Cardiff University

School of Computer Science and Informatics

CM3203

One Semester Individual Project - 40

Quantum-resistant cryptographic protocol for instant messaging

FINAL REPORT

Author: Lyubomir Kyorovski
Student Number: [redacted]

Supervisor: Philipp Reinecke
Moderator: Federico Liberatore

June 5, 2020

Abstract

When powerful quantum computers become reality, traditional means of secure communication over the network will be rendered insecure to a great extent, as the underlying mathematical problems of some cryptosystems are easily solved by such devices. The aim of this project is to develop an instant messaging protocol that is not susceptible to quantum computing attacks. Users of an application that implements the protocol should be guaranteed that the messages exchanged through it will not be tampered with or read on the way to the intended recipient. Users should also not be able to impersonate other users, i.e. send messages from another user's name. As a proof-of-concept, a web-based application that implements the designed protocol will be developed, whose intended use will be on desktop browsers. Performance measures of the application (and, therefore, the protocol) running in a desktop web browser will be taken and discussed. Additionally, the performance of the application in a mobile web browser will be measured, to gain insight as to how well the protocol applies to mobile instant messaging. The primary focus of the project is to design a protocol. The developed application that implements the protocol may introduce security risks and is not to be seen as a finished product, but rather as a prototype to be improved and extended in the future.

Acknowledgements

I would like to thank my supervisor, Dr. Philipp Reinecke, for the guidance and advice he has provided during the course of this project.

I would like to thank the developers of the Signal protocol for the astonishing work that they have done and the flawless papers they have written.

Thank you to all friends and family who helped me keep things in perspective.

Contents

Abstract 1

Acknowledgements 1

Table of Contents 2

List of Figures 4

List of Tables 5

1 Introduction 6

2 Background 9
2.1 Cryptography . . . 9
2.1.1 Symmetric cryptography . . . 9
2.1.2 Public key exchanges . . . 11
2.1.3 Digital signatures . . . 12
2.1.4 Hash functions . . . 13
2.1.5 Message authentication codes . . . 14
2.1.6 Key derivation functions . . . 15
2.1.7 Security level . . . 15
2.2 Impact of quantum computing on traditional cryptography . . . 16
2.3 Post-quantum cryptography . . . 17
2.4 Security properties of a secure instant messaging protocol . . . 18
2.5 Signal - the secure instant messaging protocol . . . 18
2.5.1 Extended Triple Diffie-Hellman (X3DH) key agreement protocol 19
2.5.2 The Double Ratchet algorithm . . . 22
2.5.3 Incorporating the X3DH protocol with the Double Ratchet algorithm to produce the Signal protocol . . . 33
2.6 Related work . . . 34

3 Specification, design and implementation 34
3.1 Assumptions . . . 34
3.2 Attacker model . . . 34
3.3 Identifying vulnerable cryptographic primitives in the Signal protocol and quantum-resistant replacements for them . . . 35
3.3.1 Analysis of Double Ratchet algorithm . . . 35
3.3.2 Analysis of X3DH key agreement protocol . . . 36
3.4 Post-quantum alternative for the X3DH key agreement protocol . . . 36

3.5 The post-quantum Signal protocol and the messenger application . . 39
3.6 Preventing side-channel attacks in the messenger application 40
3.6.1 Replay attacks 40
3.6.2 Identity key swap attacks 41
3.6.3 Cross-site scripting attacks 41
3.7 Software architecture 42
3.7.1 Design 42
3.7.2 Implementation 43

4 Results and evaluation 45
4.1 Security properties of the devised quantum-resistant secure instant messaging protocol 45
4.2 Messenger application capabilities 45
4.3 Performance testing 46
4.3.1 Methodology 46
4.3.2 Results 47

5 Future work 48
5.1 Supporting audio and video telephony 48
5.2 Supporting group messaging 49
5.3 Developing a supersingular-isogeny-elliptic-curve-based digital signature that uses supersingular isogeny Diffie-Hellman keys 49

6 Conclusions 49

Appendix A File listing 51

Appendix B Code libraries used 55

Appendix C Project README 60

Appendix D User Manual 62

References 67

List of Figures

1 Illustration of Diffie-Hellman algorithm with colours [1] 12
2 Comparison of conventional and quantum security levels of typical ciphers [2] 16
3 Impact of quantum computing on common cryptographic algorithms [3] 17
4 X3DH calculations between keys [4] 21
5 A KDF chain [5] 23
6 Two symmetric-key ratchet steps [5] 25
7 Initialisation of the Diffie-Hellman ratchet [5] 25
8 Bob's DH ratchet step [5] 26
9 Alice's DH ratchet step [5] 27
10 DH outputs as chain keys [5] 27
11 A full DH ratchet step with a KDF chain [5] 28
12 Initial step of the Double Ratchet algorithm [5] 28
13 Double Ratchet state when Alice sends her initial message [5] 29
14 Double Ratchet state when Alice receives a message from Bob [5] . . 29
15 Double Ratchet state when Alice receives a message with Bob's ratchet key not being changed, and sending some messages after that [5] . . . 30
16 Alice's final Double Ratchet state [5] 31
17 Out of order message keys [5] 32
18 Protocol 4 [6] 37
19 Model-view-controller architecture [7] 42
20 Implementation of the architecture 44

List of Tables

1 Bit lengths of public-key algorithms for different security levels [8] . . 15
2 Desktop performance metrics 47
3 Mobile performance metrics 48

1 Introduction

In recent years, a large part of all communication between individuals is carried out online - whether it is in the form of electronic mail or social networks, nearly everyone has had to transmit some data to someone else digitally. In the general use case, individuals want this data to be seen by the intended recipient and nobody else. Through the use of cryptography, the confidentiality of the data can be achieved even over insecure channels, even when a third party can view all of the transmitted messages. Throughout most of the history of mankind, this was the main purpose of cryptography, as the etymology of the word reveals[1] - the "art of writing in secret characters". But with the development of digital computers and most notably the Internet, the field of cryptography was extended to include procedures such as digital signatures, checking for the integrity of data, and many other, and ways of combining them to ensure the secure transmission of data, with additional properties such as authenticity. The combination of some of these procedures and the order in which they should be executed form cryptographic (security) protocols. Examples for such protocols include TLS [9], SSH [10] and many other, that are widely used nowadays, some standalone, others in conjunction with other plain text application protocols, such as HTTP over TLS to produce a secure version - HTTPS [11].

As the Internet became widely adopted by countries around the world, electronic mail started to become increasingly popular and used for casual conversations. However, it was not best suited for real-time conversation due to the nature of electronic mail protocols and clients, which led to the birth of instant messaging (IM) - real-time text transmission between two (or multiple) parties [12]. It is one of the primary forms of communication nowadays - nearly all social media platforms have instant messaging capabilities and there are even standalone applications for instant messaging, such as Viber[2], Skype[3] and many others. The concept of instant messaging is not limited to text messages and some implementations include audio and video telephony, file transfer, etc. Naturally, cryptographic protocols for instant messaging were developed, most of which are for end-to-end encryption, meaning that messages are revealed in their plain form only at both ends of the conversation and cannot be read at an intermediate node, e.g. a server that relays messages.

Most cryptosystems (the building blocks of security protocols) are based on mathematical problems that are easy to compute but hard to reverse [8], and are secure precisely because of that - even an adversary possessing large classical computational power resources is unable to "break" the cryptosystem, by brute-forcing or

[1] https://www.etymonline.com/word/cryptography
[2] https://www.viber.com/en/
[3] https://www.skype.com/

other means. However, with the development of the theory of quantum computation and algorithms, it became clear that quantum computers are able to solve these problems relatively easily, which puts the security of cryptographic protocols at risk [3]. While currently the development of quantum computers capable of running these algorithms at their maximal potential is held back by engineering challenges, history has shown that engineering nearly always finds a way, and when such a device is built, it is quite possible that most if not all of the cryptographic protocols that are used nowadays will be rendered insecure.

Work has begun to establish a standard for post-quantum cryptography, i.e. cryptography that is safe from attacks by quantum computers, by various organisations, most notably the US National Institute of Standards and Technology (NIST)[4]. Different post-quantum algorithms are being developed, analysed, tested, and even implemented by individuals and companies. But their practical implementation in secure communication protocols is limited due to the fact that they are simply not as well analysed and bug-proofed as traditional algorithms. Even taking this into consideration, work toward establishing post-quantum versions of cryptographic protocols should begin, so that we are not caught unprepared in the event one day we wake up and the quantum computer is there for adversaries to take advantage of.

To aid in preparation for a post-quantum world, this project's task is to develop a quantum-resistant secure instant messaging protocol, because, as can be deduced from earlier, instant messaging is one of the most popular technologies where cryptographic protocols are used. The approach taken will be as follows: choosing a secure instant messaging protocol, having its building blocks analysed for vulnerable cryptography (in the context of a post-quantum world), and finding suitable replacements for them that would convert it to a quantum-safe one. Although it may seem easy to swap out classic cryptography for post-quantum cryptography, in practice this may not be possible due to the structure of the protocol. In general, post-quantum cryptography may introduce performance and network communication penalties, such as slower run-times and/or larger key sizes. Because a secure instant messaging protocol has various desiderata (to be discussed), the post-quantum version of the protocol may not be able to achieve all of the ones available in the classic version due to the fact that the post-quantum cryptographic suite is not yet as large as the classic one. To demonstrate the use of the protocol, a messaging application will be developed that can be accessed on desktop devices using a web browser. The use of code libraries will be employed to ensure that the implementation of quantum-resistant cryptographic algorithms is correct and does not introduce security vulnerabilities, as well as to aid other parts of the development process where applicable. After the application is developed, performance of the implementation

[4] https://www.nist.gov/

will be measured, and the results will be discussed.

The intended audience for this project is any individual (or organisation) who wishes to protect their communication in a post-quantum world, as well as cypherpunks[5] and cryptographers willing to improve the developed protocol. Due to the time limitations, the web application implementing it may not be able to handle a large user-base, such as large organisations, but may be used by small ones and groups of individuals, provided they host their own instance of the application.

The scope of the project is primarily focused on developing the protocol. The developed application that implements it should be seen as a prototype and not be considered perfectly secure, as it may introduce implementation vulnerabilities due to the short time period in which it was developed. The protocol and, therefore, application, will only be capable of sending one-to-one text messages, although discussion will be provided on extending it to handle group messaging and other types of communication, namely audio and video telephony. Evaluation of the user interface, e.g. heuristic evaluation, will not be performed, nor user testing will be conducted, as the development of an application is not the main focus of the project.

The assumption made for this project is that the communication overhead (e.g. larger key size) of using quantum-resistant and its impact on throughput will be considered negligible, given that the application is aimed at desktop use, where data caps and severe network speed throttling are not present (in contrast to mobile networks, where a service provider may limit the use of data). Another reason for this assumption is the increasing speed of communication links.

The tangible outcomes of the project are:

- A specification of a quantum-resistant cryptographic protocol for instant messaging
- A web application that implements the protocol
- Performance measures of the protocol

[5] A person who uses encryption when accessing a computer network in order to ensure privacy, especially from government authorities. https://www.lexico.com/definition/cypherpunk

2 Background

2.1 Cryptography

The following sections introduce some cryptographic processes and terminology.

The main purpose of cryptography is to prevent unauthorised third parties from accessing private communication between two parties, or at least render it difficult. The two parties will be referred to as "Alice" and "Bob", and the unauthorised third party/attacker/adversary will be referred to as "Eve" - these characters were introduced by Rivest, Shamir and Adleman [13] and do not refer to humans, but rather to generic agents such as programmes and computers. The following subsections are a brief and partial summary of the book *Understanding cryptography: A textbook for students and practitioners* [8], which can be referred to for a more detailed explanation. The main security properties that cryptography aims to achieve are:

- **Confidentiality** - ensuring that no party other than Alice and Bob is able to read the information sent between them
- **Integrity** - ensuring that no third party modified or changed the information sent between Alice and Bob
- **Authentication** - ensuring that the party which Alice or Bob communicate with is authentic, i.e. they are the party that Alice or Bob intend to communicate with.

2.1.1 Symmetric cryptography

Symmetric cryptography was the only form of cryptography that existed for the better part of human history. Its main use case is when Alice and Bob want to communicate over an insecure channel (communication link). To achieve this, Alice can use a symmetric cipher, keyed by a secret k to encrypt the plaintext x, yielding a ciphertext y. Plaintext refers to any data in its raw form that is human- or computer-readable. Ciphertext is the encoded version of the plaintext, which is not readable by humans or computers without the cipher and secret used to produce it. The secret (key) is a parameter used in cryptographic algorithms that determines the functional output of a cryptographic function. Encryption is the process of converting plaintext to ciphertext, using a secret k:

$$y = E_k(x)$$

Decryption is therefore the inverse process of encryption:

$$x = D_k(y)$$

For symmetric algorithms, the key k must be the same at both ends of a conversation. There exist asymmetric encryption algorithms such as RSA [13] in which the keys used for encryption and decryption are different, however, they are not crucial for the understanding of this project.

There are two types of symmetric ciphers: stream ciphers and block ciphers. Stream ciphers encrypt bits individually, which usually involves a single XOR operation, making them fast and small. They usually require a cryptographically secure pseudo-random number generator to generate a potentially infinite number of unpredictable key bits from the secret, in order to encrypt a continuous flow of data. Block ciphers, on the other hand, encrypt data in blocks (several bits, usually 128 or 256) and are slower, but are better applicable to internet applications. One of the most popular block ciphers is AES [14].

Because block ciphers are limited to only encrypting a block of bits, there is a need for mechanisms to encrypt data larger than the block size. These mechanisms are called modes of operation, in which long data is divided into blocks. Different ones exist, the simplest one of them being the electronic codebook (ECB) mode, in which a plaintext block is mapped statically to a ciphertext block, i.e.

$$y_i = E_k(x_i)$$
$$x_i = D_k(y_i)$$

where i is the consecutive number of the block. This mode of operation is, however, insecure, as it is susceptible to frequency analysis and replay attacks. A more secure version is the cipher block chaining mode (CBC), in which each plaintext block is XOR-ed with the previous ciphertext block before encryption. As there is no ciphertext available before the first block, an initialisation vector (IV) is used, which is transmitted along with the ciphertext to the recipient. It is important that the IV value is used only once in order to reduce the possibility of cryptanalysis. The equations for encryption and decryption are as follows:

$$y_i = E_k(x_i \oplus y_{i-1}), y_0 = IV$$
$$x_i = D_k(y_i) \oplus y_{i-1}, y_0 = IV$$

These modes of operation only ensure the confidentiality of the data, but there are some that achieve authenticity and/or integrity, which are to be introduced later in the project.

To encrypt data smaller than the block size, the data needs to be padded, and there are different ways of doing that. The most popular for AES is PKCS#7 [15].

2.1.2 Public key exchanges

Symmetric cryptography needs an already established secret between Alice and Bob so that they can communicate safely, but it may be hard to agree on a secret through an insecure channel. A common strategy is to share the secret through asymmetric schemes such as RSA [13]. This way of sharing a secret is called a key encapsulation mechanism (KEM). However, cryptosystems such as the Diffie-Hellman key exchange (DHKE) have been created specifically for the purpose of key exchange [16]. DHKE is based on the discrete logarithm problem. The original specification used a combination of the properties of prime numbers to achieve its security, but variants have been developed that improve it, such as the elliptic-curve cryptography version - ECDH [17] - which comes with some benefits, such as a smaller key size. In general, all cryptography based on elliptic curves is similarly built upon a variation of the discrete logarithm problem, called the elliptic curve discrete logarithm problem.

A generalisation of the key exchange is as follows:

1. Alice and Bob generate their own (private) keys, A and B respectively, that they keep to themselves and do not reveal to anyone else.
2. With the aid of a transformation function T, which is easy to compute but hard to invert, Alice and Bob compute and exchange their transformed (public) keys, i.e. Alice sends *T(A)* to Bob and Bob sends *T(B)* to Alice.
3. Using a symmetric mixing function M, Alice and Bob compute the shared secret K by computing $K = M(A, T(B))$ on Alice's side and $K = M(B, T(A)$ on Bob's side.

