The Post-Quantum Signal Protocol

Secure Chat in a Quantum World

Ines Duits

February 5, 2019
Final version 1.3

UNIVERSITY OF TWENTE.

Services and Cybersecurity (SCS)

TNO innovation for life

Cyber Security and Robustness (CSR)

Thesis

The Post-Quantum Signal Protocol
Secure Chat in a Quantum World

Ines Duits

Graduation committee

dr. M.H. Everts
M.P.P. van Heesch, MSc
T. Attema, MSc
dr. A. Peter

February 5, 2019

Ines Duits
The Post-Quantum Signal Protocol
Secure Chat in a Quantum World
Thesis, February 5, 2019
Graduation committee: dr. M.H. Everts, M.P.P. van Heesch, MSc, T. Attema, MSc, dr. A. Peter

University of Twente
Services and Cybersecurity (SCS)
Drienerlolaan 5
7522 NB Enschede

TNO
Cyber Security and Robustness (CSR)
Anna van Buerenplein 1
2595 DA Den Haag

Abstract

The Signal Protocol provides end-to-end encryption, forward secrecy, backward secrecy, authentication and deniability for chat applications like WhatsApp, Skype, Facebook private Messenger, Google Allo and Signal. The Signal Protocol does this by using the ECDH Curve25519 key exchanges and SHA-512 key derivation. However, the ECDH key exchange is not quantum-safe; in a world were adversaries would have a quantum computer, they could get the key and read along. A post-quantum Signal Protocol requires a substitute for the ECDH key exchanges. Therefore, we look at post-quantum cryptography, which is secure against a quantum computer.

We test 10 different post-quantum key exchange mechanisms (KEMs) and the post-quantum supersingular isogeny based Diffie-Hellman (*SIDH*). Each post-quantum algorithm has different versions, which results in 44 different algorithms. In this thesis we analyse those 44 post-quantum algorithms and see how they affect the performance of Signal Protocol in terms of run time (CPU cycles), storage space requirements, bandwidth and energy efficiency. Additionally we analyse different versions of a partially post-quantum Signal Protocol. These partially post-quantum Signal Protocols are easier to implement and already are a safety measure against quantum attacks that might happen in the future.

The Signal Protocol is explained in 3 different phase: the initial setup, the first message and the message exchange. To investigate whether a post-quantum Signal Protocol is possible in practice, a likely scenario was described for each phase. For each scenario we looked at the influence the post-quantum algorithms would have on an average user, with a minimal phone in 2018. Based on our analysis, a quantum-safe Signal Protocol using both *kyper512* and *SIDH503* would result in the lowest overhead with less than 0.02 seconds per message extra delay. However, using the KEM *kyper512* requires a small change to the Signal Protocol. A complete *SIDH503* Signal Protocol would be the easiest to implement, because *SIDH* is a perfect plug and play with *ECDH*, but it will take 0.03 seconds more delay per message.

We conclude that it is feasible to have different post-quantum Signal Protocols considering the state of 2018.

Contents

1 Introduction

Throughout history, humans have been communicating in all kinds of ways: talking, writing, yodelling, smoke signals, light signals, doves, art etc. In the current digital era, a lot of the communication happens online. Almost 3.2 Billion people use social media in 2018 [Cha18] to communicate about their lives. WhatsApp is used by almost half of those users and Facebook Messenger is used by almost one third of them. However, in this world of digital communication there is a need to keep your data private, secure and confidential. Some people should be able to read the messages, while others should not. Cryptography can be used to keep communication secure, even when the communication is over an insecure channel; in which an adversary can observe all the messages. While cryptography started out as a way to hide the content of a message, nowadays cryptography can also be used for, among other things, authentication pseudorandom number generations and checking the integrity of a message [BR05].

To keep communications secure, users and computers have to follow certain security protocols. A protocol is just a collection of steps for the user to follow. A simple example is a symmetric encryption scheme or a Diffie-Hellman key exchange (which will be explained in more detail in Section 2.1). More complex protocols are combinations of these simpler cryptographic primitives. Open Whisper Systems' Signal Protocol is a more complex protocol which provides end-to-end encryption between two chatting users [Sig]. The protocol is used in chat applications like WhatsApp [Mar16c], Facebook private messaging [Mar16a], Google Allo [Mar16b], Skype [Lun18] and Signal [Sig].

The Signal Protocol combines a lot of cryptographic primitives like Elliptic Curve Diffie-Hellman (ECDH) key exchanges, symmetric encryption and key derivation functions. Most cryptographic primitives are based on mathematical principles which theoretically could be calculated and broken. However, these calculations are computational hard to perform. Current cryptographic primitives are strong enough so that an adversary with limited computational power cannot break them.

Unfortunately, with the rise of quantum computers the above statement is not true anymore and the security of some cryptographic primitives are threatened.

In the nineties Shor [Sho94] and Grover [Gro96] introduced quantum algorithms which theoretically are able to break the cryptographic principles in a lot of cryptography primitives. Elliptic Curve Diffie-Hellman and RSA are broken by these algorithms (why and how is explained in Section 2.3).

A lot of research has been performed in the field of quantum computers. Not only to improve the algorithms by Shor and Grover, but also to actually build quantum computers. Currently, quantum computers are not a threat to cryptography yet. However, in the future they might be. To anticipate on the threat of quantum computers, alternative for the broken cryptography are needed. Post-quantum cryptography is the subset of cryptography that is quantum-safe. The National Institute of Standards and Technology (NIST) is currently working on finding different standards for post-quantum cryptography. With this initiative, 69 post-quantum algorithms are analysed, tested and sometimes already implemented. Not all post-quantum algorithms are newly developed, some already exist but are not used that frequently. New cryptography requires research before it can be safely implemented into actual systems and protocols, because undiscovered bugs might form a problem.

This standardisation process needs do be done immediately, since it is the first step towards secure post-quantum cryptography. The standards should be implemented as well and that process is taking time. The theorem of Mosca [Mos15] explains when to worry about quantum computers breaking the encryption of our data. A problem occurs if the time it takes to make our system quantum-safe, y, plus the time the data should stay secure, x, are bigger than the time it takes to build a quantum computer, z.

Fig. 1.1.: The theorem of Mosca show in an image, in which x is the time that the data needs to stay secure, y is the time it takes to make the system quantum secure and z the time which it will take to make a quantum computer.

There will be a leak of data if $x + y > z$, as shown in Figure 1.1. In that case, our data could be broken by quantum computers. Therefore, research to the implementation of post-quantum cryptography in actual protocols is very useful.

In this thesis, a post-quantum Signal Protocol is created, where the problems that are encountered when implementing post-quantum cryptography in the protocol are identified. Even though it might seem easy to just substitute the current cryptography with a post-quantum version, it is not that simple. Post-quantum algorithms are sometimes slower in run time and require bigger keys.

Thereby, they are not always a perfect plug and play for current standards. In the Signal Protocol an alternative for ECDH should be found, and there are not many alternatives that can maintain the security properties the Signal Protocol has. However, different possible post-quantum Signal Protocols are evaluated.

The remainder of the introduction will discuss the contribution of this thesis in this research area (Section 1.1), give an overview of contents (Section 1.1) and provide an overview of related works (Section 1.2).

1.1 Outline of this thesis

In this thesis, we explain that it is possible to have a post-quantum Signal Protocol, considering an average user in 2018. The challenges faced when using post-quantum cryptography, how it affect the Signal Protocol and if the effects are manageable in a chat application are discussed as well.

