDP mode became more prevalent.

improvements in both latency and throughput, and introduces some modern concepts, such as connection multiplexing and network path migration [6].

Connection multiplexing is arguably one of the biggest benefits of QUIC over TLS-over-TCP, as it removes one of the biggest issues in HTTP/1 and HTTP/2 – head-of-line blocking. This issue occurs when a web client makes multiple requests to the same web server over a singular TLS-over-TCP connection and one of the requests takes a longer time or has issues with being delivered, such as dropped packets needing to be resent or a slow database connection on the back-end. This request then blocks all of the subsequent requests, slowing down the load time of the web application considerably. There are some workarounds, such as opening multiple TLS-over-TCP connections, which partially mitigate the issue. However, they come at the cost of increased overhead and latency, due to having to perform multiple TLS-over-TCP handshakes, which take a considerable amount of round-trips. HTTP/2 implements a method to multiplex requests at the application layer. Unfortunately, this only moves the problem to the TCP at the transport layer, which suffers from a similar issue [7]. QUIC removes the head-of-line blocking issue by using UDP as its transport protocol, which does not suffer head-of-line blocking issues due to having no datagram acknowledgement methods, and by implementing *streams*, which act as individual and parallel bidirectional connections. These streams implement similar reliability features as TCP on a per-stream basis, meaning that if a stream packet is dropped, it is retransmitted similar to how TCP would retransmit a lost packet. Since these streams exist within a single TLS session, established at the beginning of the communication, creating a new stream comes at no cost, allowing multiple requests to be made without blocking each other with virtually no overhead [6].

QUIC also allows sending unreliable *datagrams*, which are nearly identical to UDP datagrams in function, with the addition of their data being encrypted as they are still handled by the TLS session established by the QUIC connection. This allows some use-cases, such as video streaming or real-time communication, where packet loss is not critical to the data transfer, to bypass any and all reliability features provided by streams and achieve lower latency and higher throughput [6].

In addition to the aforementioned features, QUIC also introduces *network path migration*, which allows two QUIC endpoints to easily handle events in which a part of the network path between them has changed, such as transferring from a mobile 4G network to a known Wi-Fi network. With TCP, this situation would begin a lengthy process of re-establishing the connection when it times out. In the case of HTTP/1 or HTTP/2, multiple connections would need to be re-established, due to the way they leverage their established TCP connections for multiplexing. With QUIC, individual connections are identified using a unique connection identifier, which is attached to every packet sent, which means that all it takes to migrate to a new network path is to send a new packet, with the endpoints automatically associating the newly discovered remote socket (IP address and port) with the connection at essentially no overhead [6].

The unreliable datagrams and network path migration provided by QUIC make it a great replacement for currently used transport application protocols currently used by other VPN implementations, which is why this thesis aims to explore this use case and create a novel VPN implementation based on it.

1.2 VPN protocols

This section provides an overview of the most commonly used VPN protocols, including their functionality, strengths, and weaknesses, thus providing sufficient context for the following chapters of this thesis.

OpenVPN

OpenVPN is an open-source VPN solution built around the SSL/TLS protocol. It has a wide range of features and can be used in many types of deployment and is also one of the oldest, most commonly used and versatile VPN protocols today [8].

The security model of OpenVPN was initially based on the SSL protocol, later migrating to TLS when it replaced SSL.

OpenVPN was initially released in 2001 [9], with the idea behind it being: “Starting with the fundamental premise that complexity is the enemy of security, OpenVPN offers a cost-effective, lightweight alternative to other VPN technologies that is well-adapted for the SME and enterprise markets” [8]. This statement made sense at the time of initial release, when the only other viable alternative was the relatively more complex IPsec protocol suite. However, this changed over time - as more and more features were added, the OpenVPN source code became more lengthy and complex, totalling about 86,000 lines of C code [10].

All this accumulated complexity eventually led to OpenVPN being overly complex and difficult to maintain. This leads not only to increased difficulty auditing the code base, but also to subtle bugs arising either from unforseen side effects of changes or the increased difficulty of effective code review [11].

WireGuard

WireGuard is an open-source VPN protocol based on the ethos of simplicity and performance. It “aims to replace both IPsec for most use cases, as well as popular user space and/or TLS-based solutions like OpenVPN, while being more secure, more performant, and easier to use” [12].

The WireGuard security model was created from the ground up, with a single modern authenticated encryption scheme, ChaCha20Poly1305 (a relatively new authenticated symmetric encryption scheme with ChaCha20, a stream cipher, at its heart [13]), and a novel approach to session management and endpoint authentication. In short, this model allows WireGuard to have a fast 1-RTT handshake, low latency, and high throughput – all desirable properties for a modern VPN protocol. The entire protocol has gone through formal verification using Tamarin (a tool used for formal verification of security protocols using falsification and unbounded verification in the symbolic model [14]) and numerous security analyses performed by third parties, proving that the protocol is not only simple and performant, but also secure [15, 16].

Similarly to OpenVPN, WireGuard can be deployed as a client-to-site or site-to-site VPN. Unlike OpenVPN, however, these deployments are virtually identical, due to WireGuard’s symmetric design. This means that all endpoints, called *peers*, are equal in function and responsibilities, leading to greater versatility without the need for additional configuration.

1.3 Objectives

As mentioned in the previous sections, QUIC provides great connection properties and features, which could be leveraged by a VPN implementation. As of writing this thesis, there have been no attempts at creating an open-source VPN implementation based on QUIC and exploring its performance and security, which is why the objective of this thesis is to evaluate QUIC's feasibility in the VPN space by exploring existing VPN implementations, designing a novel VPN implementation based on the QUIC protocol, and then evaluating the performance and security of the created implementation by comparing it with some well-established VPNs, such as OpenVPN and WireGuard.

Chapter 2

Analysis

This chapter focusses on an analysis of the process of creating a novel VPN implementation, to give the reader a better understanding of the architecture and implementation choices made during the development of the implementation presented in this thesis, further referred to as Quincy[2].

2.1 VPN design

While it might seem that the currently used VPN implementations, such as OpenVPN and WireGuard, might work very similarly, their design varies greatly. This variance is due to different design philosophies and the vast sets of features VPNs can provide, such as different modes of operation (client-to-site, site-to-site, peer-to-peer), authentication schemes (basic authentication, PKI-based, TLS-based), policy-based routing, tunnel address assignment schemes, etc. Since almost all of these features come at a significant cost of code/implementation complexity and attack surface, choosing the correct feature set for given implementation goals is of utmost importance.

To give an example, OpenVPN supports client-to-site and site-to-site modes of operation, both based on the client-to-site implementation, reducing the needed complexity. It supports basic authentication, with support for multiple authentication backends such as a local SQLite database, LDAP and RADIUS, optional client certificate verification, and offers a DHCP-like tunnel address assignment [17].

In contrast, WireGuard chooses a more simplistic, yet generalised approach. Instead of distinguishing between different types of endpoints (client and server in the case of OpenVPN), the peer-to-peer model was chosen, providing decentralised deployment configurations. Instead of using basic authentication, WireGuard chooses to use the underlying public-key infrastructure on which the protocol relies to secure the packets to also serve as the authentication layer. This greatly reduced the complexity of the implementation, fitting under 4000 lines of code, compared to almost 90000 lines of code needed by OpenVPN [12].

[2]a word play on the QUIC protocol, inspired by the race of so-called Quincies in a popular animated series – Bleach

Modes of operation

The mode of operation of a VPN implementation is highly dependent on the underlying protocol used for the authentication and transport of packets between endpoints. For VPNs relying on TLS or TLS-like protocols, with a strict distinction between client and server endpoints, the client-to-site mode of operation is the most preferred, since it takes advantage of the underlying protocol, without the need for additional handling or added complexity, compared to the peer-to-peer mode of operation.

The most explored and popular modes of operations currently used are:

client-to-site A mode of operation which distinguishes between client and server endpoints, usually deployed in a star topology – with multiple clients connecting to a singular server and all traffic between clients being transferred through the server. This is the mode of operation used by many VPN implementations, most notably OpenVPN.

site-to-site A mode of operation in which two or more endpoints connect multiple existing networks, providing a secure tunnel between them. This mode of operation can be emulated with relative ease using the client-to-site mode, reducing the need for additional code complexity.

peer-to-peer A mode of operation in which two or more endpoints form a partial mesh network without distinguishing the importance of individual peers. This mode of operation has the advantage of being more resistant to failures of individual endpoints, which in other modes can cause the entire topology to fall apart, and is currently used by WireGuard.

Authentication schemes

The authentication schemes provided by a VPN form one of the important pillars of its security, as they usually serve as one of the first layers of defence against attacks and unwarranted access. These methods encompass a variety of techniques to verify the identities of users and devices seeking access to the VPN. They range from basic username-password schemes to advanced multi-factor authentication systems, and the chosen method significantly impacts the overall security stance of the VPN setup. The chosen method is highly dependent on other design choices and is one of the main influences on the perceived experience of both users and administrators. Supporting multiple authentication schemes can also provide better versatility for a wide range of deployments, which is an ethos used by OpenVPN. As stated previously, it provides multiple authentication backends, all based on the username-password scheme, only with a different underlying storage of user credentials. This allows OpenVPN to be used in a wide range of deployments, especially on the business side, where authentication against a RADIUS server or an LDAP endpoint might be highly preferred. On the other hand, WireGuard chooses a more simplistic approach, using *only* public key cryptography for user/endpoint authentication. This simplifies the code base, at the cost of lesser versatility, especially in business deployments with complicated user management.

Tunnel address assignment schemes

Most currently used VPN implementations use a mechanism to assign tunnel addresses to endpoints. Some VPNs, such as OpenVPN, employ a DHCP-like address assignment scheme, which allows for both automatic assignment from a pool of available addresses, or a static assignment per client set-up on the server endpoint. In contrast, WireGuard employs a different scheme using `AllowedIPs`, which define a subset of addresses that individual peers serve. If we have a peer A, whose `AllowedIPs` for peer B is set to `10.0.0.0/24`, then all traffic on peer A, which is routed to its active WireGuard TUN interface, with the destination address in the `10.0.0.0/24` subnet will get routed to peer B. Peers need to have their addresses set manually, with no option for dynamic assignment, which might make user management in a typical deployment more difficult [12].

The choice between different address assignment schemes depends heavily on what tooling is provided to users and administrators and what deployments or modes of operation will be supported by the VPN. For site-to-site or peer-to-peer deployments, a static assignment approach, such as what is provided by WireGuard, might be preferred, where as for client-to-site deployments, the dynamic assignment approach will provide a better user and administrator experience.

2.2 Functional requirements

This section focusses on transforming the requirements in the attached thesis assignment into actionable *functional requirements*. There are 3 general requirements stated by the assignment: “Using a QUIC library of your choice, design and implement a viable VPN protocol [...]”, “Establishing a tunnel using basic authentication (username and password)” and “Routing traffic through the tunnel using TUN/TAP interface”. Since these requirements are somewhat vague, they were transformed into the following actionable functional requirements, with *F1* corresponding to the first requirement, *F2* and *F3* corresponding to the second requirement, and *F4* and *F5* corresponding to the third requirement.

F1 The system shall use QUIC as the underlying transport protocol, relying on it for the encryption and transfer of tunnelled traffic.