This process can be illustrated as colour mixing - see Figure 1.

If we assume Eve captured the exchanged keys *T(A)* and *T(B)*, she would need to reverse only one of them to reveal one of the secret keys and use it to compute K. However, because *T()* is difficult to inverse, Eve cannot do that in a feasible time.

However, Eve may be able to execute a Man-in-the-Middle (MITM) attack, in which she would trick Bob into believing that she is Alice, and Alice would be led to believe that Eve is Bob. Eve would generate two keys, E_a and E_b, sending $T(E_b)$ to Alice and E_a to Bob. This would lead to generating two shared secrets in the system - $K_a = M(A, T(E_b))$ and $K_b = M(B, T(T(E_a))$, and Eve will also know both of them. Eve can therefore intercept all communication from Alice to Bob, decrypting transmitted data using the respective shared key, revealing the plaintext, after which the plaintext gets re-encrypted with the other shared secret and sent to the respective party.

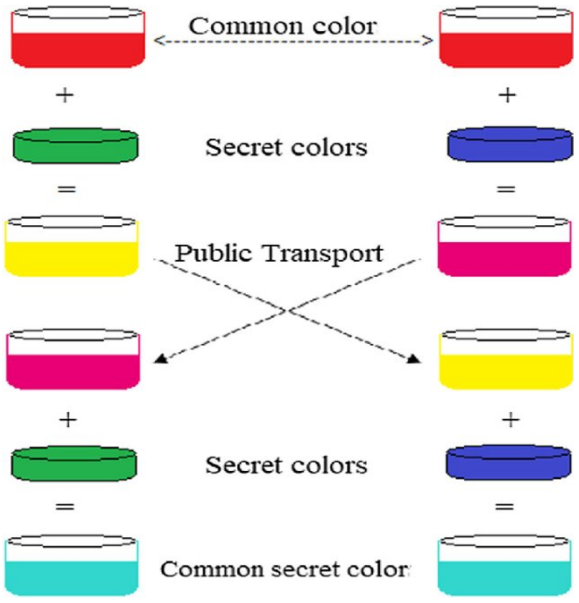


Figure 1: Illustration of Diffie-Hellman algorithm with colours [1]

To prevent a MITM attack, there must be a way for Alice and Bob to make sure that the transformed key they received truly belongs to the intended party.

2.1.3 Digital signatures

In order to communicate securely, Alice and Bob need to authenticate the data they share, such as the transformed keys in DHKE. In the real world, authenticating data such as contracts is commonly done through signature. As the digital world mimics the real world, cryptographic algorithms have been developed to create digital signatures and sign data with them. Each signatory has a key pair consisting of a private and a public key. The private key is used to sign the data, generating a signature, and the public key is used by the other party to confirm that the signature for the provided data is valid. Signing a message x to produce a signature s can be expressed as:

$$s = sig_{K_{private}}(x)$$

The signature is sent alongside the message, and the other party can verify the signature by computing:

$$x' = ver_{K_{public}}(s)$$

and checking whether $x' = x$. If the values are same, the message is authentic. In general, signature schemes are based on the prime factorisation problem or the discrete logarithm and its elliptic curve variant. RSA [13] can be used as a signature scheme, but the most commonly used nowadays is the Elliptic Curve Digital

Signature Algorithm (ECDSA) [18], which is the elliptic curve variant of the Digital Signature Algorithm (DSA) [19].

The most important security property of signature schemes is not allowing signatures to be forged, which includes creating fraudulent digital signatures for a given message or creating new messages for existing digital signatures. If Eve could do that, she would be able to impersonate the other parties. This property is normally guaranteed by the underlying mathematical problem. However, there exists another type of forgery, the existential forgery, in which Eve creates a random signature s_r and verifies it, producing a message x_r. If x_r is sent to the verifying party along with the signature s_r, the result of the verification, by construction, will be equal to x_r, hence validation succeeds; however, Eve cannot choose the message x_r. To prevent this type of forgery, padding can be used, such as the Encoding Method for Signature with Appendix – Probabilistic Signature Scheme (EMSA-PSS) [20] for RSA.

An important property of these secure digital signatures is the implied **non-repudiation**, as the issuer of a signatures cannot claim that he/she did not sign a message and deny sending it, as only the issuer has the private key with which valid signatures are produced.

2.1.4 Hash functions

There are cases in which it makes more sense to send a shorter representation of the data, rather than a full copy. For example, if a large message had to be signed, the data transmitted over the network would double, as the signature in most schemes is the same size as the message. Instead, a shorted representation of the data may be signed to achieve the same level of security.

Cryptographic hash functions are tasked with creating a shorter representation of data. A cryptographic hash-function takes a message of arbitrary size and maps it to an output with a fixed size, called the "hash" or "digest" of the message. The security requirements of a hash function are:

- **Preimage resistance** - given a hash value z, it should be computationally infeasible to find a message x that hashes to z
- **Second preimage resistance** - given a message x and its hash z, it should be computationally infeasible to find another message $x' \neq x$ with the same hash $z' = z$.
- **Collision resistance** - it should be computationally infeasible to find two messages x and x', $x \neq x'$ with the same hash $h(x) = h(x')$

The effort needed for breaking first and second preimage resistance is equivalent to

guessing the message by brute-force - trying all possible inputs to the hash function - Eve needs 2^n tries, where n is the number of output bits of a hash function. To break collision resistance however, Eve only needs to perform only $\sqrt{2^n} = 2^{\frac{n}{2}}$ operations due to the Birthday paradox [8, pp. 299–302].

The main application of hash functions is integrity checking. They are widely used as building blocks in other cryptographic algorithms. The most popular and widely used family of cryptographic hash functions are the Secure Hash Algorithms (SHA) published by NIST.

2.1.5 Message authentication codes

Message authentication codes (MAC) aim to provide message integrity and message authentication by using a symmetric key to protect messages exchanged between Alice and Bob. They differ from digital signature schemes because they do not offer non-repudiation, as the same key is used at both ends of the conversation, however, their main benefit is being faster than them. Similar to digital signatures, the procedures for signing is:

$$m = MAC_K(x)$$

Sending the MAC m alongside the message x, the recipient can verify the message by calculating

$$x' = MAC_K(m)$$

and verifying by checking whether $x' = x$.

A good MAC function should ensure that it is computationally infeasible for Eve to achieve:

- **Existential forgery** - given one or more text-MAC pairs (x_i, m_i), find a new valid pair (x', m')
- **Selective forgery** - given one or more text-MAC pairs (x_i, m_i) and a new message x', find $m' = MAC(x')$
- **Key recovery/Universal forgery** - given one or more text-MAC pairs (x_i, m_i) for a key K, recover K.

MAC functions can be constructed from other cryptographic algorithms, such as block ciphers and hash functions. One of the most commonly used MAC functions is HMAC (Hash-based message authentication code), in which the MAC is generated by prepending the key to the message, hashing them, prepending the key to the hash output, and hashing again. The security of HMAC is therefore largely based on the security of the underlying hash function.

2.1.6 Key derivation functions

Using the secret derived from a public key exchange algorithm repeatedly to encrypt messages is not desirable, as a compromise of only one of the messages will lead to a compromise of all previous and future messages (see Backward secrecy and forward secrecy). A key derivation function (KDF) is used to derived new keys from old keys and/or other data in a cryptographically secure way. This function cannot be reversed, so older keys cannot be revealed if a current key is revealed. The procedure of deriving a key can be written as:

$$K_{i+1} = KDF(K_i)$$

where K_{i+1} is the output of the KDF (the new key) and K_i is some input data.

A perfect candidate for constructing key derivation functions are hash functions, as due to the first preimage resistance property, it is infeasible for Eve to calculate K_i from K_{i+1}. Although a hash function may be directly used as key derivation functions, control over the size of the output key (e.g., the output key needs to be larger than the digest size) or derivation of keys from human-readable passwords may be needed without compromising security. For this purpose, KDFs such as HDKF, which is based on an HMAC function, and the password-based key derivation function (PBKDF) have been developed.

2.1.7 Security level

The security level of cryptographic primitives is a measure of their resistance against the best known attacks against them. It is expressed in bits - an n-bit security level means that an attacker needs to carry out 2^n operations in order to "break" the primitive. Symmetric algorithms are designed to have a security level equal to the key size.

In hybrid cryptosystems, where multiple primitives are combined, the security level is the same as the one of the "weakest link in the chain", i.e. the security level of the cryptosystem is equal to that of the weakest primitive.

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer factorisation	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete logarithm	DH, DSA, Elgamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric-key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

Table 1: Bit lengths of public-key algorithms for different security levels [8]

2.2 Impact of quantum computing on traditional cryptography

In 1994, Peter Shor [21] showed that quantum computers, devices which take advantage of the physical properties of matter and energy for the purpose of calculation, can efficiently solve one of the problems that was previously considered computationally hard. The developed polynomial-time quantum algorithm, now referred to as Shor's algorithm, is capable of efficiently factoring prime numbers and computing discrete logarithms (which includes the elliptic curve variant of the problem), which are the problems most public key cryptography, such as key exchanges, digital signatures and asymmetric encryption schemes are based upon.

Lov Grover developed an algorithm [22] that finds, with high probability, the unique input to a black-box function that produces a specified output value by running the function only (at most) $\sqrt{N}$ times, where N is the size of the function's domain. Hash function and symmetric cryptography schemes are examples of such black-box functions.

Algorithm	Key Length	Effective Key Strength / Security Level	
		Conventional Computing	Quantum Computing
RSA-1024	1024 bits	80 bits	0 bits
RSA-2048	2048 bits	112 bits	0 bits
ECC-256	256 bits	128 bits	0 bits
ECC-384	384 bits	256 bits	0 bits
AES-128	128 bits	128 bits	64 bits
AES-256	256 bits	256 bits	128 bits

Figure 2: Comparison of conventional and quantum security levels of typical ciphers [2]

The impact of the two algorithms on classical cryptography is summarised below:

It can be seen from Figure 3 that the algorithms susceptible to Grover search [22] (symmetric encryption schemes, cryptographic hashes) can thwart the attack by only increasing key size, but public-key schemes, such as digital signatures, key exchange and asymmetric encryption need whole new replacements, based on other mathematical problems.

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 3: Impact of quantum computing on common cryptographic algorithms [3]

2.3 Post-quantum cryptography

Because most of classical cryptography has been theoretically defeated by quantum computing, a new collection of algorithms has been under development that are not vulnerable to quantum(and also classical) attacks - post-quantum cryptography. The focus of this group is on public-key schemes. Currently there are several families of quantum-resistant algorithms - lattice-based, code-based, supersingular-isogeny-elliptic-curve-based, and others [3]. Although they are not susceptible to quantum attacks, they tend to perform worse in some aspects, such as key size and speed, in comparison to classical schemes [23].

Lattice-based cryptosystems are relatively efficient, and their security is provably secure against brute-force attacks, however, it is difficult to give precise estimates of security against other cryptanalysis techniques. [3]

Code-based cryptosystems are based on error-correcting codes [24], which makes them fast, but suffer from large key sizes. Attempts have been made to reduce the key sizes of such schemes, but most often this has led to successful attacks against these versions [3].

The hash-based family is made up mainly of signature schemes that are constructed using hash functions. Because the security of hash functions against quantum attacks is well understood, the security of this family is well understood too. Their main drawbacks of this family are the large signature size and the fact that such schemes can produce only a limited number of signatures before becoming insecure. This limit can be increased, but this also increases the size of the signature. [3]

Schemes based on evaluating isogenies on supersingular elliptic curves [25] are characterised with relatively short key sizes (compared to other post-quantum algorithms). In contrast to the discrete logarithm problem on elliptic curves, which can be solved relatively fast by Shor's algorithm [21], the isogeny problem on supersingular elliptic curves has no known polynomial-time attacks in both quantum and classical computing, but, as a relatively new family, the schemes based on it have the drawback of being not that well analysed, which leads to a lack of confidence in their security [3].

2.4 Security properties of a secure instant messaging protocol

In addition to confidentiality, integrity and authentication, secure protocols aim to provide other properties related to the long-term security of communications exchanged via them. For secure instant messaging protocols, the following additional properties are most commonly required:

- **Forward secrecy** - If at time t the keys used to encrypt a message x_t are compromised, an adversary should not be able to read the messages prior to x_t, i.e. messages x_0 to x_{t-1}. [26]
- **Backward secrecy (also known as Future secrecy)** - If at time t the keys used to encrypt a message x_t are compromised, an adversary should not be able to read the messages following x_t, e.g. x_{t+1} [26]
- **Deniability** - There should not exist cryptographic proof that a party took part in communication with another party. Most often, this refers to the ability of a sender to deny having sent a specific message. [26]

Deniability and authentication may seem as mutually exclusive properties, as authentication implies that the source of each message must be cryptographically confirmed, but deniability aims to provide the opposite. However, there are methods to achieve both, which are presented in the next section.

2.5 Signal - the secure instant messaging protocol

In the present day there exist multiple secure instant messaging protocols - Matrix[6], OMEMO[7] and Tox[8]. Some protocols offer peer-to-peer communication, meaning that the clients are connected directly to each other, and others use a server to relay

[6] https://matrix.org/docs/spec/
[7] https://xmpp.org/extensions/xep-0384.html
[8] https://toktok.ltd/spec

the messages between clients. As peer-to-peer communication tends to increase complexity, the latter type of protocol was considered for this project. One of the most popular such protocols nowadays is the Signal protocol (previously known as TextSecure), defined in [27, 4, 5, 28]). Given the fact that it is well analysed and proven to be secure, on multiple occasions [29, 30], along with being better documented compared to other protocols, it was chosen as the basis of this project.

The following subsections provide a short summary of [4, 5] and insight into the principle of work of the Signal protocol.

2.5.1 Extended Triple Diffie-Hellman (X3DH) key agreement protocol

It is common that during communication the receiving party is offline. A traditional Diffie-Hellman key exchange in this case will not be applicable, as for it both parties need to be online in order to compute a unique secret (shared key). It is possible that the receiving party published their public Diffie-Hellman key to a server, so that it is available to sending parties in an offline scenario, however, such re-usage of a single Diffie-Hellman key can lead to security flaws in the long term[31]. Also, the sending party and the key used in this process need to be authentic, i.e. they truly belong to the receiving party, in order to prevent MITM attack attempts. To overcome these challenges, as well as provide some additional properties, the X3DH key agreement protocol [4] was devised.

The protocol involves 3 actors:

- **Alice** wants to send Bob some data, which she wants to encrypt to prevent it being read from other parties. She also wishes to establish a shared key between her and Bob, for future communication.
- **Bob** want to enable other parties, like Alice, to establish a shared secret with him and send data to him in an encrypted format. Unfortunately, Bob may be offline when Alice attempts to do this.
- **Server** - stores messages sent from Alice to Bob, which Bob can retrieve when he comes online. The server allows Bob to publish information which the server will provide to sending parties (like Alice), which they will use to generate a shared secret, and will be able to do that even when Bob is offline. The server is assumed to be trusted.

The protocol makes use of elliptic curve cryptography and key derivation functions. A specific signature scheme, developed by the creators of the Signal protocol, is used - XEdDSA(or VXEdDSA) [27], which allows creation and verification of signatures using elliptic curve Diffie-Hellman private and public keys.

The following notation for cryptographic functions will be used:

- $X||Y$ is concatenation of byte sequences X and Y.
- $DH(DH_{privateKey}, DH_{publicKey})$ is a function which mixes two elliptic curve Diffie-Hellman (ECDH) keys and outputs a shared key as byte sequence (same as $K = M(A, T(B))$ from subsubsection 2.1.2)
- $KDF(KM)$ is a functions which derives 32 bytes (256 bits) of output from some input using the HKDF algorithm [32]. Additional parameters for this algorithm are salt - a zero-filled byte sequence with length equal to HDKF's hash function output length - and info - an ASCII string which identifies the application.