The contribution of this thesis therefor consists of:

- An analysis of the different building blocks in the Signal Protocol and how making them quantum-safe would would affect the Signal Protocol.
- An analysis of which building blocks should be substituted for post-quantum ones to create a post-quantum Signal Protocol.
- A simple implementation of different post-quantum algorithms in the Signal Protocol.
- An evaluation of the different post-quantum algorithms in the Signal Protocol and how they will affect the protocol and the user.
- An overview of the three most suitable post-quantum Signal Protocols for an average user in 2018.

We motivate and introduce this thesis in the above section. In Section 2 the preliminaries can be found. In the preliminaries, cryptography, symmetric and public key encryption are introduced. The difference in security level of cryptography in a classical and a quantum computer is discussed, and post-quantum cryptography is introduced. In Section 3 the Signal Protocol is introduced. The security claims of the Signal Protocol (end-to-end encryption, forward and backward secrecy, deniability and authentication) are analysed for every part of the Signal Protocol. In Section 4, the necessary changes for Signal Protocol to make it quantum-safe are summarised and the corresponding challenges when creating that post-quantum Signal Protocol are discussed. The possible solutions to those challenges are introduced in the form of partially post-quantum Signal Protocols, which are

useful in the transitional period from classical to quantum computers. In Section 5 explains how the different post-quantum Signal Protocols are implemented and analysed. Three scenarios for the Signal Protocol, the post-quantum algorithms and an average user are described. In Section 6 we describe the results for each scenario. Also, per scenario the best three post-quantum algorithms are chosen and those best algorithms are combined in possible best post-quantum Signal Protocols for an average user. Section 7 consists of the conclusion, a discussion and possible future research on this matter.

1.2 Related work

Signal is not the only secure chat that uses the Signal Protocol, WhatsApp [Mar16c], Facebook private messaging [Mar16a], Google Allo [Mar16b], Crytocat [Cry], Wire [Wir] and more also use it.

Wire, an encrypted instant messaging client, already looked into the possibilities of a post-quantum Signal Protocol. They created a transitional post-quantum Signal Protocol using the post-quantum algorithm NewHope [RA18]. While this is a great start, Wire's version is not yet a complete post-quantum Signal Protocol, as will be explained in Section 4.

There are also chat alternatives that do not use the Signal Protocol like Telegram [Tel], Threema [Thr], Wickr Me [Wic] and PQChat. PQChat was a promising example of a post-quantum chat application; however, it does not exist anymore.

There is not much research on post-quantum chat protocols; however, there is a lot of research into post-quantum protocols. De Vries [Vri16] implemented a post-quantum OpenVPN with which he achieved 128-bit security against quantum attacks. Bos et al. [Bos+16b] implemented the Lattice-based post-quantum algorithm: Ring Learning With Errors Problem (RLWE) into the Transport Layer Security (TLS) using OpenSSL, creating a 128-bit security level. Stabila and Mosca [SM17] reviewed two lattice based post-quantum key exchanges: *BCNS15* and *Frodo* and integrated them in TLS and analysed how they perform. In *Transitioning to a Quantum-Resistant Public Key Infrastructure*, Bindel et al. [Bin+17] not only look at post-quantum cryptography into the TLS protocol, but also how post-quantum cryptography influence other protocols, namely certificates (X.509) and email (S/MIME). Kampanakis et al. [Kam+18] also reviewed the possibilities of a post-quantum X.509 certificate.

In contrast to implementing the post-quantum algorithms into protocols, there is also a lot of research going on into creating and analysing the actual post-quantum cryptography [Che+16]. An example of this is the initiative of National Institute of Standards and Technology (NIST) which started the process of standardising post-quantum cryptography [NIS16], in which 69 different post-quantum algorithms are analysed and evaluated to find new cryptographic standards which are quantum-safe. There are a lot of papers introducing and analysing post-quantum cryptography, including but not limited to *Frodo* [Bos+16a], *New Hope* [Alk+15], *SIDH* [RS06; Cos+16].

2 Preliminaries

This section gives the preliminaries for this thesis. Concepts about cryptography, security, quantum computers, and post-quantum cryptography among others, are introduced.

In Section 2.1 the cryptographic primitives used in this thesis are explained, like symmetric and public key encryption, Diffie-Hellman and signature schemes. In Section 2.2 different terms to explain the security of cryptography are introduced. Terms like, n-bits security, Universal Composability framework, security properties and attacks like CPA, CCA and Man-in-the-middle are introduced. In Section 2.3 quantum computers and how they threat the current used cryptography are explained. Section 2.4 introduces post-quantum cryptography, cryptography which is secure against quantum computers.

2.1 Introduction to cryptography

In the following sections a brief introduction to cryptographic primitives like encryption, symmetric key encryption, public-key encryption, key exchange, Diffie-Hellman, signature schemes and functions is given. For a more detailed explanation on all the cryptographic primitives refer to Menezes' *Handbook of Applied Cryptography* [Men+96].

2.1.1 Encryption

In *cryptography* when a *plaintext* is encrypted with a *key*, the resulting text is a *ciphertext*. If the message is revealed again the ciphertext is *decrypted*. The simplest way to encrypt messages is with a symmetric key encryption scheme, as explained in Section 2.1.2.

The *current keys* are the key which are used to encrypt, or decrypt, the current message.

2.1.2 Symmetric key encryption scheme

Symmetric encryption schemes are schemes in which both parties agreed on the shared symmetric *key* and then use that *key* to encrypt and decrypt messages to and from each other. On the left on Figure 2.1, the symmetric encryption scheme is shown.

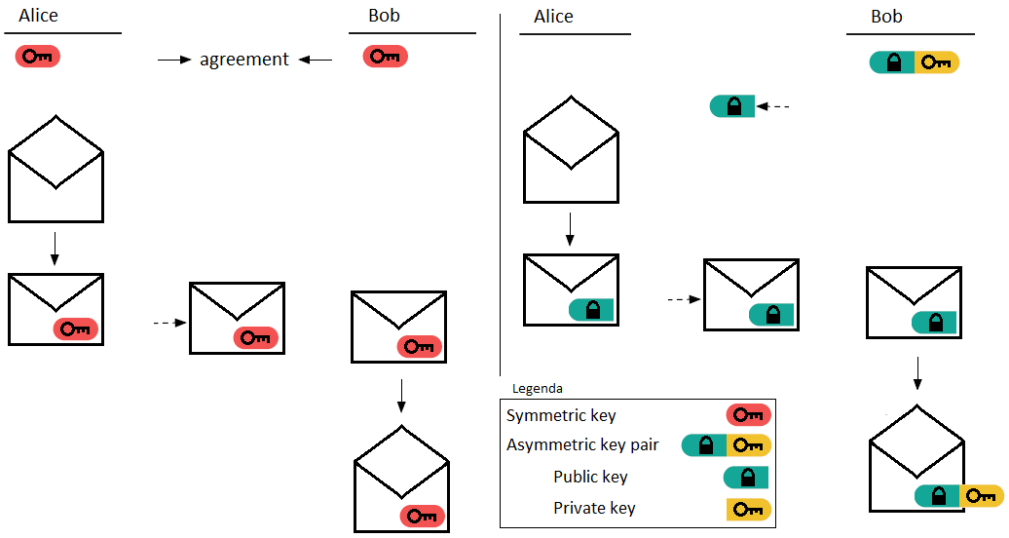

Fig. 2.1.: The symmetric encryption scheme (left); in which two users first secretly agree on a symmetric key, then they use that key to encrypt and decrypt messages. The public key encryption scheme (right); in which Alice only needs Bob's public key to send him an encrypted messages, Bob decrypts the message with his secret key.

