Semesterarbeit

Post Quantum P2P Chat

Semester: Autumn 2023

Version: 1.0
Date: 2023-12-22

Project Team: Miles Strässle
Svenja Sutter

Project Advisor: Dr. Alexandru Caracas

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

Contents

Part I

Abstract

Abstract

- **Introduction**: Our application, qChat, is a decentralized, peer-to-peer (P2P) chat application meant for future-proof privacy. It uses the latest encryption algorithms to ensure secret message exchange in the post-quantum era.
- **Problem Addressed**: qChat addresses the growing concern for digital privacy and security, particularly in the face of quantum computing's potential to break current encryption standards.

  Quantum computing poses a significant threat to current encryption methods, particularly due to advancements like Shor's Algorithm. This algorithm, in theory, allows quantum computers to break widelyused encryption schemes such as RSA and ECC much faster than conventional computers.
- **Result**: The solution involves developing a peer-to-peer chat application using post-quantum cryptography (PQC), eliminating reliance on external servers for data storage. The project has successfully created a prototype demonstrating peer-to-peer functionality with integrated post-quantum cryptography.
- **Conclusion**: These developments are significant as they provide a practical approach to secure communication, setting a precedent in the field of quantum-resistant digital communication.
- **Key Results**: The project has successfully created prototypes demonstrating P2P functionality, integrated PQC, and developed a user interface.
- **Evaluation of Results**: These developments are significant as they provide a practical approach to secure communication, setting a precedent in the field of quantum-resistant digital communication.
- **Future Implications**: The insights and technologies developed in qChat lay the groundwork for future advancements in secure communications. This also marks a significant advancement in protecting the private sphere in an increasingly interconnected post-quantum world.

Part II

Product Documentation

Chapter 1

Introduction

In today's digital world, where privacy and data security are major concerns, we often compromise our personal information for convenience. Take WhatsApp as an example, in order to use it, we have to accept terms and conditions that allow for extensive data collection.

We believe that control over personal information and private communications should be in our own hands, not stored on distant servers. That's why we are developing a peer to peer chat application as our semester assignment at the OST. Our app, qChat, prioritises privacy, so it doesn't store messages on external servers and ensures that your conversations remain confidential. Moreover, we are integrating the cutting-edge technology post-quantum cryptography. This advanced security feature is designed to protect your chats even against future threats like quantum computers. Our goal is to provide a way for people to chat without worry about their privacy.

To achieve our goals, we will evaluate and experiment with different technologies and learn about post-quantum technology and how to integrate an existing algorithm into our chat application. We plan to develop several prototypes: one dedicated to peer-to-peer (P2P) chat functionality, a proof of concept for post-quantum cryptography (PQC) encryption, and another for a basic frontend interface. Ultimately, we aim to integrate these components into a cohesive application.

In this documentation, we will first examine various state-of-the-art technologies. We will then define and evaluate our requirements and then present an overview of the chat application, focusing on its architecture and security layers. Finally, we will perform an evaluation to assess the effectiveness and security of our chat application.

Chapter 2

State of the Art

This chapter evaluates various messaging platforms, assessing their current encryption methods and preparedness for quantum computing developments.

2.1 Briar

- Overview: Briar is tailored for secure communication, functioning over Wi-Fi, Bluetooth, and the internet, with robust offline capabilities [Bri23].
- Pros: Briar utilizes the Tor network for enhanced anonymity, supports offline communication via Bluetooth and Wi-Fi, and prioritizes privacy and security.
- Cons: Briar lacks implementation of post-quantum cryptography, posing potential risks in the face of quantum computing advancements.

2.2 Signal

- Overview: Signal is known for robust encryption protocols, emphasizing user privacy and security [Sig23].
- Pros: Features state-of-the-art end-to-end encryption, securing messages from sender to receiver using their own protocol, and is exploring post-quantum cryptography.
- Cons: Signal's requirement of a phone number for registration may raise privacy concerns.

2.3 Others

- Overview: Messaging apps such as Session, Threema, Telegram, Instagram DM, Snapchat, Viber, and WeChat vary in features and security levels.

- Pros: Session offers decentralization and privacy enhancements, including no phone number requirement and routing messages over Loki Net. Threema and Telegram provide substantial security features.
- Cons: Apps like Instagram DM, Snapchat, Viber, and WeChat may have vulnerabilities and privacy issues, particularly concerning data collection and user tracking.

2.4 P2P

For privacy-focused messaging apps, P2P technology is essential. It allows direct user connections, bypassing central servers, enhancing privacy, reducing data breach risks, and supporting decentralization. XMPP [XMP23] is an example of a secure, flexible and widely used P2P protocol.

2.5 Tor

Tor (The Onion Router) is crucial for messaging apps prioritizing anonymity and circumventing internet censorship. It anonymizes internet traffic through a global network of relays, concealing users locations and activities from network surveillance and analysis [Tor23].

