POST QUANTUM CRYPTOGRAPHIC COMMUNICATION

A Major Qualifying Project
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
degree of Bachelor of Science

Authored by
Hamayel Qureshi

Advised by:
Professor Yarkin Doroz
WPI

May 10th, 2021

Abstract

The advent and eventual proliferation of quantum computers will lead to many modern encryption and digital signature algorithms being rendered obsolete. The most popular public key/asymmetric cryptosystems (PKC) which include Diffie-Hellman, Elliptical Curve Digital Signature Algorithm (ECDSA), RSA, etc., are based on discrete log problems and integer factorization. This MQP examines modern cryptosystems and their susceptibility to quantum attacks using tools like the Shor's algorithm. It also implements the Nth Degree Truncated Polynomial Ring Units (NTRU) post quantum cryptosystem in a web accessible format which will be used to develop a post quantum communication progressive web application (PWA) accessible on any range of devices.

Acknowledgements

I would like to thank Professor Yarkin Doroz for his guidance and support throughout this MQP. His expertise on the subject matter was critical in accomplishing the goals of this project.

Table of Contents

Introduction

The protection of privacy is a growing concern in the modern world, with governments and organizations across the world attempting to extrapolate information about people constantly. Whether it is adversarial nations attempting to access the electronic correspondences of each other's government officials or social media giants like Google and Facebook keeping track of their users constantly, the world is becoming less secure. It is therefore necessary that the tools we use to keep our information safe and secure are up to date and evolving alongside new technologies.

One such exciting and possibly world changing technology is quantum computing. It is important to note that quantum computing will not be a replacement for classical computing entirely, but will be able to solve certain subsets of computational problems much, much faster than classical computers. Some of these problems may include: the simulation of molecules to understand the biological interactions of medicines, thus leading to breakthrough new treatments; optimizing logistics for all kinds of industries, such as those of delivery companies; advancement of Artificial Intelligence and Machine Learning, and many more.

However, the development of quantum computing also spells the end for many of the systems that keep our data secure. Quantum computers will likely break many public key cryptographic systems which are based on integer factorization and discrete logarithmic problems. These problems are incredibly difficult and time consuming for classical computers to solve, but easy for quantum computers. As a consequence, this will result in much of our encrypted data being vulnerable to attacks by third parties. Thankfully, many new cryptographic schemes have already been developed to exist in a 'post quantum' world. It is important then to test these cryptographic schemes and apply them in real world applications, which this project aims to do. A promising candidate as a possible post quantum cryptographic standard will be studied and implemented then applied to a web application built from the ground up as a 'proof of concept' of applied post quantum cryptography

Background

Cryptography and Cryptographic Encryption

To understand and integrate post quantum cryptography into a real world application, it is first paramount to understand the basics of cryptography and some of the tools it uses to achieve secrecy and privacy. At a basic level, cryptography can be defined as a practice for secure communication in the presence of 'adversaries' or 'third parties' that may wish to intercept communications [1]. In the modern world, it comprises study and development from a wide variety of disciplines, including Mathematics, Physics, Computer Science, Electrical and Computer Engineering, and more.

Because this project focuses primarily on the 'encryption' and 'decryption' of messages using cryptographic algorithms, a very simple example of the application of cryptography in modern communication is provided below:

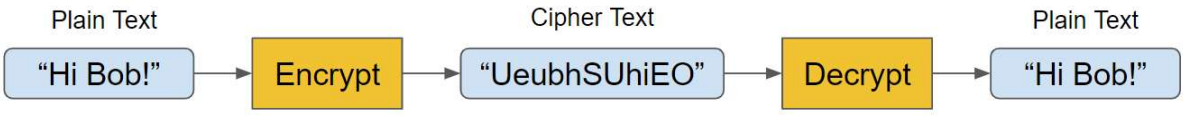

Figure 1. Basic Cryptographic Encryption/Decryption Example

In the above example, Alice may wish to send Bob a message but does not want anyone during the transit of that message to be able to tell what he said. She therefore leverages some cryptographic algorithm to encrypt her 'plain text' message to produce a 'cipher text' message that she can send to Bob. Bob is then able to use aspects of this cryptographic algorithm to decrypt the cipher text and get the plaintext message back. These aspects and cryptographic algorithms will be discussed shortly; specifically, this paper will discuss public key/asymmetric encryption, symmetric encryption, hashing (to a lesser extent), and all of this in the context of a post quantum world. However, this example sets up some of the very common nomenclature used in the discussion of cryptography. Namely, 'plain text' refers to the original message/data, 'cipher text' refers to the encrypted message/data, and the names Alice and Bob refer to sender and receiver respectively.

Symmetric Key Cryptography

Much of cryptographic encryption is done with a secret key (or sometimes a pair of keys, which will be discussed later) to encrypt and decrypt information. In symmetric key cryptography, a single such key is established and exchanged to both decrypt and encrypt information, hence the name 'symmetric.' As mentioned in **Cryptography and Cryptographic Encryption**, the encryption process renders the plain text messages unintelligible to any adversaries/third parties except for those with the symmetric key (which must be kept private!). This secret symmetric key could be an agreed upon secret code/password/phrase/etc., or as is more common a strong pseudorandom number/array of numbers from a secure random number generator (RNG) [2].

There are two main types of symmetric algorithms:

Block Ciphers:

Plaintext is broken into fixed length blocks. Each block is encrypted as one entity with any remaining data in a block being 'padded' with predetermined data (usually 0s) such that the block is always of the fixed length. Ideal block ciphers would work with 'massive key lengths' [3], though this is hardly practical and thus keys are scaled down for usability. Most modern symmetric cryptographic algorithms are based on block ciphers [3].

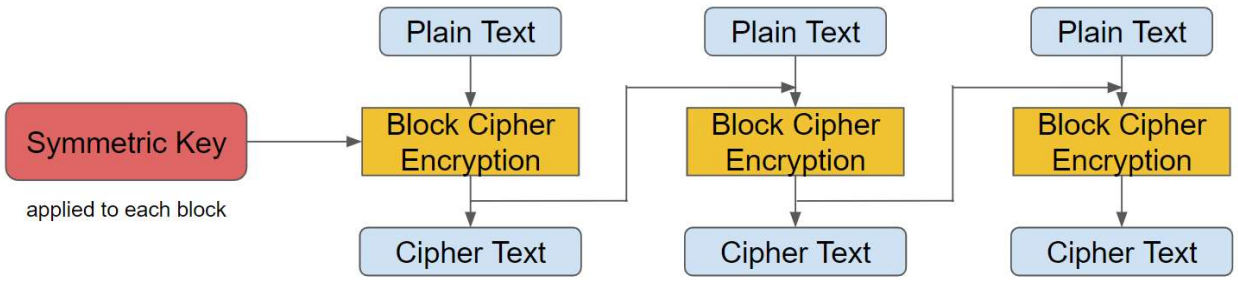

Figure 2. Basic Block Cipher Chaining Example

Stream Ciphers:

Each bit/byte of plaintext data is encrypted individually in a 'stream', hence the name. Stream ciphers can be thought of as finite state machines (FSMs) as each bit/byte is taken in a 'stream' and cipher text symbols are produced in the same order [4]. In the most basic sense, a stream cipher works by producing pseudorandom bits/bytes with a secret key and a pseudorandom number-only-used-once (nonce) which form a 'keystream' that is XORed with the plaintext, thus creating the cipher text.

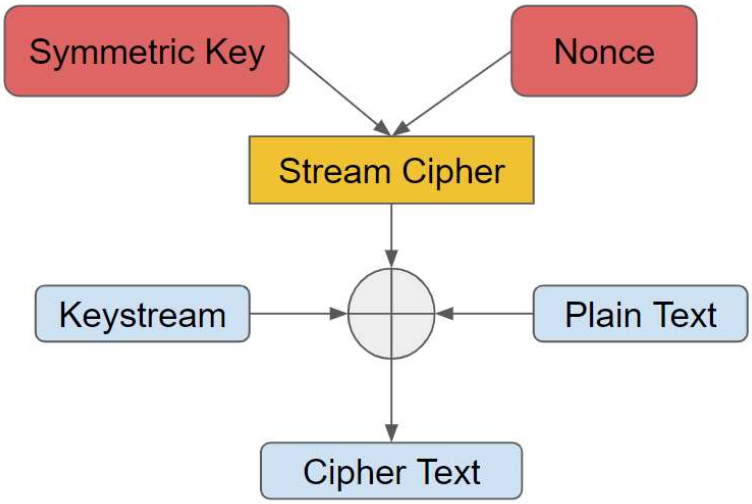

Figure 3. Basic Stream Cipher Example

Two of the most common types of symmetric algorithms are:

Data Encryption Standard (DES) / Triple Data Encryption Standard (3DES)

DES, also known as DEA (Data Encryption Algorithm), is a cryptographic algorithm based on the Feistel Cipher [5]. It maps fixed length inputs to fixed length outputs, operating on 64 bit blocks with a key size of 56 bits. Each key has 8 parity bits for transmission error detection. However, by modern standards, the 56 bit key is considered unsafe and vulnerable to brute force attacks which is why DES is considered *deprecated*. In lieu of this vulnerability, 3DES was chosen as a 'replacement' - it uses three 56 bit keys for a total of 168 bits, thus being considered safer than DES. However, 3DES operates on the same 64 bit blocks, thus making it vulnerable to the sweet32 birthday attack [6]. NIST also rates 3DES

at only 80 bits of security (vs. the 112 bit minimum - more on this later), thus *also* considered deprecated, though the federal government may still utilize it [7].

Advanced Encryption Standard (AES)

AES is by far the most popular symmetric cryptographic algorithm and will also be used extensively throughout the project, as its 256 bit variant is considered quantum secure [8], though there is a possibility of quantum algebraic attacks reducing AES to a problem of boolean equation solving [13]. AES is based on the 'Rijndael' encryption algorithms and works on a substitution-permutation network, with a certain number of rounds for the key size provided. With a key size of 128 bits/16 bytes, AES will have 10 rounds; a key size of 192 bits/24 bytes will have 12 rounds; a key size of 256 bits will have 14 rounds - each round will consist of four operations. Before moving onto the operations, it is important to mention that AES, similar to DES, operates on 'blocks' of data organized as a 128 bit/16 byte 4 by 4 matrix known as a 'state.'

The four rounds of AES are:

1. AddRoundKey/Key Expansion

The Rijndael Key Schedule is used to derive subkeys from the given key to perform 'key expansion' to generate a number of separate round keys for each round. The AddRoundKey operation XORs the state with the round's subkey.

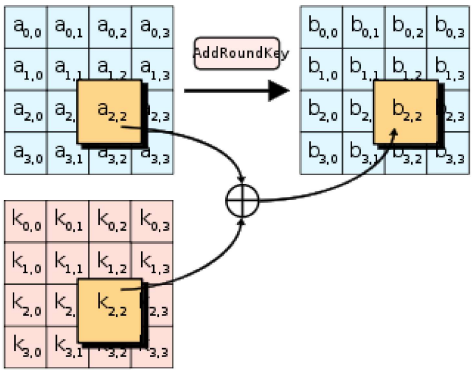

Figure 4. AddRoundKey Process[1]

[1] *https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Description_of_the_ciphers*

2. SubBytes/SBox Substitution

The substitution bytes (SubBytes) operation takes the current state (modified from the previous AddRoundKey round) and performs SBox substitution for obfuscation. In a nutshell, SBox substitution substitutes bytes from the state and replaces them with those in a complex lookup table which makes decrypting/deciphering without a key impossible on modern computers.

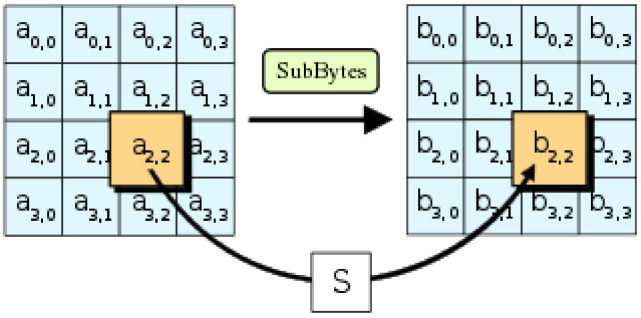

Figure 5. SubBytes Process[2]

As an example, given the following bitstream:

11000001000110011100110000100000101011001010000000010100101110000011
11010000011010110111001110001010011100110100101010100000111 0

we can perform SBox substitution using some Python code I wrote.

First, the input placed in a 4 by 4 matrix of hex values is represented as:

0xc1	0x19	0xcc	0x10
0x56	0x50	0x0a	0x5c
0x3d	0x06	0xb7	0x38
0xa7	0x34	0xaa	0x0e

It is then run through the Python script available in **Appendix E** to produce the following substituted output:

0x78	0xd4	0x4b	0xca
0xb1	0x53	0x67	0x4a
0x27	0x6f	0xa9	0x07
0x5c	0x18	0xac	0xab

[2] *https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Description_of_the_ciphers*

3. ShiftRows

Each row of the previously modified state is shifted by a certain value. The first row of the state is unchanged. The second row is shifted one place to the left. The third row is shifted two places to the left. The fourth row is shifted three places to the left. Each shifted row 'wraps around.' This step is a primary source of diffusion/obfuscation.

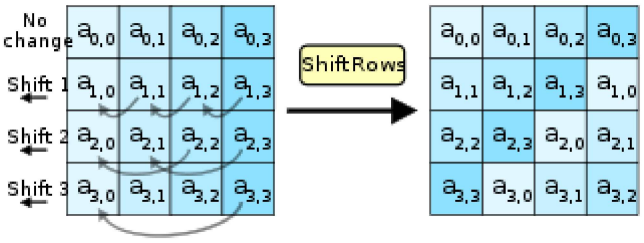

Figure 6. ShiftRows Process[3]

4. MixColumns

The MixColumns operation is a linear transformation of the previous state. Each 'column' of the state is multiplied by a 4 by 4 finite/Galois field with an output given as the inverse of the input and output. This step is a primary source of diffusion/obfuscation. This is a highly complex operation, and further reading on the matter is encouraged.

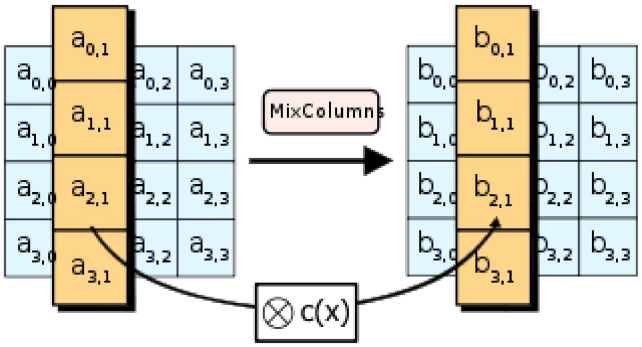

Figure 7. SubBytes Process[6]

There exist many more symmetric key cryptographic algorithms than the ones discussed here. Some of these include: International Data Encryption Algorithm (IDEA), Blowfish (which will be used for hashing purposes in this project), Rivest Cipher 4 (RC4 - a stream cipher), RC5, RC6, and more.

[3] *https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#Description_of_the_ciphers*

Public Key Cryptography

Public key cryptography, also known as 'asymmetric' cryptography, is based on computationally complex and difficult problems using a private (secret) and a public (shared) key. Both of these together form a key pair such that a message encrypted with a public key is only decryptable by its associated private key. Public key cryptography solves the issue of having to share keys, potentially insecurely, in order to establish secret/encrypted communication. Anyone with someone's public key can encrypt data, but only the person with the associated private key is able to decrypt the information - thus, public keys can securely be 'publicly' available without risk of eavesdropping. A private key should *never* be used to encrypt data - unless used for digital signatures - and should be kept private at all times [10].

As an example, consider Alice wants to send Bob a message. Bob will first need to generate a key pair (public and private). To do this, he will first choose a large random number, feed it through a key generation program, and then retrieve a public and private key.

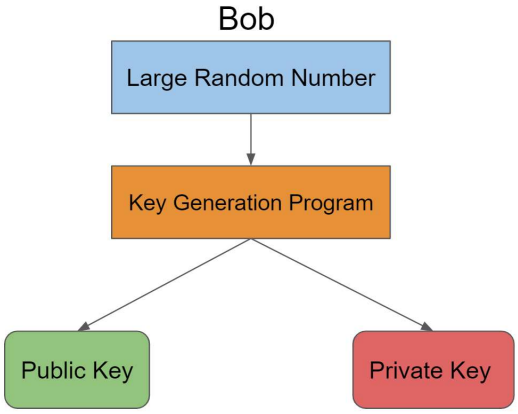

Figure 8. Simple Key Pair Generation Process

He then gives the public key to Alice or makes it publicly known, but keeps the private key secret. Alice is then able to use Bob's public key to encrypt her message, send it to Bob as cipher text so that no eavesdropper is able to decrypt the message, and finally Bob is able to use his private key to decrypt the message.

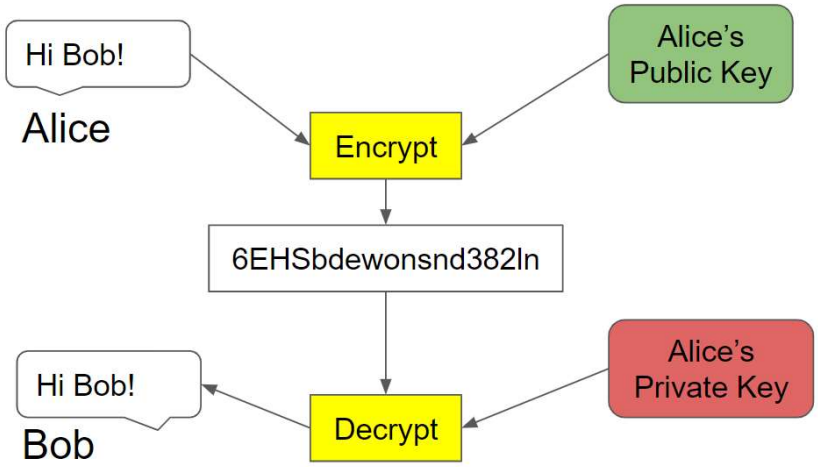

Figure 9. Public Key Encryption/Decryption Example

Public key cryptography also allows for 'digital signatures', where private keys are used for authentication instead of encryption as a form of signatures [10]. This process requires one to first hash the data and then encrypt it with the private key. Once the encrypted data (encrypted with the receiver's public key - one may choose not to encrypt the message but only rely on a digital signature to verify authenticity) and encrypted hash are received, the receiver can decrypt the encrypted hash using the sender's public key and compare the decrypted hash with a computed hash of the sent data [10]. The data is considered to be 'signed' if the hashed values are equal, and one can be reasonably certain that the sender is who they claim to be [10], as long as the private key has not been compromised. However, digital signatures will not be a factor in this project

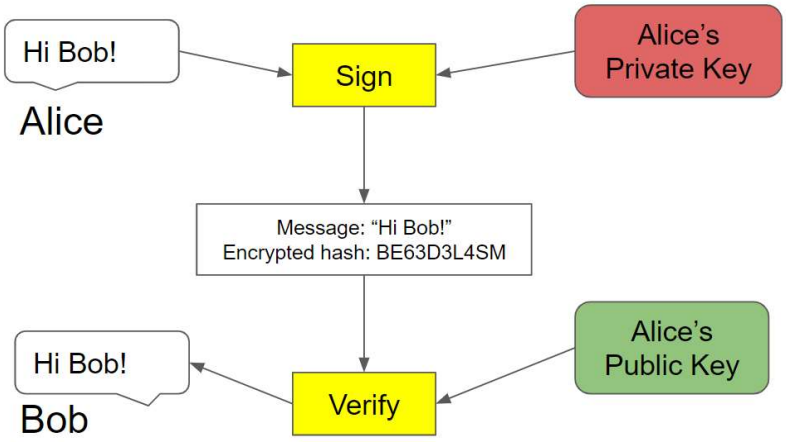

Figure 10. Digital Signature Example

Public key cryptography is slower than symmetric key cryptography due to the long key lengths and the complexity of the encryption algorithm due to the use of two keys, one of which is public [9]. In time sensitive applications, like real-time messaging, it is preferable to combine the use of public key and symmetric key cryptography. Public key cryptographic encryption, leveraging algorithms like RSA, can be used to securely transmit an AES symmetric key [9]. This will be a major factor in the security and encryption protocol of the post quantum communication application.

Some of the most popular and widely used public key cryptosystems include:

Diffie-Hellman Key Exchange: a public key protocol for securely exchanging keys over public channels based on discrete logarithm problems. It is not quantum secure [11].

ElGamal: a public key encryption scheme built upon the Diffie-Hellman Key Exchange, thus also not quantum secure.