Alice and Bob agree on a key K. When Alice wants to send Bob a message, she encrypts the message m with K, into the ciphertext c:

$$c = E(m)_K.$$

Alice sends Bob the ciphertext, and Bob decrypts the ciphertext using the K, to get the message m:

$$m = D(c)_K$$

Symmetric encryption is faster to use then public key encryption schemes (Section 2.1.3). However, the parties have to find a way to safely communicate the key. And without a way to do this securely, they will have a *key distribution problem*.

2.1.3 Public key encryption scheme

In *Public key encryption schemes* (also called *asymmetric encryption schemes*) two parties do not have to agree on a key safely before they can communicate securely,

because they do not publicly share a secret key. In public key encryption each party has two keys: a public one, A and a secret one, a, (also called *private key*). The public key is public, everybody can use it to encrypt a message that only the owner of the private key can decrypt. The public key encryption scheme can be seen on the right of Figure 2.1 right. If Alice wants to send Bob a message, she can encrypt the message using Bob's *public key*, B:

$$c = E(m)_B.$$

If Bob wants to decrypt the ciphertext he received from Alice, he uses his *private key*, b, to decrypt it

$$m = D(c)_b.$$

A few examples of a public key encryption are Diffie-Hellman (DH), ElGamal and RSA [Par13].

Most usually known public key encryption schemes are less efficient than symmetric schemes, and this makes them less practical for applications which need efficiency.

2.1.4 Key exchange

In this section, we look at how public key encryption schemes can be used in combination with symmetric encryption schemes (Section 2.1.2), to solve the key distribution problem encountered when using symmetric encryption.

Some public key encryption schemes can be used as a key exchange protocol (KEX). KEXs have the ability to create a *shared secret* between two users. That shared secret, SS, could then be used as the current key K in a symmetric encryption scheme to encrypt the message. To create a shared secret between Alice and Bob, Alice uses her own private key and Bob's public key to calculate the shared secret, SS:

$$SS = f(a, B)$$

Bob, in his turn will use Alice's public key and his own private key to generate the same shared secret:

$$SS = f(A, b)$$

The function f they use depends on the key exchange scheme they use. Both shared secrets are the same if the key exchange was successful and can be used as input for a symmetric key. In this way Alice and Bob solve the key distribution problem they had with symmetric encryption. This combination of public key encryption and symmetric encryption can be seen in Figure 2.2.

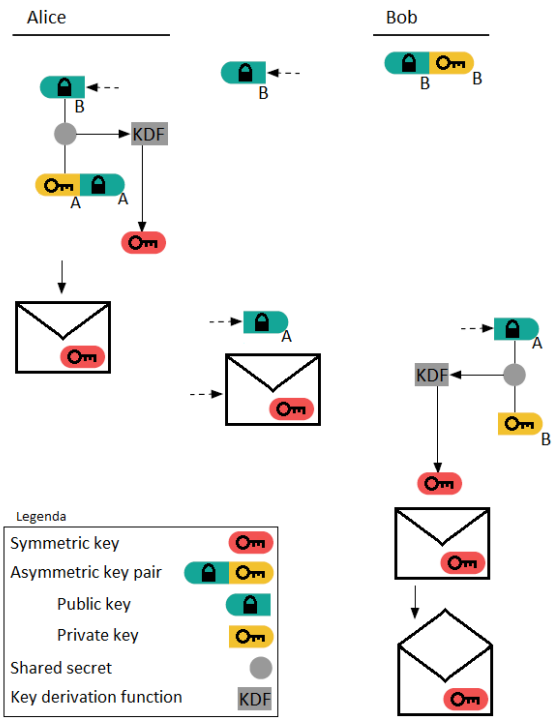

Fig. 2.2.: The combination between a symmetric encryption scheme and the public key encryption scheme. The public and private keys from the DH key pair are used to create a shared secret, which is in turn used to create a symmetric key. This key can be used to encrypt and decrypt he message.

Elliptic Curve Diffie-Hellman (ECDH), can be used as public encryption schemes and as KEX.

Another way to use public key encryption to agree on a shared secret is to use a key encapsulation mechanism (KEM). Most public key encryption schemes can be used as a KEM. If Alice and Bob want to agree on a key, Alice will create a shared secret herself. She uses Bob's public key to encapsulate that shared secret, and send it to Bob. Bob uses his private key to decapsulate the shared secret. In Section 5.3.1 we explain KEMs in more detail, and see how they could be implemented in the Signal Protocol.

An advantage of public keys for key exchanges is that it can be used *non-interactively*. This means that only one party needs to be online to agree on a key. A user can just upload his public keys to a server, where they will be stored until someone else needs them.

2.1.5 Key derivation function

Alice and Bob can use a key derivation function (KDF) to generate an actual key from their created shared secret, SS[1]. A key derivation function can be used to deviate new keys from old keys and other secret inputs [Kra10]. A KDF is one way, the old key can not be deviated from the new generated key.

Cryptographic hash functions are an example of possible key derivation functions. A hash function maps input data to a *hash value*, v, with a fixed size. For example, a hash function which maps all integer inputs, x, to a value between zero and nine, can have the following formula:

$$v(x) = hash_{10}(x) = x \mod 10.$$

Cryptographic hash functions are *one-way* and are *collision resistant*, which make them useful to use in security context. The one-way property makes it significantly hard to revert the hash value back to its original data, otherwise an adversary will be able to easily calculate an input message with the same hash value. A *low collision rate* means that two different input messages will map with a very small chance to the same hash value. Otherwise an adversary will be able to find another message with the same hash.

2.1.6 Signature schemes

Alice and Bob can communicate securely using the symmetric and public encryption scheme, but they need a way to authenticate each other to be sure they are communicating with each other. A way to authenticate the message is to sign it. A digital signature can be compared to a hand written signature. It is a way Alice can be sure the message is from Bob, by checking Bob's signature.

Public key encryption schemes can be used to create digital signatures. Alice will sign her message with her private key, a, and Bob can later verify this signature with Alice public key, A. Signing the whole message might give a big data overhead, that is why often only the footprint of a message is signed. The footprint of a message could be created by using a cryptographic hash function. For more detail on how cryptographic hash functions can be used to create a digital signature refer to [PS96].

[1] Using the created shared secret directly as key might be unwise because of forward secrecy. Forward secrecy will be explained in Section 2.2.3

2.2 Security

In this section, we first explain the difference between an active and a passive attack (Section 2.2.1). We then explain the n-bit security level, which is used to describe how strong cryptography is (Section 2.2.2). In Section 2.2.3 some security properties which could be used to describe the security of cryptographic primitives and protocols are explained.

2.2.1 Passive and active attacks

Two different attacks can be distinguished; a passive attack and an active attack. In a *passive attack* an adversary is only listening to the communication between two victim: Alice and Bob. The adversary can store all the messages, use all the public data available and use any computational power. However, he cannot interact with Alice and Bob, or change or interfere with the data they send to each other.

A passive quantum attack is the reason why some people already worry about the security of their data regarding quantum computers. Adversaries could store their non quantum-safe encrypted data now and decrypt it in the future, when quantum computers are available. The question that is asked is how long you want your data to stay secure, see the theory of Mosca in Section 1.