F2 The system shall provide a means of basic authentication to verify the identity of users.

F3 The system shall allow only authenticated users to establish a secure tunnel to the server.

F4 The system shall provide a TUN[3] interface responsible for routing traffic through the established tunnel.

F5 The system shall use a dynamic provisioning system to assign IP addresses to the created TUN interfaces.

[3]since TAP interfaces are mostly obsolete and not used in modern VPN deployments anymore, only TUN interfaces are to be supported by the implementation

Chapter **3**

Realisation

This chapter focusses on one of the main goals of this thesis – creating a minimum viable product VPN based on the QUIC protocol called Quincy. In the previous chapter, 5 functional requirements, *F1* through *F5* were created, which specify the characteristics of such an implementation.

3.1 Development process

The overall architecture and development of Quincy were approached in an iterative matter, beginning with multiple proof-of-concepts and eventually leading to the creation of a usable architecture and the resulting minimum viable product.

Language and library choice

The first chosen step in this iterative process was the selection of a language and a library on which the final product should be based. This was due to the author's belief that the programming experience and tooling of a chosen language and library are not only important for a smooth development process, but also serve as a good starting point for the overall maintainability and security of the solution. Since a VPN implementation benefits from near-direct access to low-level system primitives, due to the importance of performance and low latency of operations, the chosen language needed to be relatively low-level itself, without the added impact of garbage collectors (such as Go, C# or Java), overhead from being an interpreted language (such as Python or Ruby), etc. This ultimately resulted in a possible set of languages being limited to C, C++, Zig, or Rust. In the end, Rust was chosen not only because of its advanced tooling, zero-cost abstractions, or memory safety guarantees [18], but also because of the author's familiarity with the language.

In the Rust ecosystem, there are a number of potential QUIC libraries, most of which are backed by either large open-source communities or by the industry, the most used and recognised ones being:

quinn [19] A high-level asynchronous QUIC library, supported by multiple widely known Rust open-source contributors. It uses a third-party-audited pure-Rust implementation of the TLS protocol, *rustls* [20, 21] and provides a very user-friendly API, greatly utilising many of the zero-cost abstractions provided by the Rust language.

s2n-quic [22] A mid-level[4] asynchronous QUIC library created by Amazon for internal use, published under the Apache-2.0 licence.

quiche [23] A low-level synchronous QUIC library created by Cloudflare for their HTTP/3 edge-network support. This library forces the user to provide the entire service runtime, exposing only the primitives required for processing QUIC traffic.

All three of these libraries were used in a simple proof-of-concept application, aimed at evaluating their developer experience and overall ergonomy. The design was a simple client and server application, where data was sent bidirectionally. Due to some design choices of Rust, such as borrow checking, issues arised from using s2n-quic and quiche for simultaneous sending and receiving of QUIC datagrams, which were deemed too difficult to overcome at the time. Quinn, on the other hand, did not experience the same issues. This meant that although both quiche and s2n-quic are backed by large companies and, as such, are unlikely to be left unmaintained due to being a part of their infrastructure, neither provided a sufficiently stable developer-friendly API at the time of writing this thesis. The community-backed quinn provided a much easier-to-use and stable API, which, coupled with the audited TLS library used as the cryptographic back-end, resulted in it being chosen for the implementation.

Since quinn's API is written in async Rust, with support for multiple async runtimes, and since Rust does not provide a default async runtime in the standard library [24], this choice of the QUIC library forced the choice of an async runtime and, more importantly, influenced the choice of the library used for interacting with TUN interfaces. Due to the limited availability of TUN libraries in Rust, with only one of them supporting all major platforms (Windows, Linux, MacOS), *tun2* was chosen [25], providing both sync and async high-level APIs for interacting with TUN interfaces. The only async runtime supported by this library is *tokio* [26], the most popular multithreaded work-stealing async runtime/executor.

To sum up, the following technologies and libraries were chosen:

Rust as the programming language

tokio as the async runtime/executor

quinn as the QUIC library

tun2 as the TUN interface library

Proof of concept

With the technologies and libraries selected, an iterative development process began with a proof-of-concept application, utilising both the QUIC and TUN

[4]an implementation, which does not provide neither low-level primitives, nor a user-friendly high-level API

interfaces. The idea behind making a first proof-of-concept application with all the required components was to analyse the interoperability of individual components and to scout for any potential pitfalls. It also served as the first informal evaluation of the potential throughput and latency of the solution, which are important factors for a VPN.

The initial proof-of-concept was a simple relay between a TUN interface and a QUIC tunnel, mostly agnostic in the functionality of endpoints, apart from some required distinctions due to the architecture of the QUIC protocol (such as the inherited TLS connection client-server separation), without any authentication modules. At this point, some initial observations were made – while QUIC streams initially seemed to provide a good interface for a VPN use case, the TCP-like reliability functionality interfered with the performance of the solution in initial crude benchmarks. This shifted the attention to QUIC datagrams, which provide a very simple way to send encrypted data in unreliable datagrams, without any packet acknowledgements and interference between stacked reliability layers. When using QUIC datagrams, the benchmark measurements improved noticeably, resulting in the choice to use them for all tunnelled traffic.

The proof-of-concept also laid a foundation of the final task model, e.g. what tasks are spawned in each of the components and for what purpose, mostly due to certain performance considerations. Tasks in tokio act as lightweight “threads” – where you would use a thread for a processing task in a standard thread model, you spawn a tokio task instead. Since the executor is multithreaded, multiple tasks can be run in parallel by different threads. Additionally, the executor is *work-stealing*, meaning that if a thread “runs out” of tasks to run, it will “steal” tasks from a different thread, busy running a different task. Through a couple of iterations, the conclusion to the task model is to have 3 tasks running per-connection:

1. a task to receive packets from the TUN interface and send them to the connected QUIC endpoint as a datagram
2. a task to receive packets from the QUIC endpoint and put them into a *queue*
3. a task to take tasks from the *queue* and send them to the TUN interface

The reason for splitting packet processing in the QUIC to TUN interface pipeline into 2 tasks was made mostly due to a number of empirical measurements; the task dedicated to receiving a packet from the TUN interface and sending it to the QUIC endpoint spent most of its time waiting for a packet to be available on the TUN interface. Due to design choices in quinn, sending a datagram is non-blocking and, generally speaking, instant[5]. The potential task dedicated to receiving QUIC datagrams from the endpoint and sending them to the TUN interface spent most of its time on the TUN interface `write` calls. This caused some datagrams to be dropped due to limited buffer space of the QUIC endpoint, which, albeit configurable, cannot be infinite. When divided into two tasks, each handling its “side” of the relay, the number of dropped packets was noticeably lower.

[5]For more information, refer to the relevant issue (https://github.com/quinn-rs/quinn/issues/1738) documenting this behaviour.

Authentication system

The development of the authentication system was the most intensive iterative process in the overall development of Quincy. This was mainly due to the existence of multiple possible and equally valid approaches, each with its own set of advantages and disadvantages.

The first version of an authentication system was very similar to how many modern web frameworks handle authentication – a basic username and password authentication with a session cookie. This session cookie was not sent with every packet, but was periodically sent by the client. This approach seemed to provide more security at the time, but was quickly found redundant. Since Quincy uses QUIC in a connection-orientated way and since the TLS version 1.3 it uses is resistant to replay attacks if 0-RTT is disabled (which is true for Quincy) [27], the session cookie does not provide any additional security to the established session, making it redundant.

An important part of the multiple iterations of the authentication system was to prevent common pitfalls, such as not having proper authentication guards in the relay functions, and hence allowing unauthorised users to freely communicate over the secure tunnel established by Quincy. The initial solution – simple checks in the relay methods – was error-prone, and one of the iterations even contained a bug, which allowed the exact situation described above: unauthorised access to the established tunnel. This was eventually solved by partly leveraging the type system of Rust and making it impossible to relay packets for unauthorised connections.

A final step in the development of the authentication system was to make it modular to allow for easy extensibility in the future. The current implementation of a file-based basic username and password authentication is sufficient for simple use cases, but if Quincy were to be used in more professional deployments, support for database-, LDAP- or RADIUS-backed authentication would be very likely required. This modularity was achieved by dynamically-dispatched traits in Rust, essentially enabling countless authentication methods to be added, by implementing 2 single-function interfaces – one for client-side authentication payload generation and one for server-side payload verification – as shown in 3.1.

```rust
#[async_trait]
pub trait ServerAuthenticator: Send + Sync {
    async fn authenticate_user(
        &self,
        address_pool: &AddressPool,
        authentication_payload: Value,
    ) -> Result<(String, IpNet)>;
}

#[async_trait]
pub trait ClientAuthenticator: Send + Sync {
    async fn generate_payload(&self) -> Result<Value>;
}
```

Listing 3.1: Authentication module interfaces

3.2 Architecture

This section focusses on the final architecture, created in the iterative process described in the previous section.

Components

Quincy is composed of a number of components to separate logic, such as client/server functionality, interface operations, authentication, and address assignment.

Client A client-side component encapsulating client authentication and a packet relay for transferring traffic between its TUN interface and its QUIC endpoint.

Server A server-side component responsible for handling incoming connections and managing traffic flow between its TUN interface and multiple client QUIC endpoints.

Interface An abstraction over TUN network devices, allowing for different implementations/libraries in different operating conditions (different platforms, etc.).

Authentication Client A client-side component responsible for client authentication with a modular payload generator, matching the back-end choice of the authentication server.

Authentication Server A server-side component responsible for handling client authentication, with modular back-end support (currently only file-backed authentication) and automatic client tunnel address assignment from a pool of available addresses.

The interactions of these components are shown in the diagram 3.1.

Connection flow

Below is a more detailed step-by-step description of the connection flow, as illustrated by 3.2:

1. The client initiates a connection to the server.
2. The server accepts the connection to its socket and waits for the client to authenticate.
3. The client opens a new QUIC stream and sends its credentials to the server.
4. The server accepts the new QUIC stream, receives the client's credentials, and, if authentication is successful, sends the assigned tunnel IP address to the client using the same stream.
5. The client receives its assigned tunnel IP address, creates a TUN interface, configures it to use the received address, and closes the authentication stream.