Elliptic-curve Cryptography (ECDSA, ED25519, etc.): a public key cryptosystem based on the symmetry of elliptic curves ($y^2 = x^3 + ax + b$). Because ECC can be condensed to an integer factorization and discrete logarithm problem, it too is not quantum secure. A simple visual representation may be helpful in understanding what ECC is based on:

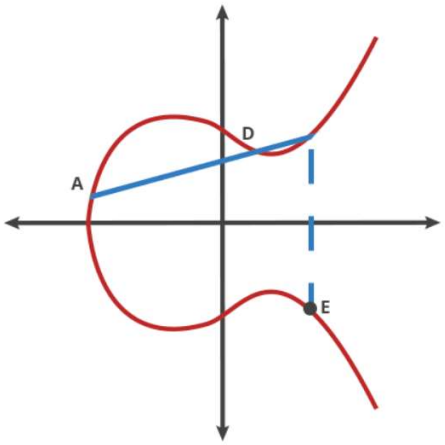

Figure 11. Simple Elliptic Curve Symmetry Example[4]

[4] https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

In fact, the majority of widely used public key cryptosystems rely on integer factorization and discrete logarithmic problems. To explore how these work, it is helpful to have a short rundown on the **RSA** (Rivest-Shamir-Adleman) cryptosystem as an example, which relies heavily on prime numbers.

Rivest-Shamir-Adleman (RSA) Key Generation, Encryption, Decryption

Key Generation

A public key is generated by choosing two large prime numbers: for the sake of simplicity, choose **11** and **17**, which will be referred to as **p** and **q** respectively. **p** and **q** must be kept private, and are part of your private key. **p** and **q** are multiplied to generate one of the public key numbers **N** which will be **11** * **7** = **187** [12]. The second public key number, which will be referred to as **e**, is computed by first subtracting 1 from **p** and **q** such that we get 10 and 16. These two values are multiplied to get the number **160** [12]. Now, **e** is chosen as a prime number that does not share prime factors with the number **160** - the prime factors of 160 are 2 and 5, so any prime number that is not one of those two values is acceptable, however we will choose **e = 7.** We are left with **p = 11, q= 17** as part of our private key and **N = 187** and **e = 7** as part of our public key, though we must keep our number **160**, which is a part of our private key, handy as well**.**

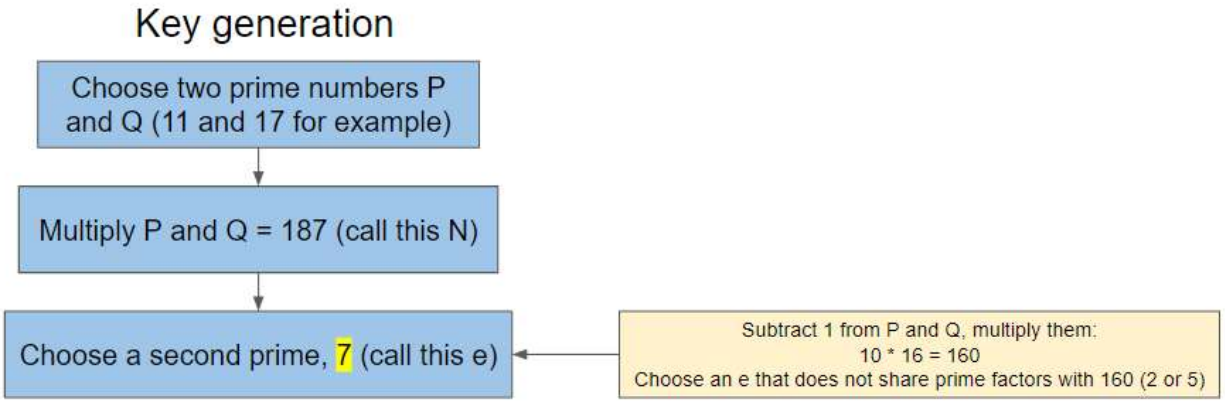

Figure 12. RSA Key Generation Example

Encryption

Create a message/data denoted by **m** - in this case the number **m = 99** (even non-numeric data is represented as numbers by one's computer after all). Retrieve the public keys **N = 187** and **e = 7.** Raise the **m** to the power of **e** such that $\mathbf{m^e = 99^7}$**.** After computing that value, we get a number greater than 93 trillion - this is where it starts getting extremely difficult and incredibly time consuming (a range of billions of years) for classical computers to attack public key cryptosystems, though in a real world scenario, much, much bigger primes than **11** and **17** would be chosen. Now, we divide $\mathbf{99^7}$ by our **N** to receive the value 498,430,667,329 with a remainder of **176** [12]. This remainder, called the modulus, is what we are interested in: it is the encrypted message. Only the person with the private key will be able to decrypt it.

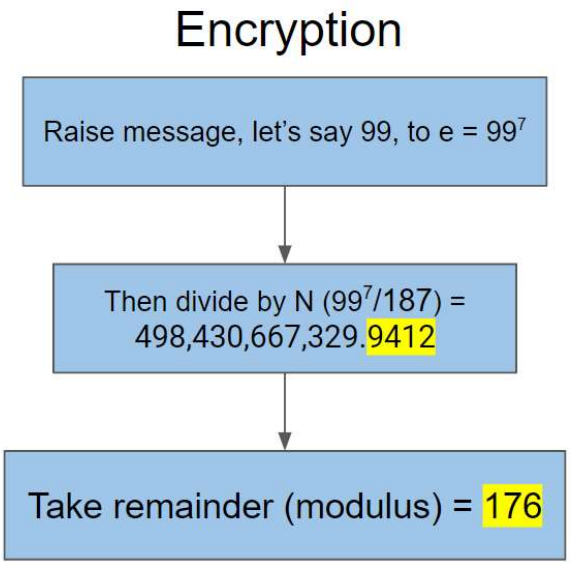

Figure 13. RSA Encryption Example

Decryption

Now that we have our encrypted cipher text **176**, we can use the private key (the numbers **p** and **q,** with the associated computed value **160**) to decrypt the message. First, we get a number **d** which is computed by adding **1** to **160** to get **161.** Now, **161** is divided by **e** (7) to get the number **d = 23.** Now, the encrypted message **176** is raised to the power **d** => $\mathbf{176^{23}}$. This value is divided by **N** ($\mathbf{176^{23}}$ **/ 187**) and the modulus (remainder) is found to be **99**, the original plain text message [12]. We have successfully decrypted our value.

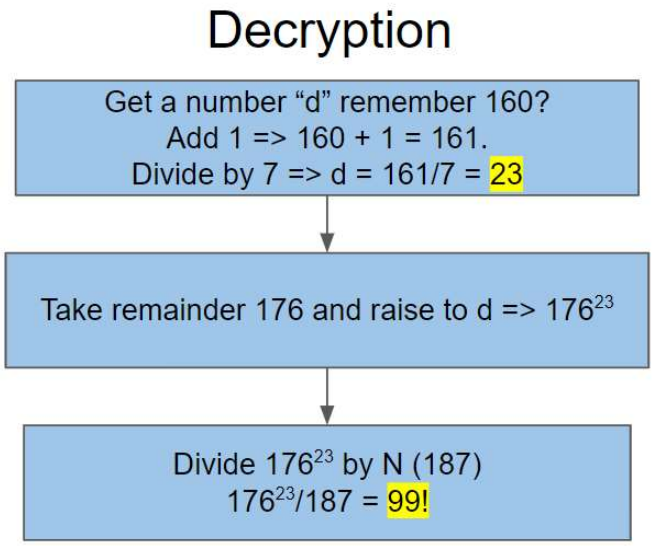

Figure 14. RSA Decryption Example

This brings us to the issue of quantum computers and how they will likely break public key cryptosystems based on prime factorization/integer factorization and discrete logarithm problems in the future. Before discussing a potential solution this project is based on, a quick rundown of quantum computers in the context of cryptography is helpful.

Quantum Computing and Its Effects On Cryptography

Quantum computers present a fundamentally different approach to computing and number crunching. Classical computers run and compute values based on binary sequences of 1s and 0s called bits whereas quantum computers operate on 'qubits' (quantum bits). Qubits can exist as either two states, written as |0⟩ or |1⟩, or some linear combination/proportion of both states known as a 'superposition.' It is easiest to visualize the 'state' of a qubit using the Bloch sphere where a qubit can be of value |0⟩ or |1⟩ at each pole of the sphere or some value in between on the surface of the sphere [13].

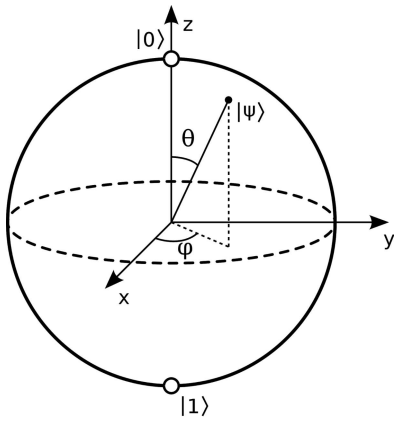

Figure 15. Bloch sphere, geometric representation of qubit values[5]

Quantum computers can take advantage of the physical properties of such qubits by executing operations not only based on the binary states |0⟩ and |1⟩ but all possible superpositions of those at the same time [13]. This allows quantum computers to be incredibly efficient at certain tasks, though they are *not* a replacement for all applications by any means. While an exciting technology with undeniable benefits for the near future, the development and eventual proliferation of quantum computers is bad news for data protection. This is especially true for many modern public key cryptosystems that, as mentioned, are based on the inherent difficulty for computers to solve prime integer factorization and discrete logarithm problems. It, for example, would be sufficient for a quantum computer to decrypt data with no knowledge of the private key whatsoever [13].

[5] https://en.wikipedia.org/wiki/Qubit

Specifically, quantum computers are able to leverage algorithms like Grover's algorithm and **Shor's algorithm**. For the purposes of this project, we will examine the latter shortly.

Cryptographic Algorithm	Type	Purpose	Impact from large-scale quantum computer
AES	Symmetric key	Encryption	Larger key sizes needed
SHA-2, SHA-3	----------	Hash functions	Larger output needed
RSA	Public key	Signatures, key establishment	No longer secure
ECDSA, ECDH (Elliptic Curve Cryptography)	Public key	Signatures, key exchange	No longer secure
DSA (Finite Field Cryptography)	Public key	Signatures, key exchange	No longer secure

Figure 16. Impact of Quantum Computing on Popular Cryptographic Algorithms[6]

The above table from the U.S's National Institute of Standards and Technology (NIST) gives a rough outline of how most popular public key cryptographic algorithms will need to be replaced with quantum-resistant ones, though symmetric key and hash protocols would benefit from large key sizes or larger outputs.

A quantifiable measure of security from cryptosystems is measured in 'bits of security.' NIST defines 'bits of security' as "a number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system" [16]. That number of steps is defined as 2^N where N is the bits of security, and 2^N as the total number of steps required to break the algorithm. All NIST approved algorithms must provide at *least* 112 bits of security. The figure below , courtesy of TechBeacon [17], gives a comparison for the bits of security RSA and AES provide on classical versus quantum computers.

[6] https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf

Type of Attack		Symmetric Encryption		Public Key Encryption		
		Key Length	Bits of Security		Key Length	Bits of Security
Classical Computers	AES-128	128	128	RSA-2048	2048	112
	AES-256	256	256	RSA-15360	15,360	256
Quantum Computers	AES-128	128	64	RSA-2048	2048	25
	AES-256	256	128	RSA-15360	15,360	31

Figure 17. Comparison of Bits of Security of AES and RSA on Quantum Computers [17]

While AES-256 remains viable according to NIST's standards at 128 bits of security, both RSA-2048 and RSA-15360 are rendered unusable at merely 25 and 31 bits of security respectively. This brings us to a brief discussion on one of the tools quantum computers may employ to break current public key cryptosystems.

Shor's Algorithm

Shor's algorithm consists of a number of highly complex operations that require an intimate understanding of quantum computing and cryptography. Therefore, further reading beyond this very short summary is encouraged. In the most basic sense, the algorithm solves the problem of finding the prime factors of a number given an integer. Compared to classical computers that solve this in sub-exponential time $O(e^{1.9(log(N))^{1/3}(log(log(N))^{2/3}})$ using tools like the general number field sieve[7], quantum computers using an algorithm like Shor's could solve this in drastically less polynomial time $O((log(N)^2(log(log(N))(log(log(log(N)))))$ [18] using quantum logic gates and the quantum Fourier transform [19]. The algorithm consists of a classical part and a quantum part [19].

[7] https://mathworld.wolfram.com/NumberFieldSieve.html

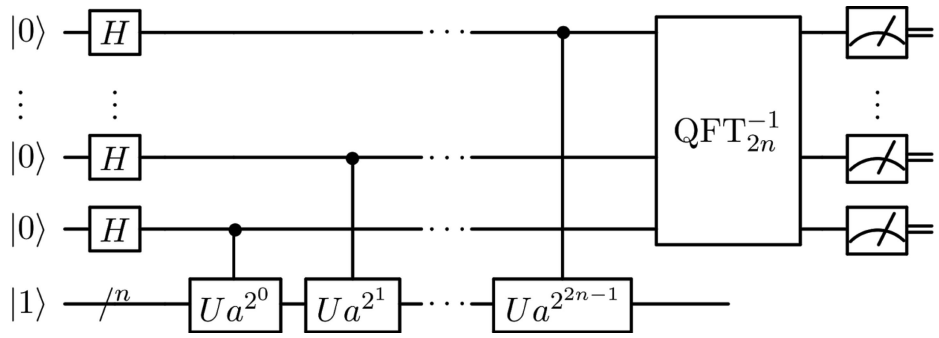

Figure 18. Shor's Algorithm Quantum Subroutine[8]

To crack RSA-2048, it would take a classical computer in the trillions of years whereas a quantum computer with 20 million qubits using Shor's algorithm could do this in 8 hours [21]. However, successfully implementing Shor's algorithm would require a quantum computer with a number of qubits substantially greater than and an error rate much lower than current quantum computers (for example, Google's Sycamore quantum computer only operates on around 50 qubits) [20].

Thankfully, current best estimates put such quantum computers at least 10-20 [20] years away giving us ample time to develop, test, implement and transition over to quantum resistant cryptographic algorithms. The issue, however, is that many popular modern communication platforms are built using non quantum secure cryptosystems. Most popular modern communication applications like Facebook's Messenger, Instagram, and WhatsApp, or even platforms lauded for their privacy features like Signal and Telegram are built on the Signal Protocol developed by Open Whisper Systems, which is **not** quantum secure. This is due to its use of ECDH (Elliptic Curve Diffie Hellman) 25519 key exchanges, which as discussed, are not quantum secure [22]. Even though the communication aspect is protected with AES-256 symmetric encryption, any adversaries with sufficient quantum computing power could decrypt the symmetric keys with Shor's algorithm during key transfer or exchange [22].

[8] https://en.wikipedia.org/wiki/Shor%27s_algorithm#/media/File:Shor's_algorithm.svg

NTRU and Quantum Resistant Cryptography

The National Institute of Standards and Technologies (NIST) is in the process of evaluating post quantum cryptosystems as candidates for standardization[9]. The third round finalists in the standardization effort for public key cryptosystems include: Classic McEliece (code based), CRYSTALS-KYBER (lattice based), SABER (lattice based), and **NTRU** (lattice based), the subject of this section and the post quantum cryptosystem to be used in this project.

There are several different types of post quantum cryptography [22] which include:

- Code based
- Isogeny based
- Lattice based
- Multivariate
- Hash based

We will briefly cover the first three types.

Code based: based on error correcting codes. Data is encrypted into 'code' with secret errors added [22]. The private key can be thought of as the code's parameters. The public key is a scrambled generator matrix [22].

Isogeny based: based on finding operations between elliptic curves called isogenies [22]. The elliptic curves themselves are part of the public key and the private key is the isogeny between the curves.

Lattice based: as the name implies, it is based on mathematical lattices and the inherent difficulty of the 'shortest vector' problem in high dimensional lattices [22]. A lattice is defined as a set of points $\boldsymbol{s}$, our secret key, and a public key $\boldsymbol{p}$ which is a scrambled version of the lattice. Data is mapped onto a point on $\boldsymbol{s}$ with an error added such that the point is

[9] https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms

still closer to the original point on ***s*** than all other points on the lattice [22]. Decrypting messages requires one to know the ***s*** so that the shortest vector to the point may be found.

The next few pages will take a somewhat detailed mathematical look into the NTRU cryptosystem - if you wish to skip the technical details, please proceed to page 27.

NTRU operations are based on objects inside a polynomial ring $R = \frac{Z[X]}{(X^{N-1})}$ with a polynomial degree at $N - 1$ such that: $a_0 + a_1x + a_2x^2 + \ldots + a_{N-1}x^{N-1}$ [23]. To discuss NTRU further beyond the simple definition of lattice based cryptography, we will use the following parameters:

- **N** - polynomials in ring R with degree N-1 (non secret)
- **q** - large modulus to which each coefficient is reduced (non secret)
- **p** - small modulus to which each coefficient is reduced (non secret)
- **f** - a polynomial private key
- **g** - a polynomial used to generate public key h from f
- **h** - polynomial public key
- **r** - random 'blinding' polynomial
- **d** - coefficient

Key Generation

NTRU key generation can be explained in 3 steps:

Step 1: Alice chooses 2 *secret* polynomials **f** and **g** in **R** (ring of truncated polynomials) [22]. Each polynomial must have an inverse.

Step 2: Alice will compute the inverse of **f** mod **q** and **f** mod **p** such that the following properties are satisfied:

- $f * fq^{-1} = 1 \ mod \ q$
- $f * fp^{-1} = 1 \ mod \ p$

Step 3: Alice will compute the product of the polynomials, **h**: $h = p * ((fq) * g) \ mod \ q$ [22]

This results in the public key **h** and the private key **f** and **fp**.

Because this project will use NTRU for key generation, examples may help better understand the process. Consider the parameters **N, p**, **q,** and **d** are chosen as 7, 3, 41, and 2 respectively. We can then compute Bob's private and public keys:

Step 1:

$$f(x) = x^6 - x^4 + x^3 + x^2 - 1$$
$$g(x) = x^6 + x^4 - x^2 - x$$

Step 2:

$$fq(x) = f(x)^{-1} \ mod \ q = 8x^6 + 26x^5 + 31x^4 + 21x^3 + 40x^2 + 2x + 32 \ mod \ 41$$
$$fp(x) = f(x)^{-1} \ mod \ p = x^6 + 2x^5 + x^3 + x^2 + x + 1 \ mod \ 3 \ \textbf{(private key)}$$

Step 3:

$$h(x) = p * fq * g \ mod \ q = 20x^6 + 40x^5 + 2x^4 + 38x^3 + 8x^2 + 26x + 30 \ mod \ 41 \ \textbf{(public key)}$$

Encryption

NTRU encryption can also be explained as a three step process:

Step 1: Alice converts message to polynomial **m** with coefficients mod **p** between $-\frac{p}{2}$ and $\frac{p}{2}$ (centered lift) [22]

Step 2: Alice chooses a polynomial **r** to obscure **m** [22]

Step 3: Alice encrypts the message as **e**: $e = r * h + m \ mod \ q$ [22]

We can continue the example from the key generation process with parameters **N, p**, **q,** and **d** as 7, 3, 41, and 2 respectively.

Step 1: Consider message **m(x)**

$$m(x) = -x^5 + x^3 + x^2 - x + 1$$

Step 2: Consider **r(x)**

$$m(x) = x^6 - x^5 + x - 1$$

Step 3:

$$e = r * h + m \ mod \ q$$

$$e(x) = 31x^6 + 19x^5 + 4x^4 + 2x^3 + 40x^2 + 3x + 25 \ mod \ 41$$ **(encrypted message)**

Decryption

NTRU decryption can also be explained as a three step process:

Step 1: Bob receives the message **e** from Alice. With the private polynomial **f** he finds a polynomial **a** such that $a = f * e + mod\ q$ where the coefficients of **a** are in an interval of length **q** [22].

Step 2: Bob solves for a polynomial **b** such that $b = a\ mod\ q$ [22].

Step 3: Bob retrieves his private polynomial **fp** and finds the plain text message **c** such that $c = fp * b\ mod\ p$

Continuing on from where the previous example let off, we can decrypt the message **e**:

Step 1:

$$a = f * e + mod\ q$$

$$a = x^6 + 10x^5 + 33x^4 + 40x^3 + 40x^2 + x + 40\ mod\ 41$$

Step 2: Centerlift modulo q such that

$$b = a\ mod\ q = x^6 + 10x^5 - 8x^4 - x^3 - x^2 + x - 1\ mod\ 3$$

Step 3: Reduce *a(x) mod p* and get **c**

$$c = fp * b\ mod\ p = 2x^5 + x^3 + x^2 + 2x + 1\ mod\ 3$$

Center lift *mod p* to finally retrieve the decrypted plain text

$$m(x) = -x^5 + x^3 + x^2 - x + 1$$

This brings us to the crux of this project: developing a real-world application built on a modern and accessible tech stack that utilizes post quantum cryptography to ensure absolute privacy despite the eventual proliferation of quantum computers. In short, this project is to be a proof of concept implementation of the NTRU protocol in a real time communication application.

Methodology

The Tech Stack

'Tech stack' refers to the amalgamation of tools, frameworks, libraries, languages, and technologies used to build the frontend, backend, and APIs (application programming interfaces) that tie everything together for the application. This section will briefly cover the choice of tech stack and tools being used to build this project.