2.6 Conclusion

This chapter has examined various messaging applications in terms of their cryptographic standards and their readiness for quantum computing. The recent integration of PQC by Signal illustrates the evolving nature of digital security. This analysis demonstrates the need for continued innovation in encryption technology to maintain the privacy and security of messages as technology advances.

Chapter 3

Requirements

Our vision of this project is a peer-to-peer chat application using a PQC algorithm for data protection.

3.1 Requirements

The functional requirements should include:

- FR1: The application must allow users to securely send both text and binary files to their peers.
- FR2: The application must provide a user registration process.
- FR3: The application must integrate a mechanism that allows users to securely exchange keys or identifiers, enabling them to establish communication and send messages to each other.
- FR4: The application must implement a secure method for key storage considering industry standards like PKI and HSM.
- FR5: The application's implementation of post-quantum cryptography (PQC) should be modular, using a wrapper architecture. This design should allow for the easy replacement of specific PQC algorithms as they are still experimental.
- FR6: Users should be able to activate or deactivate the use of post-quantum cryptography for their messages. When PQC is deactivated, the application should default to using standard encryption algorithms to secure the messages.

3.2 Evaluation Requirements

- FR1: The user can send and receive text messages. The focus of the project was on the architecture and not the functionality of the client, so we did not implement sending binary files.

- FR2: Once the user launches the client, the user registration process takes place automatically. The user receives their own onion address, certificate and PQC key pair.
- FR3: To communicate with other users, the user must exchange their onion address and certificate. To enable PQC communication, the user must give the other user his public key.
- FR4: The application stores the keys locally in a docker volume. Storing the keys in an HSM would be a possibility for further development.
- FR5: The PQC implementation is done in the C++ wrapper PQCWrapper.hpp. So it would be easy to replace the PQC algorithm.
- FR6: A checkbox on the frontend allows the user to decide whether the messages should also be PQC encrypted.

Chapter 4

Application

4.1 Client

4.1.1 User Information

When the application is first launched, a user profile is automatically created. This profile contains a unique onion address, certificate and post-quantum cryptography (PQC) key pair consisting of a private and public key (more information in chapter 5.3.2). To retrieve and view this information, users can use the 'Fetch' buttons, which gets the data from the application's Docker volume.

Additionally, the checkbox 'Enable PQC', allows users to choose whether they want to encrypt their outgoing messages using PQC. This feature provides an extra layer of security. More information can be found in section 5.3 on Post Quantum Cryptography.

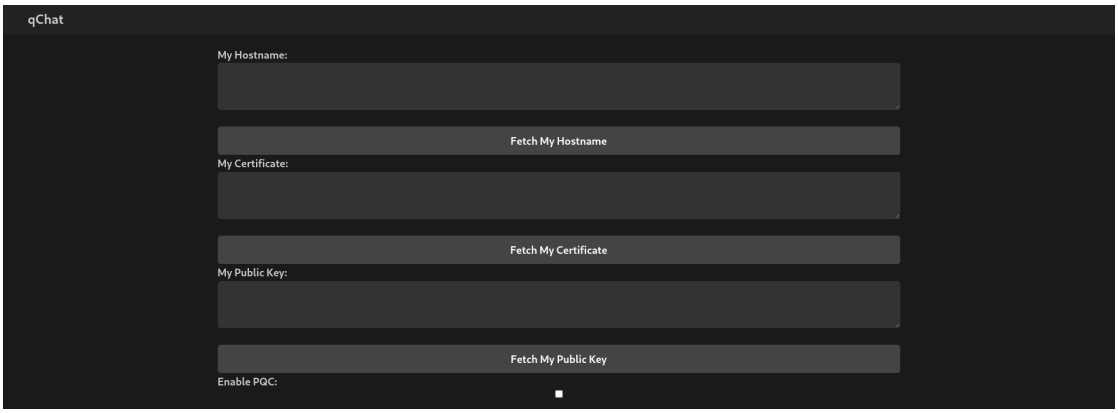


Figure 4.1: GUI: User

4.1.2 Registration Friends

Save Certificate:
This feature allows users to add friends to their communication network. This involves entering the friend's onion address and certificate. Once both users have saved each other's onion address and certificate, they can begin communicating.

Save PQC Public Key:
For enhanced message security using PQC, users need to store the public key of the person they are communicating with. This creates a shared secret that allows both parties to encrypt messages using PQC. Importantly, once one user has saved the other's public key, both can communicate encrypted without the other user having to store a public key separately. This is due to the nature of the key exchange, which will be discuessed in 5.3.2.

Figure 4.2: GUI: Friend Registration

4.1.3 Send messages

After this initial setup, the user can send text messages by selecting a recipient and composing the message. Below is an example, of how such a chat could look like when fully configured (i.e. when Alice and Bob share there Onion Hostname, XMPP Certificate and (optionally) the PQC Public key.)