In an *active attack* an adversary can interfere in the communication actively. An example of this is a man-in-the-middle attack. In a man-in-the-middle attack Eve impersonates Alice towards Bob and vice versa. The attacker, Eve, stands between the public key exchange of Bob and Alice. When Alice sends her public key to Bob, Eve intercepts the key and keeps it, sending her own public key to Bob and creating a shared secret between her and Alice. For Bob she does the same, if Bob sends his public key to Alice, Eve intercepts it and sends her own public key to Alice, creating a shared secret between her and Bob. Alice and Bob think they have received each others keys and create a shared secret with each other. However, they both created a shared secret with Eve, and Eve with them. Eve intercepts all the messages between Alice and Bob.

2.2.2 n-bits security level

To define how strong a encryption scheme is the term n *bit security level* is often used [Len04]. In Table 2.2 we see an overview of the security level of some encryption algorithms.

Algorithm	Key length (B)	Security level
SHA-1	160	61
AES-128	128	128
AES-192	192	192
AES-256	256	256
SHA-256	256	128
RSA-1024	1024	80
RSA-2048	2048	112
ECDH curve25519	32	< 128
ECC-256	256	128
ECC-384	384	256

Tab. 2.1.: The table shows the security level for a few cryptographic schemes. Data based on [Cam+15][BK04][Lan+16]. These values are against a classical computer, in Section 2.3 we will see the security levels against a quantum computer.

Cryptographic hash functions, like AES (Advanced Encryption Standard) and SHA (Secure Hash Algorithm), can be broken by the following brute force attacks:

- Preimage attack: given the hash, h, find the message, m, such that $h = HASH(m)$.
- Collision attack: find $m1$ and $m2$, such that $m1 \neq m2$, while $HASH(m1) = HASH(m2)$.

The preimage attack will make you try each possibility, trying 2^n different values. While the collision attack will only take you $2^{\frac{n}{2}}$ tries.

Public key encryption algorithms, like RSA and ECC, have in general a security level that is signification lower than their key length, because they're based on mathematical principles with certain structures that allow for better attacks than trying all (or half of the) possible keys.

2.2.3 Security properties

There are different security properties which describe a security requirement which could make your chat more secure, we define properties which are present in the Signal Protocol. The current key, mentioned in both forward and backward secrecy, is the key that is used in the encryption or decryption of the current message. In the Signal Protocol the current keys are all the keys a user has, on his phone, on a certain moment.

End-to-end encryption If a protocol provides end-to-end encryption it means that no one, no server that hosts the messages or any third-party adversary, can read the messages send between the sender and the receiver [Erm+16].

Forward secrecy If a protocol has forward secrecy it means that if the current keys at moment x are leaked, an adversary cannot read the messages prior to message x [Bor+04]. See Figure 2.3.

Backward secrecy If a protocol has backward secrecy it means that if the current keys at moment x are leaked, an adversary cannot read the messages that are send after message x [Erm+16] (also called *Future secrecy* [Fro+14] or in combination with forward secrecy: *Post-Compromise Security* [CG+17]). See Figure 2.3.

Deniability If a protocol provides deniable communication it means that both parties cannot prove using cryptography that the other party participated in the conversation [Fro+14].

Authentication It the protocol has authentication it means that both parties can be sure with whom they communicate. Authentication, on first look, seem to contradict with the deniability requirement; however, we will see there are ways in which Bob is sure he talks to Alice without being able to proof to a third party that it is indeed Alice he talked to to prevent for example a man-in-the-middle attack.

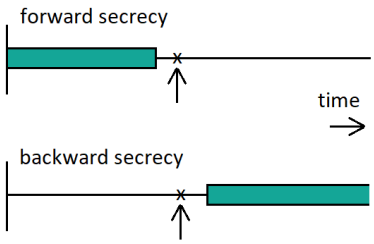

Fig. 2.3.: The properties forward and backward secrecy explained when the key is leaked at point x. The green (ticker line) is encrypted data that is still encrypted after a key compromise.

2.3 Quantum computers

Quantum computers threaten currently used cryptography. This section explains which cryptography they break and how they do that.

In 1994 Peter W. Shor created a quantum algorithm which solves prime factorisation and discrete logarithm problems in polynomial-time which is asymptotically faster than classical algorithms [Sho94], which means that it speeds up calculations needed to break certain cryptographic algorithms. In 1996, Grover created a quantum algorithm [Gro96] which improves the search process in unsorted data, resulting in a quadratic speed-up compared to conventional state-of-the-art algorithms [RP00].

The cryptography affected by quantum computers makes up for most of the public key ciphers like RSA, DSA (Digital Signing Algorithm), DH (Diffie-Hellman) and ECC (Elliptic Curve Cryptography), like ECDH and ECDHA, and other variations on these schemes [Cam+15]. RSA depend strongly on factorisation being *NP* hard (verifiable in polynomial time) for a computer to make it a secure algorithm. However, this assumption fails in the case of quantum computers. EC algorithms are based on discrete logarithm problems for elliptic curves. Both factorisation and discrete logarithms problems can be solved quicker using Shor's quantum algorithm [Sho94] in comparison to algorithms on a classical computer. A quantum computer exploiting Shor's algorithm can break these cryptographic schemes in a reasonable amount of time. There are various encryption algorithms that stay secure, as far as we know, when adversaries can use quantum computers. For example, symmetric ciphers, like AES (Advanced Encryption Standard), can be made quantum-safe by doubling the key size [Cam+15] and most hash functions stay quantum secure but it is required to create hashes twice the size [Bra+98].

Table 2.2 shows different cryptographic algorithms and their n-bit security level (see Section 2.2.2) for *conventional* computers and quantum computers for a few cryptographic schemes.

Algorithm	**Key Length** (in bits)	**Security Level** (in bits)	
		Conventional Computing	Quantum Computing
RSA-1024	1024	80	-
RSA-2048	2048	112	-
ECC-256	256	128	-
ECC-384	384	256	-
ECDH curve25519	32	< 128	-
AES-128	128	128	64
AES-192	192	192	96
AES-256	256	256	128
SHA-256	256	128	$85\frac{1}{3}$

Tab. 2.2.: The table shows the security level for a few conventional cryptographic schemes [Cam+15; BK04; Lan+16]. Some algorithms can be broken in non exponential time by a quantum computer, those are indicated by "-".

2.4 Post-quantum cryptography

Post-quantum cryptography is a subset of cryptography in which the algorithms can withstand a quantum attack. Some cryptography currently used is *quantum-safe* while other new post-quantum algorithms are especially designed to be quantum-safe: secure even when adversaries can use quantum computers. Most symmetric encryption schemes are currently considered to be quantum-safe [Ber+08], if one uses sufficiently large key sizes. The same is said for most hash functions [Ber09]. Of course we can never be sure if cryptography will stay secure forever; however, cryptography is tested and analysed to make a reliable assumptions about its security.

In Section 2.4.1 we explain on which post-quantum algorithms we focus: those submitted to the NIST standardisation process; why we do that and explain them in more detail. Section 2.4.2 describes the security level of the submitted post-quantum cryptographic schemes. In Section 2.4.3 we introduce hybrid encryption schemes, which is currently the advised way to implement post-quantum cryptography. In Section 2.4.4 we introduce the Universal Composablitity framework, which could be used to explain the security of such a hybrid scheme.