Language

Both the front and back end systems are built using TypeScript, an open source language built on top of JavaScript. TypeScript adds static type checking to JavaScript, turning it into a statically typed language from a dynamically typed one[10]. It therefore allows one to catch type errors as soon as they occur with strong IDE (integrated development environment) support, unlike in plain JavaScript where type errors are common and hard to catch.

Database

The database used for this application is PostgreSQL, a relational database management system. Although migrating over to a NoSQL database like MongoDB would be a viable option in the future if the application is to be rapidly scaled up and readied for release. Nonetheless, the table like format of relational databases makes it simpler to develop and prototype as a proof of concept. The ORM (object-relational mapping) to interact with the database is TypeORM, chosen for its strong integration with TypeScript.

Server

The server runs on Node.js, a "JavaScript runtime built on (Google) Chrome's V8 JavaScript engine"[11]. The server framework is Express.js, the most popular server framework for Node.js[12].

[10] https://www.typescriptlang.org/
[11] https://nodejs.org/
[12] https://expressjs.com/

API Layer

Rather than a traditional REST (representational state transfer) API, this application will use GraphQL. GraphQL makes it simple to query data from a database with exactly the shape and data the application requires. For example, if one wished to ask the database for users but only wanted their unique id, username, and publicKey, the query would look like:

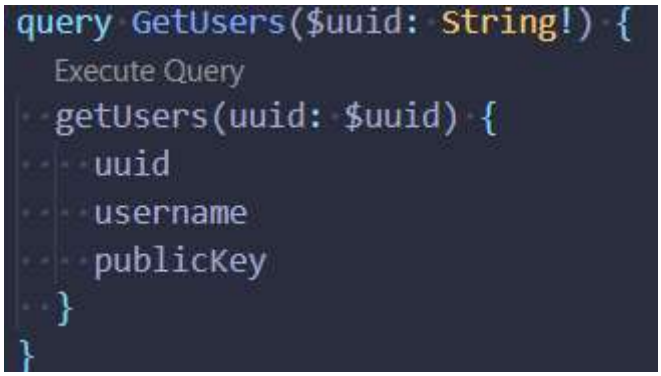

Figure 19. GraphQL getUsers Query Example

GraphQL will be integrated into the back and frontend with Apollo Server and Apollo Client which provide excellent developer tools. TypeGraphQL will be used to make the interaction between GraphQL and our database seamless.

Frontend

The frontend framework used is React. The goal of this project is to build a PWA (Progressive Web Application) that is accessible on both desktop browsers and mobile phones. It must, therefore, be 'reactive' to several different scenarios, screen sizes, etc. React provides a number of tools that make this relatively simple. The UI library used will be Material UI due to its extensive UI components list, excellent documentation, and styling capabilities. This will make prototyping and developing the UI much, much faster than manually writing all the CSS and React TypeScript code necessary to make good looking and functional components. The application *will* be custom styled with a few theme settings (dark mode is a must), but Material UI can make this process much simpler.

Cryptographic Libraries

A number of cryptographic libraries will be used in this project.

bcryptjs: the 'hash' and 'compare' modules are used on the server side to hash and verify passwords. It is a JavaScript implementation of the Blowfish cipher[13].

scrypt-js: the 'scrypt' module form scrypt-js is used to hash passwords on the server side. The 'scrypt' algorithm converts passwords into 'fixed length arrays of bytes'[14] which is necessary for encrypting private keys as we will discuss later on.

aes-js: several modules from this pure JavaScript implementation of the AES block cipher algorithm will be used throughout the project.

NTRU: an open source WebAssembly implementation of the NTRU cryptographic protocol with a JavaScript wrapper for ease of use in web applications. It was originally developed by 'Cyph'[15], a company focused on encrypted communication and telehealth. Due to development having been seemingly abandoned based on commit history, parts of it were updated to bring it to a usable state. Please refer to the background section **NTRU and Quantum Resistant Cryptography** and **Appendix A** for technical details on how NTRU and the NTRU library works.

[13] https://www.npmjs.com/package/bcryptjs
[14] https://www.npmjs.com/package/scrypt-js/v/2.0.4
[15] https://www.cyph.com/

The Protocol

To implement the application, a protocol that plans and describes how each step of the application works and encrypts information is necessary. For the following sections, 'client' refers to the frontend application that a user interacts with.

Registration Process

The client creates a 'username' and sends it to the server in plaintext - this information is not hashed, as it would make displaying and retrieving usernames difficult for both the frontend and server. It may be a good idea, in the future, to however store usernames as *un*salted hashes of the username to add an extra layer of obscurity. Continuing on, the server checks if a user with that username does not already exist, and if not, sends back an 'ok' letting the client proceed.

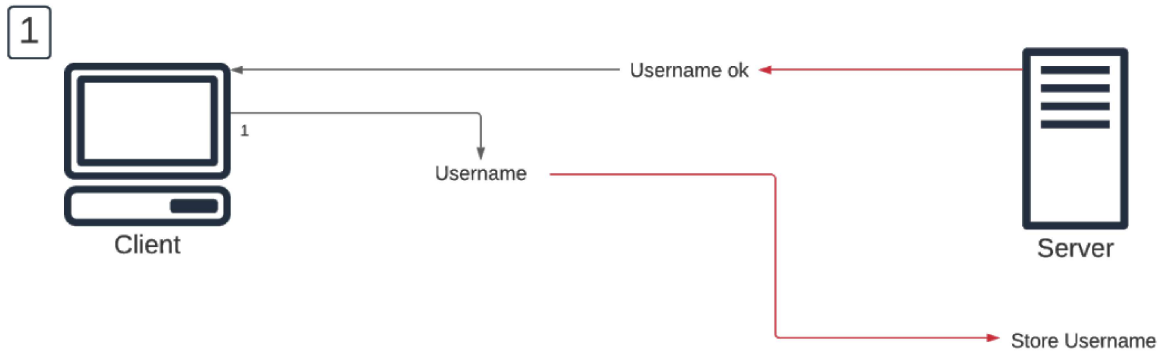

Figure 20. Registration Username Process

The client also chooses a password with some character and minimum length restrictions. The client salts and hashes the password using **scrypt** to generate a 256 bit/32 byte hash. The first half of this hash (the first 16 bytes) will be used as an encryption key and the second half (last 16 bytes) is used as an authorization key (auth key) and sent to the server. The authorization key is hashed once again using **bcrypt** on the server side to prevent pass-the-hash attacks [24] and then stored in the database.

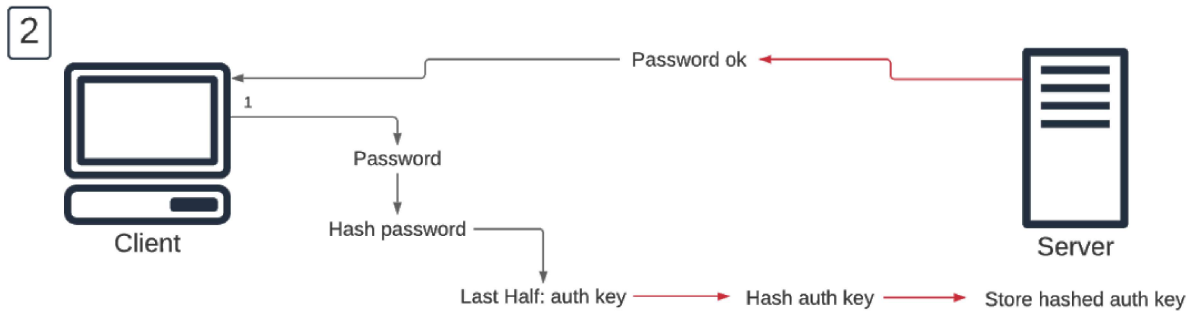

Figure 21. Registration Password Process

The custom NTRU library is used to generate a quantum secure private key. This private key is first encrypted with the authorization key and then sent to the server. It is necessary to keep this information on the server side due to the nature of this application - it is designed to run on browsers (or as a browser application on phones) which do not have permanent storage capabilities.

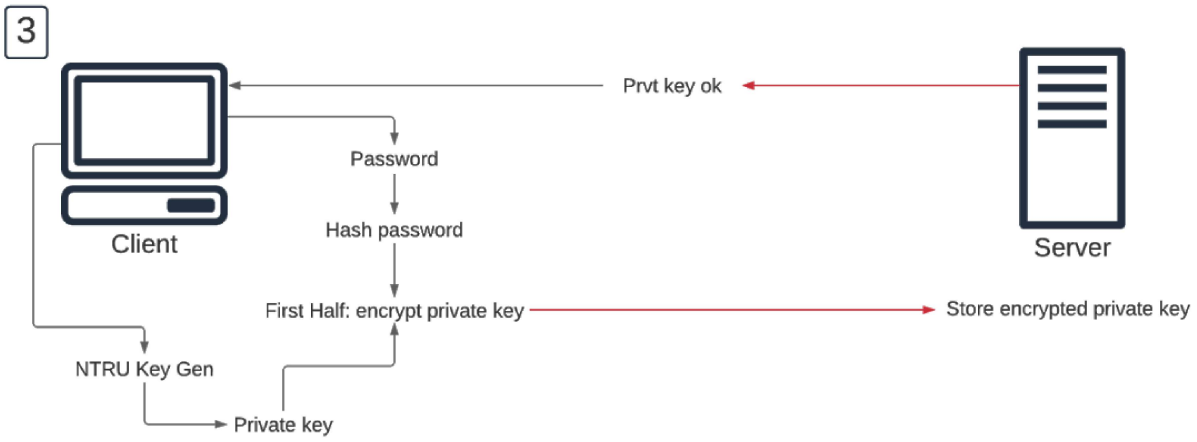

Figure 22. Registration Private Key Process

The NTRU library also produces a public key which is sent to the server in plaintext and stored.

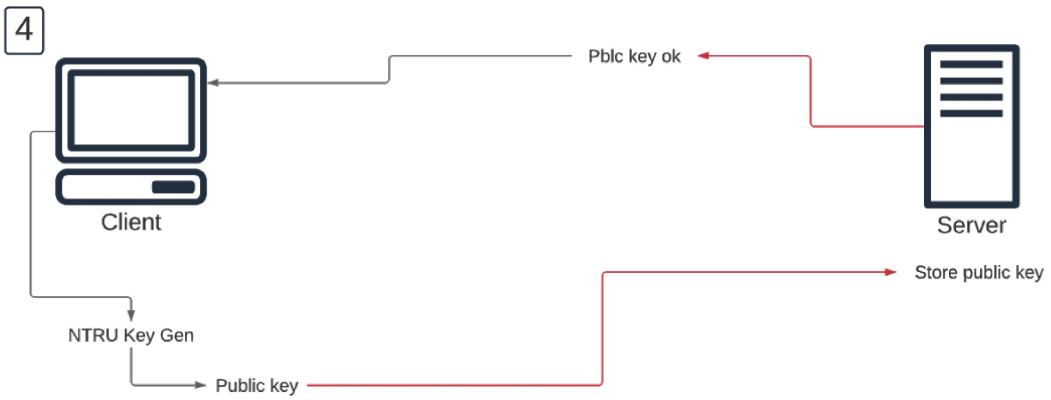

Figure 23. Registration Public Key Process

Login Process

The login process operates very similarly to the registration process. The username is sent to the server, and the server checks if the user with that username actually exists:

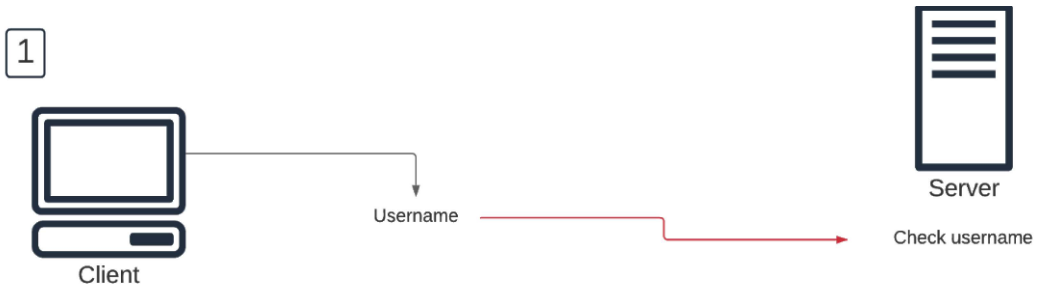

Figure 24. Login Username Process

The password is again hashed using **scrypt**. The auth key section (last 16 bytes) are sent to the server to check if the password is correct.

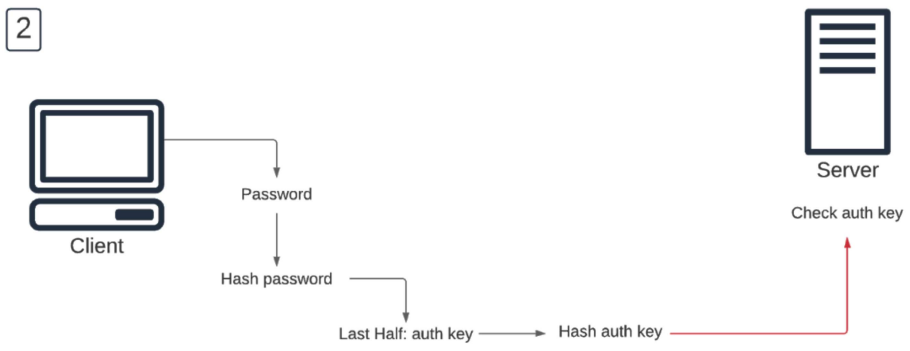

Figure 25. Login Password Process: Auth Key

The encryption key (first 16 bytes of the hashed password) is stored on the client side for use later. If the authorization key was correct, the server generates a cookie with a JSON Web Token (which will be used for validating a user's session with the server) and sends it back to the client for client storage along with the public key and encrypted private key. The encrypted private key is then decrypted using the encryption key.

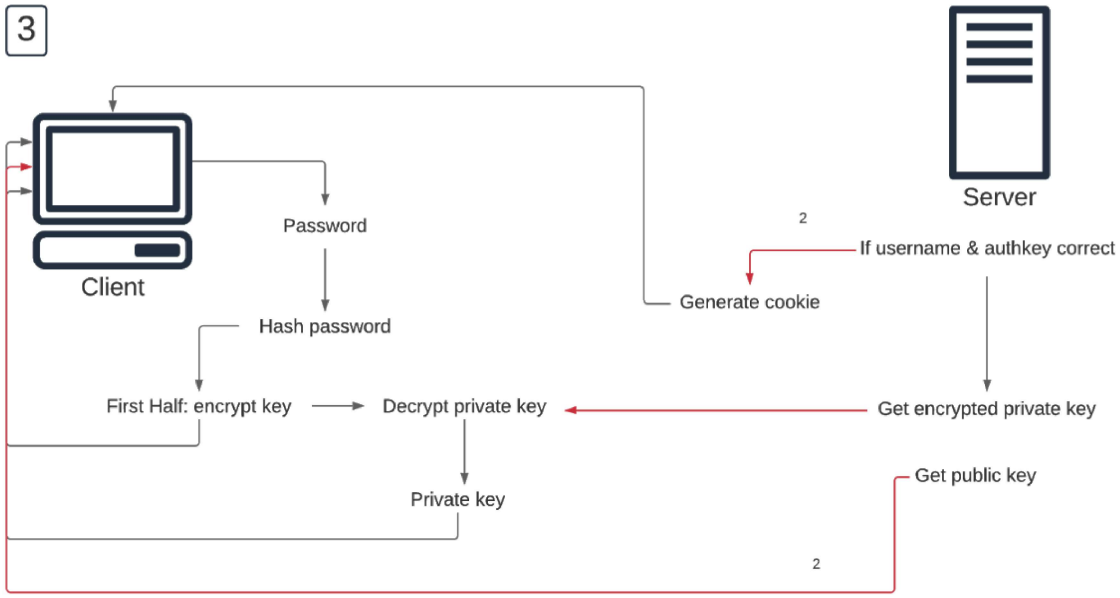

Figure 26. Login Password Process: Encryption Key and Cookies + Key Pair

Add Friend Process

As mentioned in the background section, public key cryptography is often used for establishing shared keys and safely transferring them between parties. The shared key is then used for the purposes of communication using symmetric encryption due to its speed and efficiency. The 'add' and 'accept' friend processes will establish and transfer symmetric keys between two users.

If a user wishes to add a friend, they will search the database of existing users and check for their public key.

Figure 27. Add Friend Check For Public Key Process

If the user exists, their public key is retrieved. The **aes-js** library is then used to establish a symmetric key. The shared symmetric key is encrypted with the friend's public key and stored in the database as the friend's encrypted symmetric key.

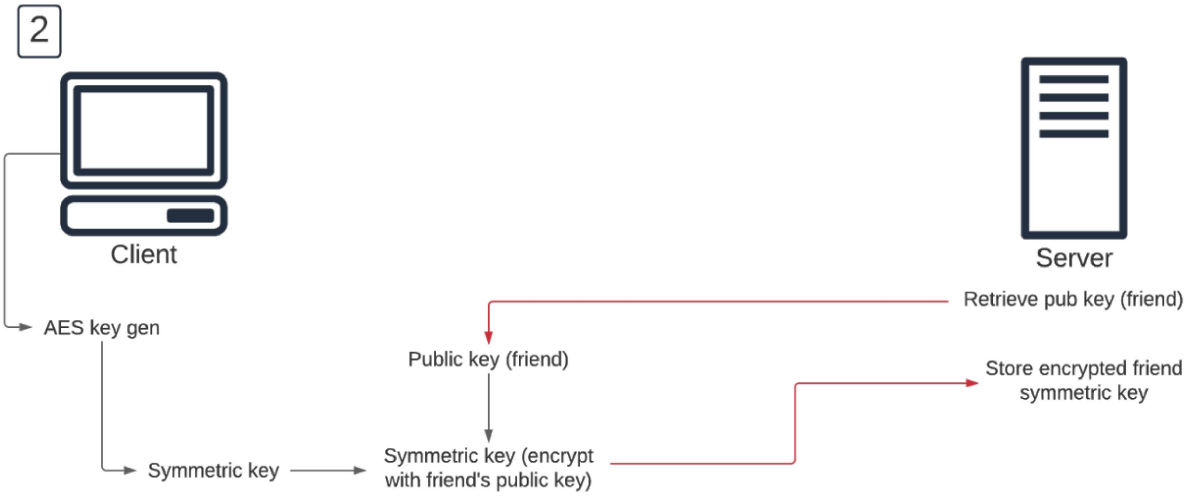

Figure 28. Add Friend Symmetric Key Process Part 1

The shared symmetric key is also encrypted with the user's encryption key and stored in the database.

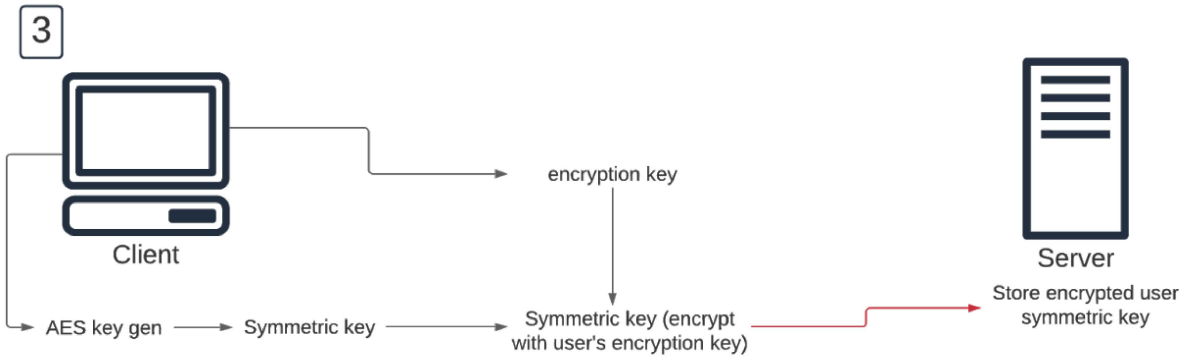

Figure 29. Add Friend Symmetric Key Process Part 2

Accept Friend Process

The user receiving the friend request may choose to deny the friend request. In this case, all associated data (in particular the shared symmetric key) is deleted from the database. However, if the user receiving the friend request chooses to accept the friend request, the user's encrypted symmetric key is retrieved from the database/server. The user is then able to use their private key to decrypt the symmetric key, then encrypt it once more with their encryption key and send and store that in the database. In this way, the shared symmetric key is successfully transferred between users in a quantum secure manner.