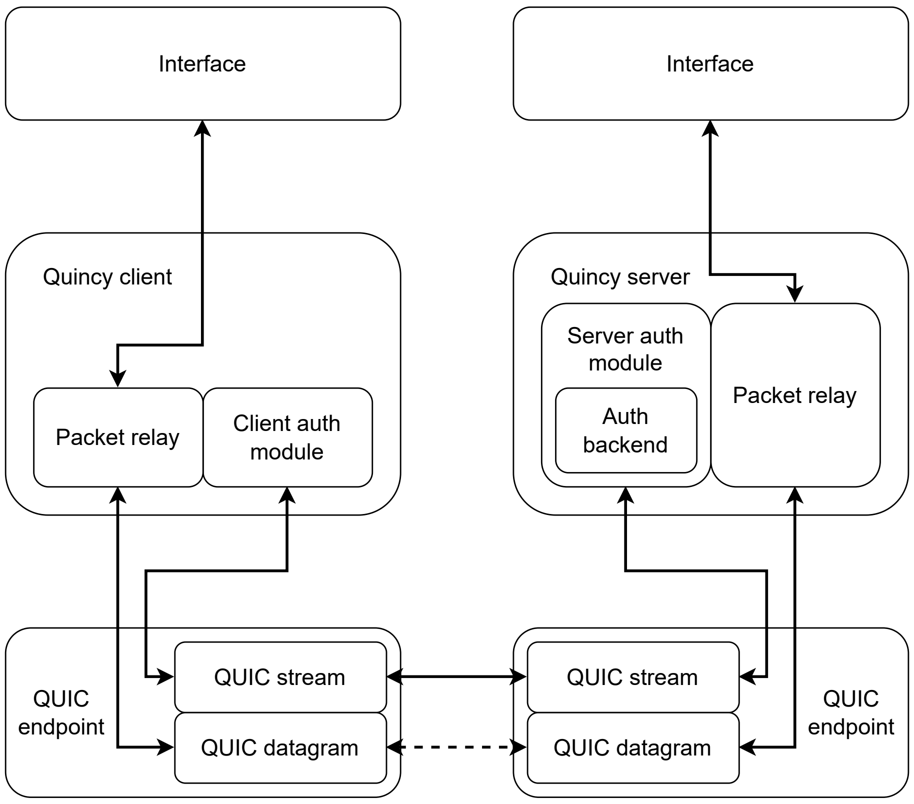

Figure 3.1: Component diagram

6. Both the client and the server begin to relay packets between the TUN interface and the other endpoint using QUIC datagrams.

An important part of the connection flow is the authentication process. As can be seen in 3.2, the process is client-proactive, which means that the client initiates the authentication process, with the server awaiting payload from the client, timing out if it is not provided within the configured authentication timeout window. When a payload is received, it is checked using the chosen authenticator module (only the `UsersFile` authenticator is supported at the time of writing this thesis). If the payload is valid, the server sends the needed connection parameters to the client and both sides start relaying packets using the established QUIC connection.

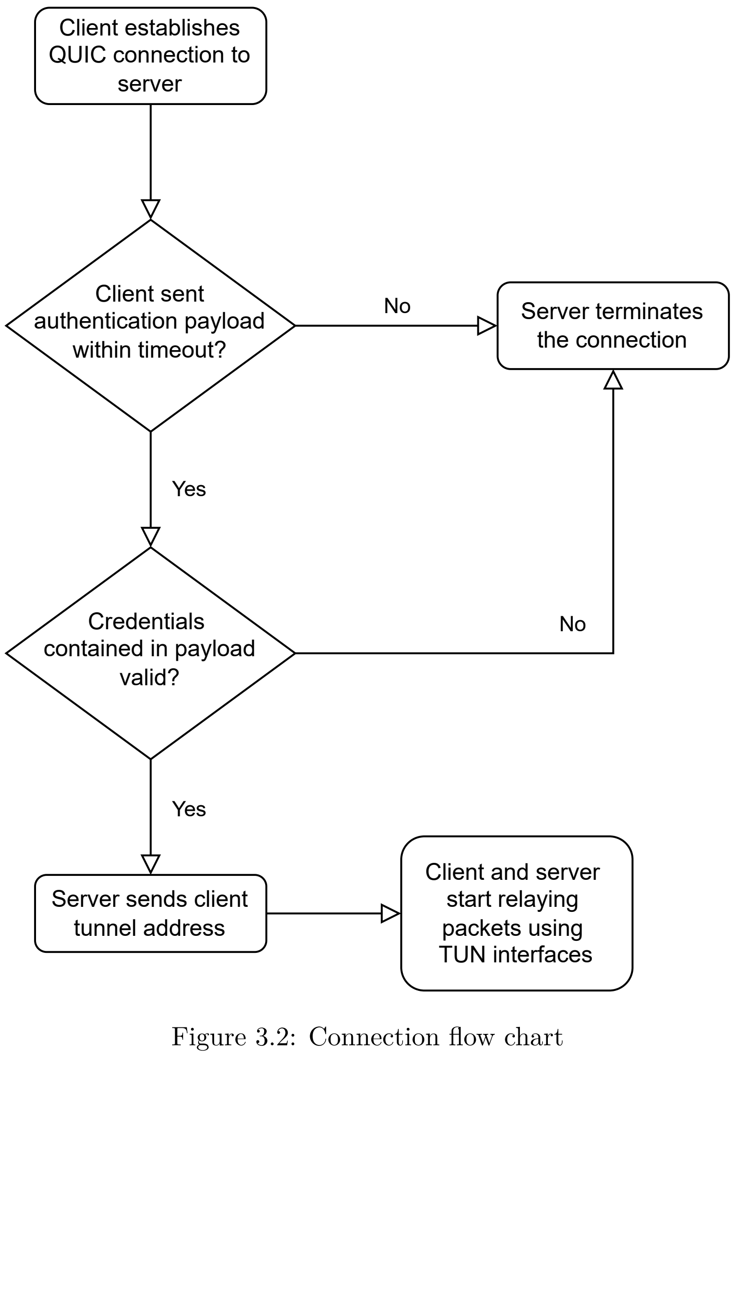
Client establishes QUIC connection to server
Client sent authentication payload within timeout?
No
Server terminates the connection
Yes
Credentials contained in payload valid?
No
Yes
Server sends client tunnel address
Client and server start relaying packets using TUN interfaces

Figure 3.2: Connection flow chart

3.3 Functional requirements

In the previous chapter, Analysis, functional requirements *F1* through *F5* were created from the thesis assignment:

F1 The system shall use QUIC as the underlying transport protocol, relying on it for the encryption and transfer of tunnelled traffic.

F2 The system shall provide a means of basic authentication to verify the identity of users.

F3 The system shall allow only authenticated users to establish a secure tunnel to the server.

F4 The system shall provide a TUN interface responsible for routing traffic through the established tunnel.

F5 The system shall use a dynamic provisioning system to assign IP addresses to the created TUN interfaces.

The functional requirement *F1* was achieved by leveraging the existing *quinn* library, which provides an async Rust API for QUIC.

The functional requirement *F2* was achieved using the modular authentication module for the client and the server.

The functional requirement *F3* was achieved by leveraging the Rust type system, essentially not allowing unauthenticated connections to relay any traffic.

The functional requirement *F4* was achieved using the existing *tun2* library, which provides a TUN interface API for Rust.

The functional requirement *F5* was achieved by implementing a dynamic address assignment component in the server authentication module.

3.4 Security considerations

With any VPN implementation, it is important for the security aspects to be a large part of the design from day 1 to prevent security issues and privacy concerns from ever arising, needing to be patched and stitched onto an already existing insecure implementation.

To achieve this goal, Quincy attempts to delegate a large part of its security to the underlying protocol, QUIC, and its reputable implementation, quinn, to minimise remaining attack surface. However, certain parts of Quincy, such as authentication, needed to be designed from the ground up, to be as secure as possible, while remaining user-friendly.

Protocol

As mentioned earlier, quinn is responsible for the implementation of the QUIC protocol used by Quincy. It is split into a number of crates[6]:

quinn-proto implementing the entirety of the QUIC protocol as a deterministic state machine,

[6]name for Rust packages

quinn-udp implementing specifically optimized UDP sockets,

quinn using *quinn-proto* and *quinn-udp* to provide a pure-Rust async API and runtime.

Quinn supports modular cryptography modules, making it usable with different cryptographic implementations of the TLS protocol. The default implementation, and the implementation used by Quincy, is *rustls*. As previously mentioned, this library has gone through a third-party security audit [21], finding no issues and overall very high code quality. This makes a strong argument for the safety of the TLS implementation itself. Additionally, since Quincy specifically only uses TLS 1.3 with the strongest cipher suites (`TLS13_AES_256_GCM_SHA384` and `TLS13_CHACHA20_POLY1305_SHA256`) [28], any potential attack surface is greatly reduced. Furthermore, known issues with the soundness of TLS 1.3, such as the security issues found with the 0-RTT functionality [27], are handled by rustls either in the form of sound defaults, or by explicitly not being implemented.

Unfortunately, the same cannot be said for quinn, which, albeit an active project with reputable maintainers, lacks any third-party security audits. Although Rust provides a good guarantee on *memory safety* of the software written in it, it only does so much to prevent logical errors. While a security audit of the library is outside of the scope of this thesis, it would be a welcome addition to the overall soundness of the QUIC ecosystem in Rust and as such would be a worthy future endeavour.

Authentication

The authentication system in Quincy is composed of 2 main components – the authentication client and server. These are modular and can possibly support numerous authentication schemes. The only scheme supported at the time of writing this thesis is the `UsersFile` scheme, which represents basic file-based authentication, which will be the focus of this subsection.

As described in the previous sections, the authentication process is client-proactive, which means that the server awaits client authentication. This makes the implementation much easier to implement, but presents a noticeable drawback. If a number of malicious clients connected and stalled the authentication process by not providing the authentication payload within the specified timeout, they could cause resource exhaustion on the server side if not considered carefully during the implementation. As of writing this thesis, the current implementation handles the process of authenticating connections in a separate task, in an effort to not block other parts of the connection flow, such as handling of new and closed connections. Although this greatly reduces the potential impact of such an attack, it does not eliminate it. If a sufficient number of connections were created in a sufficiently small time frame (less than the authentication timeout), the task executor could be slowed down by the large number of futures, increasing the latency and likely CPU and RAM usage of the server instance. With such a design, it is difficult to completely mitigate this issue, as some low-effort solutions, such as limiting the number of connections at one time, could lead to a similar Denial-of-Service scenario, in which the limit of connections would be exhausted, disallowing legitimate clients from connecting to the server. Other protocols, such as WireGuard, solve this issue by having authentication be part of the underlying protocol, making it impossible to establish a session with the server without providing valid credentials. WireGuard goes as far as to not respond to improperly authenticated packets during session establishment, effectively ignoring them without sending any response. This makes it more difficult to discover running instances of WireGuard, as well as preventing any Denial-of-Service attacks aimed at the application itself [12]. An investigation into possible mitigations of this issue presents a possible future endeavour and a continuation of this thesis.

The credentials are stored, on both sides of the connection, in regular text files. On the client side, the credentials are stored in plain text in the client configuration file, which requires users to properly set the file ownership and permissions to avoid leaking their credentials. This is a similar setup to WireGuard, which stores private keys in plain-text form in the tunnel configuration files. An option to load the credentials from a secure store, such as a system-wide key chain, would constitute a great improvement to their confidentiality. On the server side, credentials are stored in a format very similar to `/etc/shadow` on Unix systems, which means that the username is stored in plain text, while the password is stored as a salted Argon2 hash. The Argon2id mode is used to prevent side-channel attacks and to improve resistance against GPU cracking attacks, with secure default parameters [29, 30].

The session created by the authentication process is not kept by using session cookies or similar methods, relying solely on the protocol's session keeping. This is so that there is no additional session keeping, which could possibly increase the overhead of the protocol, delegating the protection against replay attacks and similar exploits to the underlying QUIC protocol. A possible improvement could be made by integrating the authentication process into the underlying protocol as an extension, using the *quinn-proto* library.

Summary

In summary, while taking reasonable security precautions, some sacrifices have been made to keep the scope of this implementation within the limits of this thesis. A security audit of the QUIC implementation – quinn – would provide a much needed verification of the soundness of its implementation of the QUIC protocol, further improving the security guarantees of Quincy. Moreover, there is room for improvements in the authentication modules, which are somewhat vulnerable to Denial-of-Service attacks and credential leaks arising from misconfigurations by users on the client side.