2.4.1 NIST submissions

The National Institute of Standards and Technology (NIST) is currently in the process of creating standards for post-quantum cryptography and in this process they are testing and analysing 69 different post-quantum algorithms [Che+16]. The submissions give a good overview of post-quantum cryptography [NIS16; RF18].

There are different types of post-quantum cryptography, and the most common discussed types are:

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

We focus on the first three. The *multivariate* and *hash-based* types are out of the scope of this paper because in this thesis we focus on the NIST submissions. The *multivariate* and *hash-based* submissions were mostly signature schemes [RF18];

no *hashed-based* KEM (Section 2.1.4) or encryption scheme were submitted and only two *multivariate* KEM submission: *DME* [Lue+17] and *CFPKM* [Cha+17]. Thereby the library we used (as will be explained in Section 5.3.3) to test the post-quantum algorithms did not have the multivariate algorithm implemented.

Thereby we choose to focus on key exchange schemes and not on signature schemes, because for passive quantum attacks the key which was exchanged or agreed upon should stay secret. If we keep using non post-quantum cryptography for key exchange, the encryption keys could retroactively be calculated using quantum computers (see the theorem of Mosca in Section 1). The authentication should be quantum-safe as well, but are not threatened by passive quantum attacks. If an attacker could fake a non quantum-safe signature created in the past, we could only strongly advice to not use those non quantum-safe signatures anymore as soon as the first threatening quantum computer is used.

In the next sections, we explain the *lattice-based*, *code-based* and *isogeny-based* cryptography in more detail. For a complete overview of post-quantum cryptography refer to Bernstein's *Introduction to Post-Quantum Cryptography* [Ber+08].

Lattice-based

Lattice-based cryptography is based on the presumably hard to solve mathematical problem for lattices: finding the shortest vector in a high dimensional lattice.

Intuitively a lattice is a set of points in space s. The basic idea for cryptography is to use this well formed lattice based space s as a secret key, and a scrambled version p of this base as a public key. The sender will map the message to a point on the well formed lattice base, then add an error in such a way that the point is still closer to the original point than any other point in the lattice. The receiver can then, because he knows the well formed base, decrypt it by finding the closest vector to the received point. It is assumed hard for an adversary, who is not aware of the well formed base, to find the closest vector point based on only the scrambled base.

Examples of lattice-based cryptographic schemes, based on the (R-)LWE hard problem, are *NTRU* for encryption and signing, *frodo* [Bos+16a] for key exchange and *NewHope* [Alk+15] for key exchange and digital signatures.

Code-based

Code-based cryptography is using error correcting codes, which were originally developed to improve communication by correcting the noise over noisy or unreliable channels, by adding control bits to verify and correct the data.

The message is converted into a code and a certain secret error is added. Because the receiver knows the code parameters, he can retrieve the original code. The adversary should not be able to distinguish the code from a random code. To achieve this, the public key is a scrambled version of a generator matrix, which was used to encrypt the message. This scramble principle is similar to what we say with lattice-based cryptography as well (Section 2.4.1). It is assumed hard for an adversary to decode a random linear code.

Examples of code-based cryptography are *McEliece* [McE78] and Niederreiter variant on that using Goppa codes [Din+11].

Isogeny-based

Elliptic Curve cryptography is based on computations on points on specific elliptic curves. The supersingular isogeny cryptography is based on finding the operation between specific elliptic curves. Those operations, that map a curve onto another curve with certain properties, are called isogenies. It is assumed hard to find the isogeny between two specific elliptic curves, unless you have more information about those curves. That information will become part of the secret key, while the public information is defined by two elliptic curves.

Examples of an Isogeny-based algorithm are Supersingular Isogeny Key Exchange (*SIKE*) and SuperSingulair Isogeny Diffie Hellman (*SIDH*). For more details about *side* and *SIKE* refer to Rostovtsev et al. [RS06] and Costello et al. [Cos+16].

2.4.2 NIST Security level

The post-quantum cryptographic algorithms can be categorised on their security strength. All the submissions for the standardisation process of NIST (see Section 2.4.1) are categorised in 5 different security strengths. In this thesis we focus on number 1, 3 and 5, and those are formulated [NIS16] as:

1 Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 128-bit key (e.g. AES128).
3 Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 192-bit key (e.g. AES192).
5 Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for key search on a block cipher with a 256-bit key (e.g. AES 256).

All three security levels; 128 bits, 192 bits and 256 bits, are assumed *"not known to be insecure"* until 2030 and beyond [Bar16]. Starting from 2030 a 112-bits security level "shall not be used for cryptographic protection".

We focus mostly on the 128 bit secure post-quantum cryptography, because it is assumed secure until at least 2030. While the 192 bits security level (NIST level 3) and 256 bits security level (NIST level 5) are assumed too high standards for decades, it is still important to research them. There may be reasons, yet unthinkable, for which we need higher security standards. It would be a waste to not at least acknowledge their existences.

2.4.3 Hybrid encryption scheme

Currently the new developed post-quantum cryptography is very new and less analysed, and therefore might have undiscovered vulnerabilities. To mitigate the risk of post-quantum cryptography a hybrid encryption scheme can be used. In a hybrid encryption scheme two or more different encryption schemes are combined. In this thesis we mean "A combination of a a not post-quantum encryption scheme and a post-quantum encryption scheme" when talked about hybrid encryption. If the post-quantum scheme is broken in a hybrid encryption scheme, at least we can still rely on the security of the non post-quantum algorithm.

For the transitional period, from classical to quantum computers, these hybrid schemes are very useful. Hybrid schemes would also protect you against a passive quantum attack.

2.4.4 Universal Composablility framework

When combining encryption schemes into hybrid schemes, or combining cryptographic primitives in new protocols the security of the primitives or the encryption

schemes might be influenced. Cannetti [Can01] introduced the Universal Composablility (UC) framework, which can be used to make statements about the security of the schemes or primitive in the UC frameworks. The UC framework allows for security analysis of complex protocols by analysing the security of the simpler building blocks: the cryptographic primitives. Cryptographic primitives that are proven secure in the UC framework remain secure when they are composed with and in other protocols. Examples of primitives that are proven to be secure in the UC framework, by Ralf Küsters and Daniel Raush [KR17], are:

- DH key exchanges (based on the DDH assumption)
- Symmetric encryption
- Public key encryption

Protocols which combine these UC secure primitives together can be sure that the primitives keep there security. The UC framework can in that way also help in proving that the whole protocol is secure in the UC framework. The UC framework is out of the scope of this thesis, for more information refer to Cannetti [Can01] and Küsters [KR17]. Refer to the works of Vajda [Vaj17] and Unruh [Unr10] for more detail about post-quantum cryptography in the UC framework.

3 The Signal Protocol

As indicated in the introduction (Section 1), we are working towards creating a post-quantum Signal Protocol. To understand how to create a post-quantum version we first introduce the Signal Protocol in this section. In Section 3.1 we give a brief introduction on the Signal Protocol and its security properties. In Section 3.2 we describe all the building blocks required for the Signal Protocol, and explain their function. In Section 3.3 we give a summary of the complete protocol, showing all the building blocks which were introduced in Section 3.2. In Section 3.4 we add a few other building blocks that are used in the Signal Protocol. These building blocks are not necessary to understand the basic working of the Signal Protocol.