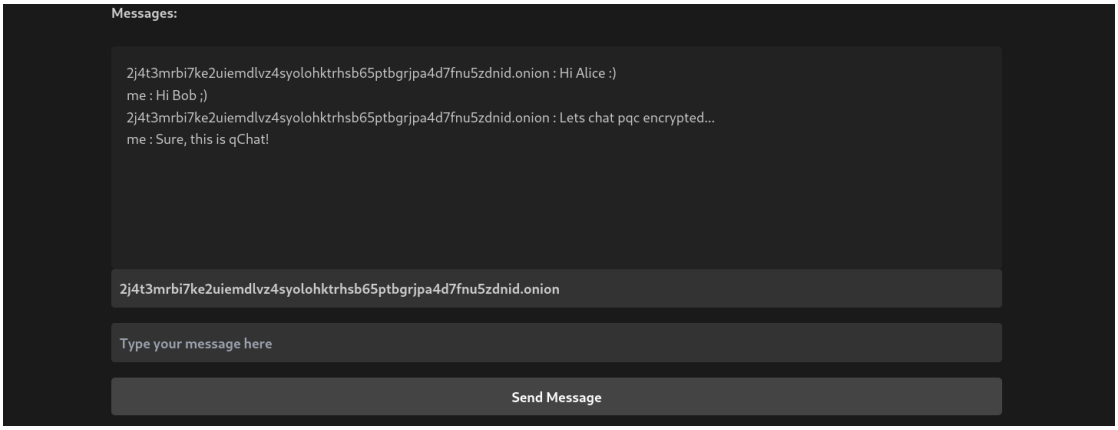


Figure 4.3: GUI: Send Messages

Chapter 5

Architecture

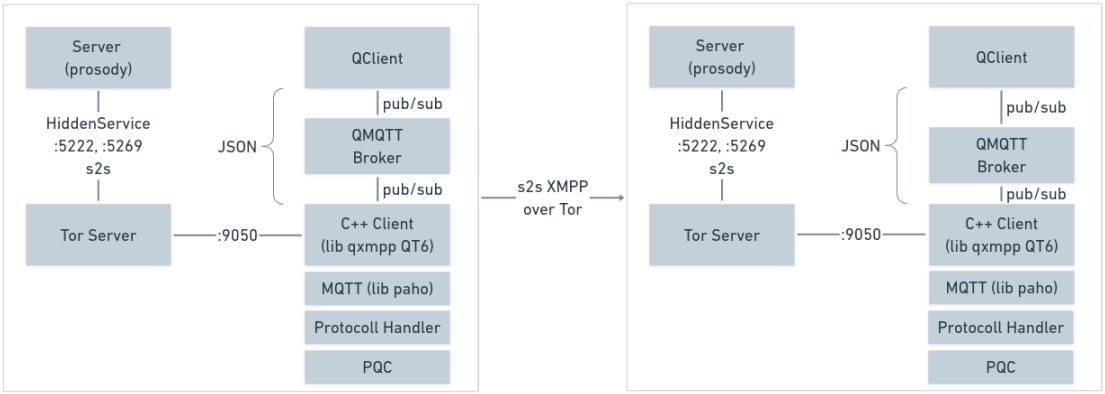


Figure 5.1: Architecture

5.1 Client Design Decisions

In the following chapters, we'll look in detail at the design and technology choices we've made, and consider their advantages and disadvantages. Our primary focus is on secure and anonymous peer-to-peer communication. To achieve this we have thoroughly evaluated the following key topics.

5.1.1 P2P Protocol

Initially, we looked at different peer-to-peer (P2P) protocols and finally focused on two: Matrix [Mat23] and XMPP [XMP23].

Protocol Comparison

XMPP Advantages

- Customizable Server Settings: XMPP offers extensive configuration options, which provides flexibility in server setup, a significant advantage for specific use cases and environments.
- Lightweight: Being more resource-efficient, XMPP uses less computational power and memory, leading to better performance, especially on limited-resource devices.
- Available Server Software: The existence of several XMPP server software options like ejabberd and Prosody simplifies the implementation process, making it easier to deploy and manage.

XMPP Disadvantages

- Limited Metadata Privacy: Unlike Matrix, XMPP doesn't inherently share conversation metadata across servers, but privacy concerns can still arise based on server configurations and client implementations.
- Complex Federations: While XMPP supports server federations, managing these federations and ensuring interoperability and security across different servers can be complex.
- Feature Set and Modernity: XMPP, being older, might not be as feature-rich or may require additional extensions for modern communication features compared to Matrix.