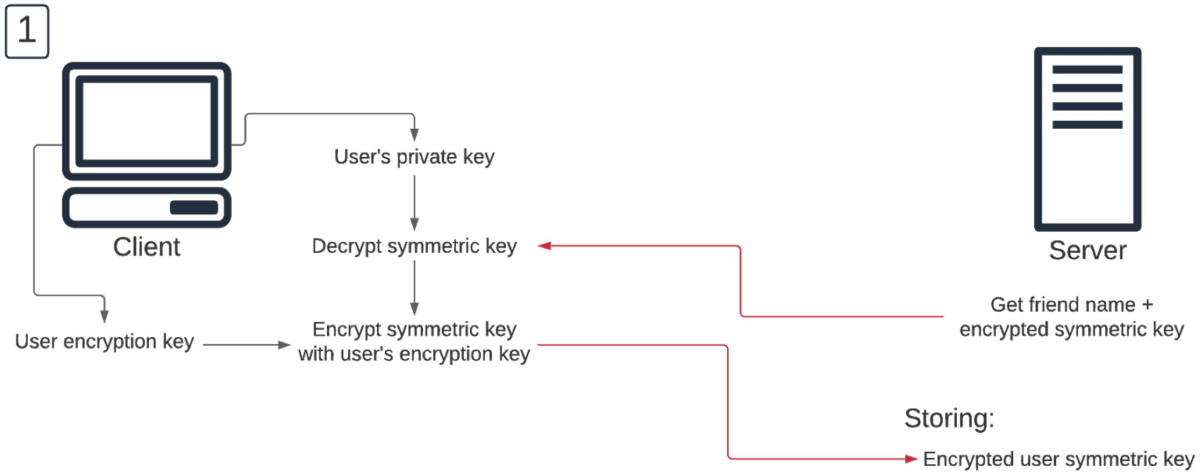

Figure 30. Accept Friend Process

Communication Process - Sending Messages

Each user on their respective client will check for each other's encrypted symmetric keys.

Figure 31. Communication Check For Symmetric Key Process

The symmetric key is then retrieved and decrypted using each user's respective encryption keys.

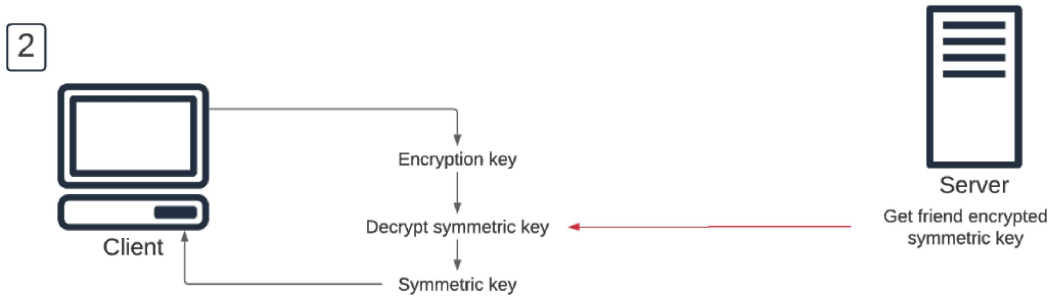

Figure 32. Communication Retrieve and Decrypt Symmetric Key Process

When the user has written a message and hits send, the client encrypts the message using AES-256 from the **aes-js** library with the symmetric key. The encrypted message is then sent to the server to be stored in the database. At no point does a plain text message ever reach the server - all messages stored are encrypted.

Figure 33. Communication Send Message Process

Communication Process - Receiving Messages

Messages from both users (sent by the user themselves and those sent by the other user) are retrieved from the database. The messages are all encrypted, so they must be decrypted using the shared symmetric key at which point the plaintext messages are successfully received.

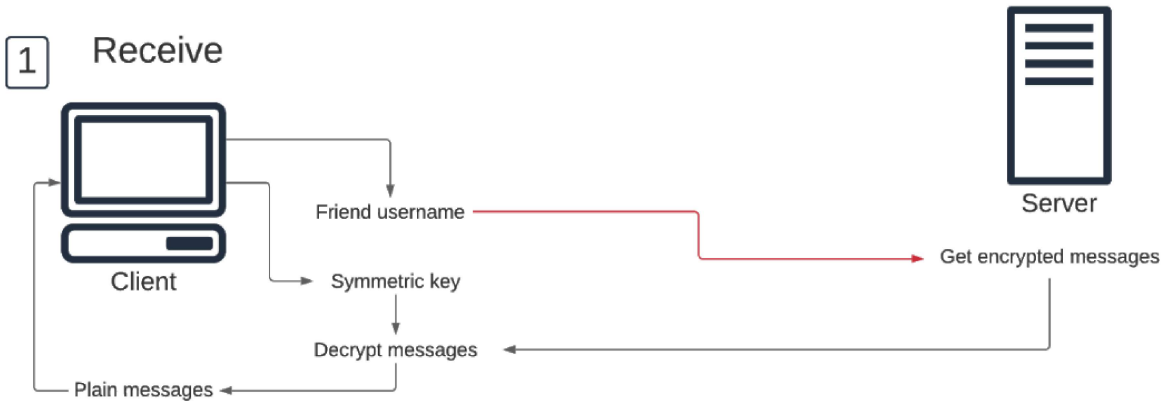

Figure 34. Communication Receive Message Process

This protocol ensures that even if an adversary or third party were to somehow get access to the whole database, they would not be able to tell nor decrypt the messages being sent in between users. Because public and private keys are established using **NTRU**, this ensures post quantum security. Essentially, this protocol achieves post quantum **End-to-End (E2E)** encryption. Users' passwords are also hashed twice, once on the client, and once again on the server - this should prevent any password leaks or decryption from the database. However, if a user's password were to be compromised by, for example, the user sharing their password online or keeping it stored in plaintext somewhere insecure, their conversation history could be accessed by an adversary/third party. 2FA (two factor authentication) would be a solid prevention tactic for this situation, though that is beyond the scope of this project.

Database Design

Before implementing the database design into code, it is necessary to plan out the database schema. The Entity Relationship Diagram below describes the data stored in each 'entity'/table. The rows, which represent the data being stored, are given self-explanatory names for ease of understanding and implementation. However, as a quick overview, the database consists of three primary entities: Users, Chats, and Messages. The Users table stores information about the users, including their usernames, passwords, public and encrypted private keys, etc. The Chats table stores information about the communication between two users. The Messages table stores information about messages being sent between users. A User may have many chats and a Chat may have two users and many messages. The 'chats_members_users' is the table that ties all three of the primary tables together.

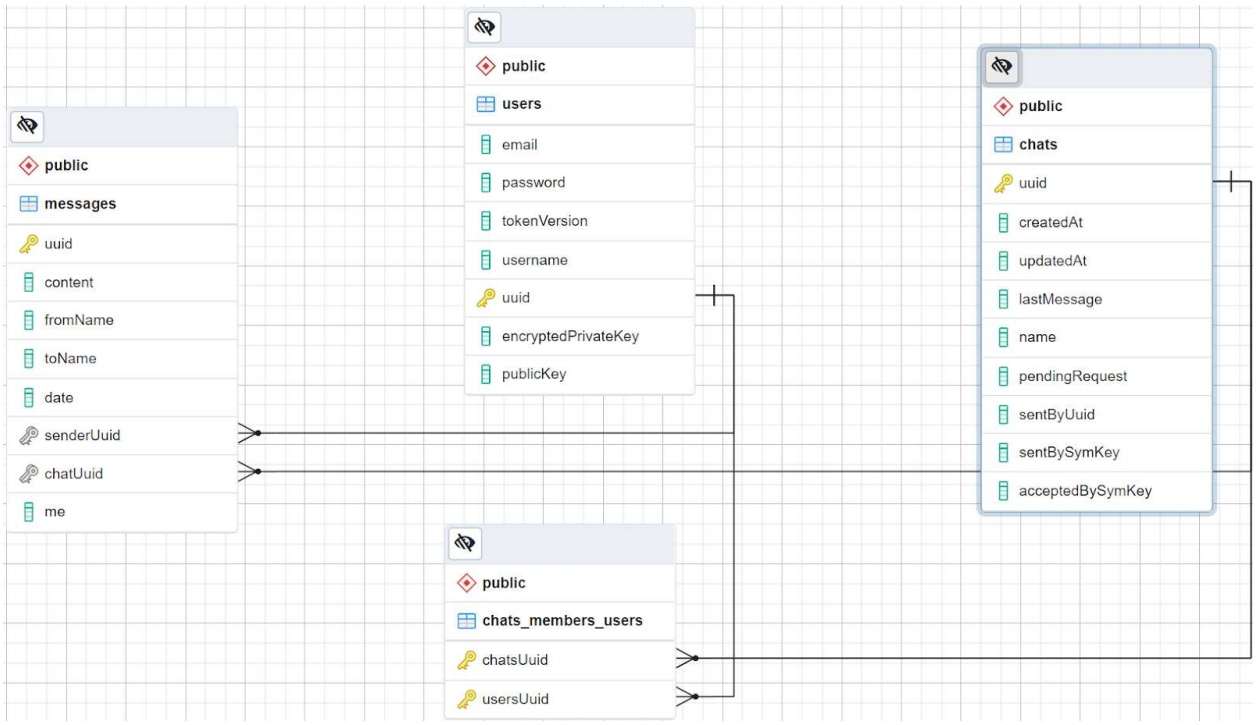

Figure 35. Database Entity Relationship Diagram

User Interface Mockups

It is generally a good idea to plan out a user interface before implementing it in code. The next few images outline how the user interface should roughly look upon completion. The design should be consistent between both desktop and mobile views, thus both a desktop and mobile view is provided for each main screen.

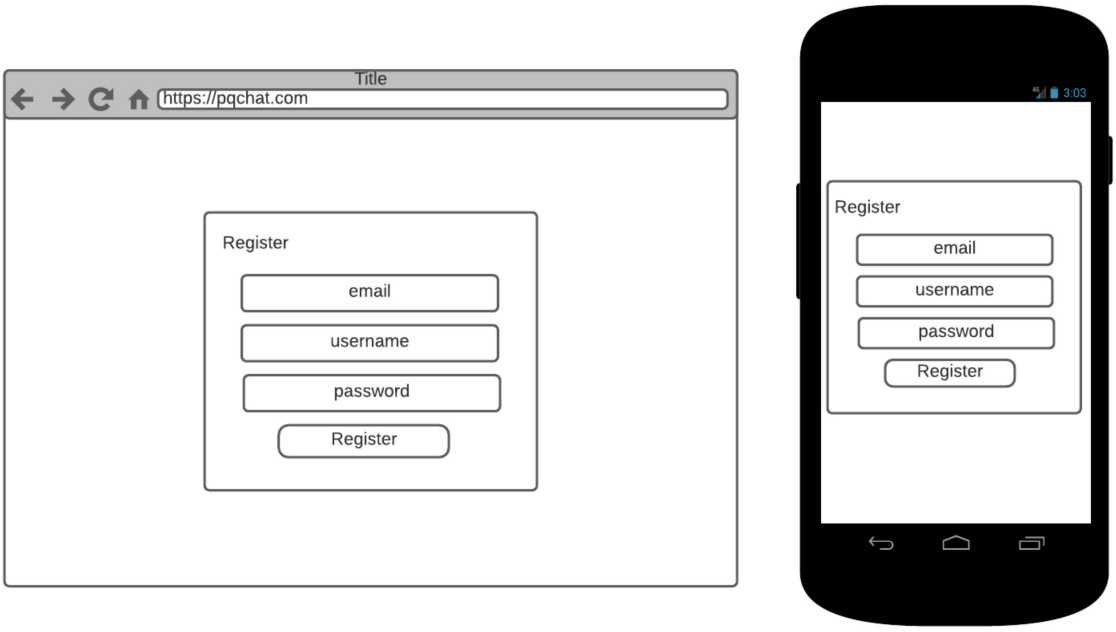

Figure 36. Register UI Mockup

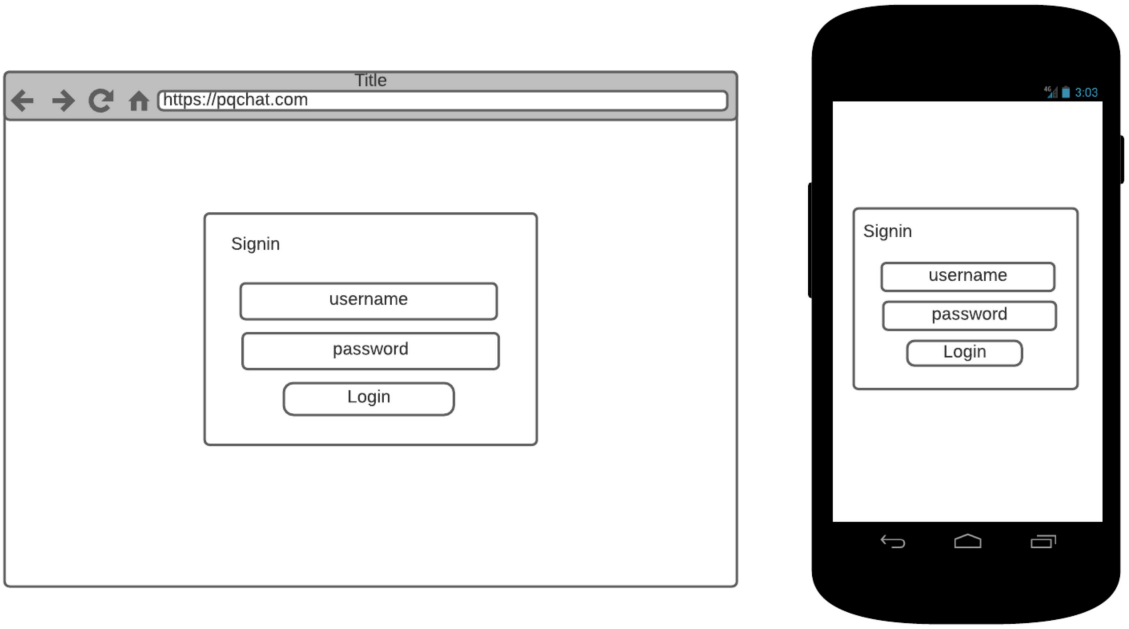

Figure 37. Login UI Mockup

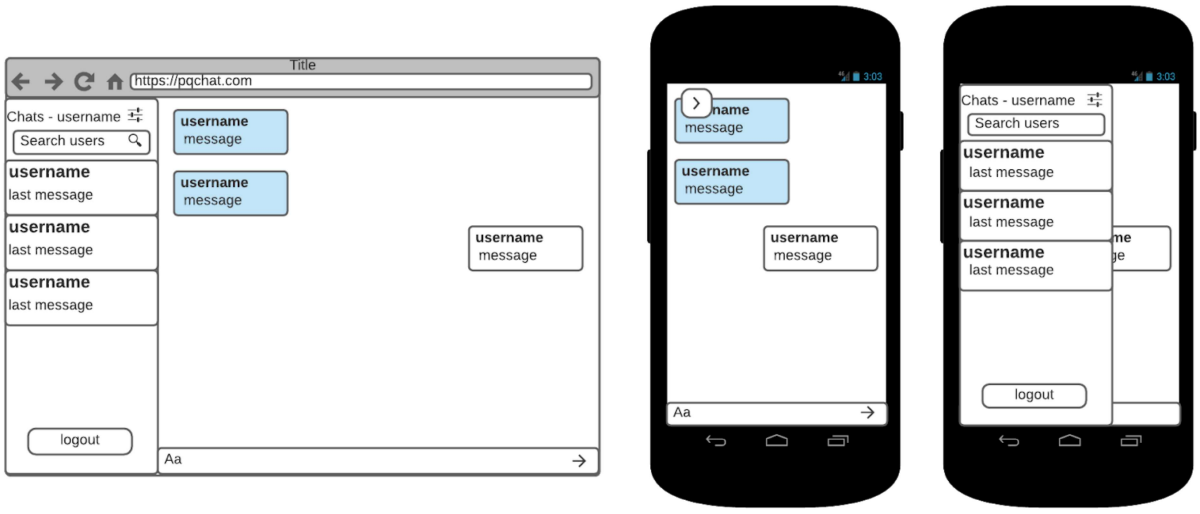

Figure 38. Main Chat Application View

Implementation

With the tech stack, protocol, database schema, and UI mockups in place, the application and all of its encompassing parts can start to take shape. This section will repeatedly reference code available in the project's GitHub repository linked in **Appendix A**, though code from the repository will be presented often where appropriate. Parts of the implementation, such as the server setup process or the styling of custom SVG icons will not be discussed in this section. Please refer to the GitHub repository for a full breakdown of how all parts of the project were implemented.

Server Side

Entities

'Entities' refer to each of the tables discussed in the section **Database Design**. Using TypeORM and TypeGraphQL, we can represent each entity as a JavaScript class.

The User entity, for example, is set up as follows:

```
@ObjectType()
@Entity("users")
export class User extends BaseEntity {
  @Field(() => String)
  @PrimaryGeneratedColumn("uuid")
  uuid: string;

  @Field()
  @Column("text", { nullable: true })
  publicKey: string; // plaintext public key

  @Field()
  @Column("text", { nullable: true })
  encryptedPrivateKey: string; // encrypted private key

  @Field()
  @Column("text", { unique: true })
  username: string;

  /* not used for login process, should be used for password
     recovery in future */
  @Field()
```

```
  @Column("text")
  email: string;

  // no @Field() here as to not expose password
  @Column("text")
  password: string;

  /* used to check if version matches saved version in user
     upon token refresh */
  @Column("int", { default: 0 })
  tokenVersion: number;

  @Field(() => [Message])
  @OneToMany(() => Message, (messages) => messages.fromName, {
    nullable: true,
    onDelete: "CASCADE",
  })
  messages: Message[];

  @Field(() => [Chat])
  @ManyToMany(() => Chat, (chat) => chat.members, {
    nullable: true,
    onDelete: "CASCADE",
  })
  chats: Chat[];
}
```

The Chat and Message entities are set up similarly as well, mapping the fields in **Figure 35** into TypeScript code. Using TypeORM, we can easily define the relationships between tables, such as the many-to-many relationship between users and chats, given that a user may have many chats and a chat may have one or more users. TypeORM takes care of setting up foreign keys between tables and junction tables that tie everything together. The full code for each of the entities is available in **Appendix B.**

Resolvers

Resolvers are a collection of functions that act upon a database schema through a GraphQL layer. Resolver functions can be of the type: query (to query for/retrieve data), mutations (to update, insert, delete data), subscriptions (similar to queries, but useful for real time updates), and more. There are several resolver functions necessary to make this application

function, but we will examine a few of them that may be of particular interest. The full code for each of the resolvers is available in **Appendix C**.

User Resolver:

Two resolver functions from the user resolver of interest are the login and register functions.

```
@Mutation(() => Boolean) // returns boolean, true worked
async register(
  @Arg("email") email: string,
  @Arg("username") username: string,
  @Arg("password") password: string,
  @Arg("publicKey") publicKey: string,
  @Arg("encryptedPrivateKey") encryptedPrivateKey: string
) {
  const hashedPassword = await hash(password, 12);

  const userByName = await User.findOne({ where: { username } });
  const userByEmail = await User.findOne({ where: { email } });

  // check if username/email taken/already exists
  if (userByName || userByEmail) {
    throw new Error("exists");
  }

  try {
    await User.insert({
      username,
      email,
      password: hashedPassword,
      publicKey,
      encryptedPrivateKey,
    });
  } catch (err) {
    console.log(err);
    return false;
  }

  return true;
}
}
```

Figure 39. User Resolver Register Function

The register function receives an email, username, password, public key, and encrypted private key from a user. The password is hashed once more on the server side. If a user by the same username or email already exists, an error is thrown to let the user know they may need to use different credentials. If not, the information is stored into the Users table.

```
// login a user
@Mutation(() => LoginResponse) // returns LoginResponse
async login(
  @Arg("username") username: string,
  @Arg("password") password: string,
  @Ctx() { res }: MyContext
): Promise<LoginResponse> {
  const user = await User.findOne({ where: { username } });

  // login error
  // username incorrect/does not exist
  if (!user) {
    throw new Error("invalid login (user does not exist)");
  }

  const valid = await compare(password, user.password);

  //password invalid
  if (!valid) {
    console.log("Invalid login attempt: ", username, password);
    throw new Error("invalid login (incorrect password)");
  }

  if (valid) {
    console.log("Valid login attempt");
  }

  // login success
  sendRefreshToken(res, createRefreshToken(user));

  return {
    accessToken: createAccessToken(user),
    encryptedPrivateKey: user.encryptedPrivateKey,
    publicKey: user.publicKey,
    user,
  };
}
```

Figure 40. User Resolver Login Function

The login function receives a username and password from the client. If the user exists, it compares the hash of the sent password with the hash stored in the database. If everything checks out, the function creates an access token for the user and returns their encrypted private key, public key, and the user object itself containing information about the user. Unauthorized users cannot access the main chat application without having all of this information sent back from the server, most importantly the access token.