3.1 Introduction to the Signal Protocol

The Signal Protocol is a protocol that allows users to update the key used for encryption. The protocol can be used to provide end-to-end encryption for voice calls, video calls, and instant messaging conversations. In the communication between Alice and Bob, the Signal Protocol can be split in three phases:

1. Key generation: occurs before there is any communication between Alice and Bob.
2. Key agreement: the first message from Alice to Bob in which they agree on the initial key.
3. Key renewal: during normal chat, the key is updated every message, after initial contact was made.

The Signal Protocol has two major parts: the Extended Triple Diffie-Hellman (X3DH) protocol and the Double Ratchet algorithm. The first two phases are done by the X3DH protocol, the normal chat phase is done by the Double Ratchet algorithm. Those protocols are, in their turn, created out of cryptographic primitives. The major cryptographic primitives used are:

- DH key exchange (using Curve25519)
- Symmetric encryption

- Key derivation function (KDF) (using SHA-512)
- Public signature schemes (using Curve25519)

These primitives, combined in both the X3DH and Double Ratchet give the Signal Protocol its desirable security properties [Fro+14]:

- End-to-end encryption
- Forward secrecy
- Backward secrecy
- Authentication
- Deniability

which are defined in Section 2.2.3. And because the Signal Protocol is used for chat application it means that the protocol has to have the property:

- *Non-interactive*: no interaction is needed to communicate or choose keys

As was explained in Section 2.1.4.

In the next section, we see how the building blocks with these properties are implemented in the Signal Protocol.

3.2 Building towards the Signal Protocol

In this section, we add the building blocks and cryptographic primitives of the Signal Protocol one by one, to eventually create the Signal Protocol. We explain per block or primitive which security properties it adds. We start building the Signal Protocol from a simple unsecured chat.

3.2.1 End-to-end encryption

The very first security property the Signal protocol should have is end-to-end encryption. This is achieved by adding symmetric encryption as explained in Section 2.1. To solve the key distribution problem they use *public key encryption* to generate a *shared secret* between the two of them, which they can use as input for a *key derivation function* to create a symmetric key. The complete buildup to this basic end-to-end chat was explained in Section 2.1. The resulting chat between Alice and Bob is shown in Figure 2.2.

3.2.2 Forward secrecy and backward secrecy in the *DH ratchet*

A protocol with only end-to-end encryption has a single point of failure. Namely, when a key is leaked, all current, previous and future messages are revealed. Ideally this scenario should leak as little information as possible. Hence, we want forward and backward secrecy (See section 2.2.3), so that if an attacker gets the key at point x he cannot read previous and future messages. Forward and backward secrecy can be implemented by using *one-time* keys, which are used one time and deleted. Leaking a one-time key will only compromise the information that has been encrypted with that key.

The *Diffie-Hellman Ratchet* (DH ratchet) [PM16a] makes it possible to have every message encrypted with another symmetric key. These symmetric keys are, in turn, generated by DH key exchanges. For every key exchange either Alice or Bob renews its DH public-private key-pair. The symmetric keys are thus only used once and every DH key-pair is only used twice.

The DH ratchet is shown in Figure 3.1, and shows how Alice sends the first message to Bob, Bob replies, after which Alice replies again.

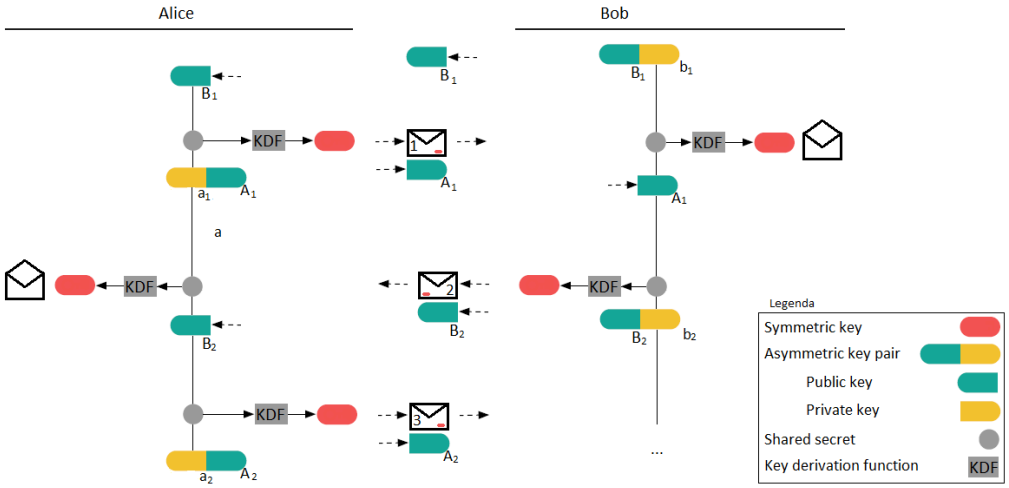

Fig. 3.1.: The DH ratchet at work, in which Alice sends first message to Bob (using B_1 and her private a_1), Bob replies to her (using his private b_2 and Alice public A_1) and Alice responds again (using B_2 and a_2). The figure is an adaptation of an image from [PM16a].

Alice will start by sending the first message to Bob. After she receives Bob's public key, B_1 she takes the following steps:

- Generate a new DH key pair, (A_1, a_1).

- Create a symmetric shared secret between her and Bob, $DH(B_1, a_1)$.
- Encrypt the message with the symmetric shared secret.
- Send Bob the encrypted message alongside her public key, A_1.

Note that for the first message Alice still needs to receive Bob's public key, we see later that Bob does not have to be online for this (see Section 3.2.4). Bob receives the message and decrypts it by following the steps:

- Create the symmetric shared secret, $DH(A_1, b_1)$.
- Decrypt the message.

Bob can send Alice a reply by taking the following steps:

- Generate a new DH key pair, (B_2, b_2).
- Create a new symmetric shared secret, $DH(A_1, b_2)$.
- Encrypt the message with the symmetric shared secret.
- Send Alice the encrypted message and his public key, B_2.

The communication continues in this way, so that the key is updated after every message.

In Figure 3.1 the forward and backward secrecy in the DH ratchet can be seen. For example, when Bob's second DH private key b_2 leaks, the shared secrets, for both the second and third messages, are no longer secret. An attacker only has to observe the public keys of Alice and create the same shared secrets. However, knowing b_2 will not help in discovering either the shared secret used to encrypt message 1 or message 4, 5 and onward, because a different key pair is used for those shared secrets.

3.2.3 Authentication in *X3DH*

At this stage, there is nothing in place to prevent an attacker to impersonate Alice and/or Bob and execute, for example, a man-in-the-middle attack. To prevent such impersonations from happening, a form of *authentication* [PM16b] is needed. This way, Alice and Bob are sure that they exchange encryption keys with each other. The authentication issue boils down to the need of verifying each others DH public keys by bounding a key to an identity.

Authentication can be done by verifying each others long-term public key, however, one-time keys are required to keep the forward and backward secrecy. In the next paragraph, we explain how the identity key and the one-time key can

be combined in the Extended Triple Diffie-Hellman Protocol (X3DH) to ensure authentication, deniability and forward secrecy. We end with a paragraph on how to confirm someone's identity key. In Section 3.2.4 we explain how the X3DH protocol is used in a non-interactive. This involves some small adjustments to the X3DH protocol, introduced in this section.