Matrix Advantages

- Rich Feature Set: Matrix is designed to support a wide range of modern communication features including VoIP, video calls, and more, making it suitable for versatile applications.
- Federation and Decentralization: Matrix supports decentralized operation and federation, enabling users on different servers to communicate seamlessly.
- Strong Metadata Sharing: Matrix shares conversation metadata across all servers, which can be beneficial for creating a unified and connected user experience.

Matrix Disadvantages

- Resource Intensive: Matrix can be more demanding in terms of computational resources compared to XMPP, potentially impacting performance on constrained devices.
- Metadata Privacy Concerns: The protocol's design involving sharing conversation metadata across servers can be seen as a disadvantage for strict privacy-focused applications, as it may expose certain information like user presence or message timestamps.

- Complexity: The advanced features and capabilities of Matrix might introduce additional complexity in deployment and maintenance, especially for smaller-scale or resource-limited environments.

In summary, XMPP's lightweight nature and customizable server settings make it suitable for scenarios where resource efficiency and flexibility are paramount. However, its limitations in metadata privacy and complexity in federation management can be drawbacks. On the other hand, Matrix's rich feature set and strong federation capabilities are advantageous for modern, feature-rich applications but come at the cost of being resource-intensive and having potential privacy concerns related to metadata sharing.

5.1.2 Tor

In a P2P application, NAT can make direct connections difficult because it hides the device's true IP address behind a router's IP address. We aim for our chat app to be as anonymous and secure as possible. If we would use STUN / TURN server for NAT traversal anonymity can be problematic. Therefore we decided to communicate over Tor.

Tor increases anonymity and privacy by routing traffic through multiple points on the internet, making it difficult to trace a message back to its origin. Tor is designed to protect against tracking, surveillance and censorship. When a user initiates a Tor session, their traffic is routed through a Guard Node. A typical Tor circuit consists of three nodes: a guard (entry) node, a middle (relay) node and an exit node.

However, Tor isn't efficient if an attacker can see where your data enters and leaves the network. To mitigate this, Tor uses Entry Guards. If you would always use a random entry and exit point, there's a chance that an attacker might be both the first and the last person in the chain. Instead, you connect through a Tor Guard. By doing this, you're ensuring that your connection always starts through a trusted and secure point. By using a fixed guard for an specific period, you reduce the chances of encountering a compromised or malicious server at the start of your Tor usage. [The23]

In summary, a guard acts as a secure entry point into the Tor network, enhancing user anonymity and privacy. And beyond the Tor network, messages are encrypted end-to-end (E2EE).

5.1.3 XMPP

XMPP [XMP23] is a communication protocol used for instant messaging. The reason why we chose XMPP is described in the subsection 5.1.1. XMPP is decentralised, so we can run our own XMPP servers. And these servers can interoperate and communicate with each other.

When a user starts an XMPP client, it connects to an XMPP server (client-to-server (c2s) connection). This process requires the user to authenticate with a username and

a password. Once authenticated, the user can begin sending messages within the same server environment. XMPP can also be configured to communicate with other servers (server-to-server (s2s) connection). This can be viewed as a federation between the XMPP Servers. When a message is sent to a federated server, the client will send the message via its own server to the recipient's server, where it will finally be forwarded to the recipient's client.

Each user on the XMPP network is identified by a unique Jabber ID (JID). In our application, this JID is represented by the username (qChat) concatenated with servers (onion) address. In our setup, the federated servers need to know the certificate of each other. The exchange of the certificates is explained in 5.1.4. In addition, XMPP uses TLS/SSL for encryption to secure communications between clients and servers.

Ejabberd and Prosody are both open-source XMPP server software that implement this protocol.

ejabberd

We started with ejabberd as our xmpp server. Since we weren't able to circumvent DNS Resolutions (which are not possible with our tor setup), we could not use ejabberd in our architecture and decided to use prosody [eja].

prosody

prosody is an xmpp server written in lua which is actively maintained. We where able to configure prosody to forward all traffic over SOCKS5 and resolve the servers onion addresses in the tor network [pro].

5.1.4 C++ Client

The C++ client serves as the core of the application, encapsulating the essential logic and functionality. The structure of this client is organised into several key components:

- **qChatProtocolHandler** - Coordinates the handling of communication protocols and stores information
- **XMPPClient** - Handles the XMPP server interactions
- **MQTTClient** - Manages MQTT communications (frontend)
- **PQCWrapper** - Implements the PQC encryption and decryption
- **JSONHandler** - Helper class which deals with JSON data processing

We use MQTT to enable communication from the frontend to the backend. The frontend sends json-messages which are interpreted by the qChatProtocolHandler. This serves as

our own protocol and enables send/receive of messages, as well as the handling of requests to fetch and store certificates and pqc-keys. The qChatProtocolHandler uses function pointers to call several functions in the MQTT- and the XMPPClient.