Chapter 4

Evaluation

This chapter delves into the evaluation of the VPN implementation described in this thesis, Quincy, and compares it with other widely used protocols, such as OpenVPN and WireGuard.

4.1 Methodology

All VPN protocols have been evaluated in numerous simulated scenarios, ranging from environments with zero latency and jumbo packets enabled, to real-life environments with simulated latency, packet loss, and packets of standard size.

To measure throughput, latency, and stability, a benchmarking suite was created, using the *iperf3* [31] benchmarking tool. iperf3 was run in both directions (client-to-server and server-to-client) for 30 seconds, with a sampling interval of 100 milliseconds. This suite is a collection of Bash scripts, whose objective was to:

- prepare the environment – create required folders, compile Quincy
- start of one the VPNs and run the iperf3 benchmarks
- collect the data and save it to a remote machine

This was to ensure effortless scalability and flexibility of the suite, allowing for testing in countless different scenarios. The benchmarks were run sequentially and on virtual machines with non-overlapping CPU cores to prevent interference.

In order to cover a wide variety of environments, the evaluation suite was run with all possible combinations of 3 main parameters: simulated latency, simulated packet loss, and MTU size. Both latency and packet loss were simulated using the `tc` [32] Linux command-line utility for network interfaces, MTU was set using the `ip` command and in the configuration files for each of the VPNs.

The resulting data are processed using a Jupyter notebook to produce graphs and other useful statistics. The entire benchmarking suite, together with all the collected data, is present in the attachments.

Latency

Latency is one of the most noticeable and influential properties of a network connection, due to its overall effect on responsiveness and bandwidth when using protocols such as TCP. With this in mind, three values have been picked to reflect the most common types of connections, ranging from personal to commercial/academic environments:

- 0 ms - CAN, or intra-datacenter connections
- 10 ms - MAN, or intra-continental WAN
- 150 ms - inter-continental WAN

Packet loss

Packet loss is another very important property of a connection, as with many protocols (especially TCP), it can lead to a noticeable decrease in responsiveness and bandwidth and when combined with higher latency, it can be catastrophic for user experience and perceived network quality. As such, it is important for a VPN to be able to withstand it to a reasonable degree. Since packet loss in modern networks is quite low and is usually guaranteed by Internet service providers to be below a certain threshold, only the following values have been chosen:

- 0 % - packet loss in most environments
- 1 % - packet loss in long-range communication (inter-continental connections, etc.)
- 5 % - packet loss in unrealiable environments (Wi-Fi/cellular connections, partial outages, etc.)

MTU

The Maximum Transmission Unit (MTU) of a network connection is also an important parameter, as it often is directly proportional to bandwidth (when supported by all devices on the network path). This can happen due to a number of factors, one of which is critical for most VPN protocols – the cost of syscalls. As described in the previous chapter, syscalls can be quite expensive to call, as they require a switch from user-space to kernel-space. Unfortunately, syscalls are, for the most part, the only way to write/read from a TUN interface from a user-space application, such as a VPN client. This means that for systems where Generic Segmentation Offload (GSO) and Generic Receive Offload (GRO) are generally not available, frequent syscalls are going to be one of the most significant bottlenecks of most VPN implementations. Due to this fact, three different MTU sizes have been chosen, to investigate this bottleneck for all evaluated VPN protocols and to provide valuable information about a wider variety of network environments:

- 1300 B - MTU value representing degraded/old connections
- 1500 B - the most common MTU value seen on the Internet as of writing
- 6000 B - MTU used in intra-datacenter communication for bandwidth-dependent applications

4.2 Environment

The evaluation environment was composed of 2 virtual machines running on a type-2 hypervisor. The relevant hypervisor specifications:

- CPU: AMD Ryzen 9 3900 XT 12C/24T (x86_64)
- RAM: 64 GB DDR4 3200 MHz ECC U-DIMM
- Operating system: unRAID 6.12.4
- Hypervisor software: QEMU/KVM (libvirtd 8.7.0)

Both virtual machines shared the same specifications and configuration:

- CPU: 6 dedicated vCPU cores (host pass-through)
- RAM: 4 GB RAM
- Operating system: Rocky Linux 9.3
- Network interface: virtio-based, 30 Gbps

Quincy

The configuration used by Quincy in the evaluation environment was very similar to the default configuration, with one notable change – the MTU being set dynamically by the evaluation script. The complete client configuration (4.1) and the server configuration (4.2) can be found below.

```
connection_string = "quincy:55555"

[authentication]
username = "test"
password = "test"
trusted_certificates = ["examples/cert/ca_cert.pem"]

[connection]
mtu = {mtu}

[log]
level = "info"
```

Listing 4.1: Quincy client configuration

```
[tunnels.tun0]
name = "tun0"
certificate_file = "examples/cert/server_cert.pem"
certificate_key_file = "examples/cert/server_key.pem"
address_tunnel = "10.0.0.1"
address_mask = "255.255.255.0"

[authentication]
users_file = "examples/users"
```

```

[connection]
mtu = {mtu}

[log]
level = "info"
```

Listing 4.2: Quincy server configuration

WireGuard

As with the Quincy configuration, WireGuard configuration was very close to the default configuration, with generated private and public keys and MTU set dynamically by the evaluation suite. The complete client (4.3) and server (4.4) configurations can be found below.

```
[Interface]
PrivateKey = cAUIq3HkJvW6PmOhqtiBfgFrfLfzI5nY1wAZK/O592c
    =
Address = 10.1.0.2/32
MTU = {mtu}

[Peer]
PublicKey = Dc9BDk3izwdM6vMAVfbBeP4rT6ASfiSi2Fx71qREahk=
Endpoint = quincy:51820
AllowedIPs = 10.1.0.1/24
```

Listing 4.3: WireGuard client configuration

```
[Interface]
PrivateKey = mMG3iQlgWJcca+093MkfXJMHuMfIIp0oA5IHQR9pFHQ
    =
Address = 10.1.0.1/24
ListenPort = 51820
MTU = {mtu}

[Peer]
PublicKey = CyqH3wfokaGTJfToqhcb+Q+dfY6+jeRyhDyldrlKWmE=
AllowedIPs = 10.1.0.2/32
```

Listing 4.4: WireGuard server configuration

OpenVPN

The OpenVPN configuration was made to be as simple and as performant as possible. This was achieved by using UDP as the transfer protocol instead of TCP, using the `fast-io` configuration option and by selecting the `AES-256-GCM` cipher, which on the selected evaluation hardware provided the best throughput. As with the other VPNs, the MTU was set dynamically by the evaluation suite. The complete client (4.5) and server (4.6) configuration can be found below.

```
client

proto udp
dev tun
tun-mtu {mtu}
mssfix {mtu - 40}
remote quincy 1194
fast-io

cipher AES-256-GCM
auth SHA256
tls-cipher TLS-DHE-RSA-WITH-AES-256-GCM-SHA384

ca /etc/openvpn/pki/ca.crt
cert /etc/openvpn/pki/client.crt
key /etc/openvpn/pki/client.key

keepalive 10 120
```

Listing 4.5: OpenVPN client configuration

```
server 10.2.0.0 255.255.255.0

proto udp
port 1194
dev tun
tun-mtu {mtu}
mssfix {mtu - 40}
fast-io

cipher AES-256-GCM
auth SHA256
tls-cipher TLS-DHE-RSA-WITH-AES-256-GCM-SHA384

ca /etc/openvpn/pki/ca.crt
cert /etc/openvpn/pki/server.crt
key /etc/openvpn/pki/server.key
dh /etc/openvpn/pki/dh.pem

tls-server
client-to-client

sndbuf 512000
rcvbuf 512000
push "sndbuf 512000"
push "rcvbuf 512000"

keepalive 10 120
```

Listing 4.6: OpenVPN server configuration

4.3 Network behaviour evaluation

In this section, different network configurations are grouped according to the behaviour of the evaluated VPNs. This allows easier comparison and interpretation of the results.

4.3.1 Ideal network conditions

The first observed behavioural group represents ideal network conditions - very low latency, no packet loss, and a wide range of MTU sizes. Under such conditions, the implementation limits of all tested VPNs can be observed.

In the first case, described by the interface configuration of 0 ms latency, 0 % packet loss, and 1500 B MTU, WireGuard has the highest throughput, Quincy lags noticeably behind, and OpenVPN has the lowest throughput (4.1, 4.2).

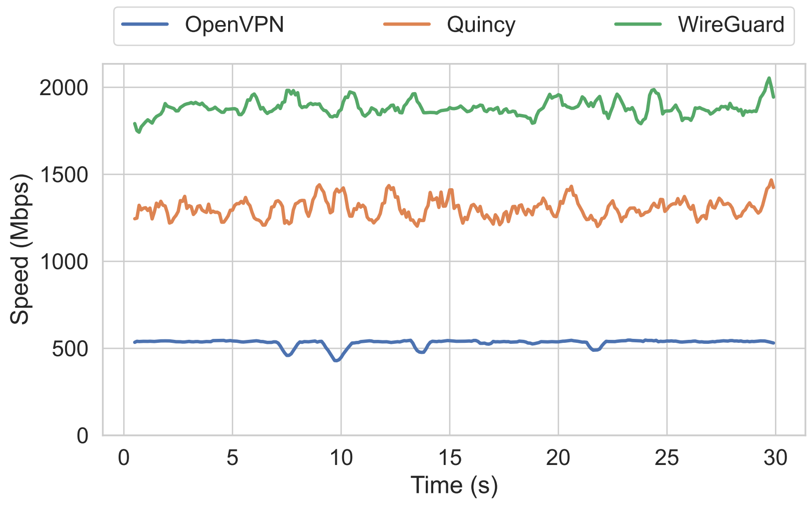

Figure 4.1: Download (0 ms latency, 0 % packet loss, 1500 B MTU)

Additionally, the throughput of both OpenVPN and Quincy is slightly asymmetrical – with server-to-client transfer being marginally faster, and client-to-server transfer being slower. In Quincy, and likely OpenVPN, this is due to differences of packet routing in the client-side and server-side implementations. WireGuard is, by design, symmetrical and does not make this distinction between two peers.

In terms of measured RTT, Quincy performed the best and had the lowest added latency of the measured VPNs, with WireGuard showing a slightly higher increase and OpenVPN adding the highest amount of latency (4.3).

In a very similar case, described by the interface configuration of 0 ms latency, 0 % packet loss, and 6000 B, overall relative throughput remains largely the same (4.4, 4.5), with the notable exception of Quincy throughput scaling.