Chat Resolver:

The Chat entity stores information about communication between two users, and as such, it also stores the encrypted symmetric keys of each user. There are two resolver functions here that are worth examining in further detail.

```
@Mutation(() => Boolean)
async createChat(
  @Arg("memberIds", () => [String]) memberIds: string[],
  @Arg("userId") userId: string,
  @Arg("sentBySymKey") sentBySymKey: string,
  @Arg("acceptedBySymKey") acceptedBySymKey: string
) {
  try {
    const user = await User.findOne({ where: { uuid: userId } });

    if (!user) throw new Error("getChats: user not authorized");

    if (memberIds.length == 1) {
      const userRepo = await this.getUserRepo(userId);

      for (let chat of userRepo.chats) {
        if (chat.members.length == 2) {
          const chatExist = chat.members.filter(
            (member) => member.uuid == memberIds[0]
          );
          if (chatExist.length >= 1) return true;
        }
      }
    }

    const members = await this.getUserObject(memberIds);

    return await this.createNewChat(
      [...members, user],
      userId,
      sentBySymKey,
      acceptedBySymKey
    );
  } catch (err) {
    console.log(err);
    return false;
  }
}
```

Figure 41. Chat Resolver Create Chat Function

The createChat function is similar to the register function from the User resolver. Given certain input parameters, it creates a chat as described in **Figures 28 and 29.** However, it also has checks in place to make sure a chat between two people does not already exist.

```
@Mutation(() => Boolean)
async acceptRequest(
  @Arg("chatId") chatId: string,
  @Arg("encryptedSymKey") encryptedSymKey: string
) {
  try {
    const chatRepo = getRepository(Chat);
    const chatToUpdate: Chat | undefined = await chatRepo.findOne({
      where: { uuid: chatId },
    });

    if (!chatToUpdate) throw new Error("Could not find chat to update");

    chatToUpdate.pendingRequest = false;
    chatToUpdate.acceptedBySymKey = encryptedSymKey;

    await chatRepo.save(chatToUpdate);

    return true;
  } catch (err) {
    console.log(err);
    throw new Error("Error accepting request");
  }
}
```

Figure 42. Chat Resolver Accept Request Function

If a user accepts a request, the newly encrypted symmetric key (encrypted with the user's encryption key) replaces the currently encrypted symmetric key in the Chat object/row. As mentioned, the rest of code for all resolvers is mentioned in **Appendix C.**

Message Resolver:

The Message resolver is relatively straightforward and contains functions for creating/sending messages and triggering updates to the server. Please refer to **Appendix C** for a complete look.

This completes a high level look at some of the important logic being handled on the server.

Client Side

The client side code base consists of a plethora of functionality. It has both the logic to encrypt and decrypt information, but also to render the User Interface. This section will briefly outline some of the code integral to providing security. Please refer to the GitHub repository available at **Appendix A** to view how the User Interface and its associated styling was done. Client side GraphQL queries and mutations are available at **Appendix D**.

Client side code is organized as follows:

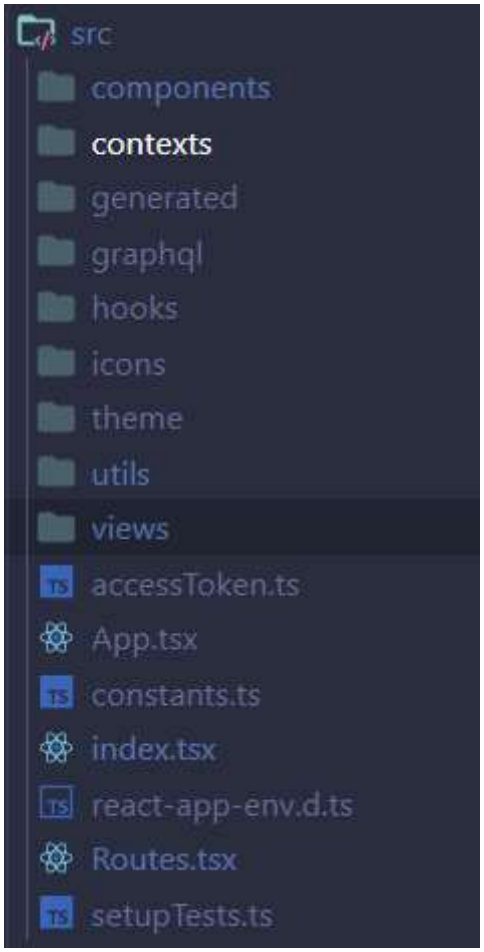

Figure 43. Client Side Code Organization

The folders 'components' and 'views' contain much of the User Interface design and the associated logic (eg. what happens when you click a button). The 'generated' and 'graphql' folders contain the client side GraphQL code for querying and mutating data in the

database. The folders 'icons', 'contexts', and 'theme' contain code pertaining to styling the User Interface. There are two folders for helper functions: 'hooks' and 'utils.' The 'utils' folder contains several helper functions written to ensure security that integrate into the User Interface and its associated logic. These functions are the subject of this section:

```
import { ntru } from "ntru";
import { default as aesjs } from "aes-js";

export const generateKeyPair = async () => {
  const keyPair: any = await ntru.keyPair();

  const privateKey = keyPair.privateKey;
  const publicKey = aesjs.utils.hex.fromBytes(keyPair.publicKey);

  return { privateKey, publicKey };
};
```

Figure 44. generateKeyPair Function

The generateKeyPair() function calls the ntru.keyPair() function to generate the NTRU public and private key pair. Each of these key pairs is represented as an array of uint8 (unsigned integers of 8 bits). The public key, as it is to simply be stored as plaintext, is converted to a hexadecimal string representation of the array to be sent to the server. It returns the private key in its uint8 array form, and the publicKey in its hexadecimal string representation form.

```
import { scrypt } from "scrypt-js";

export const scryptPassword = async (password: string): Promise<any> => {
  const passwordArray: Uint8Array = new TextEncoder().encode(password);

  const salt: Uint8Array = new TextEncoder().encode("salt"); // randomly gene

  const hashedPassword: Uint8Array = await scrypt(
    passwordArray,
    salt,
    1024,
    8,
    1,
    32
  );

  const firstHalf = hashedPassword.slice(0, 16);
  const lastHalf = hashedPassword.slice(16, 32);

  const scryptedPassword: string = new TextDecoder().decode(lastHalf);
  return [scryptedPassword, firstHalf];
};
```

Figure 45. scryptPassword Function

The scryptPassword() function breaks a plaintext password into an encryption key and an authorization key as discussed in the protocol section. The plaintext password is encoded into a uint8 array which is then hashed using the 'scrypt' function from the scrypt-js library. The parameters for the scrypt function are outlined in the variable declaration for hashedPassword.

```
import { default as aesjs } from "aes-js";

export const encryptPrivateKey = (
  privateKey: Uint8Array,
  encrArray: Uint8Array
) => {
  const aesCtr = new aesjs.ModeOfOperation.ctr(encrArray);
  const encryptedBytes = aesCtr.encrypt(privateKey);
  const encryptedPrivateKey = aesjs.utils.hex.fromBytes(encryptedBytes);

  return encryptedPrivateKey;
};
```

Figure 46. encryptPrivateKey Function

The encryptPrivateKey() function takes in the privateKey (a uint8 array) and the encryption key (also a uint8 array) and uses the AES CTR (counter) module to encrypt the private key. The encrypted private key is then converted into a hexadecimal string representation and returned so that it can be sent back to the server.

```
import { default as aesjs } from "aes-js";

export const decryptPrivateKey = (
  encryptedPrivateKey: string,
  encrArray: Uint8Array
) => {
  const aesCtr = new aesjs.ModeOfOperation.ctr(encrArray);
  const encryptedBytes = aesjs.utils.hex.toBytes(encryptedPrivateKey);
  const decryptedBytes = aesCtr.decrypt(encryptedBytes);

  return decryptedBytes;
};
```

Figure 47. decryptPrivateKey Function

The decryptPrivateKey() function essentially performs the opposite of the encryptPrivateKey() function. Because the encryption key is essentially a symmetric key, the AES CTR module is used to decrypt the encrypted private key with the encryption key. The original uint8 array form of the private key is returned for local storage.

```
import { randomBytes } from "crypto";
import { default as aesjs } from "aes-js";

export const generateEncryptedSymKeys = (
  userEncryptionArr: Uint8Array,
  friendPublicKey: Uint8Array
) => {
  const symKey = randomBytes(32);

  console.log("Sym key", symKey);

  const friendPublicKeyBytes = friendPublicKey.slice(300, 332); // choosing

  const aesCtr1 = new aesjs.ModeOfOperation.ctr(userEncryptionArr);
  const encryptedSentBySymKeyBytes = aesCtr1.encrypt(symKey);

  const aesCtr2 = new aesjs.ModeOfOperation.ctr(friendPublicKeyBytes);
  const encryptedAcceptedBySymKeyBytes = aesCtr2.encrypt(symKey);

  const encryptedSentBySymKey = aesjs.utils.hex.fromBytes(
    encryptedSentBySymKeyBytes
  );
  const encryptedAcceptedBySymKey = aesjs.utils.hex.fromBytes(
    encryptedAcceptedBySymKeyBytes
  );

  return {
    encryptedSentBySymKey,
    encryptedAcceptedBySymKey,
  };
};
```

Figure 48. generateEncryptedSymKeys Function

The generateEncryptedSymKeys() function generates a shared symmetric key and encrypts it for the user sending a friend request and for the friend receiving the request. The shared symmetric key is created using the 'randomBytes' function from the native library 'crypto.' The randomBytes are of length 32 (or 256 bits) to ensure 128 bits of post quantum security with AES-256. The shared symmetric key is encrypted using the friend's public key and returned as a hexadecimal string representation. The shared symmetric key is also encrypted using the user's encryption key and returned as a hexadecimal string

representation. This key (encrypted for the user and friend) is then sent to the server for storage and later retrieval.

```
import { default as aesjs } from "aes-js";

export const decryptEncryptSymKey = (symKey: string) => {
  const symKeyBytes = aesjs.utils.hex.toBytes(symKey);
  const privateKeyBytes: any = Object.values(
    JSON.parse(sessionStorage.getItem("privateKey")!)
  ).slice(300, 332);
  const encrArray: any = Object.values(
    JSON.parse(sessionStorage.getItem("encrArray")!)
  );

  const aesCtr = new aesjs.ModeOfOperation.ctr(privateKeyBytes);
  const decryptedSymKeyBytes = aesCtr.decrypt(symKeyBytes);

  console.log("Decrypted symkey bytes", decryptedSymKeyBytes);

  const aesCtr2 = new aesjs.ModeOfOperation.ctr(encrArray);
  const encryptedSymKeyBytes = aesCtr2.encrypt(decryptedSymKeyBytes);

  const encryptedSymKey = aesjs.utils.hex.fromBytes(encryptedSymKeyBytes);

  return encryptedSymKey;
};
```

Figure 49. decryptEncryptSymKey Function

The decryptEncryptSymKey() function is called when a user accepts an incoming friend request. Because the symmetric key was previously encrypted using the user's public key, the user must first decrypt it with their private key (which is grabbed from session storage). The decrypted private key is then encrypted again using the user's encryption key and returned so that it may be stored server side.

```
import { default as aesjs } from "aes-js";

export const encryptMessage = (symKey: Uint8Array, message: string) => {
  const messageBytes = aesjs.utils.utf8.toBytes(message);

  const aesCtr = new aesjs.ModeOfOperation.ctr(symKey, new aesjs.Counter(5));
  const encryptedMessageBytes = aesCtr.encrypt(messageBytes);

  const encryptedMessage = aesjs.utils.hex.fromBytes(encryptedMessageBytes);

  return encryptedMessage;
};
```

Figure 50. encryptMessage Function

The encryptMessage() function is used to encrypt a message with a symmetric key. The function takes in the symmetric key and the plain text message. The message is then converted into a uint8 array, after which it is encrypted using the AES CTR module and returned as a hexadecimal string representation.

```
import { default as aesjs } from "aes-js";

export const decryptMessage = (symKey: Uint8Array, message: string) => {
  const messageBytes = aesjs.utils.hex.toBytes(message);

  const aesCtr = new aesjs.ModeOfOperation.ctr(symKey, new aesjs.Counter(5));
  const decryptedMessageBytes = aesCtr.decrypt(messageBytes);

  const decryptedMessage = aesjs.utils.utf8.fromBytes(decryptedMessageBytes);

  return decryptedMessage;
};
```

Figure 51. decryptMessage Function

The decryptMessage() function converts ciphertext back into plaintext. It does this by taking in a symmetric key and the ciphertext, feeding the symmetric key and ciphertext into the AES CTR module and returning the decrypted plaintext as a string.

All of the aforementioned functions ensure E2E encryption is guaranteed and that symmetric keys are at no point susceptible to attacks or leaks due to the use of NTRU (barring a user leaking their own password due to negligence).

The following function is used during the logout process:

```
import { setAccessToken } from "../accessToken";

export const deleteStore = () => {
  setAccessToken("");
  sessionStorage.removeItem("accessToken");
  sessionStorage.removeItem("userUuid");
  sessionStorage.removeItem("userUsername");
  sessionStorage.removeItem("ntruPublicKey");
  sessionStorage.removeItem("ntruPrivateKey");
  sessionStorage.removeItem("encrArray");
};
```

Figure 52. deleteStore Function

This function is called to remove all stored items from session storage upon a user logging out. The log out process also informs the server that the user is logging out and invalidates their access token.

To protect routes (eg. preventing access to the the chat section without being logged in), the routes are guarded using a simple AuthGuard component:

```
const GuardedRoute: React.FC<GuardedRouteProps> = (props) => {
  const { component: Component, exact, path } = props;
  const auth = sessionStorage.getItem("accessToken");

  if (auth && (path === "/login" || path === "register"))
    return <Redirect to="/chat" />;

  return (
    <Route
      path={path}
      exact={exact}
      render={() => (auth ? <Component /> : <Redirect to="/login" />)}
    />
  );
};
```

Figure 53. AuthGuard Component

This component simply blocks a user from accessing certain paths/areas in the application and forces them to login if they do not have a valid accessToken.

Results

With the methodology and design implemented into code, we can examine how the application and its associates systems have taken shape.

NTRU Encryption Demo

Part of the implementation process included creating a demonstration of the NTRU Encryption and Decryption protocol running on a browser to test and showcase the NTRU JavaScript and WebAssembly library. This demo is available at the route '/ntru' if you run the application on a local machine.

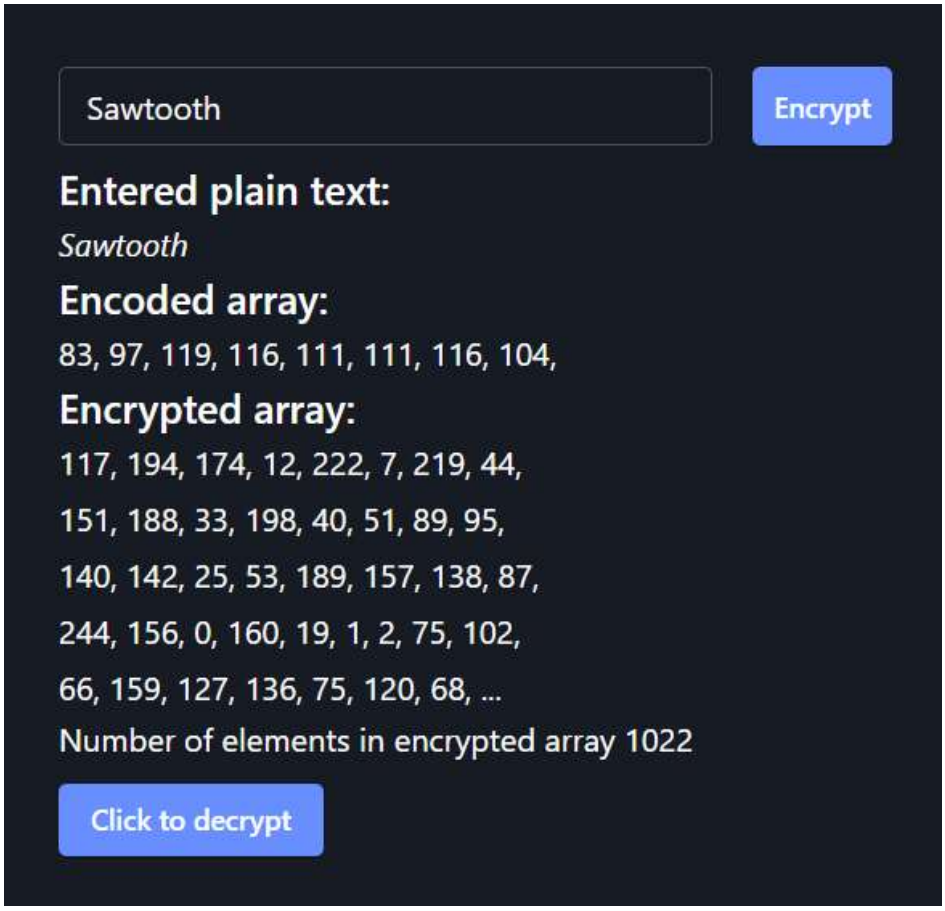

Figure 54. NTRU Encryption Demo

Any plaintext can be entered in the textbox; in the above example the plaintext message "Sawtooth." Was entered. The plaintext message is encoded into a uint8 array. Then, an NTRU public and private key are generated, and the encoded message is encrypted using NTRU, to produce the output under 'Encrypted array'.

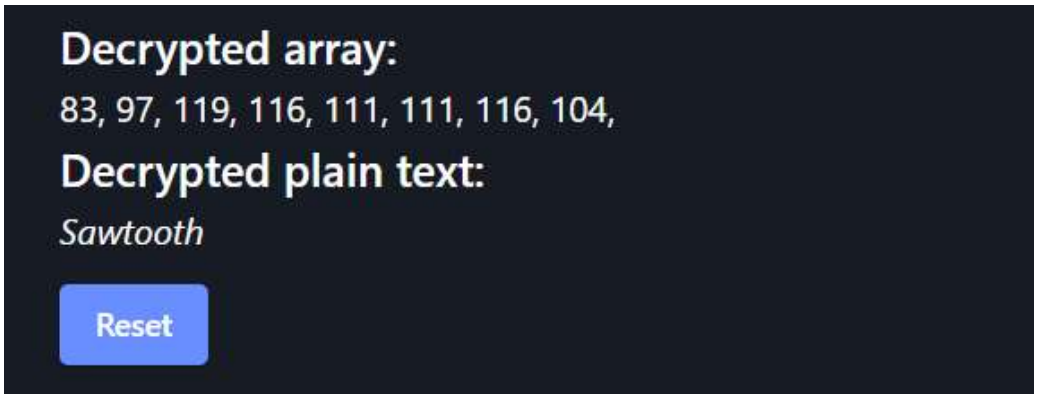

Figure 55. NTRU Encryption Demo

The private key can then be used to decrypt the message and retrieve the encoded plaintext. The encoded plaintext uint8 array can then be decoded to produce the original plaintext message "Sawtooth."

Database

Using the interactive terminal 'psql' for PostgreSQL, we can examine whether the code for our entities described in the 'Implementation' section were successfully created. Using the '\d' command, we can view the list of tables/entities and relations in the database.

```
                List of relations
Schema |         Name          | Type  |
-------+-----------------------+-------+
public | chats                 | table |
public | chats_members_users   | table |
public | messages              | table |
public | users                 | table |
```

Figure 56. List of Relations In Database

Using the '\d' command followed by the name of the table will list all the properties and associated relations (foreign keys, restraints, etc.) of that table.

```
                        Table "public.users"
       Column        |  Type   | Collation | Nullable |      Default
---------------------+---------+-----------+----------+--------------------

 email               | text    |           | not null |
 password            | text    |           | not null |
 tokenVersion        | integer |           | not null | 0
 username            | text    |           | not null |
 uuid                | uuid    |           | not null | uuid_generate_v4()
 encryptedPrivateKey | text    |           |          |
 publicKey           | text    |           |          |
Indexes:
    "PK_951b8f1dfc94ac1d0301a14b7e1" PRIMARY KEY, btree (uuid)
    "UQ_fe0bb3f6520ee0469504521e710" UNIQUE CONSTRAINT, btree (username)
Referenced by:
    TABLE "messages" CONSTRAINT "FK_39bdf5ecd7fc006102715b33e9f" FOREIGN KE
Y ("senderUuid") REFERENCES users(uuid)
    TABLE "chats_members_users" CONSTRAINT "FK_571e1d59a81e1767944f72430ee"
 FOREIGN KEY ("usersUuid") REFERENCES users(uuid) ON DELETE CASCADE
```

Figure 57. Users Table In Database

```
                                Table "public.chats"
     Column      |            Type             | Collation | Nullable |        Default
-----------------+-----------------------------+-----------+----------+-----------------------
 uuid            | uuid                        |           | not null | uuid_generate_v4()
 createdAt       | timestamp without time zone |           | not null | now()
 updatedAt       | timestamp without time zone |           | not null | now()
 lastMessage     | character varying           |           |          |
 name            | character varying           |           |          |
 pendingRequest  | boolean                     |           | not null | true
 sentByUuid      | character varying           |           | not null |
 sentBySymKey    | character varying           |           | not null | ''::character varying
 acceptedBySymKey | character varying          |           | not null | ''::character varying
Indexes:
    "PK_4741e8cb46af785df554407dbcb" PRIMARY KEY, btree (uuid)
Referenced by:
    TABLE "messages" CONSTRAINT "FK_7c2c1970d7a191422f4bcdc3306" FOREIGN KEY ("chatUuid") REFERE
NCES chats(uuid)
    TABLE "chats_members_users" CONSTRAINT "FK_996d0c05aa3ba744a6aeea82952" FOREIGN KEY ("chatsU
uid") REFERENCES chats(uuid) ON DELETE CASCADE
```

Figure 58. Chats Table In Database

```
                                Table "public.messages"
   Column   |            Type             | Collation | Nullable |      Default
------------+-----------------------------+-----------+----------+--------------------
 uuid       | uuid                        |           | not null | uuid_generate_v4()
 content    | text                        |           |          |
 fromName   | text                        |           |          |
 toName     | text                        |           |          |
 date       | timestamp without time zone |           | not null | now()
 senderUuid | uuid                        |           |          |
 chatUuid   | uuid                        |           |          |
 me         | boolean                     |           | not null | false
Indexes:
    "PK_6aaade3b9853de4a3f61ca174c5" PRIMARY KEY, btree (uuid)
Foreign-key constraints:
    "FK_39bdf5ecd7fc006102715b33e9f" FOREIGN KEY ("senderUuid") REFERENCES users(uuid)
    "FK_7c2c1970d7a191422f4bcdc3306" FOREIGN KEY ("chatUuid") REFERENCES chats(uuid)
```

Figure 59. Messages Table In Database

```
       Table "public.chats_members_users"
  Column   | Type | Collation | Nullable | Default
-----------+------+-----------+----------+---------
 chatsUuid | uuid |           | not null |
 usersUuid | uuid |           | not null |
Indexes:
    "PK_9017c2b62b115460cfe127e0090" PRIMARY KEY, btree ("chatsUuid", "usersUuid")
    "IDX_571e1d59a81e1767944f72430e" btree ("usersUuid")
    "IDX_996d0c05aa3ba744a6aeea8295" btree ("chatsUuid")
Foreign-key constraints:
    "FK_571e1d59a81e1767944f72430ee" FOREIGN KEY ("usersUuid") REFERENCES users(uuid) ON DELETE CASCADE
    "FK_996d0c05aa3ba744a6aeea82952" FOREIGN KEY ("chatsUuid") REFERENCES chats(uuid) ON DELETE CASCADE
```

Figure 60. Chat Table In Database

As seen in the above figures, the database was successfully set up with all the tables and relations described in **Figure 35.** The 'messages' table will be referred to to double check all messages being received and stored are encrypted.