5.1.5 CLI Integration into Frontend

First we developed a command line interface (CLI). Once this was operational, we moved on to integrating a frontend with django [Dja23] and styled it with tailwind [Tai23]. Since we adopted a containerized approach with separate services, we faced challenges in coordinating and managing these different components. For instance, the XMPP Server needs to load XMPP Certificates in the Linux Truststore. We triggered this with a watchfile which gets modified by the frontend. We isolated the frontend from the cli client, therefore we had to use MQTT to enable communication via our own protocol.

5.1.6 MQTT

As described in the EMQX blog post [EMQ23], MQTT (Message Queuing Telemetry Transport) is a simple and efficient messaging protocol used to send data between devices. It's designed to be lightweight and uses TLS/SSL for secure communication. In our project, we use MQTT to connect the frontend to the C++ client.

Here's how MQTT works:

MQTT Client:
This is any application or device that uses MQTT to send or receive messages. We're using the Eclipse Paho MQTT C++ library.

MQTT Broker:
The broker manages connections, subscriptions and routes messages. We use Eclipse Mosquitto as our broker.

Topics:
In MQTT, message routing is based on topics, which are essentially routing channels. For example, a topic in our system is qchat/api.

Publish-subscribe pattern:
Clients can both publish messages to topics and subscribe to receive messages on specific topics. This results in effective communication and enables multi-device environments.

Quality of Service (QoS) Levels:

- QoS 0: Delivers a message at most once, with no guarantee of delivery.
- QoS 1: Ensures a message is delivered at least once.

- QoS 2: Guarantees that each message is received only once.

Operational Workflow:
Interaction begins with clients connecting to the broker over a TCP/IP connection, secured with TLS/SSL encryption. Clients either publish or subscribe to topics. The broker processes the incoming messages and forwards them to subscribers of the respective topics.

XMPP Certificates

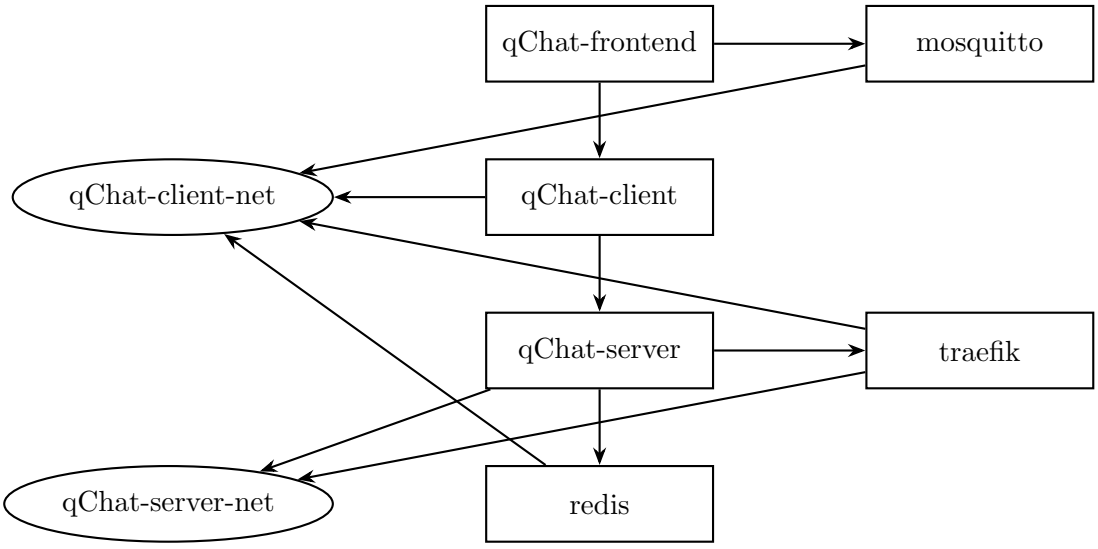
Since effectively the servers of two users are communicating (s2s), the xmpp server needs to know about the certificate of the users friend. We used a filewatcher to recognise a new friends certificate and load it in the linux trust store. At the moment we share docker volumes with the xmpp certificate of the users friends such that the C++ Client can handle the request from the frontend and update a watchfile, which triggers the update of the certificate in the trust store.

PQC Keys

The PQC-public keys are stored in json format and handled by the qChatProtocolHandler as well.

5.2 Docker Overview

Here is an overview of the relations of the docker services in the docker compose.



5.2.1 Services

- **qChat-frontend** runs the frontend.

- **qChat-client** runs the Cli Client.
- **qChat-server** runs the XMPP server (Prosody) which is configured to route via Tor.
- **mosquitto** is the MQTT Broker.
- **redis** is to buffer websocket data.
- **traefik** is used as a reverse proxy.