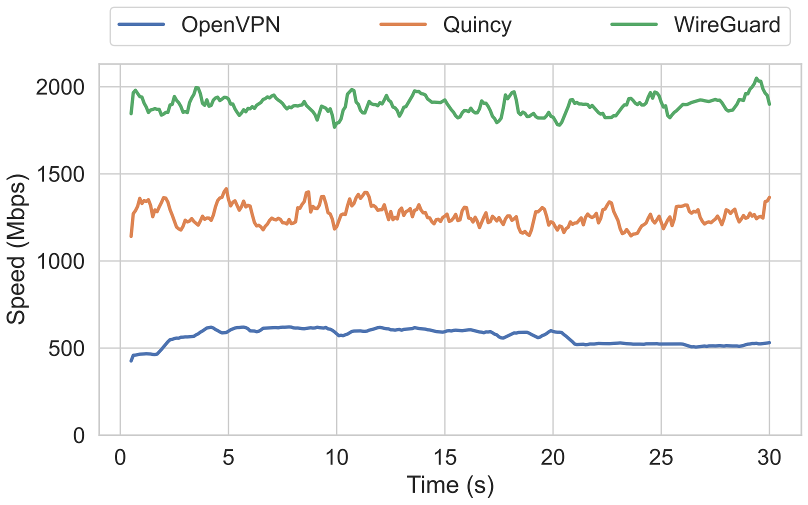

Figure 4.2: Upload (0 ms latency, 0 % packet loss, 1500 B MTU)

Both OpenVPN and WireGuard scale linearly with the MTU size – a 4x increase in MTU size leads to a roughly 4x increase in throughput. However, this is not the case for Quincy, whose throughput increases to only about three times the throughput at the MTU of 1500 B. This might be due to Quincy using a different runtime model (an async runtime with a user-space scheduler, instead of a thread-based model with the OS scheduler) and warrants further investigation. The trends in measured RTT are very similar to the previous case, with the WireGuard RTT stabilising and slightly decreasing (4.6).

Overall, under ideal network conditions, according to the measured data, Quincy heavily outperforms OpenVPN, and both Quincy and OpenVPN are outperformed by WireGuard. This is likely due to the overhead of having to make *syscalls* from user-space for each read/write operation on the TUN interface. As WireGuard is a kernel module, it can make these syscalls at a much lower, albeit not non-existent, cost. In summary, under these conditions, WireGuard provides the best characteristics in terms of throughput, with Quincy taking the lead in latency.

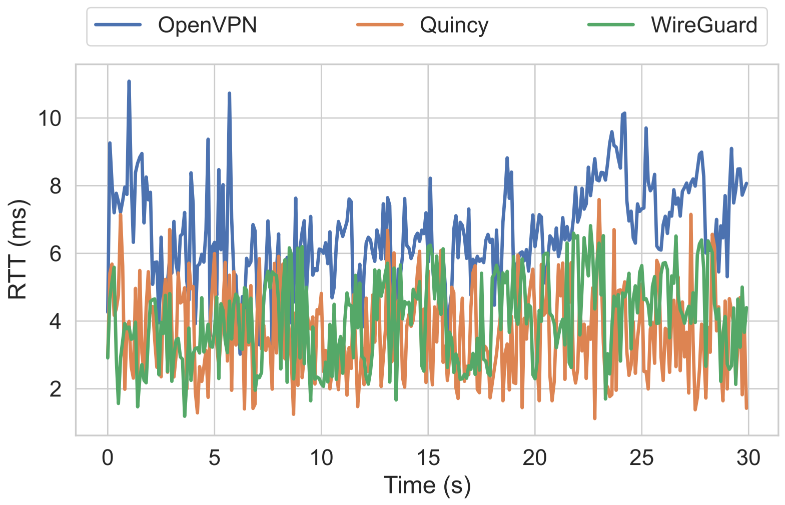

Figure 4.3: RTT (0 ms latency, 0 % packet loss, 1500 B MTU)

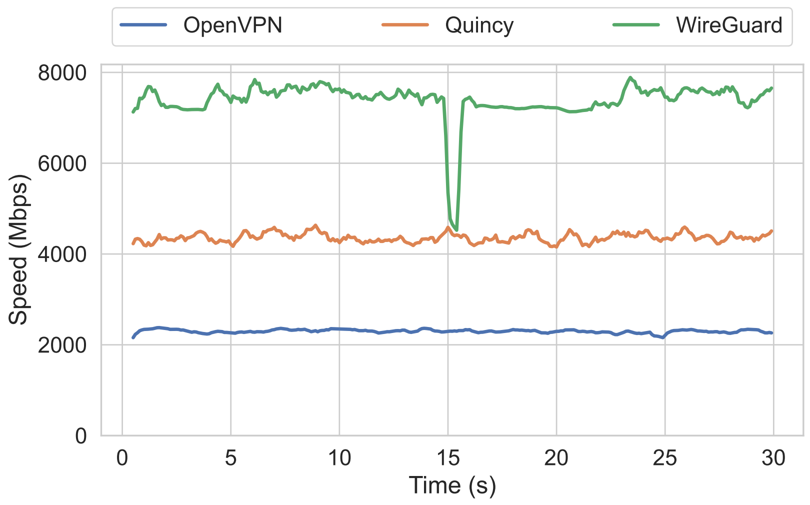

Figure 4.4: Download (0 ms latency, 0 % packet loss, 6000 B MTU)

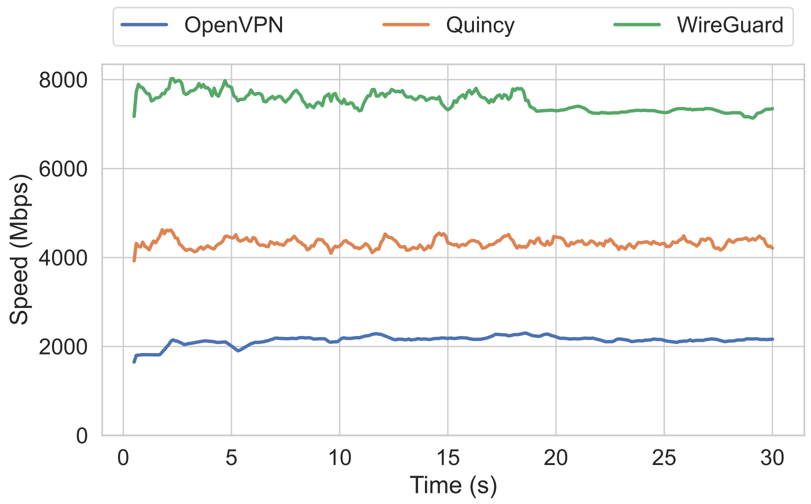

Figure 4.5: Upload (0 ms latency, 0 % packet loss, 6000 B MTU)

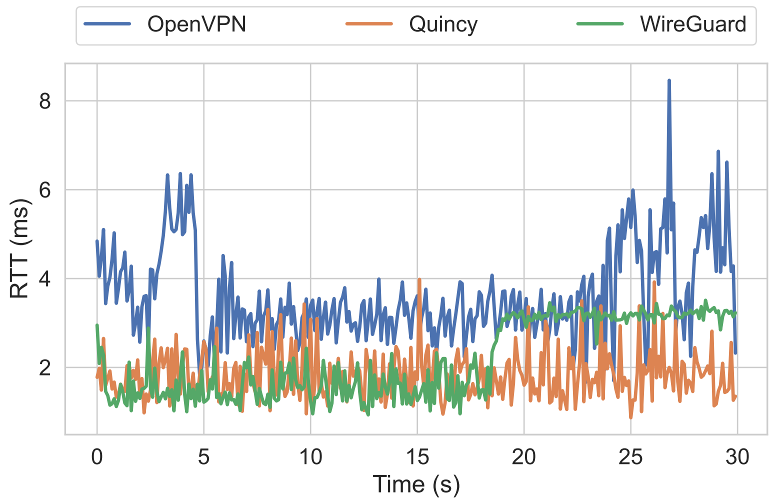

Figure 4.6: RTT (0 ms latency, 0 % packet loss, 6000 B MTU)

4.3.2 Common network conditions

The second observed behavioural group represents common network conditions, with relatively low latency, low packet loss, and standard MTU sizes ranging from 1300 to 1500 bytes. Such conditions are the most representative of real-world performance in the established evaluation suite.

In the first example of such conditions, 10 ms latency, 0 % packet loss, and 1500 B MTU, the observed behaviour is somewhat similar to ideal network conditions, with Quincy seeing higher overall throughput than WireGuard and OpenVPN underperforming significantly.

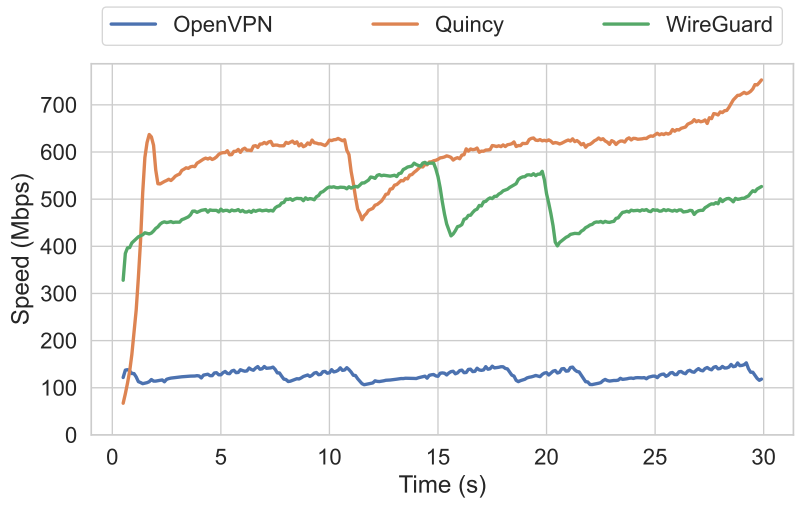

Figure 4.7: Download (10 ms latency, 0 % packet loss, 1500 B MTU)

All evaluated VPNs experience the "Saw-Tooth" behaviour inherent to the default configuration of TCP (4.7). This is due to the evaluation suite using iperf3 in the default TCP mode and provides a view of the properties of the VPN connection, as experienced by the application protocols that use it. Both Quincy and WireGuard display similar connection properties, with WireGuard possibly suffering from worse packet pacing, leading to marginally lower throughput.

In terms of observed latency (4.8), both Quincy and WireGuard add less than 1 ms of RTT, with WireGuard performing slightly better, increasing the latency less. Notably, Quincy experiences sudden latency spikes multiple times during the measurement, as can be seen on the histogram of its measured RTT (4.9). OpenVPN experiences worse performance, similar to the measured throughput, increasing the total RTT by up to 8 ms.

Another example of relatively common conditions, such as marginally unreliable (wireless) networks, is represented by the test parameters of 10 ms latency, 1 % packet loss, and 1500 B MTU. In these conditions, all VPNs behave somewhat similarly due to the inherent limitations of unreliable networks with measurable packet loss. However, Quincy seems to outperform both Wire-

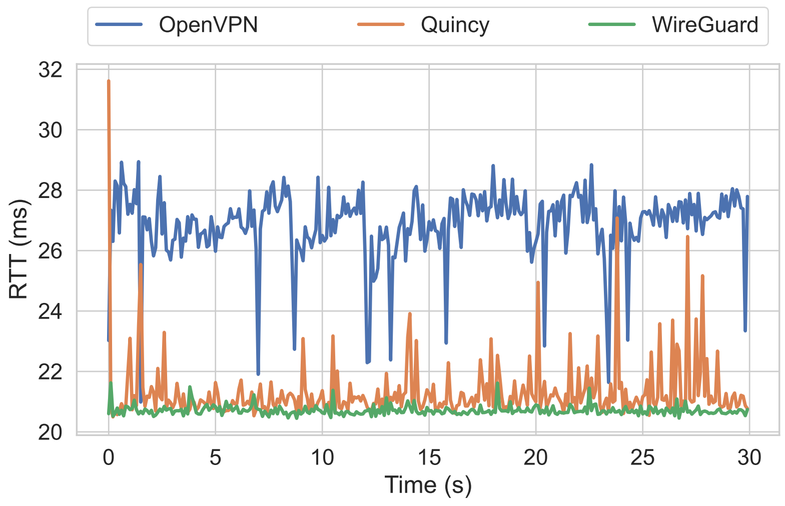

Figure 4.8: RTT (10 ms latency, 0 % packet loss, 1500 B MTU)

Guard and OpenVPN in terms of throughput by a slight margin (4.10). The RTT measurements for all VPNs were nearly identical, with occasional latency spikes (4.11).