Register

The register page was successfully created with the planned User Interface between both desktop and mobile views. Moreover, all of the associated logic such as storing hashed passwords and the key pair (with the private key being encrypted) was implemented.

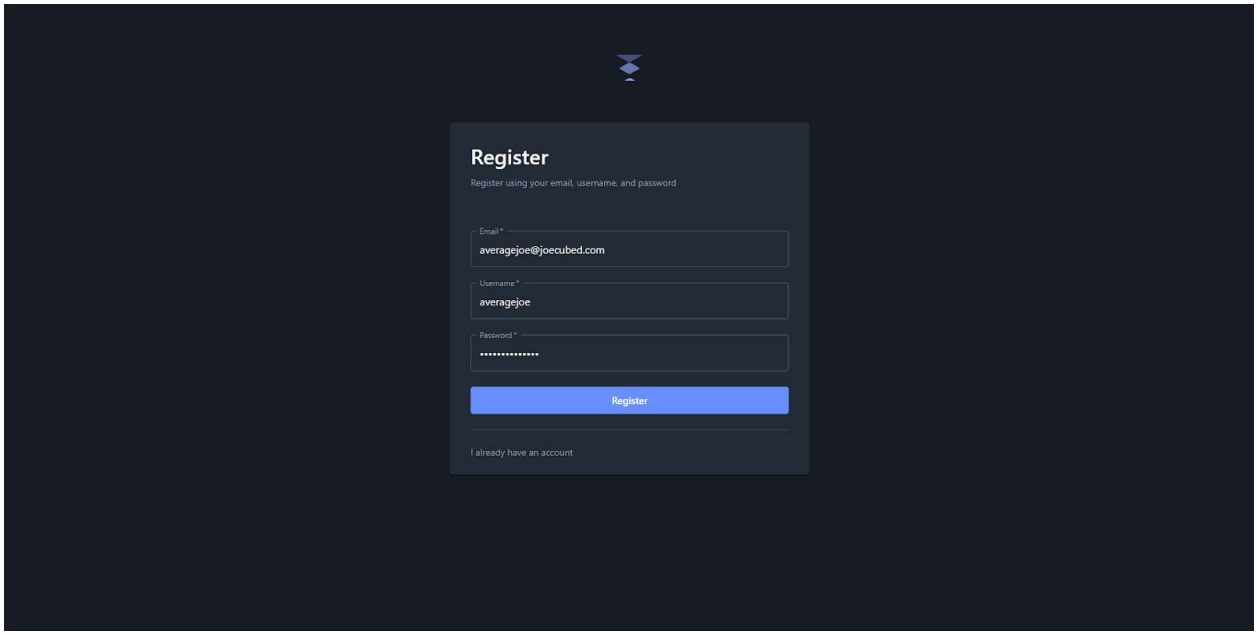

Figure 61. Register Desktop View

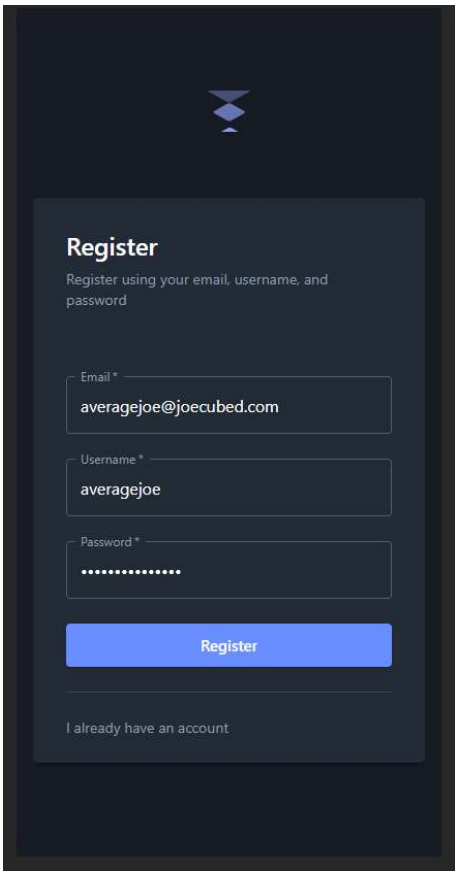

Figure 62. Register Mobile View

If a user attempts to create an account with an insecure password, they receive the following message:

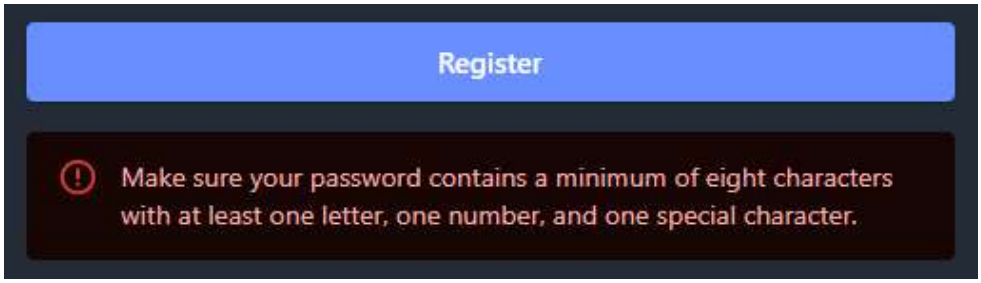

Figure 63. Register Insecure Password

If a user chooses an email or username that is already taken, they receive the following message:

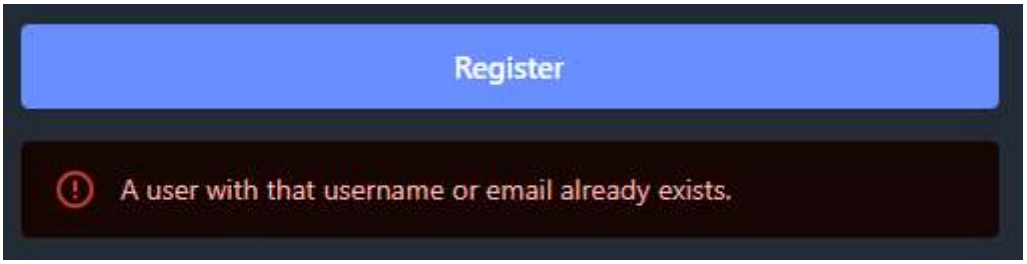

Figure 64. Register Username/Email Taken

If all credentials are chosen properly, the information is sent back to the server for storage.

Login

The login page is very similar to the register page in its layout. For demonstration purposes, a default user with the username ‘demo’ and password ‘Password!123’ is provided.

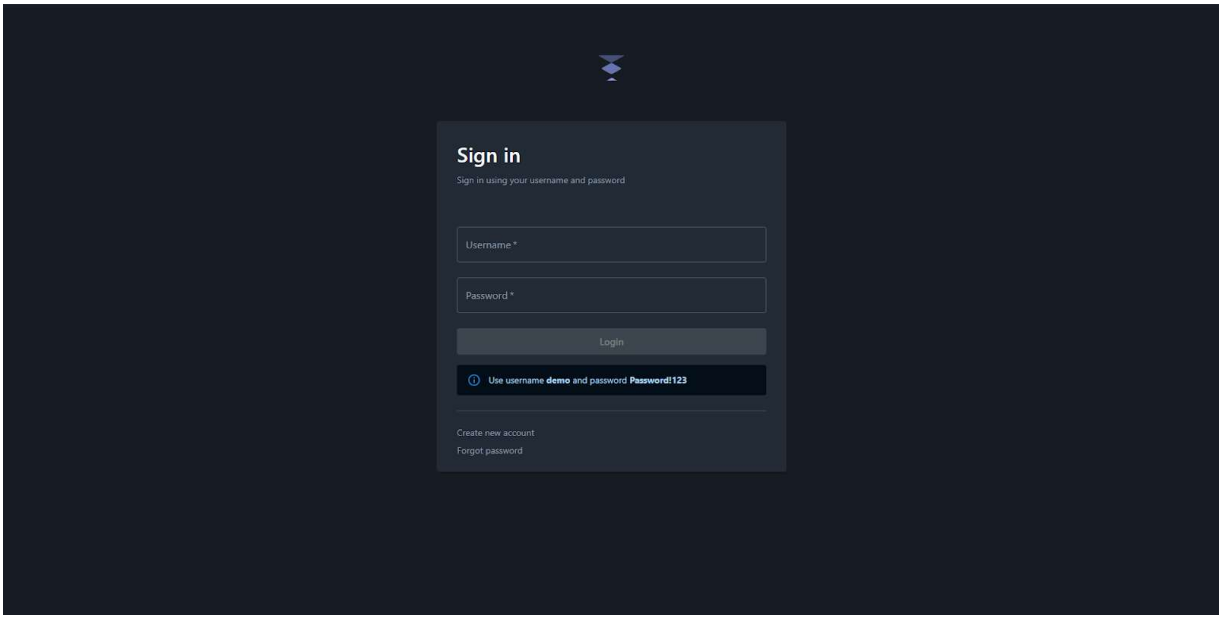

Figure 65. Login Desktop View

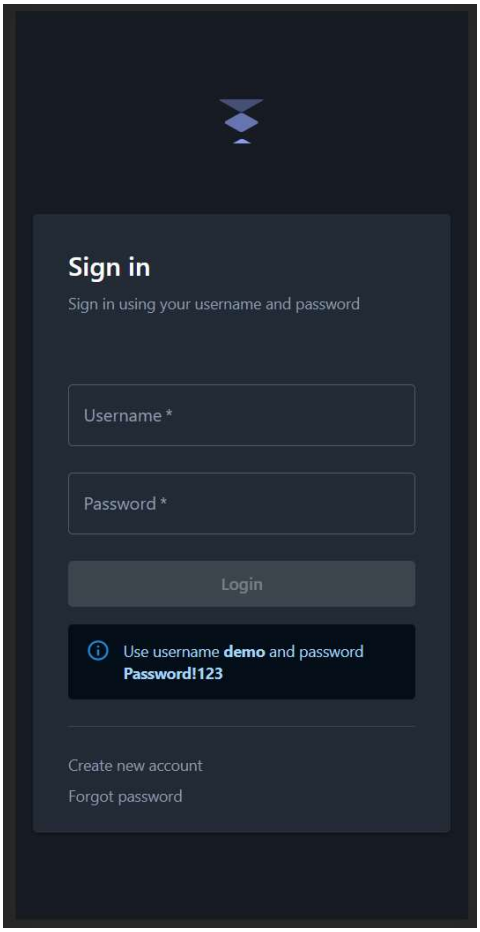

Figure 66. Login Mobile View

If a user logs in with incorrect credentials, either due to an invalid username or password, they are presented with the following message:

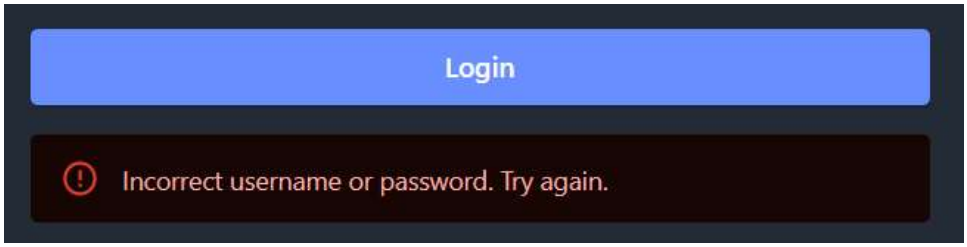

Figure 67. Login Incorrect Credentials

However, if the user does use the correct credentials, the server sends the client the user's access token, public key and encrypted private key. The client uses the encryption key (extrapolated from the user's password) to decrypt the private key. This and other relevant information is stored in session storage and the access token is stored in local storage:

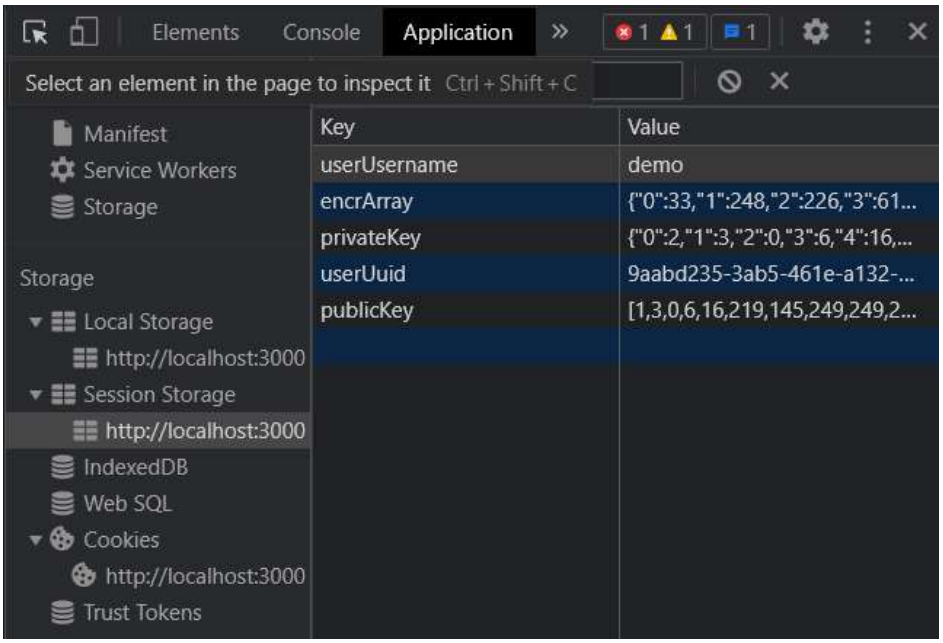

Figure 68. Session Storage With User Info

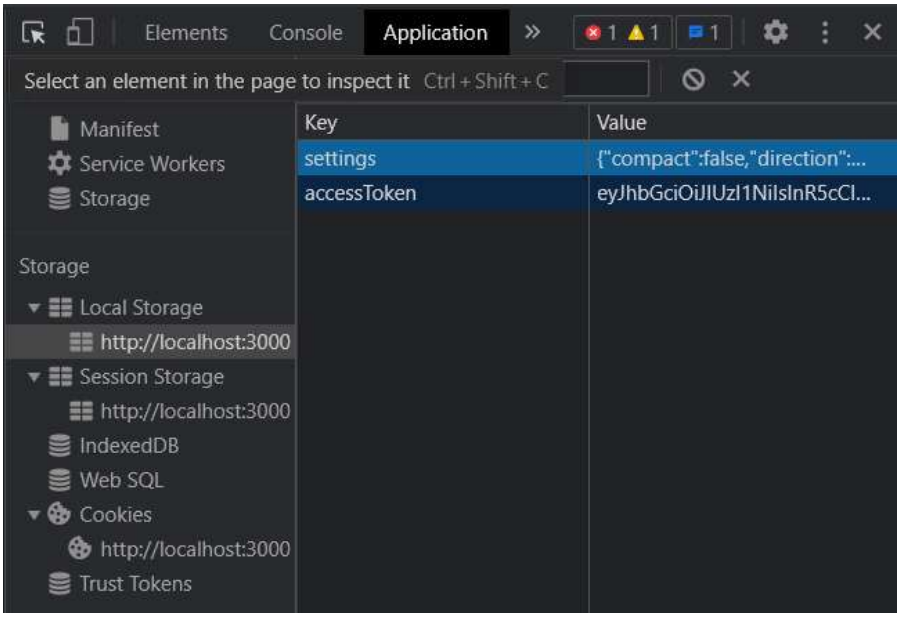

Figure 69. Local Storage With Access Token

The client is then redirected to the main chat view.

Chat

As mentioned, upon successful login, users are brought to the main chat view.

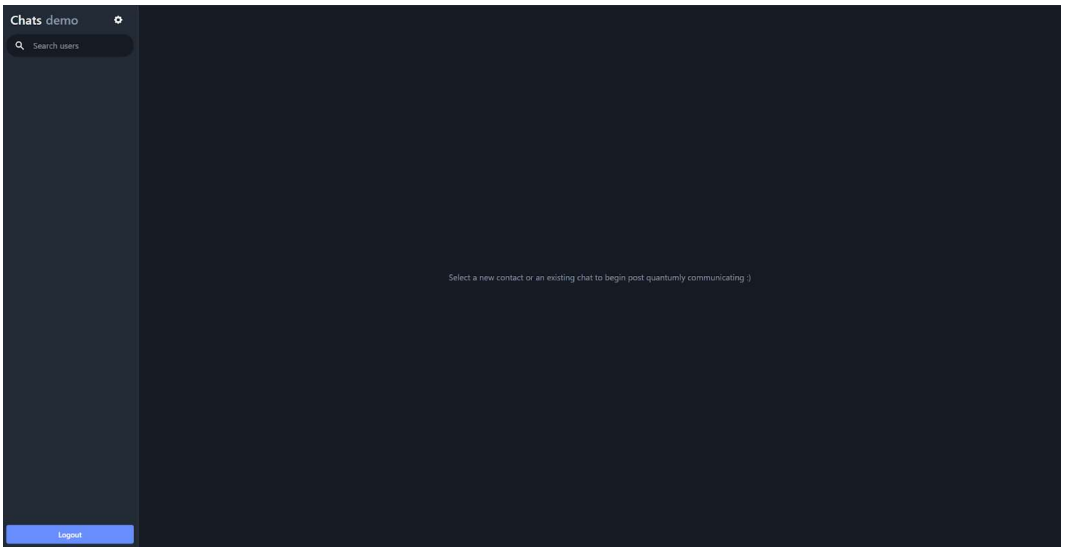

Figure 70. Desktop Chat View

On mobile phones, the left side panel is hidden with the option to show the panel by pressing the menu button in the top left corner of the screen.