Combining long-term and one-time keys in X3DH The Extended Triple Diffie-Hellman protocol (X3DH) combines the long-term identity key (IK) with the one-time key (OTK), to generate an initial shared secret, which has forward secrecy, authentication and deniability. Alice and Bob will initiate their secure chat with one identity key and one one-time key. In Table 3.1 we see a quick overview of the keys that Alice and Bob both need during the X3DH, and their purpose.

Key pair	Name	Type	Purpose
IK	Identity Key	Long-term	Authentication
OTK	One-Time Key	One-time	Forward secrecy

Tab. 3.1.: The keys both Alice and Bob needs to communicate with each other.

The four keys are combined in the Extended Triple Diffie-Hellman (X3DH) in three different DH shared secrets which can be seen in Figure 3.2, and are later combined to the initial shared key, K_{init}, with a key derivation function (see Section 2.1.5).

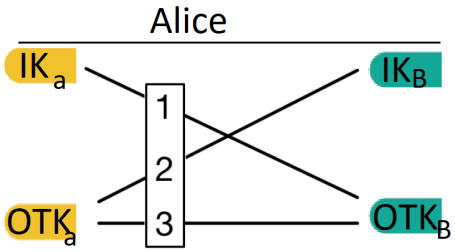

Fig. 3.2.: Alice creates the 3 shared secrets between her private keys and Bob's public key. Bob will create the same shared secrets, but he will use his private keys and Alice's public keys.

The four keys are combined in such a fashion to provide forward secrecy, authentication and deniability. The purpose of each of the DH shared secrets is given in Table 3.2.

There is no shared secret between the identity keys, because that would result in a secret without any forward secrecy. The shared secret between both one-time keys (DH_3) does not involve any authentication. The leak of authentication could have been solved by signing (see Section 2.1) the one-time keys, however, that would result in losing the deniability claim: a user cannot deny that it was

Secret	Alice	Bob	Main purpose
DH_1	IK_a	OTK_B	Authentication of Alice, forward secrecy for Bob
DH_2	OTK_a	IK_B	Authentication of Bob, forward secrecy for Alice
DH_3	OTK_a	OTK_B	Forward secrecy, however, no authentication

Tab. 3.2.: The three DH secrets that are generated in the X3DH protocol, between which keys they are created and their main purpose.

him who sent the key to initiate communication because he signed it. There are possibilities for deniable signatures, but they are complex to compute and thus not suitable for chat [Mar13].

Deniability Doing the triple handshake as in Figure 3.2 allows both users to authenticate each other using their private identity key to create shared secrets DH_1 and DH_2, and still have deniability. Alice can never publish a cryptographic proof that it was Bob, and not herself, who created the shared secrets.

The initial shared key, that Alice and Bob can use to send each other messages, will be a concatenation of the three DH shared secrets which is put through a KDF, as described by the formula:

$$K_{init} = KDF(DH_1||DH_2||DH_3)$$

Verifying the identity key The verification of the identity keys (ID) for both parties *should* be done before the protocol starts, otherwise there is no guaranteed authentication. Eve could create a fake identity key for Bob and Alice may communicate with Eve instead of Bob, without knowing it. The long-term identity key could be verified in real life or using a public key infrastructure, in which we use certificates to verify that someone is who he says he is. The Signal Protocol uses the first “real life verifying” option, and uses a so called “security code” in the application with which users can verify each others identities.

3.2.4 Uploading to a server

The public keys needed in the X3DH protocol are uploaded in advance to a trusted server, so when Alice needs Bob’s keys, she can just contact the server.

In the initial phase of the protocol, before users are able to send each other messages, each user has to generate several Diffie-Hellman key pairs and upload the public part of those keys to the server. A user will generate one identity key

pair, IK, one signed pre-key pair, SPK and 100 one-time pre-key pairs, OTK's, as can be seen in Table 3.3.

Key pair	Type	Number of keys on server	Signed
IK	Long-term	1	No
SPK	Short-term	1	Yes, with IK
OTK	One-time	100	No

Tab. 3.3.: The table shows per users which keys are uploaded to the server, how many, what type, and which one are signed.

The public Diffie-Hellman keys on the server are grouped in *pre-key bundles*, each bundle containing the public identity key IK, the public signed pre-key SPK, one public one-time pre-key OTK and the signature of the SPK. Every time the server is almost out of OTK's, the user has to generate 100 new OTK and one new SPK pair, of which he uploads the public keys to the server. If there are no new OTK keys the protocol can run without the OTK.

Using a pre-key bundle and thus both a SPK and an OTK does not make major changes in how the X3DH protocol works. When Alice receives Bob's pre-key bundle, both the SPK and the OTK are involved in the creation of the encryption key for messages, by creating four DH shared secrets as can be seen in Figure 3.3.

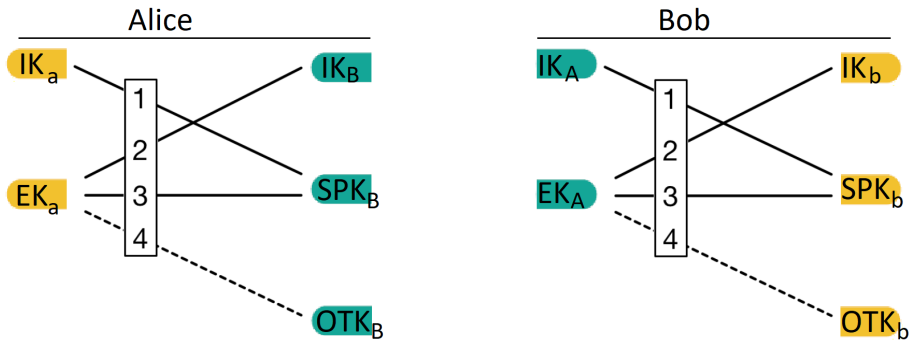

Fig. 3.3.: The figure shows schematically how Alice (left) and Bob (right) create the four shared secrets between them, using both OTK and SPK, if Alice initiated contact. If there is no OTK available DH_4 is not created.

Note that the difference compared to Figure 3.2 is that SPK now creates the third shared secret (DH_3) and the OTK is involved in generating the fourth DH shared secret (DH_4). The four DH secrets are again concatenated and put through a KDF:

$$K_{init} = KDF(DH_1||DH_2||DH_3||DH_4)$$

The other major difference is that Alice's OTK is called an ephemeral key (EK), because that key is the only key that is created during run time and not in the

initialisation phase. If the server would ran out of OTK's the protocol will run without the OTK and without the fourth DH shared secret.

3.2.5 Creating the *Double Ratchet* for *efficiency*

In Section 3.2.3 and 3.2.4 we saw that the X3DH protocol provides a deniable and non-interactive way of authentication. However, having to do the X3DH protocol every time a message is sent is inefficient. To maintain the authentication of the X3DH protocol, but keep the number of generated DH shared secrets low, the *symmetric ratchet* is used. The symmetric ratchet is combined with the *DH ratchet*, into the *Double Ratchet* to have both a new encryption key for each message and authentication.

The symmetric ratchet The symmetric ratchet uses a chain of key derivation functions (see Section 2.1.5), so that the next key will be obtained from the current key [CG+16], as can be seen in Figure 3.4.