In summary, under common network conditions, Quincy and WireGuard behave similarly, both in terms of throughput and latency, with Quincy experiencing marginally better throughput than WireGuard in some scenarios. OpenVPN drastically underperforms both in throughput and in latency in cases where network conditions are not the limiting factor. The relative improvement of Quincy compared to WireGuard might be due to better packet pacing provided by the QUIC protocol and warrants further investigation.

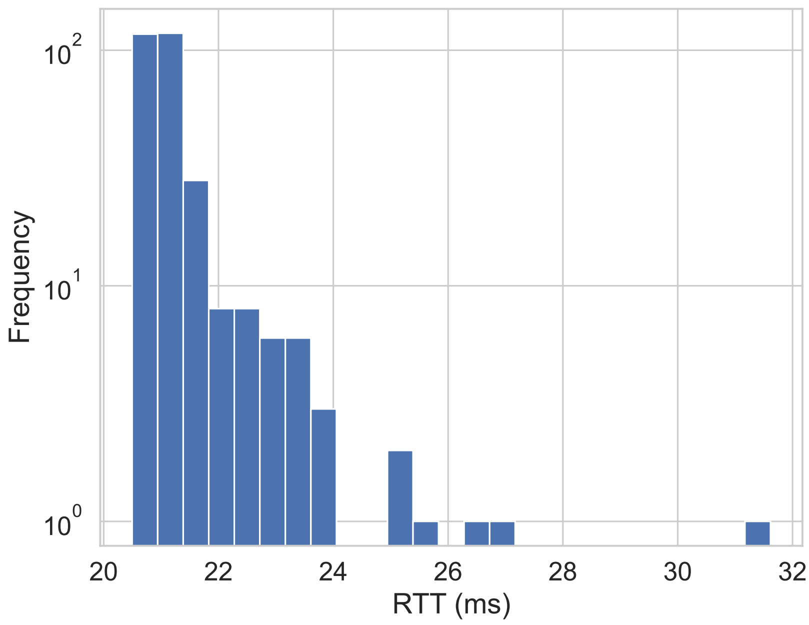

Figure 4.9: Quincy RTT distribution (10 ms latency, 0 % packet loss, 1500 B MTU)

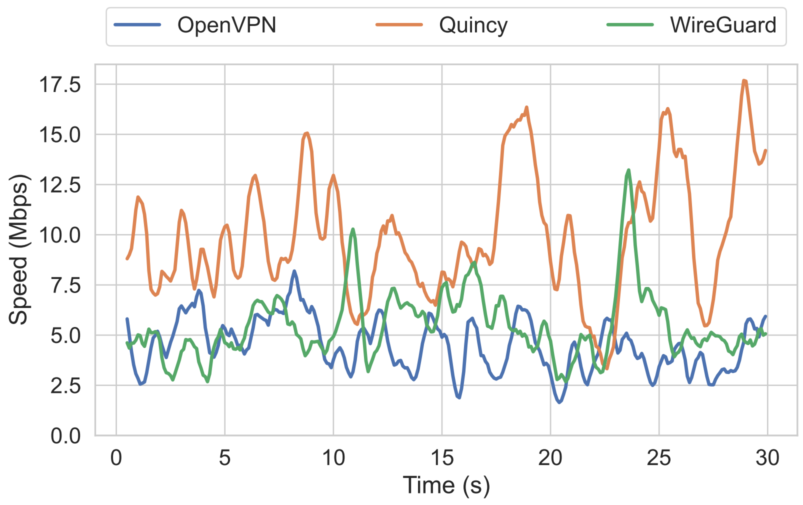

Figure 4.10: Download (10 ms latency, 1 % packet loss, 1500 B MTU)

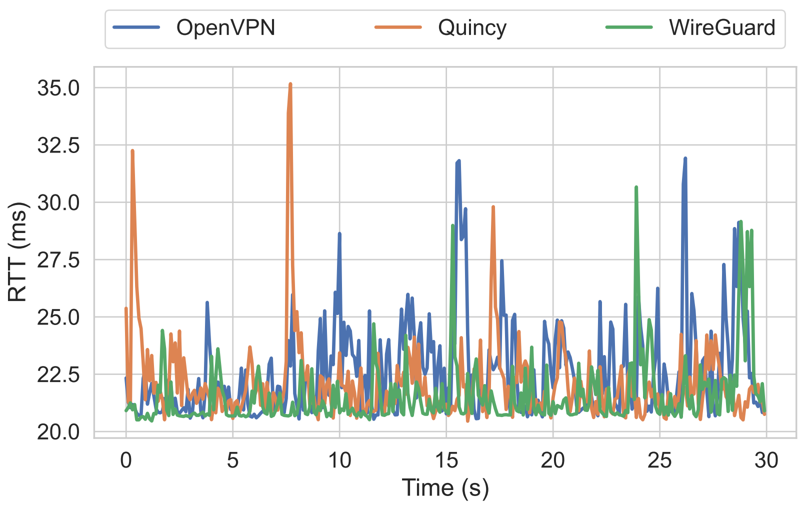

Figure 4.11: RTT (10 ms latency, 1 % packet loss, 1500 B MTU)

4.3.3 Poor network conditions

The last observed behavioural group represents poor network conditions – networks with high latency, noticeable packet loss, and decreased MTU sizes, such as unreliable intercontinental connections, Wi-Fi or cellular networks with high congestion (public places, hotels, events, etc.). Data from this behavioural group show how the evaluated VPNs handle conditions in which the use of a VPN might be required for privacy and security.

The first example of such conditions is described by the network parameters of 10 ms latency, 5 % packet loss and 1500 B MTU. In such conditions, all evaluated VPNs behave similarly, being mostly limited by the connection itself. In terms of throughput (4.12), WireGuard and OpenVPN show very similar results with a relatively stable connection. Quincy shows a slight increase in throughput at the cost of less overall stability (4.13). In terms of latency, all VPNs show very similar results, with Quincy and WireGuard experiencing a low number of high latency spikes of up to 100 ms, which could be due to the detection of link congestion by iperf3 or the packet pacing algorithms of the VPNs (4.14).

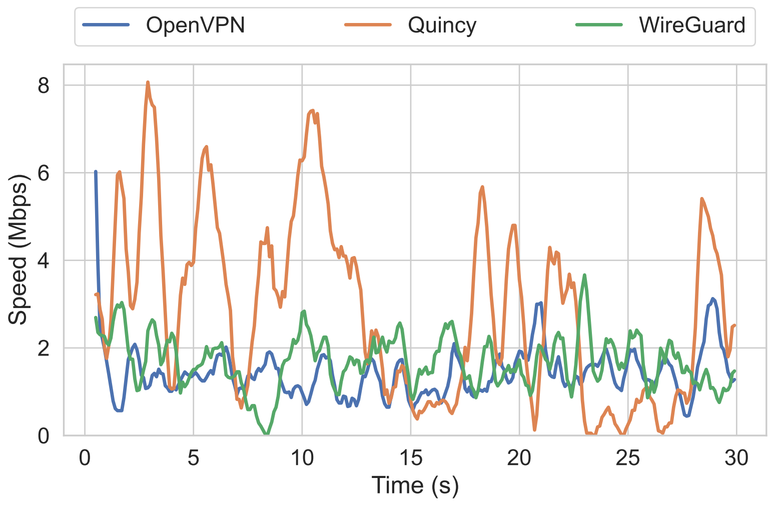

Figure 4.12: Download (10 ms latency, 5 % packet loss, 1500 B MTU)

The second example of poor network conditions, with 150 ms latency, 1 % packet loss, and 1500 B MTU, represents less ideal intercontinental connections or connections with a large number of wireless links. In these conditions, OpenVPN and WireGuard behave nearly identically, both in terms of measured throughput and latency (4.15, 4.16). Unlike the other VPNs, Quincy is observably much more aggressive with its packet pacing, getting to and keeping at a much higher transfer speed for most of the test. This behaviour seems to come at a cost to the overall stability and latency of the connection, which spiked significantly and was around 70 ms higher for most of the test (4.17). Although it might be better for throughput-intensive network loads,

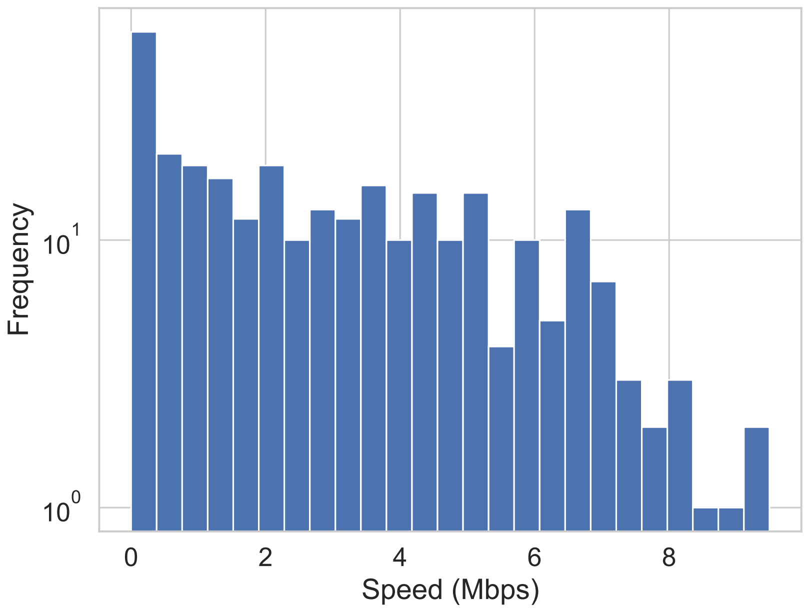

Figure 4.13: Quincy download distribution (10 ms latency, 5 % packet loss, 1500 B MTU)

such as media consumption or downloading/uploading large files, it comes at a detriment to the experience for real-time applications, such as communication protocols (VoIP, WebRTC, etc.). Further investigation is needed to determine the root cause of this behaviour.

The most extreme example of poor network conditions, with 150 ms latency, 5 % packet loss, and 1500 B MTU, shows very similar results to the previous example. In terms of throughput, OpenVPN and WireGuard behave very similarly, with WireGuard achieving marginally higher throughput in the download (server-to-client) direction. Quincy's throughput was observably higher, at the cost of less stability (4.18). Regarding latency, both OpenVPN and WireGuard again performed very well, with OpenVPN achieving slightly lower overall RTT. Quincy observed similar problems as in the previous case, with latency spikes and significantly worse RTT, about 75 ms higher than other evaluated VPNs (4.19, 4.20).