5.2.2 Volumes

Here are the relevant volumes in the compose:

- **qChat_server-data** stores tor credentials.
- **qChat_server-config** stores prosody config
- **qChat_prosody-data** stores prodody data
- **qChat_userfile** stores information about the user (hostname, certificates, pqc keys)
- **qChat_friends** stores information about the users friend (hostname, certificates, pqc keys)

To seperate privileges his volumes are mounted to the respective service which uses them. The volumes itself are just simple directories on the host system and are not encrypted.

5.3 PQC: Post Quantum Cryptography

5.3.1 NIST PQC Algorithms

The National Institute of Standards and Technology (NIST) has completed its third round of the Post-Quantum Cryptography (PQC) standardization process on July 5, 2022 [Nat22]. Quantum computing is advancing and this progress could potentially undermine the security of the encryption methods we use today. Recognizing the urgency of preparing for a quantum future, NIST has launched an initiative to support cryptographic protocols that are resistant to quantum computer-based attacks. They have selected four cryptographic algorithms intended to be standardized to secure information against the quantum computing threat:

Round 4 will look at other promising key protection methods: BIKE, Classic McEliece, HQC and SIKE. The full details of these methods are still being worked out [Nat23].

Due to the fact that Crystal was standardised in Round 3, we will be using CRYSTALS-KYBER in our project.

Public-Key Encryption/KEMs	Digital Signatures
CRYSTALS-KYBER	CRYSTALS-Dilithium
	FALCON
	SPHINCS+

Table 5.1: Algorithms to be Standardized

CRYSTALS-KYBER

CRYSTALS-KYBER is a Key Encapsulation Mechanism (KEM) using structured lattices, the key to lattice-based cryptography. Its security is based on the hardness of the mathematical problems associated with lattice structures, which are considered hard for quantum computers, and are in the Bounded Error Quantum Polynomial Time (BQP) complexity class. Central to its security is the LWE (Learning With Errors) problem, which is reduced to complex lattice challenges such as the CVP (Closest Vector Problem) and SVP (Shortest Vector Problem). This makes KYBER quantum resistant. The mathematical definition and further information can be found on their website [pqc].

AES-256's reliance on symmetric key cryptography contributes to its quantum resistance. So Kyber encapsulates the AES-256 symmetric key, providing resistance to potential quantum computer attacks due to its key length. A quantum computer can use Grover's Algorithm [Gee23] to brute force a symmetric key in about half the time compared to a classical computer. This means that a 256-bit key in a quantum world offers security comparable to a 128-bit key against classical attacks, which is still considered very secure. Kyber is available in three different parameter sets:

- Kyber-512, targeting security equivalent to AES-128.
- Kyber-768, targeting security equivalent to AES-192.
- Kyber-1024, aiming for security equivalent to AES-256.

For our project, we have selected the Kyber-768 parameter set, as it provides a robust security level exceeding 128 bits against both classical and quantum threats, as recommended by the developers of the official Kyber implementation. The key size of the private key is 2400 bytes and the public key is 1184 bytes.

The official implementation of Kyber is written in C. It provides the following interface to generate a keypair, to encrypt and decrpyt:

```
int pqcrystals_kyber768_ref_keypair_derand
(uint8_t *pk, uint8_t *sk, const uint8_t *coins);

int pqcrystals_kyber768_ref_keypair
(uint8_t *pk, uint8_t *sk);
```

```
int pqcrystals_kyber768_ref_enc_derand
(uint8_t *ct, uint8_t *ss, const uint8_t *pk, const uint8_t
    *coins);

int pqcrystals_kyber768_ref_enc
(uint8_t *ct, uint8_t *ss, const uint8_t *pk);

int pqcrystals_kyber768_ref_dec
(uint8_t *ss, const uint8_t *ct, const uint8_t *sk);
```

Some functions end with `_derand`. These are used in scenarios where explicit control over the randomness in the encryption process is required. We use the other methods to ensure high entropy randomness.

5.3.2 PQC Workflow

We have developed a prototype to demonstrate the successful use of Kyber. The prototype included key generation, providing a secret and public key, encryption and decryption of messages. After proof of concept for Kyber, we integrated it into the client.