Alice and Bob each have a long-term identity ECDH key pair - IK_A and IK_B, respectively. Bob has a signed ECDH key pair SPK_B, which he changes periodically, and a set of ECDH one-time key pairs $OTPK_B$, each of which is used only once - for each run of the X3DH protocol with Bob, a unique $OTPK_B$ is used. Bob's key pairs are called "prekeys" because they are published to the server prior to Alice initiating the protocol. During a protocol run, Alice will generate an ephemeral key pair EK_A. After a successful run of the X3DH protocol, a 32-byte secret key SK will be shared between the two parties.

The protocol consist of 3 phases:

1. Bob publishes the public keys of IK_B and prekey key pairs (SPK_B, $OTPK_B$) to the server.
2. Alice fetches a "prekey bundle" from the server, and uses it to generate SK, which she can then use in some way to send an initial message to Bob
3. Bob receives Alice's initial message and processes it, he too generating SK

In the first phase, Bob generates a set of ECDH key pairs. He uses one of them as IK_B. Bob assigns another one as his signed prekey key pair SPK_B and signs its public key with IK_B, producing a signature $S_{SPK_B} = sig_{IK_B}(SPK_B)$. The remaining ones, Bob assigns as his one-time prekeys - $OTPK_B^1, OTPK_B^2, etc.$. Bob uploads the public keys of each key pair, along with the signature S_{SPK_B} to the server. Bob can upload new $OTPK_B$s at other times, e.g. when the server informs him of a shortage of $OTPK_B$s. Bob will change and publish SPK_B along with S_{SPK_B} at some time interval, but will keep track of previous SPK_Bs.

When Alice wants to perform an X3DH key agreement with Bob, she contacts the server to get a "prekey bundle", which contains Bob's identity key IK_B, Bob's signed prekey SPK_B, Bob's prekey signature S_{SPK_B} and, if available, one of Bob's

one-time prekeys $OTPK_B$. The server deletes from its records the $OTPK_B$ that was sent to Alice. Alice then verifies the prekey signature, terminating the protocol in case it is not valid. Alice generates an ephemeral key pair EK_A, which she uses alongside her identity key IK_A and keys from Bob's "prekey bundle" to generate the shared secret. If the bundle does not contain a $OTPK_B$ because there were no more $OTPK_B$s available on the server, the shared key SK is generated in the following way:

$$\begin{aligned} DH_1 &= DH(IK_A, SPK_B) \\ DH_2 &= DH(EK_A, IK_B) \\ DH_3 &= DH(EK_A, SPK_B) \\ SK &= KDF(DH_1||DH_2||DH_3) \end{aligned}$$

If the bundle contains $OTPK_B$, the procedure is (DH_1, DH_2, DH_3 being the same as above):

$$\begin{aligned} DH_4 &= DH(EK_A, OTPK_B) \\ SK &= KDF(DH_1||DH_2||DH_3||DH_4) \end{aligned}$$

DH_1 and DH_2 provide mutual authentication, as the keys used in to derive are bound to the identities of Alice and Bob, while DH_3 and DH_4 provide forward secrecy, as EK_A and $OTPK_B$ are used only once. Figure 4 shows the calculations between keys. Alice then generates an "associated data" byte sequence AD which contains

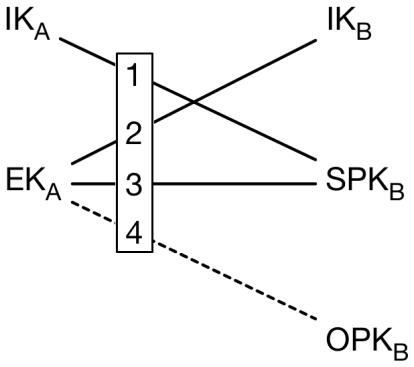


Figure 4: X3DH calculations between keys [4]

identity information for both parties: $AD = IK_A||IK_B$. Additional data may be added to AD, which uniquely identifies Alice and Bob, such as their username, phone numbers, etc. Alice then sends bob the initial message, which contains IK_A, EK_A, the identifiers of Bob's prekeys that were used by Alice, along with some initial ciphertext encrypted with an AEAD encryption scheme [33], using AD as the associated data parameter, and SK as the key. Upon receiving the message,

Bob generates SK in a similar fashion using IK_A, EK_A and his respective "prekey bundle" keys and decrypts the ciphertext.

Because the initial message is not signed by Alice using a digital signature scheme, she is able to deny sending it. This means that X3DH provides deniability, since neither Alice nor Bob publishes a cryptographic proof of the contents of their communication. In order to ensure authentication, i.e. Alice is sure that she communicates with Bob and vice versa, they may compare public keys/the additional data AD generated at both ends of the conversation manually (in person).

A malicious third party may attempt to exhaust all $OTPK_B$ from the server, which will degrade the forward secrecy of the protocol as DH_4 will never be generated. The server should have some countermeasures in place to prevent this.

Using SK to encrypt all messages exchanged between Alice and Bob is not ideal, as if SK is revealed to Eve at some point in time, all previous and future messages will be revealed. Therefore, the inclusion of other means is needed to provide forward and backward secrecy in the long term - the Double Ratchet algorithm.

2.5.2 The Double Ratchet algorithm

The Double Ratchet algorithm [5] can be used by two parties to exchange encrypted messages based on a secret key established previously between them. It allows the parties to derive new keys for each message sent in the conversation in a way that does not allow earlier keys to be calculated from later ones. In addition, by attaching new ECDH keys to their messages and using them when generating a new keys, this protocol prevents valid new keys to be generated from old ones, in case old keys are leaked to a malicious third party. By doing so, the Double Ratchet algorithm provides both forward and backward secrecy.

KDF chains are extensively used in the algorithm. They make use of key derivation functions (in this case HMAC[9] [34] and HDKF[32]). Using the KDF and a key K, they generate a byte sequence, some of which they use as an output key, and the remaining part is as a key for the next invocation of the KDF (i.e. the remaining part replaces K). Figure 5 illustrates a KDF chain ("Input" is an arbitrary parameter, which may be "null"):

KDF chains provide the following security properties:

- **Resilience** - the output keys appear random to an adversary which does not know the input keys which were used to derive them. This remains true if the adversary is in control of the KDF inputs.

[9] Although a MAC function, HMAC also fits the definition of a KDF

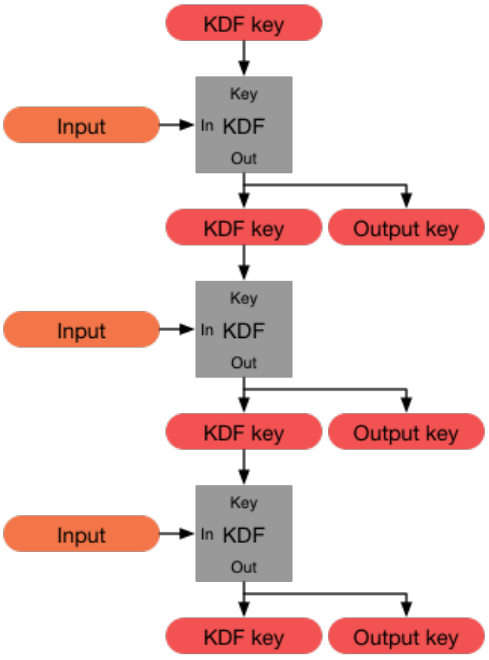


Figure 5: A KDF chain [5]

- **Forward security** - Previous output keys appear random to an adversary who learns the KDF key at some point in time.
- **Break-in recovery** - Newly generated output keys appear random to an adversary that has obtained a KDF key at some point in time. This property holds only when the future values of the "Input" parameter are unpredictable by the adversary.

When Alice and Bob establish a Double Ratchet session, each party creates three chains - **root chain**, **sending chain** and **receive chain**. Alice's sending chain matches Bob's receiving chain, and vice versa. There are 3 KDF keys (one for each chain) and both Alice and Bob have a copy of these keys.

When Alice and Bob exchange messages, they also exchange Diffie-Hellman keys, which are mixed to produce a secret, which is fed as input to the root chain. The output keys from the root chain are then used as input for the sending and receiving chains. This part of the algorithm is called the Diffie-Hellman ratchet.

The output keys from the sending and receiving chains are used to encrypt and decrypt messages, respectively. These chains advance whenever a message is sent or received. The KDF keys used in these 2 chains are called **chain keys**. This part of the algorithm is called the **symmetric key ratchet**. The KDF inputs of these chains are constant, so break-in recovery is not available. The only purpose of these chains is to ensure that each message is encrypted using a unique key, which can be omitted after it is used, providing better forward secrecy. The calculation of a new message and chain key is called a **ratchet step** in the symmetric-key ratchet. In Figure 6, two of these steps are shown.

Message keys are not used to derive other keys, and therefore they can be permanently stored without undermining the security of the algorithm. Storing previous message keys allows for successful handling of out-of-order messages, which will be introduced later in this section.

To enable break-in recovery, the KDF inputs to the sending and receiving chains must not be constant. The Double Ratchet algorithm achieves this by combining a symmetric key ratchet with a Diffie-Hellman (DH) ratchet. which updates the chain keys based on Diffie-Hellman outputs. To construct an DH ratchet, Alice and Bob each generate a Diffie-Hellman key pair that they assign as their current **ratchet key pair**. Then, every message they send has a header that contains the sender's current ratchet public key. When the receiving party sees that a new ratchet public key is received, a **DH ratchet step** is performed (the DH ratchet **"ticks"**), replacing the receiving party's current ratchet key pair with a new key pair. Thus, if Eve obtains a current ratchet private key value, this value will not be valid for long,

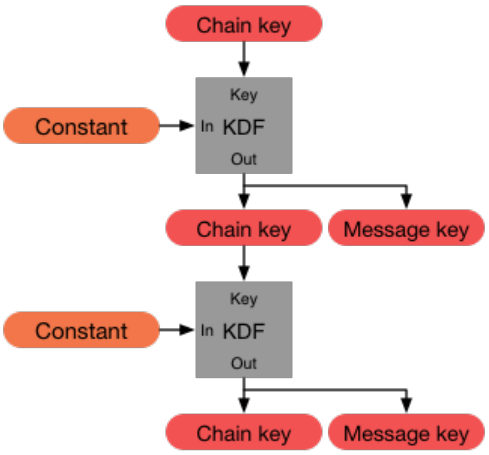


Figure 6: Two symmetric-key ratchet steps [5]

as a new private key will be generated eventually, which will again start to produce DH ratchet output unknown to her.

The Diffie-Hellman ratchet begins by Alice trying to contact Bob. Alice knows Bob's ratchet public key, but Bob doesn't yet know Alice's ratchet public key. Alice mixes her ratchet private key with Bob's public ratchet key to produce an DH output, which is shown in Figure 7.

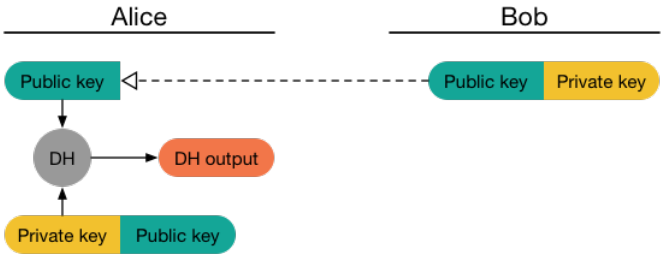


Figure 7: Initialisation of the Diffie-Hellman ratchet [5]

Alice sends an initial message that contains her ratchet public key. When Bob receives one of these messages, Bob's DH ratchet ticks, calculating the DH output between his private ratchet key and Alice's ratchet public key, which results in an

DH out which is identical to that of Alice. Bob then calculates a new DH output. This procedure is shown in Figure 8.

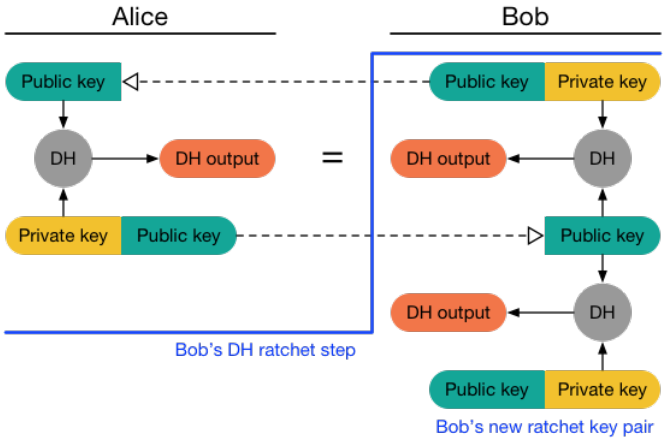


Figure 8: Bob's DH ratchet step [5]

Then, when Bob sends a message, he advertises his new ratchet public key in the header. Analogous to Bob, when Alice receives a message, she performs a DH ratchet step, which generates a new ratchet key pair for her and deriving two DH outputs - one that is identical to Bob's new DH output, and one new one, which is presented in Figure 9

The DH outputs of each tick of the DH ratchet are used to derive the send and receive chain keys. Alice uses the first DH output to generate a sending chain key. When Bob receives the initial message, he uses the first DH output to create a receiving chain key, which matches Alice's sending key, and the second to create a sending chain. With each tick of the DH ratchet, a new sending chain is introduced. A visual summary of the process is available in Figure 10.

To increase resilience and break-in recovery, instead of directly using DH outputs as chain keys, the use of a KDF chain is employed. The chain created is called the **root chain**, and the DH outputs are used as KDF inputs to it. The KDF outputs of the root chain are used as send and receive keys. In each tick of the DH ratchet, the root chain is updated twice, using the outputs from the DH ratchet. The process is illustrated in Figure 11

Incorporating the Diffie-Hellman ratchet and the symmetric key ratchet gives the Double Ratchet algorithm. When a message is sent/received, the symmetric key

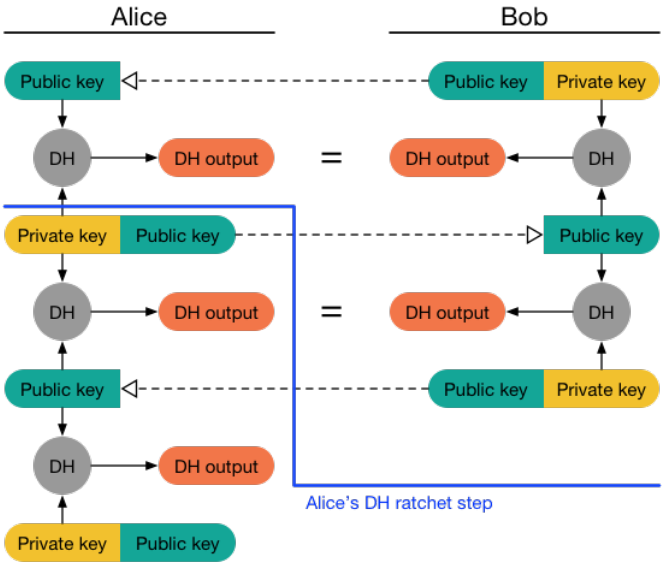


Figure 9: Alice's DH ratchet step [5]

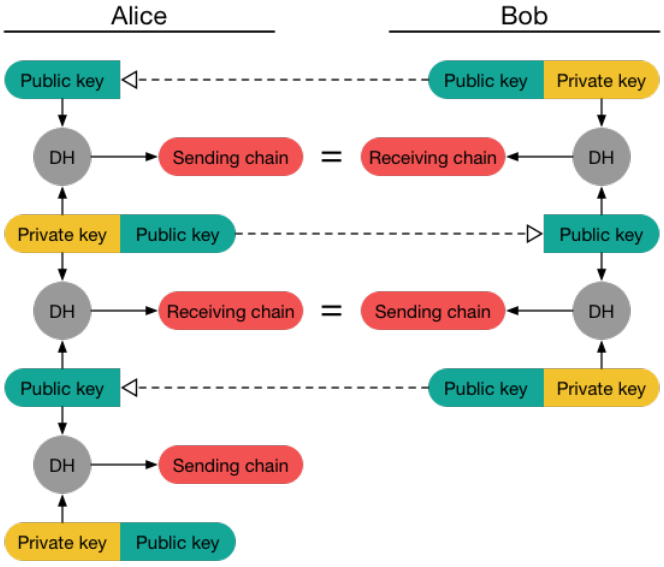


Figure 10: DH outputs as chain keys [5]

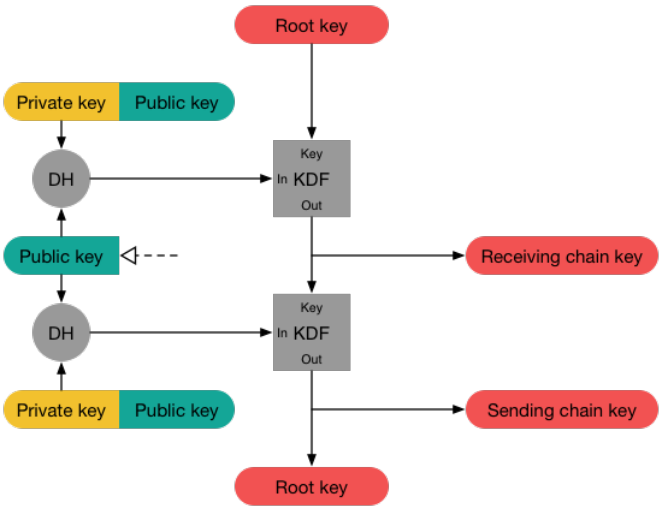


Figure 11: A full DH ratchet step with a KDF chain [5]

ratchet ticks to derive a message key. When a new ratchet public key is received, the DH ratchet ticks before the symmetric key ratchet, to produce new chain keys. When Alice tries to contact Bob, she initialises the algorithm with Bob's ratchet public key, and a shared secret SK which serves as the initial root key RK. She also generates her ratchet key pair, computes a DH output, and feeds it into the root chain to compute a new root key RK and a sending chain key CK. This is shown in Figure 12.