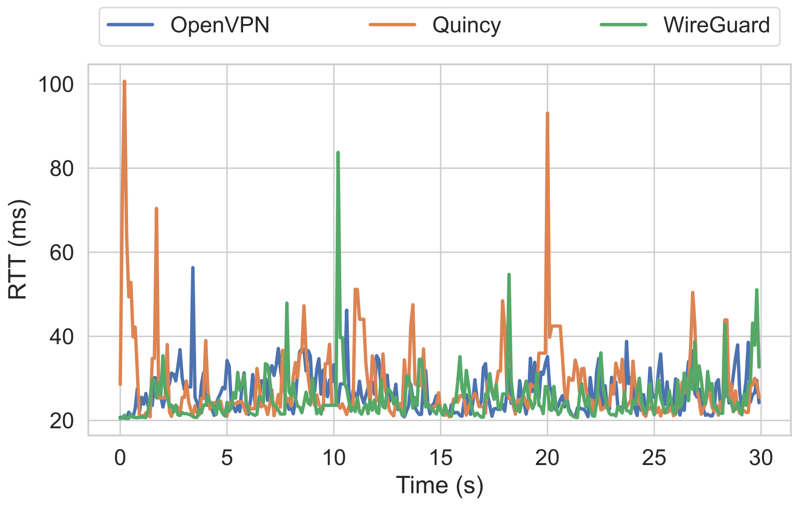

Figure 4.14: RTT (10 ms latency, 5 % packet loss, 1500 B MTU)

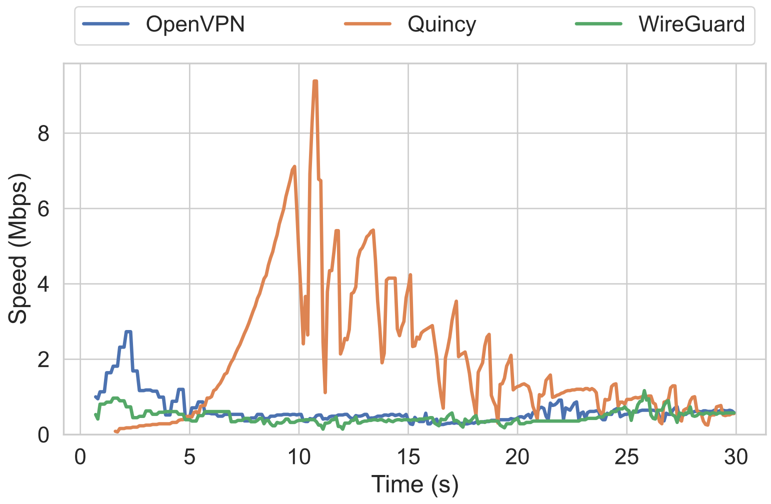

Figure 4.15: Download (150 ms latency, 1 % packet loss, 1500 B MTU)

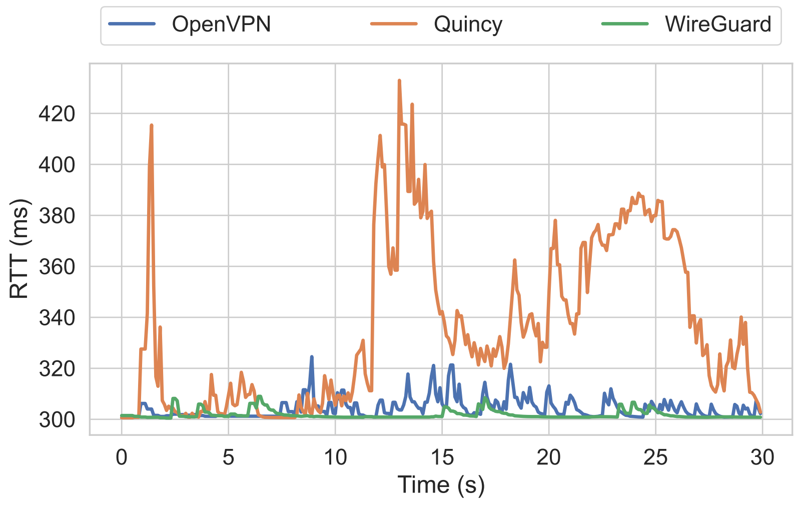

Figure 4.16: RTT (150 ms latency, 1 % packet loss, 1500 B MTU)

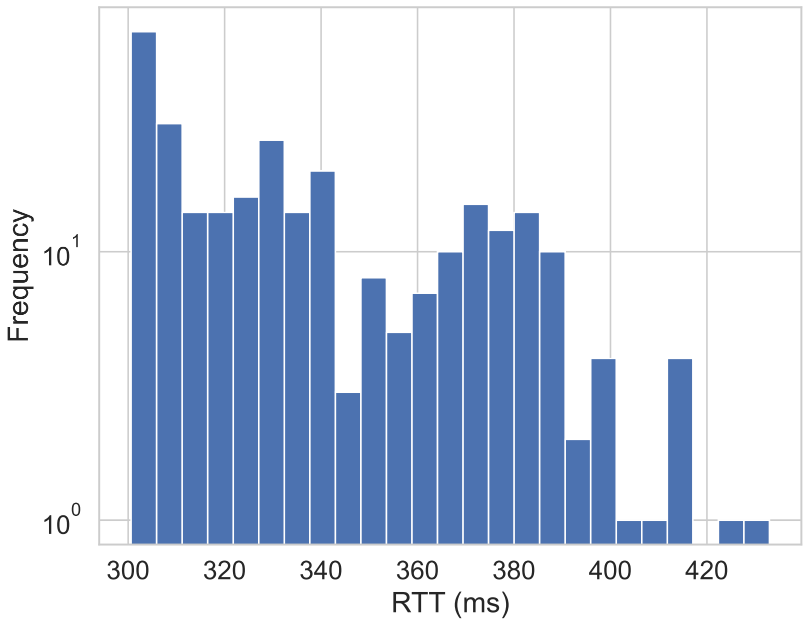

Figure 4.17: Quincy RTT distribution (150 ms latency, 1 % packet loss, 1500 B MTU)

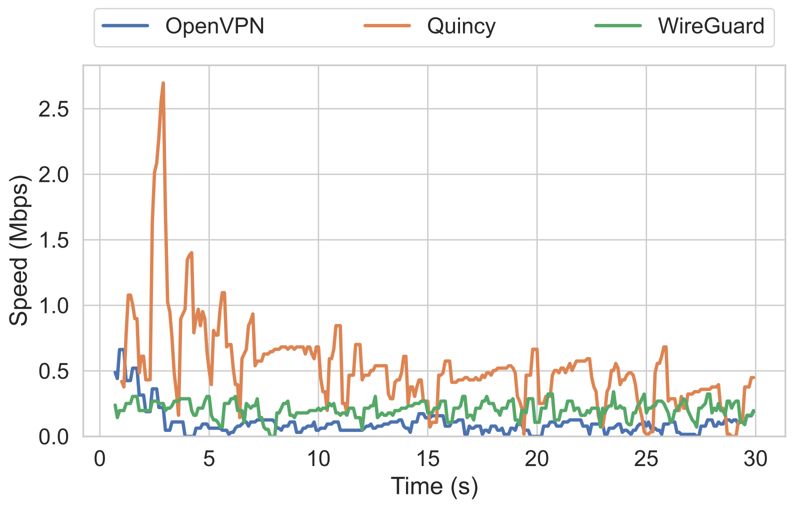

Figure 4.18: Download (150 ms latency, 5 % packet loss, 1500 B MTU)

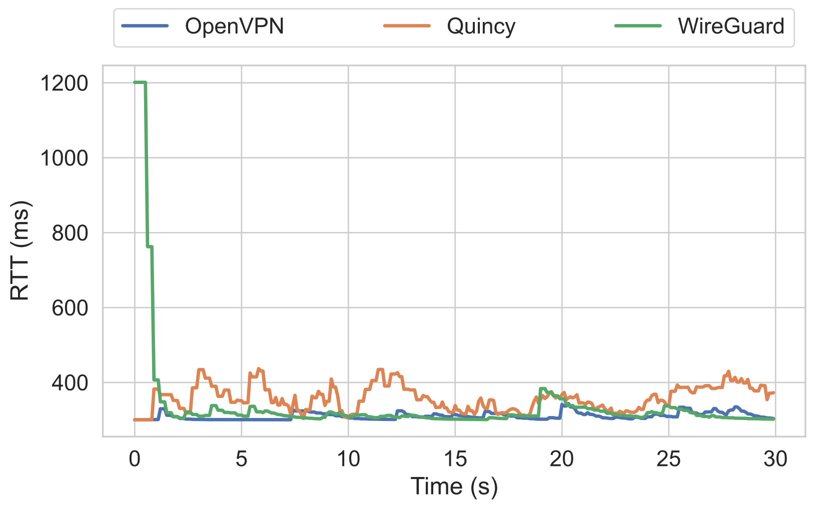

Figure 4.19: RTT (150 ms latency, 5 % packet loss, 1500 B MTU)

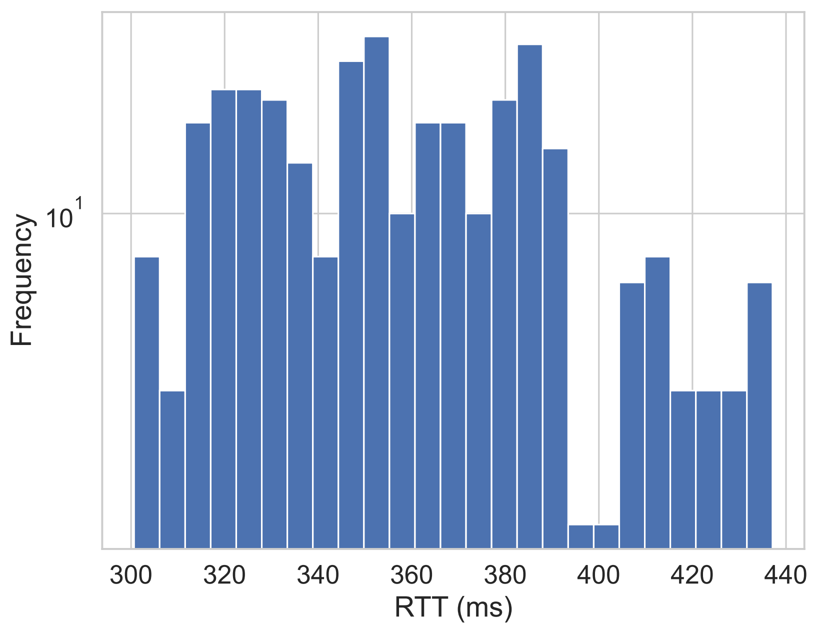

Figure 4.20: Quincy RTT distribution (150 ms latency, 5 % packet loss, 1500 B MTU)

4.4 Discussion

In this chapter, multiple groups of simulated network conditions were used to evaluate the performance of Quincy, OpenVPN, and WireGuard, ranging from ideal conditions, such as CAN or MAN networks, to poor conditions, such as inter-continental WAN connections or unrealiable cellular/Wi-Fi networks.