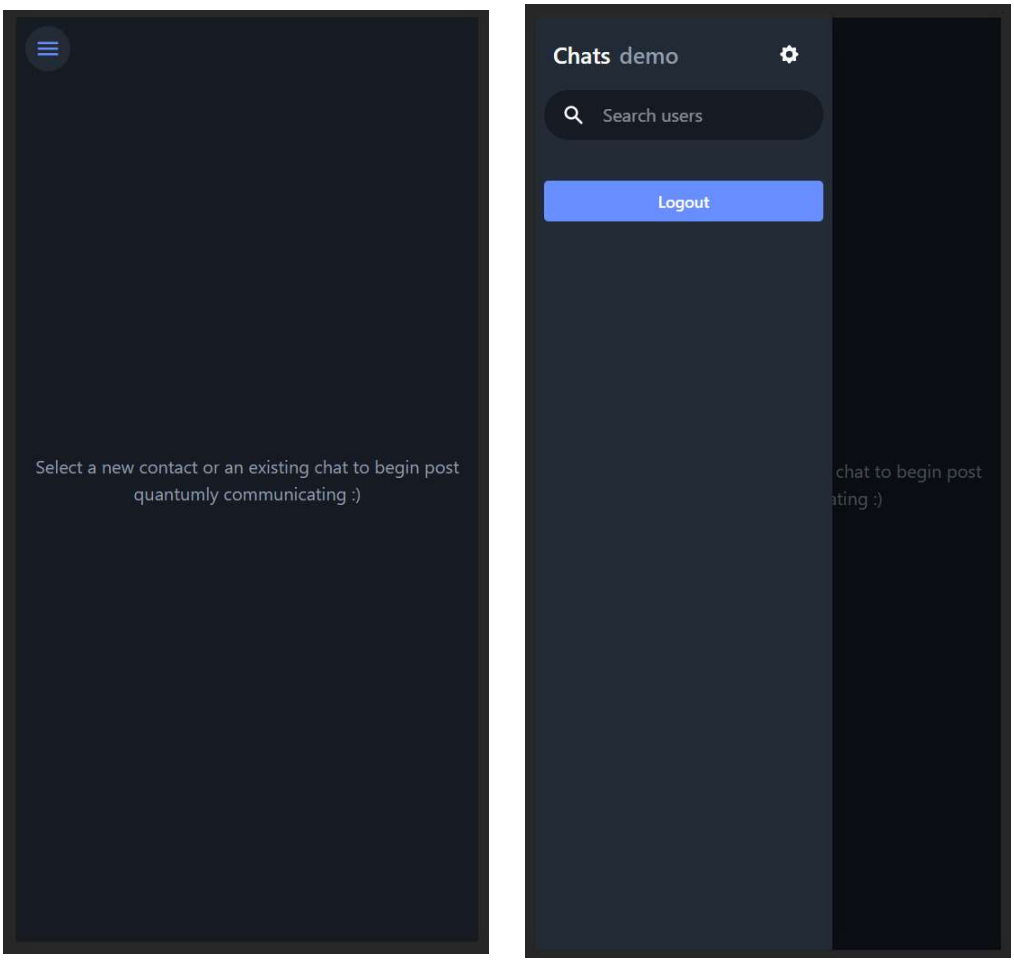

Figure 71. Mobile Chat View

To search for a user to add as a contact/friend, a user must use the search bar with the search icon. A user may enter an empty space for all users registered on the platform, or search for a specific user. To add someone as a friend, they must then click on the user in the list of users. The friend request will then show as 'pending.'

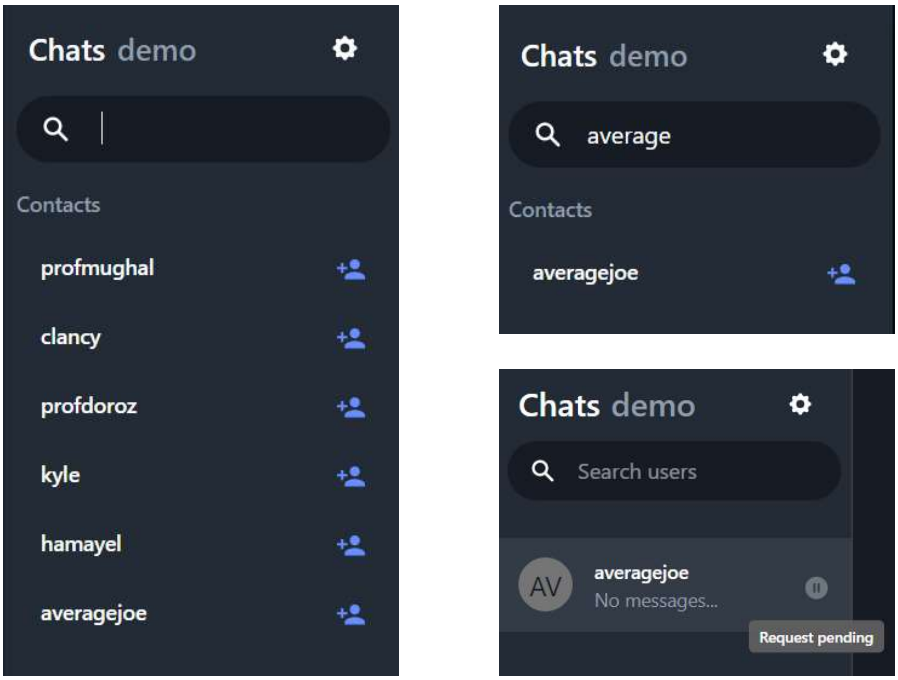

Figure 72. User Search & Friend Request View

This will create a row in the Chats table with pendingRequest set to to true and the encrypted symmetric key.

pendingRequest boolean	sentByUuid character varying	sentBySymKey character varying	acceptedBySymKey character varying
true	9aabd235-3ab5-461e-a...	4cb43a897fcd2c01a96...	42e5db4fb04381cb91...

Figure 73. Friend Request - Chat Created

The user receiving the request can accept or deny the friend request. If the friend request is denied, the row in the Chats table is deleted. If it is accepted, pendingRequest is set to false and the 'acceptedBySymKey' is replaced with the newly created symKey. The chat between the two users is then successfully established and they may message each other.

pendingRequest boolean	sentByUuid character varying	sentBySymKey character varying	acceptedBySymKey character varying
false	9aabd235-3ab5-461e-a...	4cb43a897fcd2c01a96...	a1d4d5558b8a3c7a12...

Figure 74. Friend Request - Chat Accepted

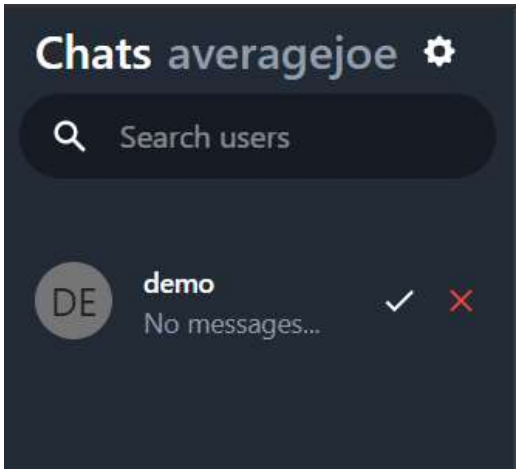

Figure 75. Friend Request Accept/Deny View

Assuming the friend request is accepted, the users will be presented with the following screens (note the user on the right is on a phone and using a 'Light' theme with rounded edges turned on - to change themes and settings a user needs to click the settings icon).

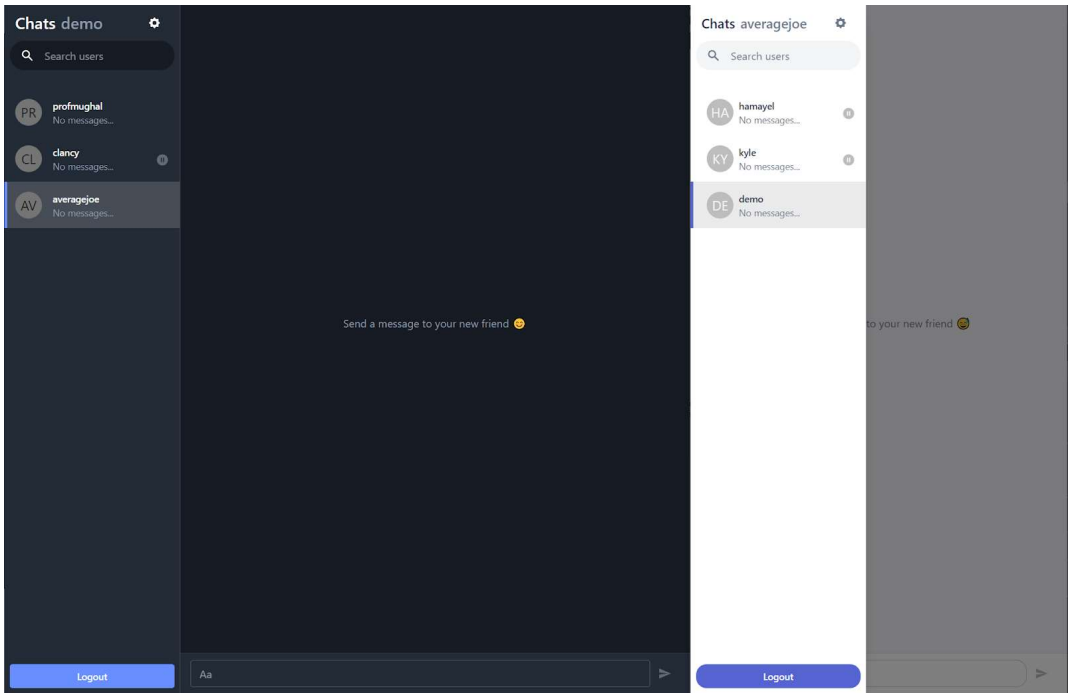

Figure 76. Request Accepted View

The users may now send each other E2E encrypted messages that will be kept secret even with the proliferation of quantum computers.

To send messages to each other, the users must simply type their messages into the message box at the bottom and hit enter or the send icon. Messages sent by the user are shown on the right in the theme's primary color and messages sent from the friend are shown on the left in the theme's secondary color.

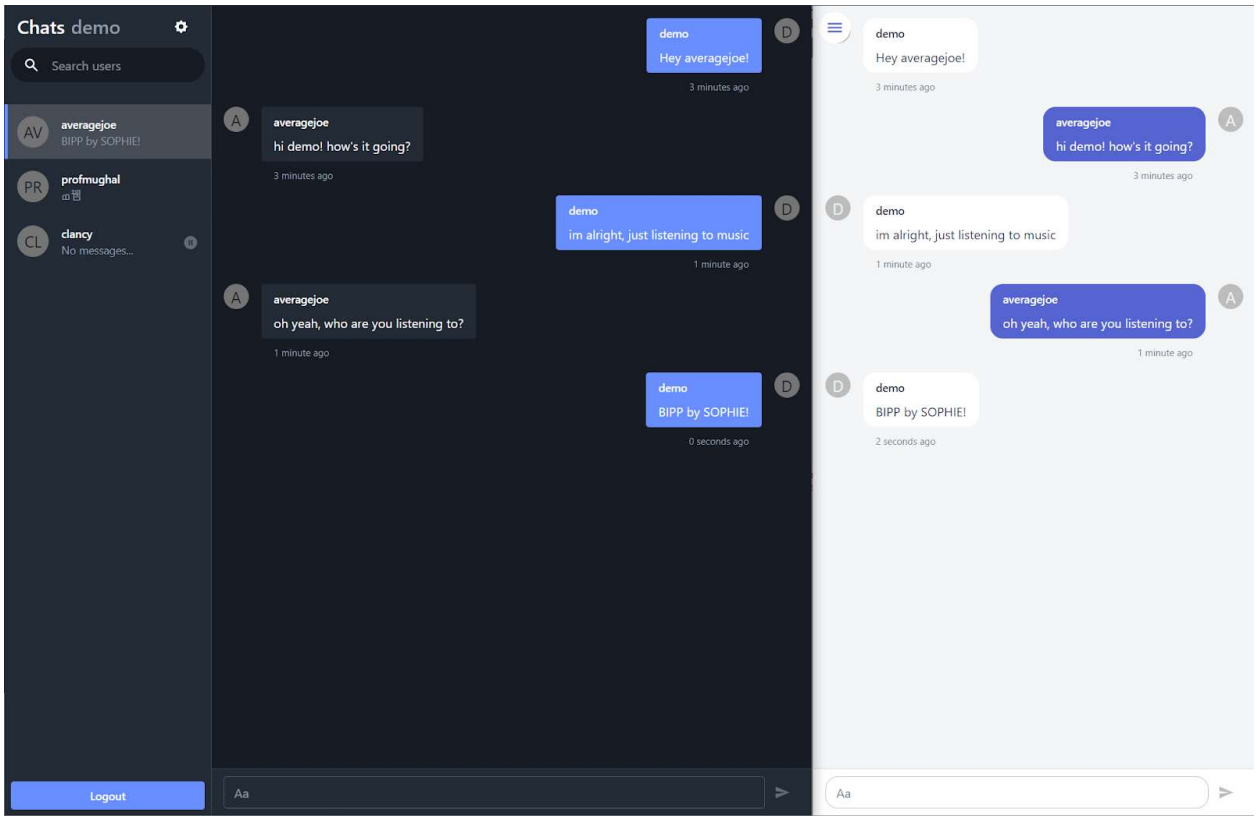

Figure 77. Chat Between Two Users

To double check that the messages are indeed E2E encrypted, we need to make sure the server does not receive and store the plaintext version of the messages at any point. We can simply query the messages table and check the data in the 'content' column.

	uuid [PK] uuid	content text	fromName text	toName text	date timestamp without time zone	senderUuid uuid	chatUuid uuid
1	99d94521-f4...	46c0356d	demo	[null]	2021-05-09 20:43:06.636946	9aabd235-3ab5-...	de949ef8-d3d3-...
2	1c56eeb1-57...	d4ad14d77c365a6a3e880efefc8f8d	demo	[null]	2021-05-09 20:44:54.385898	9aabd235-3ab5-...	3eeb168f-53eb-...
3	a05ac9ff-d42...	f4a14d93782d50397f8704e3b4998c632742988212e1ee62	averagejoe	[null]	2021-05-09 20:45:02.228027	d6fc8c03-41a5-...	3eeb168f-53eb-...
4	858c4a53-d3...	f5a54d967132567f379b47b4f99fdf7e730e969e0feae7342a2c46decda0c970a3bb04	demo	[null]	2021-05-09 20:46:59.946508	9aabd235-3ab5-...	3eeb168f-53eb-...
5	1d973375-bf...	f3a04d8e782157347f9803fbb38bde6f731b90985be3e02e302e08c3cce78471bfed	averagejoe	[null]	2021-05-09 20:47:07.736843	d6fc8c03-41a5-...	3eeb168f-53eb-...
6	04659e81-64...	de813da73d2246380ca03bdcdaaf8d	demo	[null]	2021-05-09 20:47:45.074704	9aabd235-3ab5-...	3eeb168f-53eb-...

Figure 78. Messages Table With Encrypted Messages

As seen in the content field, all of the messages are hexadecimal string representations of the encrypted message. At no point does the server receive the plaintext message. If any hackers or adversaries were to gain access to the database, conversations between users would be fully protected. Most importantly, quantum computers utilizing tools like the Shor's algorithm would not be able to break the encryption protocol, at least in any reasonable amount of time, thus achieving the objective of this project.

Conclusion and Future Work

This project set out with the goal of implementing a web accessible version of the NTRU cryptosystem and utilizing it in a PWA chat application as a proof of concept. Doing this required careful study of existing cryptographic systems and a deep understanding of how they functioned and could be utilized in a modern tech stack. The project began with a thorough planning of all the systems (database schema, user interface design, etc.) that would need to be built to make this application come to fruition. Then, these designs were implemented in code and tested in an iterative process, figuring out what worked and what could be improved. All of this resulted in one of the first implementations of post quantum encryption using NTRU in an E2E encrypted PWA chat application. The project did come with challenges, a few of which were related to learning some of the technologies chosen for the project, such as GraphQL and TypeScript. There are also many opportunities for future MQPs to build upon the project in its current state. Some of these include:

- Implement end to end testing with frameworks like Jest.
- Implement 'forgot password' functionality, thus utilizing the email field.
- Implement new symmetric key creation after a certain number of messages.
- Implement initialization vectors to provide further obfuscation in cipher text.
- Implement two factor authentication to protect data if passwords are compromised.
- Move to WebSockets for updates instead of frequent polling. This would drastically decrease the resource use both on the server and client.
- Scale the application up and move systems off from a local machine to the cloud using AWS, Azure, or another cloud platform.

Works Cited

[1] Ronald L. Rivest, (1990). "Cryptography". In J. Van Leeuwen (ed.). Handbook of Theoretical Computer Science

[2] Crane, Casey. “Symmetric Encryption Algorithms: Live Long & Encrypt.” Hashed Out by The SSL Store™, 14 Jan. 2021, www.thesslstore.com/blog/symmetric-encryption-algorithms/.

[3] Dawn M. Turner, Peter Smirnoff. Symmetric key Encryption - why, where, and how it’s used in banking. CRYPTOMAThIC, 2019 https://www.cryptomathic.com/news-events/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking

[4] Khalid, Ayesha, et al. Domain Specific High-Level Synthesis for Cryptographic Workloads. Springer Nature, 2019.

[5] Nachef, Valerie, et al. Feistel Ciphers Security Proofs and Cryptanalysis. Springer International Publishing, 2017.

[6] Bhargavan, Karthikeyan and Leurant, Gaetan, Sweet32: Birthday attacks on 64-bit block ciphers in TLS and OpenVPN, https://sweet32.info/

[7] Barker, Elaine (January 2016). "NIST Special Publication 800-57: Recommendation for Key Management Part 1: General" (PDF) (4 ed.). NIST.

[8] Bonnetain, X., Naya-Plasencia, M. and Schrottenloher, A. 2019. Quantum Security Analysis of AES. IACR Transactions on Symmetric Cryptology. 2019, 2 (Jun. 2019), 55-93. DOI:https://doi.org/10.13154/tosc.v2019.i2.55-93.

[9] Eric Conrad, Seth Misenar, Joshua Feldman, Chapter 4 - Domain 3: Security Engineering (Engineering and Management of Security), CISSP Study Guide (Third Edition), Syngress, 2016, Pages 103-217

[10] Thomas W. Edgar, David O. Manz, Chapter 2 - Science and Cyber Security, Syngress, 2017, ISBN 9780128053492,

[11] De Feo, Luca; Jao; Plut (2011). "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies" PQCrypto 2011

[12] Smith, Belinda. “Prime Numbers Keep Your Encrypted Messages Safe.” ABC News, ABC News, 19 Jan. 2018, www.abc.net.au/news/science/2018-01-20/how-prime-numbers-rsa-encryption-works/9338876.

[13] C. Cid, Information Security Group, University of London, Algebraic Analysis of AES, https://www.cosic.esat.kuleuven.be/ecrypt/AESday/slides/AES-Day-CarlosCid.pdf, October 2012.

[14] OLEJNIK, Lukasz, and Thomas ZERDICK. "Quantum Computing and Cryptography." European Data Protection Supervisor, European Union, edps.europa.eu/sites/default/files/publication/07-08-2020_techdispatch_quantum_computing_en_0.pdf.

[15] National Institute of Standards and Technology (2016). Report on Post-Quantum Cryptography. (Computer Security Resource Center), 04/28/16: NISTIR 8105 (Final) https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf

[16] National Institute of Standards and Technology (2012). Recommendation for Applications Using Approved Hash Algorithms (Computer Security Division) https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf

[17] Martin, Karen. Waiting for Quantum Computing: Why Encryption Has Nothing to Worry About. 22 Jan. 2019, techbeacon.com/security/waiting-quantum-computing-why-encryption-has-nothing-worry-about

[18] Shor, Peter W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." SIAM Journal on Computing, vol. 26, no. 5, 1997, pp. 1484–1509., doi:10.1137/s0097539795293172.

[19] Beckman, David, et al. "Efficient Networks for Quantum Factoring." Physical Review A, vol. 54, no. 2, 1996, pp. 1034–1063., doi:10.1103/physreva.54.1034.

[20] Garisto, Dan. "Quantum Computers Won't Break Encryption Just Yet." Protocol, Protocol - The People, Power and Politics of Tech, 8 Apr. 2021, www.protocol.com/manuals/quantum-computing/quantum-computers-wont-break-encryption-yet.

[21] Emerging Technology from the arXiv. "How a Quantum Computer Could Break 2048-Bit RSA Encryption in 8 Hours." MIT Technology Review, MIT Technology Review, 2 Apr. 2020, www.technologyreview.com/2019/05/30/65724/how-a-quantum-computer-could-break-2048-bit-rsa-encryption-in-8-hours/.

[22] Duits, Ines. "The Post-Quantum Signal Protocol: Secure Chat In A Quantum World." University of Twente, University of Twente, 2019, pp. 1–84.

[23] Maheshwari, Anil. “NTRU Cryptosystem and Its Analysis.” School of Computer Science, Carleton University http://people.scs.carleton.ca/~maheshwa/courses/4109/Seminar11/NTRU_presentation.pdf

[24] Ewaida, Bashar. “Pass-the-Hash Attacks: Tools and Mitigation.” SANS Institute, SANS Institute, 2021, pp. 2–35.