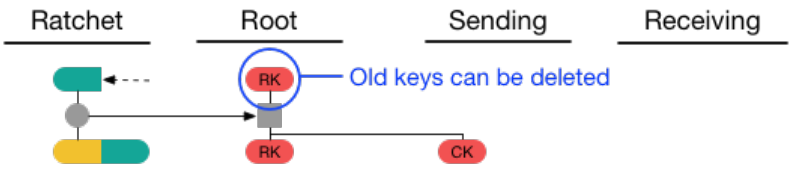


Figure 12: Initial step of the Double Ratchet algorithm [5]

When Alice sends her first message $A1$, she performs a symmetric-key ratchet step on her sending chain key, which produces a message key (labelled the same as the message - $A1$), which she uses to encrypt the message. The new chain key CK is stored and the message key $A1$ along with the old chain key can be deleted. This process is presented in Figure 13

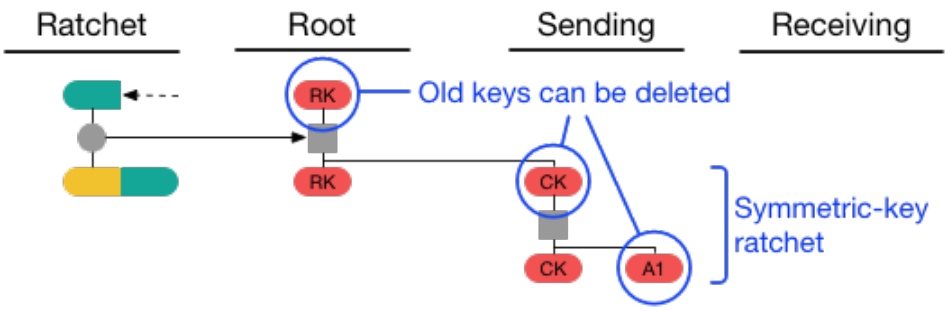


Figure 13: Double Ratchet state when Alice sends her initial message [5]

If Alice receives a message $B1$ from Bob, its header will contain his new ratchet public key $B1$. Alice performs an DH ratchet step to derive new chain keys. Then she ticks the symmetric-key ratchet of the receiving chain to get the message key to decrypt the received message - see Figure 14.

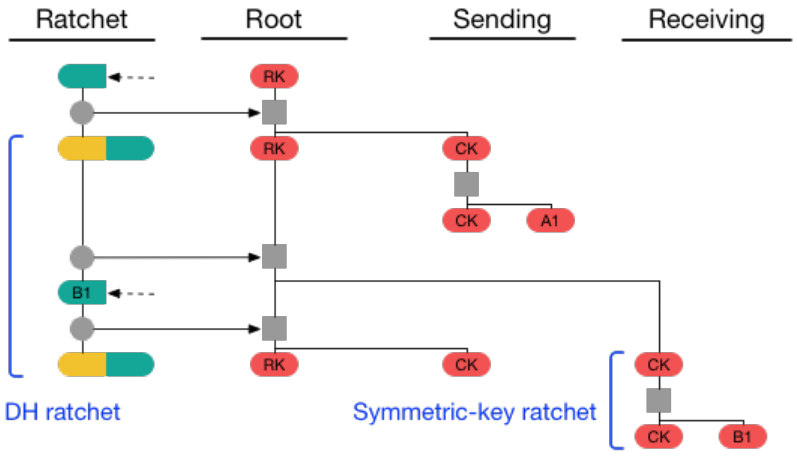


Figure 14: Double Ratchet state when Alice receives a message from Bob [5]

Then, if Alice sends a message $A2$, receives another message $B2$ which has the same ratchet public key in its header as $B1$, then sends messages $A3$ and $A4$, Alice's sending chain will perform three ticks, and her receiving one will ratchet once - see Figure 15.

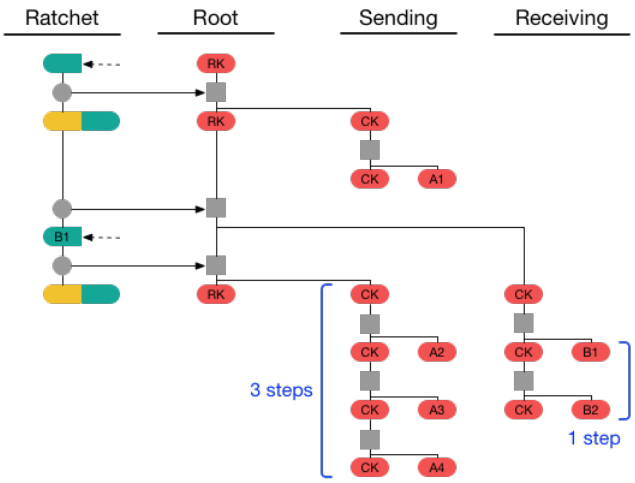


Figure 15: Double Ratchet state when Alice receives a message with Bob's ratchet key not being changed, and sending some messages after that [5]

To continue the example, if Alice receives messages $B3$ and $B4$ with Bob's new ratchet public key and then sends a message $A5$, Alice's state will be as shown in Figure 16.

It is possible that a message is lost or delayed during transit. To handle such messages, the Double Ratchet algorithm keeps track of how many messages are sent in the current and previous chains. In the header of each message, a number N is appended, which represents the message's number in the sending chain, as well as a number PN, representing the number of message keys in the previous sending chain. This enables the recipient to skip over to the relevant message key, while storing skipped message keys, in case the delayed message arrives at some point in time. When a message is received and an DH ratchet is triggered, the number of skipped messages is equal to the received PN minus the length of the current receiving chain. The received N then represents the number of skipped messages in the new receiving chain, which was created after the tick of the DH ratchet. Respectively, if a DH ratchet step is not performed, the number of skipped messages in the chain is received number N minus the length of the receiving chain. In Figure 17, an example is provided (a continuation of the state presented in Figure 16, but with $B2$ and $B3$ delayed in transit). Message $B4$ triggers Alice's DH ratchet instead of $B3$. Message $B4$ contains $PN = 2$ and $N = 1$ in its header. When Alice receives $B4$, she will have a receiving chain length of 1, because she only received $B1$ previously, so she skips over to the message key $B4$, storing $B2$ and $B3$ so that the respective messages can

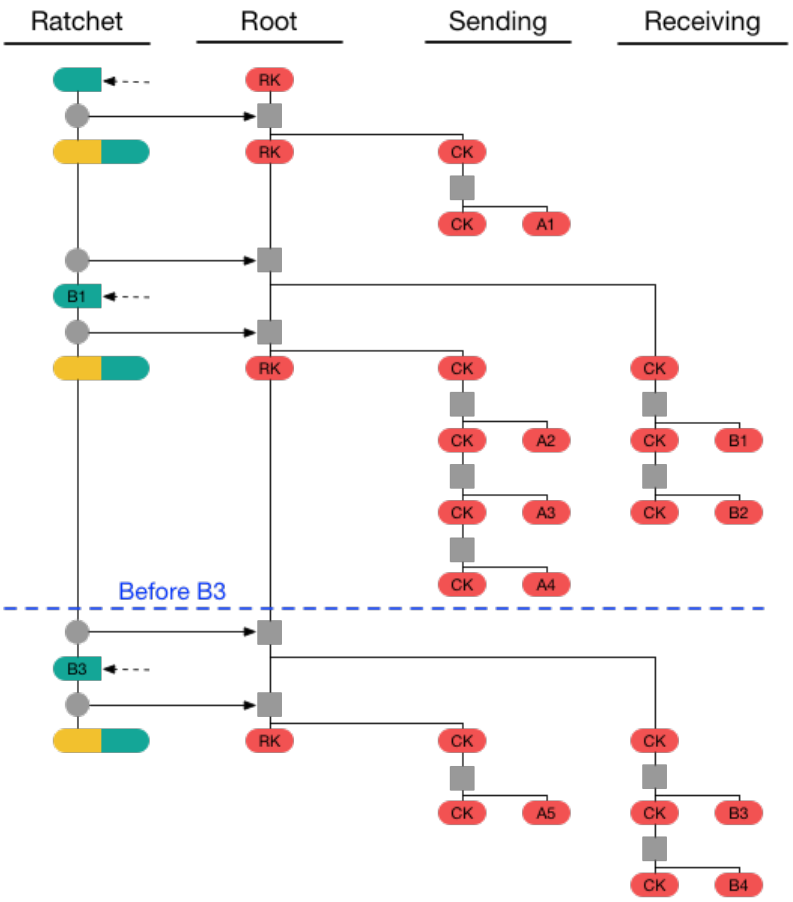


Figure 16: Alice's final Double Ratchet state [5]

be decrypted if they arrive later.

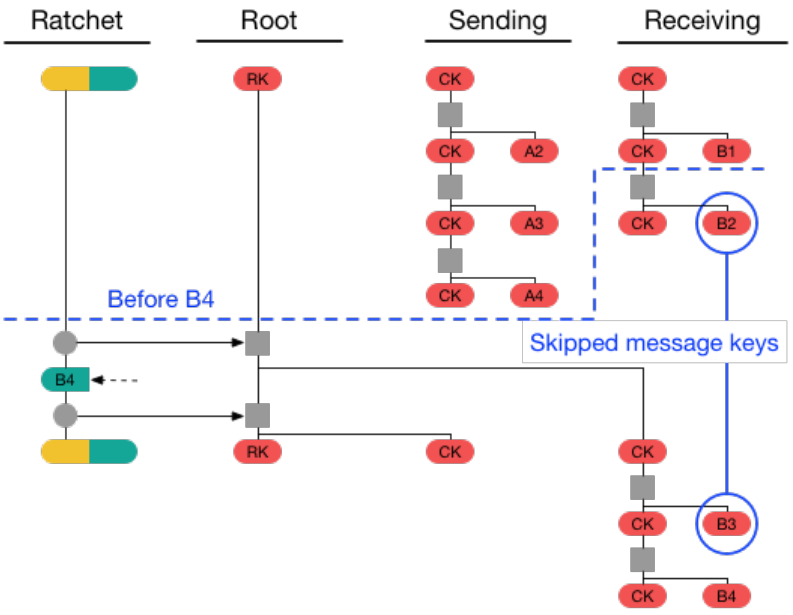


Figure 17: Out of order message keys [5]

The relevant variables (keys, chain lengths, etc.) are stored in a **state** variable. There exists a state variable for each session (conversation between two parties). A code implementation of the algorithm along with details about the variables that are stored in the state variable are available in [5, pp. 19–21].

The cryptographic primitives used in the algorithm, as presented in [5, pp. 30], are:

- **Elliptic curve Diffie-Hellman** [17] for generation and mixing of Diffie-Hellman keys
- **HKDF** [32] **with SHA-256 or SHA-512** [35] for generating keys in the root chain
- **HMAC** [34] **with SHA-256 or SHA-512** for generating message keys in the sending and receiving chains
- **AES-CBC with HMAC and HKDF** for encryption and decryption of messages (with some additional data AD). These 3 cryptographic primitives are used to construct an AEAD [33] encryption scheme, which provides integrity and authentication in addition to confidentiality.

There is a version of the Double Ratchet algorithm that introduces header encryption, to prevent third parties from learning the PN and N of messages sent in a session. However, this will increase the scope and complexity of the project, while not providing crucial additional security properties and is therefore not considered in this report (due to time constraints).

2.5.3 Incorporating the X3DH protocol with the Double Ratchet algorithm to produce the Signal protocol

The outputs from the X3DH protocol are used by the Double Ratchet algorithm to form the Signal protocol as follows:

- The SK output from X3DH becomes the root key RK input in the Double Ratchet initialisation
- The AD output from X3DH becomes the AD input to Double Ratchet encryption and decryption
- Bob's signed prekey (SPK_B) from X3DH becomes Bob's initial ratchet public key, which Alice uses to initiate her Double Ratchet

The "initial ciphertext" for X3DH will be any message encrypted using a key from Alice's initial sending chain. Alice will send her initialisation parameters - public key from the EK_A key pair and her public identity key IK_A - in the header of every message she sends until she receives a response from Bob. This is done so that Bob can generate the same secrets in the event that any of the initial or subsequent messages are lost during transit.

The Double Ratchet algorithm provides forward secrecy and good backward secrecy. Backward (Future) secrecy is achieved only when the Diffie-Hellman ratchet ticks, as when it does not, the KDF input to the receiving and sending chain is effectively a constant, and if an adversary obtains a key from the symmetric-key chain at time t, they will be able to derive all the keys for all messages sent after time t, until the Diffie-Hellman ratchet ticks. So, we say that the Double Ratchet algorithm achieves forward secrecy only in the root chain.

Because SK is derived from authenticated keys, there exists an implicit authenticity for message keys derived in the Double Ratchet algorithm. Therefore, neither Alice nor Bob have to sign their messages to prove their identity to the other party. This also allows for deniability, as in theory both Alice and Bob can produce the same key to encrypt a message - there is no cryptographic proof which one of them did it.

2.6 Related work

In [36], key encapsulation mechanisms are explored as a possible replacement for vulnerable key exchanges. Although possible, this method is impractical and/or less secure when it comes to a post-quantum version of X3DH, as in an offline setting, the receiving party must have published a one-time prekey which was encapsulated with the sending party's public encapsulation key. It is in fact impossible because when a newly registered sending party wants to send a message to an offline receiving party, as the receiving party cannot generate a one-time prekey for each sending party that is created in the future. It may be possible for the receiving party to submit all of the $OTPK$s encapsulated with the server's public encapsulation key, the server will then decapsulate and store them in their raw form and when a sending party requests a "prekey bundle", the server will encapsulate the bundle with the sending party's public encapsulation key. However, this introduces a central point of failure, because in the event the server is compromised, the raw keys will be leaked completely breaking security.