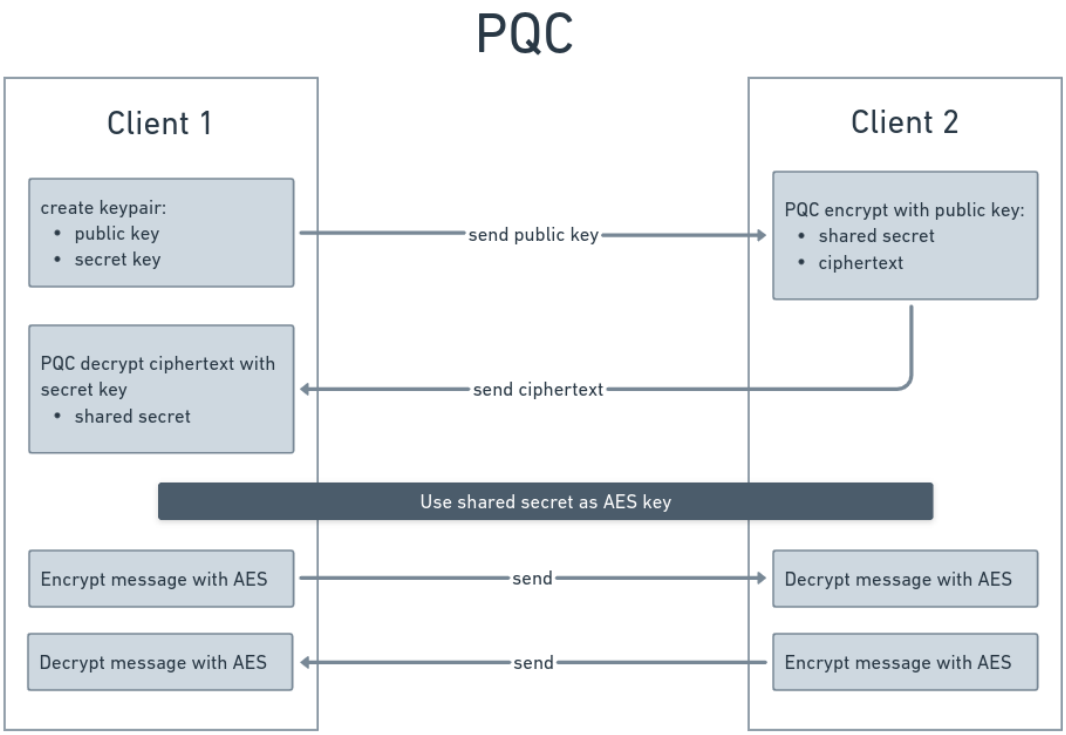


Figure 5.2: Workflow

1. Client 1 generates a key pair (`pqcrystals_kyber768_ref_keypair`).
2. Client 1 sends the public key to Client 2.
3. Client 2 encapsulates a secret (`pqcrystals_kyber768_ref_enc`), generating a ciphertext (1088 bytes) and a shared secret.
4. Client 2 sends the ciphertext to Client 1.
5. Client 1 decapsulates the ciphertext (`pqcrystals_kyber768_ref_dec`), recovering the shared secret.

Now Client 1 and Client 2 have the same shared secret, which can be used as a key for symmetic encryption. As the shared secret in Kyber is 32 bytes, we use AES-256.

5.4 Testing

To test the connections during development, we used Pidgin [Pid23]. Pidgin is a chat program which lets you log into accounts on multiple chat networks simultaneously. It

is compatible with XMPP, so we were able to send a message from one onion address to another one.

For the ultimate test of qChat's capabilities, we successfully conducted a direct message exchange between our computers.

5.5 Security Layers

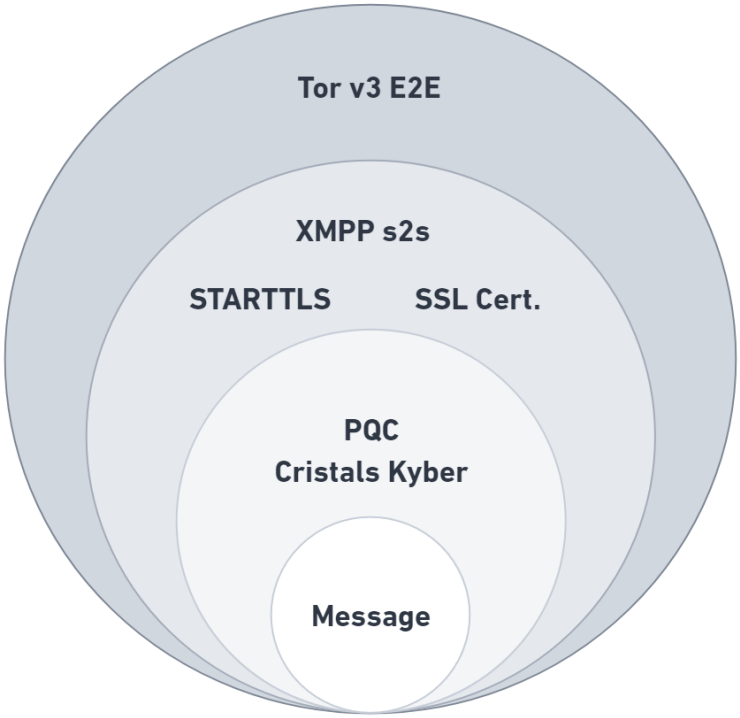


Figure 5.3: Security Layers

First, the message is encrypted using AES-256. The symmetric key was initially negotiated in the key exchange 5.2 with the Kyber PQC algorithm, which enhances confidentiality by protecting it from future quantum computer threats.