Appendix A

The GitHub repository for this project is at: github.com/hamayelq/MQP-Post-Quantum

Appendix B

User Entity

```
import { Field, ObjectType } from "type-graphql";
import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  BaseEntity,
  OneToMany,
  ManyToMany,
  //   ManyToOne,
} from "typeorm";
import { Chat } from "./Chat";
import { Message } from "./Message";

@ObjectType()
@Entity("users")
export class User extends BaseEntity {
  @Field(() => String)
  @PrimaryGeneratedColumn("uuid")
  uuid: string;

  @Field()
  @Column("text", { nullable: true }) // remove nullable in production
  publicKey: string; // plaintext public key

  @Field()
  @Column("text", { nullable: true }) // remove nullable in production
  encryptedPrivateKey: string; // encrypted private key

  @Field()
  @Column("text", { unique: true })
  username: string;

  /* not used for login process, should be used for password
     recovery in future */
  @Field()
  @Column("text")
  email: string;

  // no @Field() here as to not expose password
  @Column("text")
  password: string;

  /* used to check if version matches saved version in user
     upon token refresh */
```

```
  @Column("int", { default: 0 })
  tokenVersion: number;

  @Field(() => [Message])
  @OneToMany(() => Message, (messages) => messages.fromName, {
    nullable: true,
    onDelete: "CASCADE",
  })
  messages: Message[];

  @Field(() => [Chat])
  @ManyToMany(() => Chat, (chat) => chat.members, {
    nullable: true,
    onDelete: "CASCADE",
  })
  chats: Chat[];
}
```

Chat Entity

```
import { Field, ObjectType } from "type-graphql";
import {
  Column,
  Entity,
  PrimaryGeneratedColumn,
  BaseEntity,
  CreateDateColumn,
  UpdateDateColumn,
  OneToMany,
  JoinTable,
  ManyToMany,
} from "typeorm";
import { Message } from "./Message";
import { User } from "./User";

@ObjectType()
@Entity("chats")
export class Chat extends BaseEntity {
  @Field(() => String)
  @PrimaryGeneratedColumn("uuid")
  uuid: string;

  @Field(() => String)
  @Column({ nullable: true })
  name: string;

  @Field(() => Message)
  @OneToMany(() => Message, (messages) => messages.chat, {
```

```
    nullable: true,
    onDelete: "CASCADE",
  })
  @JoinTable()
  messages: Message[];

  @Field(() => [User])
  @JoinTable()
  @ManyToMany(() => User, (member) => member.chats, { onDelete: "CASCADE" })
  members: User[];

  @Field(() => String)
  @Column({ nullable: true })
  lastMessage: string;

  // pending field, boolean, for friend 'request'
  @Field(() => Boolean)
  @Column({ default: true })
  pendingRequest: boolean;

  // sentByUuid,
  @Field(() => String)
  @Column()
  sentByUuid: string;

  //sentBySymKey
  @Field(() => String)
  @Column({ default: "" })
  sentBySymKey: string;

  //acceptedBySymkey
  @Field(() => String)
  @Column({ default: "" })
  acceptedBySymKey: string;

  @Field(() => Date)
  @CreateDateColumn({ name: "createdAt" })
  "createdAt": Date;

  @Field(() => Date)
  @UpdateDateColumn({ name: "updatedAt" })
  "updatedAt": Date;
}
```

Message Entity

```
import { Field, ObjectType } from "type-graphql";
import {
  Entity,
  PrimaryGeneratedColumn,
  Column,
  BaseEntity,
  CreateDateColumn,
  ManyToOne,
} from "typeorm";
import { Chat } from "./Chat";
import { User } from "./User";

@ObjectType()
@Entity("messages")
export class Message extends BaseEntity {
  @Field(() => String)
  @PrimaryGeneratedColumn("uuid")
  uuid: string;

  @Field(() => User)
  @ManyToOne(() => User, (user) => user.messages)
  sender: User;

  @Field(() => Chat)
  @ManyToOne(() => Chat, (chat) => chat.messages)
  chat: Chat;

  @Field()
  @Column("text", { nullable: true })
  fromName: string;

  @Field()
  @Column("text", { nullable: true })
  toName: string;

  @Field()
  @Column("text", { nullable: true })
  content: string;

  @Field()
  @Column("boolean", { default: false })
  me: boolean;

  @Field(() => Date)
  @CreateDateColumn({ name: "date" })
  "date": Date;
}
```

Appendix C

User Resolver

```
import {
  Arg,
  Ctx,
  Field,
  Mutation,
  ObjectType,
  Query,
  Resolver,
  UseMiddleware,
} from "type-graphql";
import { User } from "../entity/User";
import { hash, compare } from "bcryptjs";
import { MyContext } from "../MyContext";
import { createAccessToken, createRefreshToken } from "../auth/auth";
import { isAuth } from "../auth/isAuth";
import { sendRefreshToken } from "../auth/sendRefreshToken";
import { getConnection } from "typeorm";
import { verify } from "jsonwebtoken";

@ObjectType()
class LoginResponse {
  @Field()
  accessToken: string;
  @Field()
  encryptedPrivateKey: string;
  @Field()
  publicKey: string;
  @Field(() => User)
  user: User;
}

export type contextType = {
  user: User;
};

@Resolver()
export class UserResolver {
  // bye!
  @Query(() => String)
  @UseMiddleware(isAuth)
  bye(@Ctx() { payload }: MyContext) {
    return `Your user id is: ${payload!.uuid}`;
  }
```

```
  // find users in db
  @Query(() => [User])
  async getUsers(@Arg("uuid") uuid: string) {
    const user = await User.findOne({ where: { uuid: uuid } });

    if (!user) {
      return [];
    }

    console.log(
      `getUsers request made by user with uuid ${uuid ? uuid : "NULL"}`
    );

    const users = await User.find();
    const filteredUsers: User[] = users.filter((user) => user.uuid !== uuid);
    return filteredUsers;
  }

  // return currently logged in user
  @Query(() => User, { nullable: true })
  me(@Ctx() context: MyContext) {
    // console.log(context.req.headers.authorization);
    const authorization = context.req.headers.authorization; // read header

    if (!authorization) {
      return null;
    }

    try {
      const token = authorization.split(" ")[1]; // bearer, token (so token value)
      const payload: any = verify(token, process.env.ACCESS_TOKEN_SECRET!); // verify token
is correct
      context.payload = payload as any; // set payload inside context
      return User.findOne(payload.userId);
    } catch (err) {
      console.log(err);
      return null;
    }
  }

  // this is a bad idea, should do this in a function that can be called...
  @Mutation(() => Boolean)
  async revokeRefreshTokensForUser(
    @Arg("userId", () => String) userId: string
  ) {
    await getConnection()
      .getRepository(User)
      .increment({ uuid: userId }, "tokenVersion", 1);
```

```
    return true;
  }

  // login a user
  @Mutation(() => LoginResponse) // returns LoginResponse
  async login(
    @Arg("username") username: string,
    @Arg("password") password: string,
    @Ctx() { res }: MyContext
  ): Promise<LoginResponse> {
    const user = await User.findOne({ where: { username } });

    // login error
    // username incorrect/does not exist
    if (!user) {
      throw new Error("invalid login (user does not exist)");
    }

    const valid = await compare(password, user.password);

    //password invalid
    if (!valid) {
      console.log("Invalid login attempt: ", username, password);
      throw new Error("invalid login (incorrect password)");
    }

    if (valid) {
      console.log("Valid login attempt");
    }

    // login success
    sendRefreshToken(res, createRefreshToken(user));

    return {
      accessToken: createAccessToken(user),
      encryptedPrivateKey: user.encryptedPrivateKey,
      publicKey: user.publicKey,
      user,
    };
  }

  @Mutation(() => Boolean)
  async logout(@Ctx() { res }: MyContext) {
    try {
      sendRefreshToken(res, "");
      // or
      // res.clearCookie
      return true;
    } catch (err) {
```

```
      console.log(err);
      return false;
    }
  }

  // register a user
  @Mutation(() => Boolean) // returns boolean, true worked
  async register(
    @Arg("email") email: string,
    @Arg("username") username: string,
    @Arg("password") password: string,
    @Arg("publicKey") publicKey: string,
    @Arg("encryptedPrivateKey") encryptedPrivateKey: string
  ) {
    const hashedPassword = await hash(password, 12);

    const userByName = await User.findOne({ where: { username } });
    const userByEmail = await User.findOne({ where: { email } });

    // check if username/email taken/already exists
    if (userByName || userByEmail) {
      throw new Error("exists");
    }

    try {
      await User.insert({
        username,
        email,
        password: hashedPassword,
        publicKey,
        encryptedPrivateKey,
      });
    } catch (err) {
      console.log(err);
      return false;
    }

    return true;
  }
}
```

Chat Resolver

```
import {
  Arg,
  Field,
  Mutation,
  ObjectType,
  Query,
  Resolver,
} from "type-graphql";
import { getRepository } from "typeorm";
import { User } from "../entity/User";
import { Chat } from "../entity/Chat";

export type contextType = {
  user: User;
};

@ObjectType()
class GetChatSymKeyResponse {
  @Field(() => String)
  encryptedSymKey: string;
}

@Resolver()
export class ChatResolver {
  @Mutation(() => Boolean)
  async createChat(
    @Arg("memberIds", () => [String]) memberIds: string[],
    @Arg("userId") userId: string,
    @Arg("sentBySymKey") sentBySymKey: string,
    @Arg("acceptedBySymKey") acceptedBySymKey: string
  ) {
    try {
      const user = await User.findOne({ where: { uuid: userId } });

      if (!user) throw new Error("getChats: user not authorized");

      if (memberIds.length == 1) {
        const userRepo = await this.getUserRepo(userId);

        for (let chat of userRepo.chats) {
          if (chat.members.length == 2) {
            const chatExist = chat.members.filter(
              (member) => member.uuid == memberIds[0]
            );
            if (chatExist.length >= 1) return true;
          }
        }
```

```
    }

    const members = await this.getUserObject(memberIds);

    return await this.createNewChat(
      [...members, user],
      userId,
      sentBySymKey,
      acceptedBySymKey
    );
  } catch (err) {
    console.log(err);
    return false;
  }
}

@Mutation(() => Boolean)
async createNewChat(
  members: User[],
  userId: string,
  sentBySymKey: string,
  acceptedBySymKey: string
) {
  try {
    const chat = Chat.create({
      members: [...members],
      lastMessage: "",
      sentByUuid: userId,
      sentBySymKey,
      acceptedBySymKey,
    });
    await chat.save();
    return true;
  } catch (err) {
    console.log("Save Chat FAILED\n");
    console.log(err);
    return false;
  }
}

@Mutation(() => Boolean)
async acceptRequest(
  @Arg("chatId") chatId: string,
  @Arg("encryptedSymKey") encryptedSymKey: string
) {
  try {
    const chatRepo = getRepository(Chat);
    const chatToUpdate: Chat | undefined = await chatRepo.findOne({
      where: { uuid: chatId },
```

```
    });

    if (!chatToUpdate) throw new Error("Could not find chat to update");

    chatToUpdate.pendingRequest = false;
    chatToUpdate.acceptedBySymKey = encryptedSymKey;

    await chatRepo.save(chatToUpdate);

    return true;
  } catch (err) {
    console.log(err);
    throw new Error("Error accepting request");
  }
}

@Mutation(() => Boolean)
async denyRequest(@Arg("chatId") chatId: string) {
  try {
    const chatRepo = getRepository(Chat);
    const chatToUpdate: Chat | undefined = await chatRepo.findOne({
      where: { uuid: chatId },
    });
    if (!chatToUpdate) throw new Error("Could not find chat to update");

    await chatRepo.remove(chatToUpdate);
    return true;
  } catch (err) {
    console.log(err);
    throw new Error("Error deleting request");
  }
}

@Query(() => GetChatSymKeyResponse)
async getChatSymKey(
  @Arg("chatId") chatId: string,
  @Arg("userId") userId: string
) {
  try {
    const chat = await Chat.findOne({ where: { uuid: chatId } });

    const symKey =
      chat?.sentByUuid === userId
        ? chat.sentBySymKey
        : chat?.acceptedBySymKey;

    return { encryptedSymKey: symKey };
  } catch (err) {
    console.log(err);
```

```
    throw new Error("Error getting chat symkey");
  }
}

@Query(() => [Chat])
async getChats(@Arg("userId") userId: string) {
  try {
    const user = await User.findOne({ where: { uuid: userId } });

    if (!user) throw new Error("getChats: user not authorized");

    const userRepo = await this.getUserRepo(userId);

    const chats = [];
    for (let chat of userRepo.chats) {
      if (chat.members.length == 1) {
        await chat.remove();
      } else if (!chat.name) {
        const member = chat.members.filter(
          (member) => member.uuid !== user.uuid
        )[0];
        chats.push({ ...chat, name: member.username });
      } else {
        chats.push(chat);
      }
    }


    return chats;
  } catch (err) {
    console.log(err);
    return [];
  }
}

@Query(() => [User])
async getUserObject(memberIds: string[]) {
  const members = await Promise.all(
    memberIds.map(async (memberId) => {
      const user = await User.findOne({ uuid: memberId });
      return user;
    })
  );
  return members as User[];
}

@Query(() => User)
async getUserRepo(userId: string) {
  const userRepo = getRepository(User);
```

```
    const user = await userRepo.find({
      relations: ["chats", "chats.members"],
      where: { uuid: userId },
    });

    return user[0];
  }
}
```

Message Resolver

```
import {
  Arg,
  Field,
  Mutation,
  ObjectType,
  Query,
  Resolver,
} from "type-graphql";
import { PubSub, withFilter } from "apollo-server-express";
import { getRepository } from "typeorm";
import { Chat } from "../entity/Chat";
import { Message } from "../entity/Message";
import { User } from "../entity/User";

export type contextType = {
  user: User;
};

@ObjectType()
class GetMessageResponse {
  @Field(() => String)
  uuid: string;
  @Field(() => String)
  lastMessage: string;
  @Field(() => Date)
  createdAt: Date;
  @Field(() => Date)
  updatedAt: Date;
  @Field(() => [User])
  members: User[];
  @Field(() => [Message])
  messages: Message[];
}

@Resolver()
export class MessageResolver {
```

```
@Mutation(() => Boolean)
async createMessage(
  @Arg("chatId") chatId: string,
  @Arg("content") content: string,
  @Arg("userId") userId: string
) {
  const user = await User.findOne({ where: { uuid: userId } });

  if (!user) {
    throw new Error("createMessage: user unauthorized");
  }

  const chat: Chat | undefined = await Chat.findOne({ uuid: chatId });
  if (!chat) {
    throw new Error("createMessage: can't find chat");
  }
  chat.lastMessage = content;

  const newMessage = await this.createNewMessage(content, user, chat);

  try {
    await chat.save();
  } catch (err) {
    console.log(err);
  }

  return this.triggerSubscription(chatId, newMessage);
}

@Mutation(() => Message)
async createNewMessage(content: string, user: User, chat: Chat) {
  const message = Message.create({
    content,
    sender: user,
    fromName: user.username,
    chat,
  });
  try {
    await message.save();
  } catch (err) {
    console.log(err);
  }
  return message;
}

@Mutation(() => Boolean)
async triggerSubscription(chatId: string, message: Message) {
  try {
    pubSub.publish(GET_CHAT_SUB, { getNewMessages: message, chatId });
```

```
      return true;
    } catch (err) {
      console.log(err);
      return false;
    }
  }

  @Query(() => GetMessageResponse)
  async getMessages(
    @Arg("chatId") chatId: string,
    @Arg("userId") userId: string
  ) {
    const user = await User.findOne({ where: { uuid: userId } });

    if (!user) {
      throw new Error("getMessage: user unauthorized");
    }

    let chats: Chat[] = await this.getChatRepo(chatId);

    if (!chats[0].members.some(({ uuid }) => uuid === userId)) {
      throw new Error("getMessage: user unauthorized");
    }

    let messages: any = [];

    try {
      messages = chats[0].messages.map((message) => {
        if (message.sender.uuid === user.uuid) return { ...message, me: true };
        return { ...message, me: false };
      });

      messages.sort((a: any, b: any) => +new Date(b.date) - +new Date(a.date));
    } catch (err) {
      console.log(err);
    }

    return { ...chats[0], messages };
  }

  @Query(() => Chat)
  async getChatRepo(chatId: string) {
    const chatRepo = getRepository(Chat);
    return await chatRepo.find({
      relations: ["members", "messages", "messages.sender"],
      where: { uuid: chatId },
    });
  }
```

```
  getNewMessages = () => {
    return withFilter(
      () => pubSub.asyncIterator(GET_CHAT_SUB),
      (payload, variable) => payload.chatId === variable.chatId
    );
  };
}

export const GET_CHAT_SUB = "GET_CHAT_SUB";
export const pubSub = new PubSub();
```

Appendix D

GraphQL Queries and Mutations

```
mutation Register(
  $email: String!
  $username: String!
  $password: String!
  $publicKey: String!
  $encryptedPrivateKey: String!
) {
  register(
    email: $email
    username: $username
    password: $password
    publicKey: $publicKey
    encryptedPrivateKey: $encryptedPrivateKey
  )
}

mutation Login($username: String!, $password: String!) {
  login(username: $username, password: $password) {
    accessToken
    encryptedPrivateKey
    publicKey
    user {
      uuid
      username
    }
  }
}

mutation Logout {
  logout
}

mutation CreateChat(
  $memberIds: [String!]!
  $userId: String!
  $sentBySymKey: String!
  $acceptedBySymKey: String!
) {
  createChat(
    memberIds: $memberIds
    userId: $userId
    sentBySymKey: $sentBySymKey
    acceptedBySymKey: $acceptedBySymKey
  )
```

```
}

query GetUsers($uuid: String!) {
  getUsers(uuid: $uuid) {
    uuid
    username
    publicKey
  }
}

query GetChats($userId: String!) {
  getChats(userId: $userId) {
    uuid
    name
    sentByUuid
    lastMessage
    pendingRequest
    sentBySymKey
    acceptedBySymKey
    updatedAt
    members {
      uuid
      username
      publicKey
    }
  }
}

query GetChatSymKey($chatId: String!, $userId: String!) {
  getChatSymKey(chatId: $chatId, userId: $userId) {
    encryptedSymKey
  }
}

mutation AcceptRequest($chatId: String!, $encryptedSymKey: String!) {
  acceptRequest(chatId: $chatId, encryptedSymKey: $encryptedSymKey)
}

mutation DenyRequest($chatId: String!) {
  denyRequest(chatId: $chatId)
}

mutation CreateMessage($chatId: String!, $content: String!, $userId: String!) {
  createMessage(chatId: $chatId, content: $content, userId: $userId)
}

query GetMessages($chatId: String!, $userId: String!) {
  getMessages(chatId: $chatId, userId: $userId) {
    uuid
```

```
    messages {
      uuid
      date
      content
      me
      sender {
        uuid
        username
      }
    }
  }
}
```

Appendix E

AES Substitution Box Implementation

```
import textwrap


def hex2binary(hexlist):
    #converting hex number to its binary representation
    binaryList = []
    for hexnum in hexlist:
        item = bin(int(hexnum, 16))[2:].zfill(8)
        binaryList.append(item)
    return binaryList


def hex2decimalaccess(hexlist):
    #converting hex number to 2 decimal numbers to access the SBOX
    #for substitution. Ez clap.
    accessDecimals = []
    for hexnum in hexlist:
        decimalVals = []
    for hexvalue in hexnum[2:]:
        decimalVals.append(int(hexvalue, 16))
        accessDecimals.append(decimalVals)

    return accessDecimals

class AES:
    def __init__(self, input, SBOX):
        self.input = input
        self.SBOX = SBOX


    def convert2eight(self):
        #converting input to 8 bit chunks
        eightbitlist = textwrap.wrap(self.input, 8)
        return eightbitlist


    def convert2hex(self):
        #converting list to hex values
        hexlist = []
        for eightbits in self.convert2eight():
            converted = hex(int(eightbits, 2))
            if(len(converted) == 3):
                converted = "0x" + converted[2:].zfill(2)
                hexlist.append(converted)
```

```
        else:
                hexlist.append(converted)
        return hexlist


    def sboxSubstitution(self):
        #performing substitution with SBOX table
        accessList = hex2decimalaccess(self.convert2hex())
        convertedList = []
        for accessValues in accessList:
            for i in range(1):
                item = hex(self.SBOX[accessValues[i]][accessValues[i+1]])
                item = "0x" + item[2:].zfill(2)
                convertedList.append(item)
        return convertedList


    def substitutedBinary(self):
        #converting hex list to binary output
        final = "\n"
        output = hex2binary(self.sboxSubstitution())
        final = final.join(output)
        return final

aAES = AES(bitinput, SBOX)
print("\n".join(aAES.convert2eight()))
print(", ".join(aAES.convert2hex()))
print(", ".join(aAES.sboxSubstitution()))
print(aAES.substitutedBinary())

SBOX = [
 [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
0xfe, 0xd7, 0xab, 0x76],
 [0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
0x9c, 0xa4, 0x72, 0xc0],
 [0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
0x71, 0xd8, 0x31, 0x15],
 [0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
0xeb, 0x27, 0xb2, 0x75],
 [0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
0x29, 0xe3, 0x2f, 0x84],
 [0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
0x4a, 0x4c, 0x58, 0xcf],
 [0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
0x50, 0x3c, 0x9f, 0xa8],
 [0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21,
0x10, 0xff, 0xf3, 0xd2],
 [0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
0x64, 0x5d, 0x19, 0x73],
```

```
 [0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb],
 [0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62,
0x91, 0x95, 0xe4, 0x79],
 [0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea,
0x65, 0x7a, 0xae, 0x08],
 [0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
0x4b, 0xbd, 0x8b, 0x8a],
 [0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
0x86, 0xc1, 0x1d, 0x9e],
 [0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9,
0xce, 0x55, 0x28, 0xdf],
 [0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
0xb0, 0x54, 0xbb, 0x16],]
```