The paper also does not consider signature schemes, which are also broken by quantum computing. Due to the specific way the signature scheme of Signal (XEdDSA/VXEdDSA [27]) is used, the developed post-quantum alternatives of the protocol are in large part unrealistic. Naturally, because of the lack of signatures, these versions also inherently lack authentication, which is one of the most important properties of a secure messaging protocol.

In contrast, this report provides an independently developed version of the protocol, which can run in an offline setting and provides authentication.

3 Specification, design and implementation

3.1 Assumptions

The protocol is assumed to be running over an HTTPS connection. It is assumed that a client can verify the identity of the server. Malicious third parties are assumed to not have administrative access to the server.

3.2 Attacker model

To better imagine security goals, an attacker model is introduced, that is, what a malicious attacker is capable of and will attempt, in order to break the security of the protocol.

The attacker owns a true quantum computer, with a sufficient number of qubits to

execute a Grover search [22] [37] and run Shor's algorithm [21], so is able to break any cryptosystem based on integer factorisation/discrete logarithms. The attacker is not able to impersonate a server and trick a client(user) to connect to it. The attacker cannot impersonate a server's identity. The attacker is unable to decrypt HTTPS traffic in real time due to the fact that ephemeral ECDH keys are used and computing the private key of every key pair takes a long (but still relatively short compared to traditional computers) amount of time, as estimated in [38]. The attacker is able to capture HTTPS traffic and decrypt it later. The attacker is not able to prevent communication from server to client and from client to server(drop packets). The attacker is capable of executing cross-site scripting attacks.

3.3 Identifying vulnerable cryptographic primitives in the Signal protocol and quantum-resistant replacements for them

The Signal protocol makes extensive use of elliptic curve cryptography and it can be seen in Figure 3 that such cryptographic schemes are completely "broken" by Shor's algorithm [21], so they need replacing, but other primitives, such as key derivation functions, message authentication codes and symmetric encryption schemes only need an increase in key size to thwart a Grover search. To identify what replacements are needed, we examine the X3DH key agreement protocol and the Double Ratchet algorithm.

3.3.1 Analysis of Double Ratchet algorithm

The only public key scheme used in the Double Ratchet algorithm is the elliptic curve Diffie-Hellman algorithm, for generating and mixing ratchet key pairs in the Difie-Hellman ratchet. A post-quantum version of the Diffie-Hellman key exchange has already been developed to serve as a plug-and-play replacement for non-quantum-safe DH variants - the Supersingular isogeny Diffie–Hellman (SIDH) [39, 40]. So, to make the Double Ratchet algorithm quantum-resistant, the following steps are needed:

1. Replace ECDH with SIDH in the Diffie-Hellman ratchet
2. Increase the key size of the HMAC, HKDF and AES-CBC functions, as well as the hash output size of the hash function (SHA) used as a building block in HMAC and HKDF.

3.3.2 Analysis of X3DH key agreement protocol

In the X3DH protocol, a key derivation function is used and, as before, increasing its key size is enough to make it quantum safe. In the key exchange procedure, two public key cryptographic schemes are used - ECDH for Diffie-Hellman key generation/shared secret derivation, and XEdDSA/VXEdDSA [27] for creating and verifying signatures. Attempting to replace ECDH here with SIDH creates a problem. XEdDSA/VXEdDSA was specifically developed to handle digital signature operations with ECDH keys; however, a post-quantum alternative to XEdDSA/VXEdDSA that can use SIDH keys in the same manner is not available at the time of writing. Therefore, it is impossible to create a completely analogous quantum-safe version of X3DH. To overcome this obstacle, a quantum-resistant alternative for the X3DH key exchange was devised and is described in subsection 3.4.

3.4 Post-quantum alternative for the X3DH key agreement protocol

The X3DH key agreement protocol provides a way for parties to mutually authenticate each other and establish implicitly authenticated keys. This protocol is an extension of the triple Diffie-Hellman protocol presented in [41] and adds the ability to run in a setting when one party is offline while preserving the security properties. During this project, multiple attempts were made to achieve the same using post-quantum primitives, but it was found that triple Diffie-Hellman has requirements which cannot be achieved due to the fact that the triple Diffie-Hellman specification is strictly based around the properties of the discrete logarithm problem in regular elliptic curves, which is quantum-solvable.

Exploring further, triple Diffie-Hellman is based on the key agreement "Protocol 4" (also known as "Double Diffie-Hellman") described in [6]. As mentioned in [41], Protocol 4 has been conjectured by its authors to be secure, but it was never proven secure or insecure, mostly because it was defined at a relatively abstract level, rather than in a more detailed manner, such as triple Diffie-Hellman. However, because it is abstractly defined, it provides the possibility of it being implemented with quantum-safe primitives.

Protocol 4 is presented in Figure 18. The notation is as follows (adapted from [6]):

- i is the identity of the sender
- j is the identity of the intended recipient
- k is the agreed key

- P is a pair $P = (\Pi, \mathcal{G})$, where Π specifies how honest players behave, and $\mathcal{G}$ generates key pairs for each entity
- $\mathcal{H}$ is a hash function (serves the purpose of a key derivation function)
- α^{XY} is the result of mixing Diffie-Hellman keys X and Y
- S_p and R_p are respectively a party p's sending and receiving Diffie-Hellman keys. S_p's are tied to the identity of p, whereas R_ps are ephemeral and are generated whenever a party tries to contact the other party.

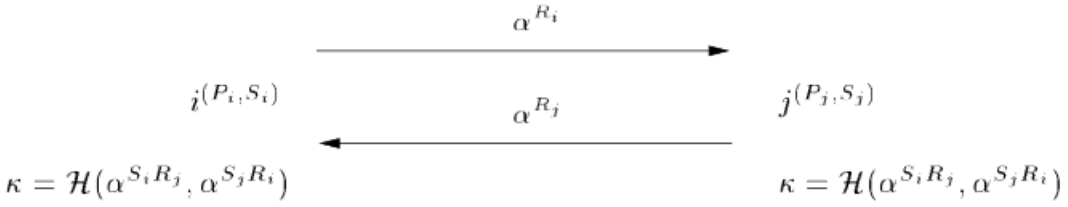


Figure 18: Protocol 4 [6]

If we assume that i is Alice and j is Bob with their respective identity keys. We can say that S_j is equivalent to Bob's SPK_B from X3DH, as it is also bound to his identity, because it is signed. Furthermore, we can consider R_j to be one of Bob's one-time prekeys ($OTPK_B$), as it is ephemeral too (it is deleted after it is used). Alice's keys, S_i and R_i (referred to from now on as S_A and R_A) will both be ephemeral (equivalent to EK_A, but split into 2 to accommodate the specification of Protocol 4), however, they also need to be authenticated, so that Bob can be sure that the party he performed the key agreement with is truly Alice.

Some quantum-safe signature scheme must be used for signing and verifying the keys. The hash-based signature scheme SPHINCS [42] is recommended, as it is provably quantum-resistant. A downside of using is the fact that it is able to sign 2^{50} messages, after which forging signatures becomes highly possible, though 2^{50} is a large number, and a party would most certainly not sign that many messages in its lifetime. For Diffie-Hellman key generation and secret derivation, the only post-quantum variant available at the time of writing is SIDH.

First, Bob will generate a digital signature key pair IK_B. Bob will also generate a set of $OTPK_B$s and one SPK_B. He will sign all of them with his signature private key and assign them an identification number. Bob will then publish his public signature key IK_B, his signed prekey SPK_B along with its signature S_{SPK_B}, and the set of $OTPK_B$s with their respective signatures $S_{OTPK_B^1}$, $S_{OTPK_B^1}$, etc. Alice would also have previously performed this procedure.

When Alice wants to contact Bob, she will request a pre-key bundle from the server, which contains Bob's identity key IK_B, signed pre-key SPK_B and its signature S_{SPK_B}, and a one-time pre-key $OTPK_B$ with its signature S_{OTPK_B}. Alice will verify the signatures with IK_B, and if verification fails, she will abort the execution of the protocol. Alice will generate S_A and R_A, mix them with SPK_B and $OTPK_B$ and use them as the key input to a KDF (a recommended one is HKDF) to produce the shared key SK:

$$DH_1 = DH(S_A, OTPK_B)$$
$$DH_2 = DH(R_A, SPK_B)$$
$$SK = KDF(DH_1||DH_2)$$

Alice will also compute an additional data parameter AD as in X3DH, which she uses during encryption - $AD = username_{Alice}||IK_A||IK_B||username_{Bob}$.

Alice will then send an initial message encrypted with SK and include the public keys of S_A and R_A in its header, attaching the identifiers of which SPK_B and $OTPK_B$ she used. Alice will also sign the initial message with her identity key IK_A. Upon receiving the initial message, Bob will retrieve Alice's identity key IK_A from the server and use it to verify that the message and, therefore, the keys S_A and R_A in its header are authentic. Bob will then perform an analogous calculation to Alice's to derive SK:

$$DH_1 = DH(OTPK_B, S_A)$$
$$DH_2 = DH(SPK_B, R_A)$$
$$SK = KDF(DH_1||DH_2)$$

He will also compute AD in the same way as Alice. At the end of the protocol, both parties will have established the same authenticated shared secret SK and the additional data parameter AD.

This protocol will not support a key exchange when there are 0 of Bob's $OTPK$s on the server, as this will lead the protocol to use only SPK_B to derive a shared secret. Using only SPK_B could hinder forward secrecy, since SPK_B is reused and known to all parties(malicious or not).

The signing of all Diffie-Hellman keys is done so that the generated secrets DH_1 and DH_2 are fully authenticated, as in X3DH, DH_1 and DH_2 are also authenticated. This is done so that in SK, the same $n - bit$ security level is achieved with regard to authenticity (i.e. the amount of authenticated bits used to derive SK is the same in both protocols). However, this makes achieving deniability in this key agreement protocol impossible, as Alice has to sign her initial messages with her identity key. In short, this protocol sacrifices deniability for better authenticity and forward secrecy. This protocol can be executed even when Bob is offline.

As this protocol is an interpretation of Protocol 4, it may be possible to prove its security, however, this requires further work, which could not fit in the time frame of this project.

3.5 The post-quantum Signal protocol and the messenger application

The protocol presented in subsection 3.4 is integrated with the post-quantum Double Ratchet algorithm in the same way as X3DH is integrated with the traditional Double Ratchet algorithm - the output SK from the key agreement protocol is used as the initial root key RK, and SPK_B is used as Bob's initial ratcheting public key. Alice generates a new ratcheting key pair which she combines with SPK_B to produce her sending chain. She attaches it along with S_A and R_A to the header of every message she sends to Bob and signs it with her identity key IK_A, so that Bob can confirm that Alice's initial keys are authentic. Alice continues to attach these values to her messages until she receives a response from Bob. When she does, she can stop attaching S_A and R_A, and also stop signing her messages, as all later keys will be implicitly authenticated.

The cryptographic primitives used in the implementation of the post-quantum Signal protocol in the messenger application will be:

- **SIDHp751** for Diffie-Hellman key operations. It provides shared secrets with a 128-bit quantum security level; public keys are 1152 bytes, and private keys are 1248 bytes long.
- **SPHINCS** based on the BLAKE-256 hash function [43]. It provides signatures with 128-bit quantum security level; public keys are 1056 bytes, and private keys are 1088 bytes long. The signatures generated by the function have a size of 41 000 bytes (≈41 kilobytes).
- Traditional cryptographic primitives for key derivation and encryption, as described in [5, pp. 30]. Their key size is increased to make them at least at a 128-bit security level. Taking into consideration the Grover search [22], which effectively halves the security level of black-box functions such black-box functions (because $\sqrt{2^n} = 2^{\frac{n}{2}}$), to achieve 128-bit quantum security, the following is needed:
  - 256 bits for **AES-CBC** symmetric keys
  - At least SHA-256 as the hash function used in **HMAC** and **HKDF**. Because of the way **HMAC** and **HKDF** use the hash function, it is possible that their security level is lower than the security level of the

hash function. To compensate for that, SHA-512 will be used instead.

The implementations of SIDH and SPHINCS used in the application are provided by a third-party JavaScript library, and the implementation of traditional cryptography functions (AES-CBC, HKDF, HMAC) are provided by the Web Crypto API [10] available in modern browsers. A full list of all code libraries used in the messenger application is available in Appendix B.

The Double Ratchet protocol is implemented in the application as described in [5, pp. 19–21]. The renewal time for signed prekeys (SPKs) is set to 2 days, as in the original Signal protocol[11]. The number of one-time prekeys will be set to 11 instead of the traditional 100[12] to adjust for storage, as SIDH keys are significantly larger in size compared to ECDH keys. The server will require a party to replenish its $OTPK$s when there are less than 5 of its $OTPK$s left on the server. The server administrator will be able to tune these values.

3.6 Preventing side-channel attacks in the messenger application

It is possible that the security of cryptographic schemes is undermined by some flaw in the application itself. In this section, 3 such flaws are identified, and countermeasures are suggested.

3.6.1 Replay attacks

It is possible that an attacker captures a message packet which Alice sends to Bob. The attacker could later attempt to resend this message to Bob. The server will forward the message to Bob and, depending on timing, Bob would successfully process the message, as it will have a valid structure and bear a valid signature. This could result in Bob's Double Ratchet state with Alice becoming corrupted.

To prevent such replay attacks, a challenge-response authentication scheme [44, pp. 198–199] may be used. Each time Alice wants to send a message, the server will provide her with a challenge in the form of a random byte sequence. Alice will append this sequence to the message packet and sign the whole packet with her identity key IK_A. The server keeps track of challenges served to Alice and has set some expiration time to them to prevent the server's storage from being cluttered

[10] https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API

[11] https://github.com/signalapp/SignalServiceKit/blob/5c3b30104a7a642d7ff70250e0b81ca645938dec/src/Account/TSPreKeyManager.m#L20

[12] https://github.com/signalapp/SignalServiceKit/blob/5c3b30104a7a642d7ff70250e0b81ca645938dec/src/Account/TSPreKeyManager.m#L28

by unused challenges. Upon receiving the message packet, the server will check whether its signature is valid, and if so, it will forward the message to Bob and mark the challenge byte sequence as used and delete it from its memory. If not, the server will omit the message. If Eve attempts to replay the packet, the server will detect that the challenge is already used and will not pass the verification check. The inclusion of such a random challenge variable in each message packet prevents a malicious third party from predicting the message signature even if the message packets themselves are the same.

3.6.2 Identity key swap attacks

It is possible that a malicious third party, Eve, tricks a party into thinking that her identity key, IK_E, is the other party's identity key. This would allow Eve to further trick the party into thinking that a pre-key bundle PKB_E she created actually belongs to the other party. So signature checks will pass in the key agreement protocol, resulting in the shared secret being shared not between Alice and Bob but between Alice and Eve. This enables Eve to carry out a Man-in-the-Middle attack.

To be sure that the identity key truly belongs to relevant party, Alice and Bob can choose to meet in person an compare their keys and ensure that they are the same at both ends of the conversation. In the current implementation, the SPHINCS public keys are 1056 bytes long. Assuming an ASCII encoding, each key will be 1056 characters long, which will be infeasible to compare. To allow humans to compare such long strings easily, a hash-visualisation scheme [45] can be used. The particular scheme used in the application is the "The drunken bishop" algorithm [46], which is typically used in OpenSSH[13] for generating "randomart" - visualising generated keys. The implementation of this algorithm in the messenger application is provided by a third-party library. The data visualised by the algorithm is a SHA-512 hash of the additional data AD.

3.6.3 Cross-site scripting attacks

Information for a user's account will be stored only on the user's browser, so the user's account is unlikely to be compromised if the server is breached by a malicious party. However, it may be possible that an adversary can retrieve the user's account information by sending a malicious message. Such a malicious message will contain an HTML script tag, in which JavaScript code is present that reads the user's account info and sends it to the malicious party.