After securing the message with Kyber, the XMPP server initiates a TLS (Transport Layer Security) session to further encrypt the message. The TLS layer not only adds another level of confidentiality, but also contributes to integrity by ensuring that the data is not tampered with during transmission. The server uses a Secure Socket Layer (SSL) certificate for authentication, confirming the identity of the communicating parties and protecting against man-in-the-middle attacks.

Once the TLS session is established, the XMPP server sends the encrypted message through the Tor network. We use Tor version 3, which enhances privacy and anonymity, key aspects of confidentiality. This layer ensures that the identity of the communicating parties and the route of the data transmission are concealed. Tor uses Distributed Hash Tables (DHT) to improve availability. In a DHT system, data is distributed across a network of nodes, meaning there is no single point of failure.

This combination of PQC, TLS, and Tor v3 provides a robust security model that aims to protect against a wide range of potential threats.

Chapter 6

Evaluation

6.1 Project Overview

6.1.1 Achievement of Goals

In this project, our primary objective was to develop a secure peer-to-peer chat application using PQC for message encryption. We are pleased to report that we have successfully achieved this goal, as we have effectively demonstrated a working implementation of secure, quantum-resistant communication.

6.1.2 Areas for Improvement

We had originally planned to include features such as sending binary text files. However, due to time constraints, this functionality could not be implemented, as can be seen from the requirements evaluation in Section 3.2. Nevertheless, the proof of concept of integrating PQC into a peer-to-peer chat application was successfully demonstrated.
Further improvements, particularly in the area of user management and local key storage, are required for practical application and improved user experience. Possible directions for further development include the integration key vaults or docker secrets as well as optimizing the communication between frontend and backend which is done with MQTT at the moment.

6.2 Client Performance and Efficiency

6.2.1 Message Latency

Initial message delivery takes around 10 seconds. The first message involves setting up this circuit and establishing a secure, anonymous route, which takes time. Subsequent messages use the already established circuit, hence they are faster

6.2.2 Message Capacity

The application efficiently handles messages up to 100,000 characters. Larger messages, like 1 million characters, lead to a temporary service disruption. This issue has not been analysed in great detail. The log files indicate that this is related to the xmpp server which could not handle this vast amount of data. The message it self will be split into blocks by AES which causes alot of xmpp send requests to the server.
There is a change to message dropping due to MQTT misbehaviour. XMPP will handle the messages correctly, but due to spontaneous reconnects in MQTT, a message could be dropped. The log indicates that this issue is only related to the frontend; we did not see any drops in the cli client itself.

6.3 User Perspective

6.3.1 Functionality

The project functions as intended, with minor areas needing attention like the initial message delay and rare message drops. These issues do not detract from the overall success of the project.

Bibliography

[Bri23] Briar Project. Briar project - secure messaging, anywhere, 2023. Accessed: 2023-12-20.

[Dja23] Django Software Foundation. Django — the web framework for perfectionists with deadlines, 2023. Accessed: 2023-12-20.

[eja] ejabberd Documentation. `https://docs.ejabberd.im/`. Accessed: 2023-12-12.

[EMQ23] EMQX. The easiest guide to getting started with mqtt, 2023. Accessed: 2023-12-12.

[Gee23] GeeksforGeeks. Introduction to grover's algorithm - geeksforgeeks, 2023. Accessed: 2023-12-20.

[Mat23] Matrix.org. Matrix — open network for secure, decentralized communication, 2023. Accessed: 2023-12-20.

[Nat22] National Institute of Standards and Technology (NIST). Pqc candidates to be standardized and round 4, 2022. Accessed: 2023-12-12.

[Nat23] National Institute of Standards and Technology. Round 4 submissions - post-quantum cryptography, 2023. Accessed: 2023-12-20.

[Pid23] Pidgin. Pidgin - universal chat client, 2023. Accessed: 2023-12-20.

[pqc] PQ-Crystals: Post-Quantum Cryptography from Lattices. `https://pq-crystals.org/index.shtml`. Accessed: 2023-12-12.

[pro] Prosody IM Documentation. `https://prosody.im/doc`. Accessed: 2023-12-12.

[Sig23] Signal. Signal - private messenger, 2023. Accessed: 2023-12-20.

[Tai23] Tailwind Labs Inc. Tailwind css - a utility-first css framework for rapidly building custom designs, 2023. Accessed: 2023-12-20.

[The23] The Tor Project. Understanding entry guards, 2023. Accessed: 2023-12-12.

[Tor23] Tor Project. Tor project — anonymity online, 2023. Accessed: 2023-12-20.

[XMP23] XMPP Standards Foundation. Xmpp — extensible messaging and presence protocol, 2023. Accessed: 2023-12-20.