Under ideal conditions with very low latency, no packet loss, and varied MTU sizes, WireGuard observes the highest throughput, Quincy lags noticeably behind, and OpenVPN shows the lowest throughput of the tested VPNs. Both Quincy and OpenVPN exhibit asymmetrical throughput, likely due to different implementations of client-side and server-side routing, whereas WireGuard shows symmetrical throughput in both directions, due to its peer-to-peer design. The measured latency is the lowest for Quincy, with WireGuard and OpenVPN adding slightly more latency. Overall, Quincy greatly outperforms OpenVPN, and both fall short of WireGuard. WireGuard showcases the best characteristics in throughput, likely due to its implementation residing in the kernel-space, while Quincy leads in latency.

In scenarios representing common network conditions with low latency, low-to-none packet loss, and standard MTU sizes, Quincy and WireGuard behave similarly both in terms of throughput and latency. Quincy outperforms WireGuard in throughput under specific conditions, while OpenVPN consistently underperforms in all scenarios. The "Saw-Tooth" behaviour inherent to the default TCP configurations is observed for all VPNs, with Quincy and WireGuard displaying similar connection properties. In scenarios simulating slightly unreliable networks with some packet loss, Quincy slightly outperforms WireGuard in throughput.

Under poor network conditions, including high latency, packet loss, and reduced MTU sizes, individual VPNs show how they handle challenging conditions. In less extreme conditions, all VPNs behave very similarly, generally limited by the connection. Quincy shows a slight throughput increase at the cost of stability. In more extreme conditions, OpenVPN and WireGuard behave similarly, while Quincy demonstrates noticeably higher transfer speeds, ultimately sacrificing stability and latency.

Overall, the performance of Quincy, both in terms of throughput and latency, is better than the performance of OpenVPN and approaches that of kernel-level WireGuard, even reaching beyond it in certain scenarios. As some behavioural anomalies occurred during specific evaluation cases, such as the latency and throughput spikes in poor network conditions, further investigation of the QUIC protocol is required to fully understand its viability in the realm of VPNs.

Conclusion

The objective of this Bachelor's thesis was to test the feasibility of the QUIC protocol in the VPN space. To achieve this goal, a comprehensive analysis of the current VPN space was performed, researching individual network protocols and existing VPN implementations. Afterwards, through an iterative design and development process based on this analysis, a new VPN implementation, based on the QUIC protocol, was created. During the subsequent evaluation, it was found to provide all required functionality while achieving satisfactory security and performance, nearing the throughput and latency of the WireGuard implementation in the Linux kernel in many of the tested scenarios, with both implementations achieving much higher throughput and lower latency compared to OpenVPN. Therefore, it successfully met all the requirements specified in the thesis assignment.

The created implementation has a sufficient feature set to allow for usual VPN deployments, such as providing secure access to home and small business local networks, being used as a site-to-site tunnel between multiple private networks, or allowing users from countries with Internet censorship and privacy-invasive laws to access the Internet in a secure manner, retaining their online privacy.

Several potential improvements could be made to allow for a wider range of possible use cases, such as validating the security and overall soundness of the QUIC protocol implementation provided by *quinn*; implementing additional authentication back-ends based on LDAP, SSO or PKI; or improving the performance of the TUN interface using Generic Segmentation Offload and Generic Receive Offload.

Bibliography

1. EDDY, Wesley. *Transmission Control Protocol (TCP)* [online]. RFC Editor, 2022. Request for Comments, no. 9293. Available from DOI: `10.17487/RFC9293`.
2. POSTEL, Jon. *User Datagram Protocol* [online]. RFC Editor, 1980. Request for Comments, no. 768. Available from DOI: `10.17487/RFC0768`.
3. *Transport Layer Security* [online]. Mozilla Foundation, 2023 [visited on 2024-01-03]. Available from: `https://web.archive.org/web/20231225052013/https://developer.mozilla.org/en-US/docs/Web/Security/Transport_Layer_Security`.
4. VILLANUEVA, John Carl. *An Introduction To Cipher Suites | SSL/TSL Cipher Suites Explained* [online]. 2022. [visited on 2024-05-10]. Available from: `https://www.jscape.com/blog/cipher-suites`.
5. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3* [online]. RFC Editor, 2018. Request for Comments, no. 8446. Available from DOI: `10.17487/RFC8446`.
6. IYENGAR, Jana; THOMSON, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport* [online]. RFC Editor, 2021. Request for Comments, no. 9000. Available from DOI: `10.17487/RFC9000`.
7. VARSHNEY, Abhishek. *Head-of-line (HOL) blocking in HTTP/1 and HTTP/2* [online]. 2021. [visited on 2024-05-07]. Available from: `https://engineering.cred.club/head-of-line-hol-blocking-in-http-1-and-http-2-50b24e9e3372`.
8. *Overview of OpenVPN* [online]. [visited on 2024-01-06]. Available from: `https://web.archive.org/web/20230408102834/https://community.openvpn.net/openvpn/wiki/OverviewOfOpenvpn`.
9. *openvpn2-historical-cvs/CHANGES* [online]. [visited on 2024-01-07]. Available from: `https://github.com/OpenVPN/openvpn2-historical-cvs/blob/d32aa83c2adc6f9fb70e2f0cc32c344f4864e95d/CHANGES#L20`.
10. *OpenVPN/openvpn – lines of code* [online]. [visited on 2024-01-07]. Available from: `https://api.codetabs.com/v1/loc/?github=OpenVPN/openvpn&ignored=doc,distro,contrib,build,dev-tools,sample&branch=v2.6.8`.

11. BÉDRUNE, Jean-Baptiste; BOUYAT, Jordan; CAMPANA, Gabriel. *OpenVPN 2.4.0 Security Assessment* [online]. OSTIF, 2017. Available also from: `https://ostif.org/wp-content/uploads/2017/05/OpenVPN1.2final.pdf`.
12. DONENFELD, Jason A. *WireGuard: Next Generation Kernel Network Tunnel* [online]. 2020. [visited on 2024-01-07]. Available from: `https://www.wireguard.com/papers/wireguard.pdf`.
13. NIR, Yoav; LANGLEY, Adam. *ChaCha20 and Poly1305 for IETF Protocols* [RFC 7539]. RFC Editor, 2015. Request for Comments, no. 7539. Available from DOI: `10.17487/RFC7539`.
14. TEAM, The Tamarin. *Tamarin-Prover Manual – Security Protocol Analysis in the Symbolic Model* [online]. 2024. [visited on 2024-05-10]. Available from: `https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf`.
15. DONENFELD, Jason A.; MILNER, Kevin. *Formal Verification of the WireGuard Protocol* [online]. 2017. [visited on 2024-01-11]. Available from: `https://www.wireguard.com/papers/wireguard-formal-verification.pdf`.
16. DOWLING, Benjamin; PATERSON, Kenneth G. *A Cryptographic Analysis of the WireGuard Protocol* [online]. 2018. [visited on 2024-01-11]. Available from: `https://eprint.iacr.org/2018/080.pdf`.
17. *Reference manual for OpenVPN 2.6* [online]. [visited on 2024-04-14]. Available from: `https://openvpn.net/community-resources/reference-manual-for-openvpn-2-6/`.
18. *The Rust Programming Language* [online]. [visited on 2024-04-29]. Available from: `https://doc.rust-lang.org/book/`.
19. *Quinn repository* [online]. [visited on 2024-01-17]. Available from: `https://github.com/quinn-rs/quinn`.
20. *Rustls repository* [online]. [visited on 2024-03-24]. Available from: `https://github.com/rustls/rustls`.
21. HEIDERICH, Dr.-Ing. M.; KREIN, MSc. N.; KOBEISSI, Dr. N.; KOPF, Dipl.-Inf. G. *Security Review & Audit Report rustls 05. – 06. 2020* [online]. [visited on 2024-01-17]. Available from: `https://github.com/rustls/rustls/blob/fa81bd23c00d71f1c53509be9ea58701265b0c4d/audit/TLS-01-report.pdf`.
22. *s2n-quic repository* [online]. [visited on 2024-04-30]. Available from: `https://github.com/aws/s2n-quic`.
23. *Quiche repository* [online]. [visited on 2024-04-30]. Available from: `https://github.com/cloudflare/quiche`.
24. *Asynchronous Programming in Rust* [online]. [visited on 2024-05-01]. Available from: `https://rust-lang.github.io/async-book/08_ecosystem/00_chapter.html`.
25. *rust-tun repository* [online]. [visited on 2024-01-17]. Available from: `https://github.com/ssrlive/rust-tun`.

26. *Tokio repository* [online]. [visited on 2024-01-17]. Available from: `https://github.com/tokio-rs/tokio`.

27. FISCHLIN, Marc; GÜNTHER, Felix. *2017 IEEE European Symposium on Security and Privacy (EuroS&P).* Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates. 2017. Available from DOI: `10.1109/EuroSP.2017.18`.

28. *Cipher suites recommendations* [online]. [visited on 2024-03-31]. Available from: `https://developers.cloudflare.com/ssl/reference/cipher-suites/recommendations/`.

29. BIRYUKOV, Alex; DINU, Daniel; KHOVRATOVICH, Dmitry. *Argon2: the memory-hard function for password hashing and other applications* [online]. 2017. [visited on 2024-03-26]. Available from: `https://github.com/P-H-C/phc-winner-argon2/blob/master/argon2-specs.pdf`.

30. *Default parameters of the Rust implementation of Argon2* [online]. [visited on 2024-03-26]. Available from: `https://docs.rs/argon2/0.5.3/argon2/struct.Params.html`.

31. *iPerf homepage* [online]. [visited on 2024-01-18]. Available from: `https://iperf.fr/`.

32. *tc – Linux manual page* [online]. [visited on 2024-01-23]. Available from: `https://web.archive.org/web/20231225153001/https://man7.org/linux/man-pages/man8/tc.8.html`.

Appendix A

Acronyms

AES	Advanced Encryption Standard.
API	Application Programming Interface.
CAN	Campus/Company Area Network.
CPU	Central Processing Unit.
DHCP	Dynamic Host Configuration Protocol.
ECC	Eliptic Curve Cryptography.
GPU	Graphics Processing Unit.
GRO	Generic Receive Offload.
GSO	Generic Segmentation Offload.
HTTP	Hypertext Transfer Protocol.
IETF	Internet Engineering Task Force.
LDAP	Lightweight Directory Access Protocol.
MAN	Metropolitan Area Network.
MTU	Maximum Transmission Unit.
PKI	Public Key Infrastructure.
RADIUS	Remote Authentication Dial-In User Service.
RAM	Random Access Memory.
RSA	Rivest–Shamir–Adleman.
RTT	Round Trip Time.

SHA	Secure Hashing Algorithms.
SSL	Secure Sockets Layer.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.
UDP	User Datagram Protocol.
VoIP	Voice over Internet Protocol.
VPN	Virtual Private Network.
WAN	Wide Area Network.
WebRTC	Web Real-Time Communication.

Appendix **B**

Contents of attachements

```
executables .......................... directory with executables
  linux-x86_64 ....................... Linux x86_64 executables
  macos-arm64 ........................ MacOS ARM64 executables
  windows-x86_64 ..................... Windows x86_64 executables
  config ............ directory with configuration files and certificates
  README.md ............... a guide for running the provided executables
sources .............................. directory of source codes
  quincy ............................. implementation sources
  quincy-benchmark ........... evaluation suite and collected data
  thesis ............................. LaTeX source codes of the thesis
```