This attack can be easily hindered - the messenger client will replace $<$ and $>$ in messages with their relative HTML entity codes (&lt; and &gt;) before appending

[13] https://www.openssh.com/

them to the page's HTML, thus making the HTML parser not recognise the message as containing valid HTML tags. In the application, an HTML template engine is used, which automatically performs this procedure.

3.7 Software architecture

3.7.1 Design

As the proof of concept application will be web-based, a user interface needs to be developed. Caution should be taken, because some implementations of graphical interfaces lead to strongly coupled components [47], which make a project difficult to extend, maintain and conduct unit testing on. To prevent such implementation, the use of the model-view-controller (MVC) architecture [7] (Figure 19) will be employed, allowing for the separation of presentation logic from the main application logic, achieving stronger cohesion and looser coupling. Furthermore, the data layer can be separated from the main logic, which benefits testability greatly - for an example, sets of data for which the output from the component is known can be created and injected in said component via mock objects. The unit test will then verify that the output of the test matches the expected output. In general, this leads to faster development times, as the time for unit testing is shortened. The data layer in this application is mainly tasked with retrieving and storing information to a database and storing temporary (run-time) data. The data layer can also be effortlessly extended to handle retrieving data from third-party APIs (e.g. an email verifier service) if need be. To connect to the database and perform operations on the stored data, data transfer objects (DTOs) are used, which are provided by an object document mapper (ODM) library.

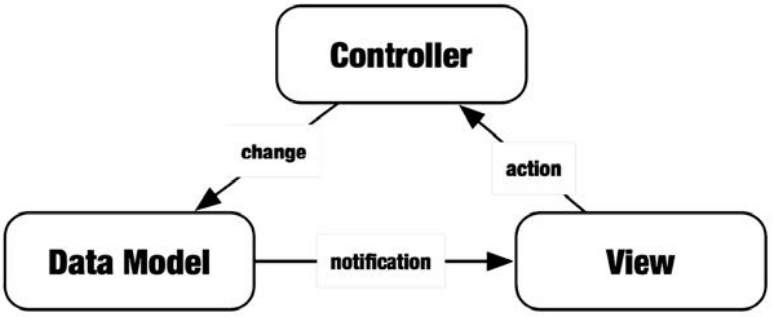


Figure 19: Model-view-controller architecture [7]

The architecture also aims to fulfil the principle of dependency inversion [48] and implements a version of the dependency injection design pattern [49], to allow even looser coupling between components. Given the scale of the project, usage of more

design patterns may lead to an "over-architecture", which would slow down development times, while not having any greater benefits. For example, a further layer of abstraction between models and the means of storing them persistently is possible by introducing a repository pattern [50], but this would be largely unnecessary given that the chosen language to develop the application in, JavaScript, is weakly typed. If it is to be developed further, other design patterns may be considered, but, in general, the structure of the application will closely follow the SOLID principle set [51].

3.7.2 Implementation

The implementation of the MVC architecture in Node.js (with Express[14] used as the web framework) is as follows: Model (DTO) schemas are dynamically loaded into a models module. This module is used by the data loader after establishing a connection to the database to create the actual DTO objects. The data loader then passes these DTOs to the individual data modules, simultaneously loading them into a single data module (data layer), which the controllers use to manipulate the models. The controller loader loads all controllers into a module. Each controller is responsible for parsing the information passed on to it and presenting a view (which is not necessarily only a web page with user interface, but may be a view of some data, e.g. a binary stream, JSON string, etc.). The routes are responsible for defining the routes(URL paths) the application can be accessed at and mapping them to a controller. All routes are loaded into the Express application by the routes loader. The architecture is shown in Figure 20.

Git was used as the version control system, and Cardiff University's GitLab instance was used as the remote git repository manager. Unit testing in the application has been set up along with GitLab Continuous Integration (CI), which runs the unit test when a new commit is pushed to the remote repository. However, the application was not unit tested extensively due to the time constraints of this project.

[14] https://expressjs.com/

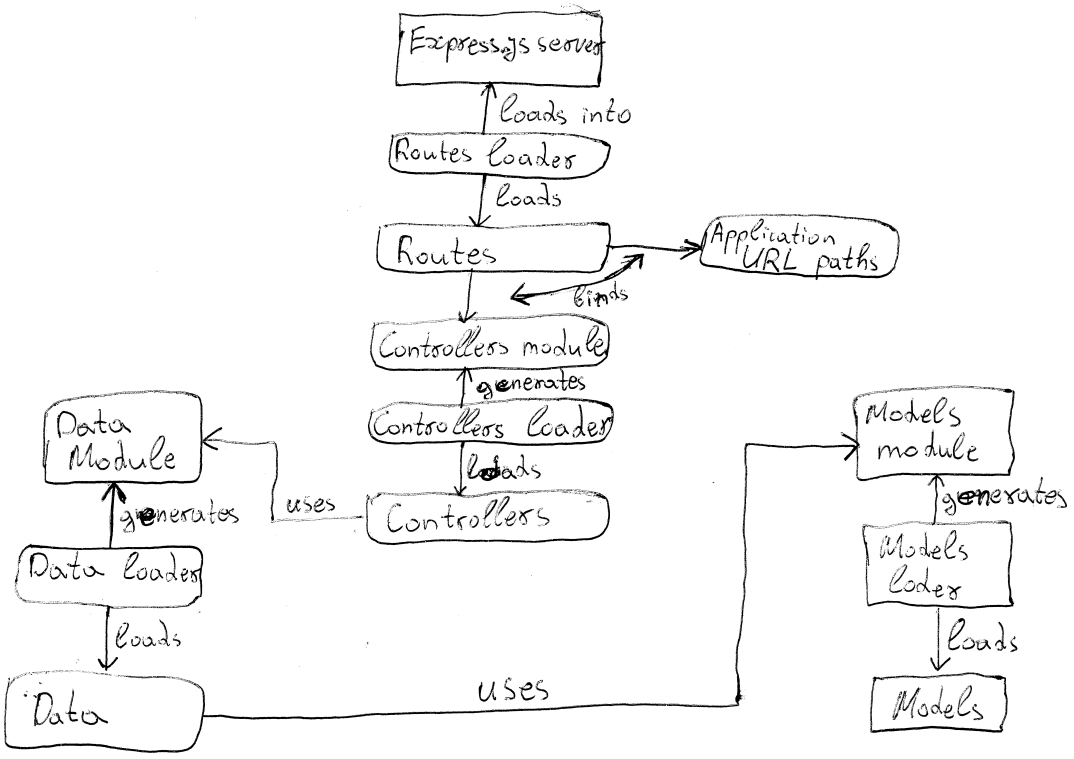


Figure 20: Implementation of the architecture

4 Results and evaluation

4.1 Security properties of the devised quantum-resistant secure instant messaging protocol

Because the structure of the Double Ratchet algorithm is essentially not changed, the post-quantum version of the protocol achieves the same properties as the traditional version - forward secrecy, backward secrecy and deniability.

As mentioned in the specification for the replacement for X3DH (subsection 3.4), the devised key agreement protocol does not meet the deniability requirement. However, after Alice's initial message is processed by Bob, future messages have the deniability property, as it is guaranteed by the Double Ratchet protocol. This applies to the raw version of the post-quantum Signal protocol.

When we take into consideration the challenge-response scheme used to prevent replay attacks, it is apparent that deniability can not be achieved, as each message is signed with the sending party's identity key - there is cryptographic proof that specifically that party sends that message.

In summary, the raw version of the post-quantum Signal protocol provides the same security properties as the traditional version, with the exception of deniability, which is only partial. The version implemented in the developed messenger application will completely lack deniability, as it is sacrificed in favour of protection against side-channel attacks.

4.2 Messenger application capabilities

The developed application is capable of handling user registration(only one user account can be active per browser), user discovery (users are able to find and initiate conversations with other users), exchanging end-to-end encrypted messages between users, user profile exporting and importing (used to transfer user accounts between browsers), and user identity key verification. The application also handles the replenishing of a user's $OTPK$s and SPKs. A file listing of the application, along with a summary of the purpose of each file, is available in Appendix A.

Two versions of the application are supplied: one that uses a single thread for cryptographic operations and one that uses multiple threads. These are released are two independent versions, as the multi-threaded version may have issues related to concurrency control, and the single-threaded version is therefore considered more stable.

Guides on how to configure and use the application are available in Appendix C and

Appendix D respectively.

The application was developed to run on and is proven to work in the web browsers Google Chrome and Mozilla Firefox.

4.3 Performance testing

4.3.1 Methodology

The performance of the application was tested in two environments: desktop and mobile. While primarily developed for the desktop, the application is capable of being used on mobile devices, due to the fact that it is a web-based application. The focus is on desktop performance, but measuring performance on mobile provides a good insight into how post-quantum cryptography will affect mobile devices, as they are often used for instant messaging. Furthermore, only time of execution, i.e. the speed, will be measured, as this is property that likely most users will be interested in.

The desktop machine that was used to benchmark the application has the following configuration:

- **CPU** - Intel(R) Core(TM) i7-8750H, 6 cores / 12 threads running at 3.9GHz
- **RAM** - Single Channel 16GB DDR4 running at 2666MHz
- **Operating System** - Windows 10, Version 10.0.18363 Build 18363

The mobile device used to benchmark the application has the following configuration:

- **CPU** - Qualcomm Snapdragon 820, 4 cores / 4 threads, two running at 2.2GHz and two at 1.6GHz
- **RAM** - 6 GB LPDDR4
- **Operating System** - Android 9.0

The browser used to run the benchmark in was selected to be Google Chrome, as it is, at the time of writing, the most widely used browser, which means that most users are likely to experience the same results as the ones shown here. The version of Chrome that was used is 83.0.4103.97 for the desktop and 83.0.4103.96 for the mobile.

The message size used for the message encryption/decryption benchmark was set to 60 characters. This value was chosen, as it should be close to the average length of a text message [52].

The benchmark was run for multiple iterations, and the average was taken to negate the impact of any system anomalies that occurred while it was running. On the desktop, the benchmarks were run for 1000 iterations in the single-threaded setting and for 100 in the multi-threaded setting. On mobile devices, 10 iterations were performed in the single-threaded setting and 5 in the multi-threaded. The reason that the number of iterations on mobile is so low is to prevent the impact of thermal throttling, which leads to reduction in CPU performance, which subsequently results in inaccurate results.

Only the parts of the application which involve post-quantum cryptography will be measured, as the increase in key size does not bear a noticeable performance impact on traditional cryptographic schemes:

- **accountGen** - user account generation, performing the post-quantum key agreement protocol
- **initEncrypt** - sending the initial message in the conversation, which involves agreeing on a shared key using the protocol specified in subsection 3.4, initiating the post-quantum double ratchet algorithm, deriving a sending chain message key, encrypting and signing the message packet.
- **initDecrypt** - receiving the initial message, verifying its signature, deriving the shared key using the keys provided in its header, deriving a receiving chain message key, and decrypting the message.
- **tickDHDecrypt** - decrypting non-initial messages, which advance the Diffie-Hellman ratchet.

4.3.2 Results

Event	Thread count											
	1	2	3	4	5	6	7	8	9	10	11	12
	Average run time in milliseconds											
accountGen	8882	4502	3523	2730	2536	2089	2058	2019	1986	1964	1952	1490
initEncrypt	2698	2130	1949	1975	1974	1993	1981	1967	1977	1958	1980	1934
initDecrypt	1740	1456	1382	1366	1442	1403	1450	1449	1395	1454	1459	1383
tickDHDecrypt	1098	1149	1138	1139	1071	1074	1075	1086	1094	1138	1096	1131

Table 2: Desktop performance metrics

In both Table 2 and Table 3 it can be seen that the time it takes to complete tasks involving post-quantum cryptographic operations takes seconds, in contrast to their traditional cryptography counterparts, which only need milliseconds. Message encryption and decryption times in both a desktop and a mobile setting are large - in the single-threaded variant, message processing may take as long as ≈ 5 seconds

Event	Thread count			
	1	2	3	4
	Average run time in milliseconds			
accountGen	38811	23658	19007	15374
initEncrypt	11922	9680	10644	10000
initDecrypt	7582	6418	6476	6789
tickDHDecrypt	4837	4839	5271	4919

Table 3: Mobile performance metrics

on a desktop and up to 20 seconds on a mobile. In general, the most often executed costly operation will be *tickDHDecrypt*, which takes ≈ 1 second on the desktop and ≈ 5 seconds on the mobile, which should be acceptable. *accountGen* will be ran only once by a user, when they generate their account, and it being relatively slow is acceptable. *initEncrypt* and *initDecrypt* will also run only once - to decrypt and encrypt the initial message, respectively, so a delay there is also acceptable.

Looking at multi-threading results, we see an improvement when increasing thread count in both desktop and mobile. In *accountGen*, adding more threads results in faster account generation times, especially on the desktop, where employing the use of 12 threads reduces the time for account generation from $8882ms$ to $1490ms$, which is a $\approx 83\%$ decrease. For message operations, a decrease is only observed in *initEncrypt* and *initDecrypt* when going from 1 to 2 threads, which gives a 10 to 20% decrease. For *tickDHDecrypt*, no improvement is observed when adding additional threads.

In general, it is observed that post-quantum cryptography is rather difficult to compute for mobile devices, judging by the large amount of time it takes to perform the measured operations. Therefore, it is certain that post-quantum cryptography cannot be applied as extensively in mobile applications, because it requires more computational power, which subsequently leads to more power usage, leading to shorter battery life. Also, it can be concluded that mobile applications that use post-quantum cryptography will suffer from significant delays.

5 Future work

5.1 Supporting audio and video telephony

As voice and video calling are popular nowadays, the protocol may also be extended to handle these forms of communication. In general, voice and video feeds are divided into **packets**. To ensure that the security properties of the protocol

are preserved during voice communication, a decision must be made on how to handle the encryption and decryption of these packets without introducing lag in the audio/video feed. As seen in subsection 4.3, it takes a long time (≈3-4 seconds, without accounting for any network delay) for each new message to be encrypted and decrypted. This makes it practically impossible for each packet to be sent in a way that ensures complete future secrecy. A potential solution may be to periodically send a packet which introduces new ratchet keys. By doing so, periodic future secrecy can be achieved, as if a malicious third party obtains the key used to encrypt a packet, this key will only be able to decrypt packets for some short time (depending on how often a ratcheting packet is sent). Furthermore, the usage of symmetric stream ciphers can be explored to speed up performance [8, pp. 30–51].

5.2 Supporting group messaging

Currently, the protocol specification and the developed application only support one-to-one messaging. However, it is common for instant messengers to support conversations between multiple parties. This may create additional challenges for the newly devised protocol, but some ideas for enabling group messaging are available in [53].

5.3 Developing a supersingular-isogeny-elliptic-curve-based digital signature that uses supersingular isogeny Diffie-Hellman keys

As was found in this project, a post-quantum version of X3DH with equivalent security properties cannot be established due to the lack of a direct post-quantum alternative for the cryptographic primitive XEdDSA/VXEdDSA [27], which allows elliptic curve Diffie-Hellman keys to be used as signature keys. However, it was identified that there is a direct replacement for ECDH, this being SIDH. Therefore, it is suggested that a supersingular-isogeny-elliptic-curve-based digital signature scheme analogous to XEdDSA/VXEdDSA should be developed, which can use SIDH keys for signing/verification. Work has been done toward creating signature schemes based on supersingular isogenies [54], but it remains to be seen whether such a scheme can be incorporated with SIDH.

6 Conclusions

This project aimed to deliver a quantum-resistant cryptographic protocol from instant messaging. In addition, a browser application was developed to demonstrate the protocol in action and to take its performance metrics.

Performance was tested in both single- and multi-threaded scenarios. It became apparent that the inclusion of post-quantum cryptography severely impacts throughput. The current version of the protocol failed to achieve deniability because every message is signed in the challenge-response scheme, which prevents replay attacks. Also, the replacement for X3DH completely lacks deniability, as the initial messages are signed by the sender, as there should be some way for the recipient to know that the sender's keys in the initial message are authentic. However, if the challenge-response scheme is removed, deniability can be achieved with the current version of the protocol. In large parts, the current state and availability of post-quantum cryptographic algorithms limits the possibility of achieving the same security properties as the original Signal protocol specification.

In general, the objectives outlined in the initial plan:

1. Research currently available secure instant messaging protocols and their implementations.
2. Research quantum-resistant cryptographic algorithms.
3. Design a quantum-resistant secure instant messaging protocol.
4. Develop a web application that implements the devised protocol.
5. Evaluate the security of the protocol.
6. Evaluate the security of the application.

were achieved to their full extent, with the exception of objective 5 and 6, as they were omitted in large parts to adjust for the shortened time frame of the project.

The developed application provides the ability for users to register, communicate (with their messages being encrypted end-to-end and protected from quantum computing attacks), verify their identities, and transfer their accounts to other browsers. Although functional, the application was only created for demonstration purposes, and its security is not fully analysed, so it can be improved in the future.

Appendix A File listing

File	Purpose
src/app.js	Application server entry point; Application server setup
src/public/css/messenger.css[15]	Style sheet for the messenger web application
src/public/html/messenger.html[15]	HTML skeleton for the messenger web application
src/public/images/avatar.png[16]	Default user profile picture
src/public/images/favicon.ico	Favicon for the application
src/public/js/account-transferring.js	Logic for importing and exporting user accounts in the messenger web application
src/public/js/benchmarks/account-gen.js	Benchmark for user account generation
src/public/js/benchmarks/message-ops.js	Benchmark for message operations (sending/receiving messages and key exchanges)
src/public/js/benchmarks/utils.js	Statistics utilities
src/public/js/constants.js	Constants for the messenger web application
src/public/js/crypto-helper.js	Wrapper/abstraction for the cryptographic primitives provided by the WebCryptoAPI, SPHINCS and SIDH libraries, along with utilities for work with JavaScript ArrayBuffers and typed arrays in the browser
src/public/js/db-init.js	Initialisation file for the in-browser database that is used to store user's account

[15] Adapted from https://bootsnipp.com/snippets/1ea0N

[16] https://vectorified.com/avatar-icon-png#avatar-icon-png-1.png

src/public/js/helpers.js	Various client-side helper functions that define commonly used procedures/code snippets, helping eliminate repetation in code
src/public/js/main.js	Controls whether the register page or the main application page is loaded
src/public/js/messenger.js	Client side logic for the messenger application
src/public/js/ratcheting.js	Implementation of the Signal Double Ratchet
src/public/js/register.js	Client side logic for registering an user account
src/public/js/workers/sidh-worker.js	WebWorker that offloads the SIDH library's computations from the main thread to a secondary thread, in order to prevent the browser from freezing
src/public/js/workers/sphincs-worker.js	WebWorker that offloads the SPHINCS library's computations from the main thread to a secondary thread, in order to prevent the browser from freezing
src/server/config/app/index.js	Bootstrap of the Express (server) application
src/server/config/app/post-routes-config.js	Error handler for the Express (server) application
src/server/config/constants.js	Constants for the messenger web server
src/server/config/index.js	Configuration variables for the messenger web server
src/server/config/socket/index.js	Web socket configuration for the messenger web server

src/server/controllers/controllers-loader.js	Finds and loads all controllers into the "Controllers" module
src/server/controllers/index.js	Linked to *controllers-loader.js*
src/server/controllers/main/benchmark-controller.js	Controller that serves the benchmark page
src/server/controllers/main/home-controller.js	Controller that serves the home page
src/server/controllers/messaging/messaging-controller.js	Controller that handles sending/retrieving messages
src/server/controllers/security/challenge-controller.js	Controller that handles the challenge-response functionality
src/server/controllers/user/user-controller.js	Controller that handles operations on user accounts - registering, SPK and OTPK updates, username querying
src/server/data/data-loader.js	Finds and loads all data management functions into the "Data" module; Bootstraps Mongoose Models and initiates connection to database
src/server/data/index.js	Linked to *data-loader.js*
src/server/data/messages/message-data.js	Data management functions for messages
src/server/data/security/challenge-data.js	Data management functions for challenges
src/server/data/user/socket-data.js	Data management functions for sockets, mapping socket IDs to usernames
src/server/data/user/user-data.js	Data management functions for user accounts
src/server/models/models-loader.js	Finds and loads all models into the "Models" module
src/server/models/index.js	Linked to *models-loader.js*
src/server/models/message/message-model.js	Schema definition for the "Message" model
src/server/models/security/challenge-model.js	Schema definition for the "Challenge" model

src/server/models/user/otpk-model.js	Schema definition for the "Otpk" model
src/server/models/user/user-model.js	Schema definition for the "User" model
src/server/routes/routes-loader.js	Finds and loads all routes into the Express application
src/server/routes/index.js	Linked to *routes-loader.js*
src/server/routes/main/benchmark-route.js	Routes for the benchmark page
src/server/routes/main/home-route.js	Routes for the home page
src/server/routes/rest-api/challenge-route.js	API routes for the challenge-response functionality
src/server/routes/rest-api/messaging-route.js	API routes for the messaging functionality
src/server/routes/rest-api/user-route.js	API routes for the user account management functionality
src/server/utils/crypto-utils.js	Wrapper/abstraction for the cryptographic primitives provided by the NodeJS "crypto" and SPHINCS libraries, along with helper functions
src/server/utils/date-utils.js	Utilities for working with dates in NodeJS
src/server/utils/file-system-utils.js	Utilities for working with the filesystem in NodeJS
src/server/utils/scheduled-tasks.js	Scheduled tasks (deletion of expired challenges)
src/server/utils/validator.js	Validation utilities
src/server/views/benchmark.pug	Benchmark view template
src/server/views/main.pug	Register page view template
tests/controllers/main-controller-tests.js	*main-controller.js* unit tests
benchmark_results_parser.py	Python script to process the .json files generated by the benchmark page and display summary statistics of the results

Appendix B Code libraries used

Library	Usage	License	Version
express[1]	The server framework for the messsenger application	MIT	4.17.1
express-session[2]	User session management and server-side session variables (this library was used only during development and is not being actively used in the final version of the application)	MIT	1.17.0
body-parser[3]	Parsing of HTTP request bodies to JavaScript objects	MIT	1.19.0
cookie-parser[4]	Parsing of the HTTP "Cookie" header and converting it to a JavaScript object (this library was used only during development and is not being actively used in the final version of the application)	MIT	1.4.5
frontend-dependencies[5]	Used to adapt node package manager (npm) to manage front-end dependencies	MIT	1.1.8
moment[6]	Used for date modification operations which are not available in plain JavaScript	MIT	2.24.0
mongoose[7]	Definition database schemas and models; Connection to the database	MIT	5.9.7
node-schedule[8]	Scheduling of the operation that deletes expired challenges, making it run on a specified time interval while the server is running	MIT	1.3.2
pug[9]	Used as the template engine for Express	MIT	2.0.4
serve-favicon[10]	Efficient serving of the website's favicon	MIT	2.5.0

socket.io[11]	Real-time communication between the client application and the web server, allowing for instant delivery of new messages	MIT	2.3.0
sphincs[12]	Digital signature operations using the SPHINCS cryptosystem	BSD-2-Clause	2.4.0
toastr[13]	Notifications in the client application	MIT	2.1.4
sidh[14]	Key generation and secret derivation using the SIDH(v2) cryptosystem	BSD-2-Clause	4.1.6
vue[15]	Used in the client application to provide reactive(real-time) HTML rendering upon updating the underlying application information such as messages and contacts(also functions as a template engine)	MIT	2.6.11
bootstrap[16]	Styling of HTML elements	MIT	4.4.1
jquery[17]	Mainly used to operate on the HTML DOM tree(selection and modification of elements)	MIT	3.5.0
socket.io-client[18]	The client side library for *socket.io*, used to connect to the server's socket.io	MIT	2.3.0
localforage[19]	Used as an abstraction to IndexedDB	Apache-2.0	1.7.3
promise-worker[20]	Used to add support for the SIDH and SPHINCS web workers to be being called as JavaScript promises	Apache-2.0	2.0.1
webworker-promise[21]	Used in the multi-threaded version of the application to enable the web workers to use multiple threads	MIT	0.4.2
font-awesome[22]	Provides the icons in the client application	OFL-1.1 AND MIT	4.7.0

randomart-js[23]	Used to produce randomart from the hash of the additional data that is generated for each contact	MIT	0.1.1
chai[24]	Used as the assertion framework for unit tests	MIT	4.2.0
eslint[25]	Used to check the code style/syntax against a configuration	MIT	6.8.0
eslint-config-google[26]	Used as the configuration for *eslint*	Apache-2.0	0.14.0
mocha[27]	Used as the unit test framework (running and description of unit tests)	MIT	7.1.1
nodemon[28]	Used to automatically restart the server upon a server source code file change(during development)	MIT	2.0.2
sinon[29]	Used to create mock objects in unit tests	BSD-3-Clause	9.0.1

Package URLs

1. https://www.npmjs.com/package/express/v/4.17.1
2. https://www.npmjs.com/package/express-session/v/1.17.0
3. https://www.npmjs.com/package/body-parser/v/1.19.0
4. https://www.npmjs.com/package/cookie-parser/v/1.4.5
5. https://www.npmjs.com/package/frontend-dependencies/v/1.1.8
6. https://www.npmjs.com/package/moment/v/2.24.0

7. https://www.npmjs.com/package/mongoose/v/5.9.7
8. https://www.npmjs.com/package/node-schedule/v/1.3.2
9. https://www.npmjs.com/package/pug/v/2.0.4
10. https://www.npmjs.com/package/serve-favicon/v/2.5.0
11. https://www.npmjs.com/package/socket.io/v/2.3.0
12. https://www.npmjs.com/package/sphincs/v/2.4.0
13. https://www.npmjs.com/package/toastr/v/2.1.4
14. https://www.npmjs.com/package/sidh/v/4.1.6
15. https://www.npmjs.com/package/vue/v/2.6.11
16. https://www.npmjs.com/package/bootstrap/v/4.4.1
17. https://www.npmjs.com/package/jquery/v/3.5.0
18. https://www.npmjs.com/package/socket.io-client/v/2.3.0
19. https://www.npmjs.com/package/localforage/v/1.7.3
20. https://www.npmjs.com/package/promise-worker/v/2.0.1
21. https://www.npmjs.com/package/webworker-promise/v/0.4.2

22. https://www.npmjs.com/package/font-awesome/v/4.7.0

23. https://www.npmjs.com/package/randomart-js/v/0.1.1

24. https://www.npmjs.com/package/chai/v/4.2.0

25. https://www.npmjs.com/package/eslint/v/6.8.0

26. https://www.npmjs.com/package/eslint-config-google/v/0.14.0

27. https://www.npmjs.com/package/mocha/v/7.1.1

28. https://www.npmjs.com/package/nodemon/v/2.0.2

29. https://www.npmjs.com/package/sinon/v/9.0.1

Appendix C Project README

Quark chat

Installation

- MongoDB version 4.2.1(or later) must be installed, running at its default port (27017) without authentication
- Node version 12.16.2(or later) must be installed. If you have nvm installed, just run `nvm install && nvm use` the project's directory (this nvm feature does not work in a Windows environment)

1. Clone this repository
2. Open a terminal and navigate to the repository
3. Run `npm install` and wait for the dependencies to be installed
4. Run `npm start` to start the application. Configuration options are available (see the Configuration section).
5. Open a web-browser. By default, the app is available at http://localhost:8080

   NOTE: In order to be used on devices other than localhost, the app needs to be accessed over an https connection due to WebCrypto API limitations. A quick way to overcome this is to use a tunnelling reverse proxy such as ngrok

Configuration

Some application variables can be controlled by the user. These are to be passed as environmental variables, i.e.:

- Defined for this specific instance of the app

```
VARIABLE1=value1 VARIABLE2=value2 npm start
```

or

- Defined globally in the shell configuration file (e.g. .bashrc, .zshrc, .bash_profile, etc.)

```
export VARIABLE=value
```

The following variables are user configurable:

Variable	Type	Description	Default value
CONNECTION_STRING	MongoDB connection string	Connection string for the mongodb database	mongodb://localhost/QuarkChatDB
NODE_ENV	string	One of `development` or `production`. When this variable is set to `production`, the CONNECTION_STRING variable must be also set and error messages are not printed to the console	development
PORT	integer	The port the application runs at	8080
CHALLENGE_TIMEOUT	integer	The timeout for the challenges served to users in ***ms***	60000
SPK_RENEWAL_INTERVAL_UNIT	string	The time unit for users' Signed pre-key change interval	d

Variable	Type	Description	Default value
SPK_RENEWAL_INTERVAL_VALUE	integer	The value of SPK_RENEWAL_INTERVAL_UNIT	2
LOW_OTPK_WARNING_THRESHOLD	integer	The amount of one-time pre-keys for a user which, upon reached, makes the server request from the client(user) to submit more one-time pre-keys	5

Development

- Run with `npm run dev` instead of `npm start` for the application to automatically restart upon changing a file
- Run unit tests with `npm test`
- Check syntax and code style with `npm run lint`

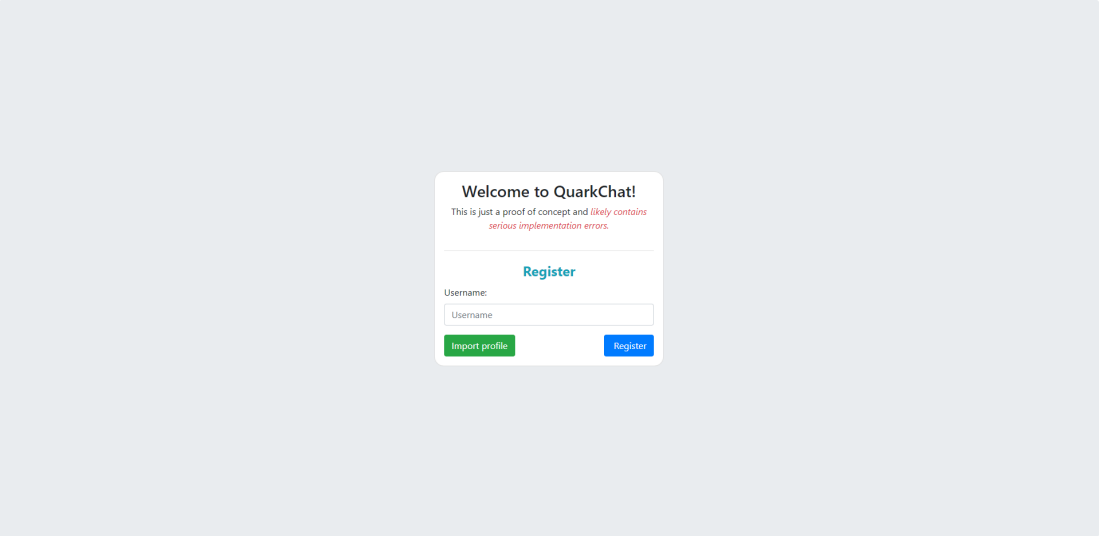
Appendix D User Manual

Quark chat - User Manual

The goal of this document is to display the main functionalities of the application Quark chat and how to use it.

Login screen - Import and Register

`Image Displaying the Login Page`



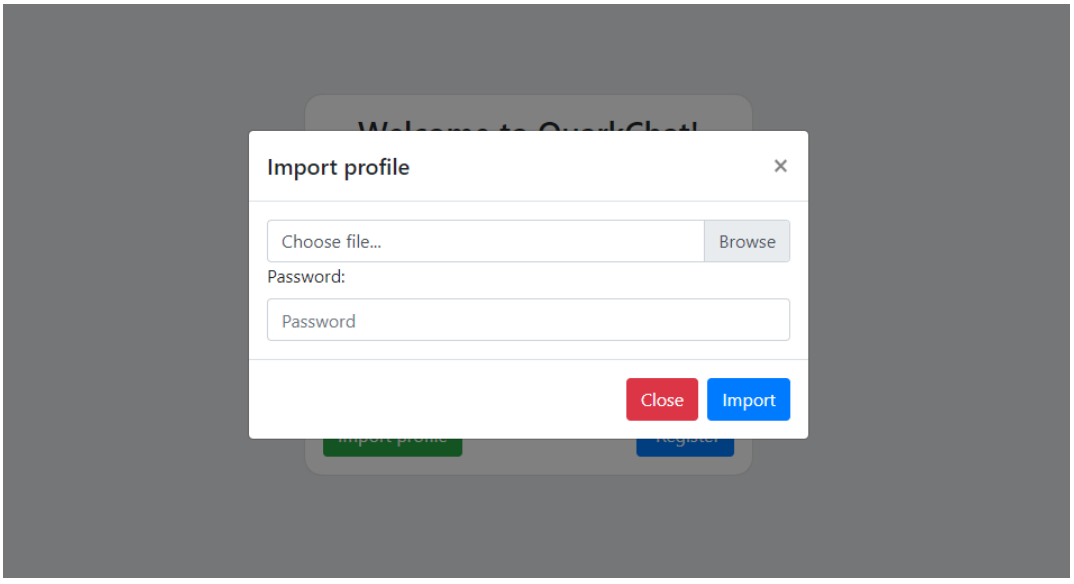
1. Register

- You will have to create a new account when you use Quark chat for the first time.
- Think of a username that is alphanumeric and between 3 and 30 characters and click "Register".

2. Import

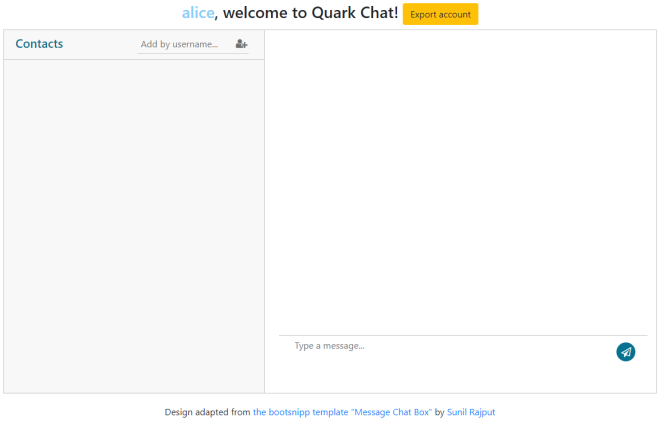
- If you already used Quark chat and you have an account click on "Import".
- In the new prompt you will have to select the profile file(.qc) generated during the last session after using the option "Export".
- After typing the password(created during export) click "Import".

`Image Displaying the Import Dialog`



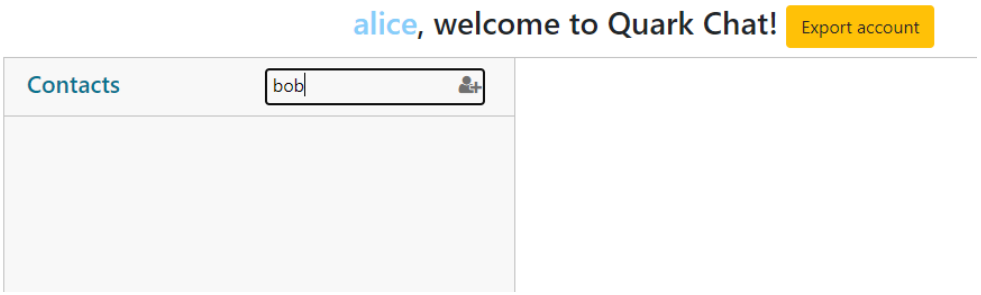
Main Page - Adding a User and starting a conversation for the first time

`Image Displaying the Main Page`



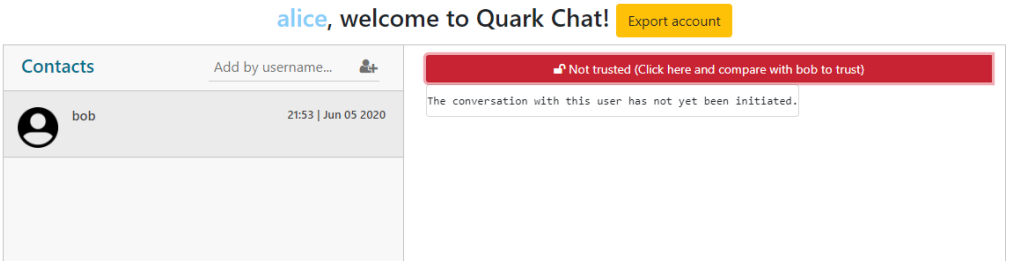
- Before you can start a chat, you will need to add some Users to the Contacts List. That is achieved by typing the name of the user in the "Add by username..." field and clicking on the button next to it. Notice that you can only add Users that are already registered in Quark Chat.

`Image Displaying how to add a User`

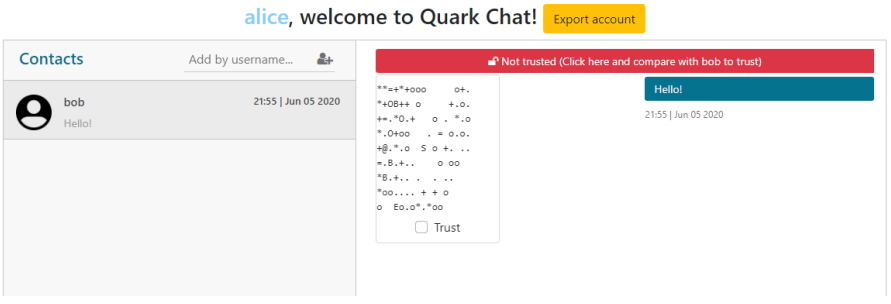


- To start a conversation, select someone from the contacts list by clicking on him. Notice that on top of the list are the people that wrote to you most recently. Messages that you haven't read are with displayed with a bold font.
- When you open the chat with someone for the first time you can see it marked as "Not trusted" in Red.
- To indicate the chat as "Trusted" first you will have to initiate the conversation. That happens automatically after sending a message to the other User.
- To confirm that their public keys are authentic, the two users can meet in person and compare the image generated when they click the red bar. If the two match, they can click the "Trust" button, the red bar becomes green and reads "Trusted" to signify that they have confirmed their identities. If the two images are not the same, then it is likely that their public keys have been tampered with and are possibly a subject to Man-in-the-Middle attack.

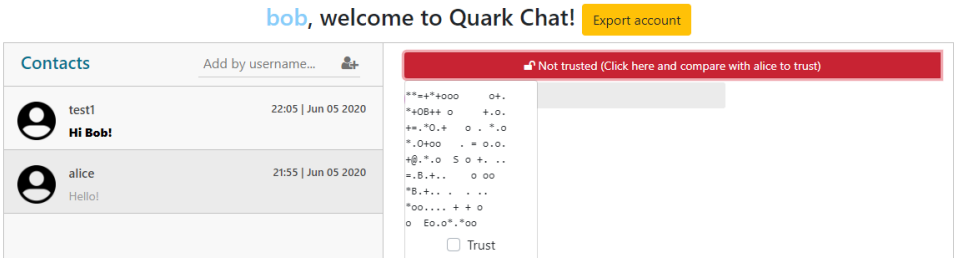
`Image Displaying the chat before initialization`



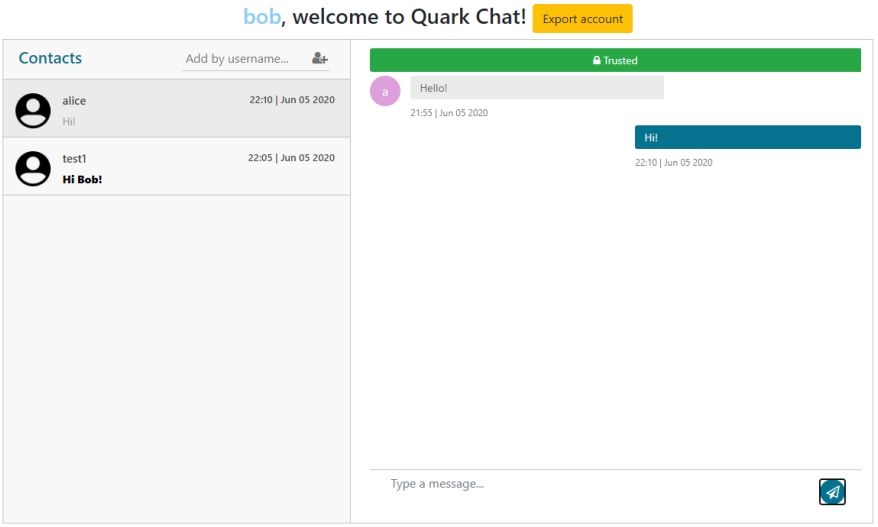
`Image Displaying the "randomart" after initialization and before marking the chat as "Trusted"(from Alice's perspective)`



`Image Displaying the "randomart" after initialization and before marking the chat as "Trusted"(from Bob's perspective)`



`Image Displaying the chat after marking it as "Trusted"`

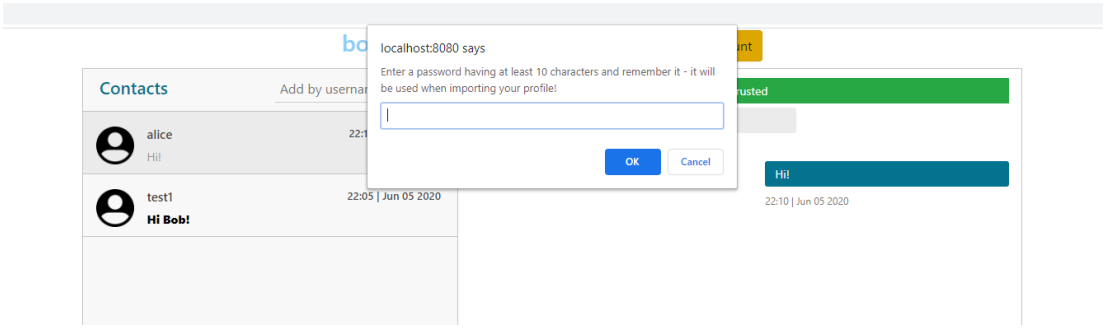


Design adapted from the bootsnipp template "Message Chat Box" by Sunil Rajput

Main Page - Exiting the application

- To exit Quark chat click on Export account.
- Export of the account is needed due to the fact that account information is stored only locally(in the browser, rather than in a remote database) and that is the only way to log in into the same account again in the future.

`Image Displaying the Export prompt`



Benchmarking utility

To benchmark the messaging application on your machine, you can visit `/benchmark` and run the tests

References

[1] Syeda Batool and Waseem Hafiz. A novel image encryption scheme based on arnold scrambling and lucas series. *Multimedia Tools and Applications*, 06 2019.

[2] Robert Campbell. Evaluation of post-quantum distributed ledger cryptography. *The Journal of the British Blockchain Association*, 2:1–8, 05 2019.

[3] Lily Chen, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology, 2016.

[4] Moxie Marlinspike and Trevor Perrin. The x3dh key agreement protocol. *Open Whisper Systems*, 2016.

[5] Moxie Marlinspike and Trevor Perrin. The double ratchet algorithm. *Open Whisper Systems*, 2016.

[6] Simon Blake-Wilson, Don Johnson, and Alfred Menezes. Key agreement protocols and their security analysis, 1997.

[7] James Bucanek. *Model-View-Controller Pattern*, pages 353–402. Apress, Berkeley, CA, 2009.

[8] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[9] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[10] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006.

[11] Eric Rescorla. HTTP Over TLS. RFC 2818, May 2000.

[12] Sonu Aggarwal, Gordon Mohr, Mark Day, and Jesse R. Vincent. Instant Messaging / Presence Protocol Requirements. RFC 2779, February 2000.

[13] Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21:120–126, 01 1978.

[14] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[15] Burt Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315, March 1998.

[16] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[17] Joanna Lang. The elliptic curve diffie-hellman (ecdh), 2015.

[18] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[19] Patrick Gallagher. Digital signature standard (dss). *Federal Information Processing Standards Publications, volume FIPS*, pages 186–3, 2013.

[20] Jakob Jonsson and Burt Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, February 2003.

[21] Peter Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. *Proceedings of 35th Annual Symposium on Foundations of Computer Science*, feb 1996.

[22] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[23] Vladimir Valyukh. *Performance and Comparison of post-quantum Cryptographic Algorithms*. PhD thesis, Linköping University, Department of Electrical Engineering, 2017.

[24] Lorenz Minder. *Cryptography based on error correcting codes*. PhD thesis, Verlag nicht ermittelbar, 2007.

[25] Steven D Galbraith and Frederik Vercauteren. Computational problems in supersingular elliptic curve isogenies. *Quantum Information Processing*, 17(10):265, 2018.

[26] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to/dev/random. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 203–212, 2005.

[27] Trevor Perrin. The xeddsa and vxeddsa signature schemes. *Open Whisper Systems*, 2016.

[28] Moxie Marlinspike and Trevor Perrin. The sesame algorithm: Session management for asynchronous message encryption. *Open Whisper Systems*, 2017.

[29] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[30] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 435–450, 2017.

[31] Alfred Menezes and Berkant Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. *IJACT*, 2:154–158, 01 2010.

[32] Dr. Hugo Krawczyk and Pasi Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010.

[33] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008.

[34] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[35] Quynh H Dang. Secure hash standard, 2015.

[36] Ines Duits. The post-quantum signal protocol : Secure chat in a quantum world, February 2019. http://essay.utwente.nl/77239/.

[37] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. Applying grover's algorithm to aes: quantum resource estimates. In *Post-Quantum Cryptography*, pages 29–43. Springer, 2016.

[38] Martin Roetteler, Michael Naehrig, Krysta Svore, and Kristin Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *Advances in Cryptology – ASIACRYPT 2017*, pages 241–270, 11 2017.

[39] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[40] Craig Costello. Supersingular isogeny key exchange for beginners. In *International Conference on Selected Areas in Cryptography*, pages 21–50. Springer, 2019.

[41] Caroline Kudla and Kenneth G Paterson. Modular security proofs for key agreement protocols. In *International conference on the theory and application of cryptology and information security*, pages 549–565. Springer, 2005.

[42] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. Sphincs: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015.

[43] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. Cryptology ePrint Archive, Report 2013/322, 2013. https://eprint.iacr.org/2013/322.

[44] Henk CA Van Tilborg and Sushil Jajodia. *Encyclopedia of cryptography and security*. Springer Science & Business Media, 2014.

[45] Adrian Perrig and Dawn Song. Hash visualization: A new technique to improve real-world security. In *International Workshop on Cryptographic Techniques and E-Commerce*, pages 131–138, 1999.

[46] Dirk Loss, Tobias Limmer, and Alexander von Gernler. The drunken bishop: An analysis of the openssh fingerprint visualization algorithm, 2009.

[47] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, Citeseer, 1994.

[48] Robert C Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.

[49] Ekaterina Razina and David S Janzen. Effects of dependency injection on maintainability. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, page 7, 2007.

[50] Philippe Lalanda. Shared repository pattern. In *Proceedings of the 5th Pattern Languages of Programs Conference (PLoP 1998)*. Citeseer, 1998.

[51] Samuel Oloruntoba. S.O.L.I.D: The First 5 Principles of Object Oriented Design, March 2014. https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

[52] K, wrap it up mom, March 23 2017. https://crushhapp.com/blog/k-wrap-it-up-mom.

[53] Moxie Marlinspike. Private Group Messaging, 2014. https://signal.org/blog/private-groups/.

[54] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. A post-quantum digital signature scheme based on supersingular isogenies. In *International Conference on Financial Cryptography and Data Security*, pages 163–181. Springer, 2017.