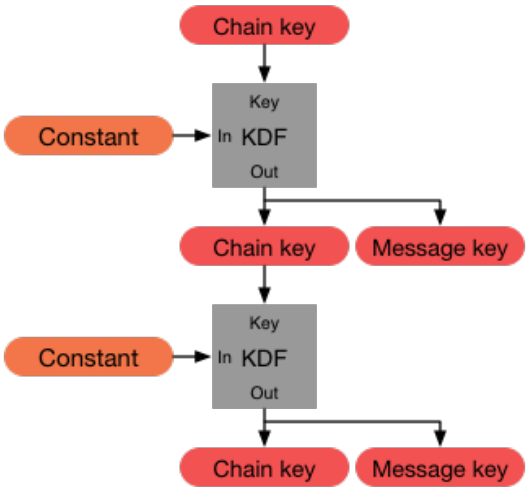

Fig. 3.4.: The symmetric ratchet in which a lot of KDF functions are chained. The figure is taken from [PM16a].

The KDF will output a message key, which is used to encrypt or decrypt a message, and a new chain key, which is used as input for the next KDF. The output is split, so that if an adversary gets hold on a message key, he can still not use that as an input to the KDF to generate future message keys.

The symmetric ratchet has two input values, the chain key and a constant value. The chain key is obtained from the previous KDF output. For the initial chain

key the key created in the X3DH protocol is used. Therefore the authentication flows through the rest of the keys created by the symmetric ratchet. The one-way property of the KDF gives the symmetric ratchet it's forward secrecy for all those keys.

The constant input to the KDF will be the symmetric shared secret created by the DH ratchet. In this way the symmetric ratchet and the DH ratchet are combined into the Double Ratchet.

The Double Ratchet algorithm The Double Ratchet algorithm combines the symmetric ratchet and the DH ratchet.

To have forward secrecy, backward secrecy, authentication and deniability, the DH ratchet and the symmetric ratchet are combined into the *Double Ratchet* Protocol. The symmetric ratchet will not only use the chain key as input but also the DH shared secret created with the DH ratchet. The resulting Double Ratchet for Alice, can be seen in Figure 3.5. When Alice wants to send Bob a message using the

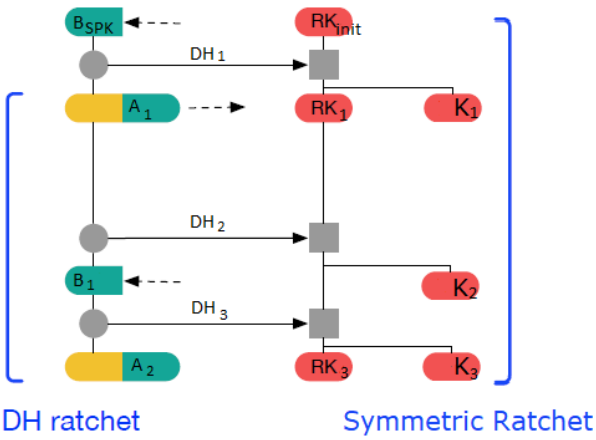

Fig. 3.5.: The figure shows from Alice view, how the DH ratchet and the symmetric ratchet are combined into the Double Ratchet. The figure is an adaptation of an image from [PM16a].

Double Ratchet algorithm, she puts the initial key, K_{init}, obtained from the X3DH protocol, in the symmetric ratchet as RK_{init}. The other input to the symmetric ratchet is the first DH shared secret she created with Bob using the DH ratchet. To keep the Double Ratchet non-interactive she does not need to ask for a DH key from Bob, but she will uses his SPK in combination with her new generated private key a_1, to create the shared secret DH_1. The KDF outputs a message key K_1 and a new root chain key RK_1. The message key K_1, she uses to encrypt the message to Bob, after which Alice sends both the encrypted message and her public key, A_1, for the DH ratchet to Bob.

When Alice receives a reply from Bob, she needs the chain key RK_1 she receives from the symmetric ratchet. To decrypt the message, Alice uses her old private DH key, a_1 and the new DH key Bob just sent her, B_1, to create a shared secret DH_2. She puts DH_2 and the RK_1 into the symmetric ratchet, to get the message key K_2 which she uses to decrypt the message from Bob.

When Alice wants to reply again to Bob, she first generate a new DH key pair, (A_2, a_2), generates a new DH shared secret (DH_3), put this together with RK_2 into the symmetric ratchet and receives a new sending key K_3. They will again continue to use the Double Ratchet in this fashion to communicate.

We summarise the utility of the Double Ratchet algorithm. The Double Ratchet is used to update the message key every message. The message keys have forward and backward secrecy, because of the combination of the symmetric ratchet and the DH ratchet. Because the Double Ratchet get's an initial chain key from the X3DH protocol, the authentication flows through the rest of the keys.

3.3 The Signal Protocol in a nutshell

All the building blocks from Section 3.2 can be combined into the Signal Protocol. The Signal Protocol then has end-to-end encryption, forward and backward secrecy, authentication, deniability and is non-interactive. Both the X3DH and Double Ratchet together give the Signal Protocol its end-to-end encryption and forward and backward secrecy. The X3DH protocol is responsible for making the Signal Protocol non-interactive, deniable and authenticated, while these properties of course flow trough the rest of the Signal Protocol.

The Signal Protocol will roughly consists of 3 phases (in the case that Alice initiates the contact):

1. Initial setup: uploading the keys to the server.
2. The first message: the first message from Alice to Bob.
3. Message exchange and key update: during normal chat, renew the key for all messages, after initial contact was made.

The first two phase are done by the X3DH protocol, the *message exchange and key update* phase is done by the Double Ratchet algorithm. We explain each phase in the following three sections.

3.3.1 Phase 1 - Initial setup

The first phase in the Signal Protocol is the same as the initial phase of the X3DH protocol as described in Section 3.2.4. Alice and Bob both upload 100 key bundles to the server, containing the public identity key IK, the public signed pre-key SPK, one public one-time pre-key OTK and the signature of the SPK. This phase is shown in Figure 3.6.

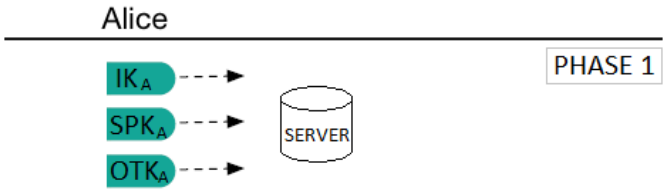

Fig. 3.6.: Phase 1 of the Signal Protocol. In phase 1 the users upload their pre-key bundles to the server.

3.3.2 Phase 2 - The first message

In phase 2, Alice request the pre-key bundle from Bob and will use this to send him the first message, as can be seen in Figure 3.7. She will follow the steps from the X3DH protocol, to create the initial key, K_{init}. She puts K_{init} in the Double Ratchet, together with the shared secret created between her new generated private key a_1 and Bob's SPK, to create the first sending key, K_1, and the root key for the chain, RK_1. She uses K_1 to encrypt a message to Bob and sends Bob the keys he need to decrypt the message: IK_A, EK_A and A_1, together with the encrypted message. Bob will decrypt the message, as soon as he appears online, following the same X3DH as Alice.

3.3.3 Phase 3 - Message exchange and key update

In the third phase Alice and Bob just have a normal chat conversation. For Bob to reply to Alice he will follow the steps:

- Bob creates a new DH key pair (B_1, b_1).
- He generates the shared secret between his new generated private key b_1 and Alice's public key A_1.
- He puts his chain key, RK_1, and the shared secret in the KDF.
- The KDF will output a new chain key RK_2 and the encryption key, K_2 for Bob.
- Bob encrypts the message with K_2.

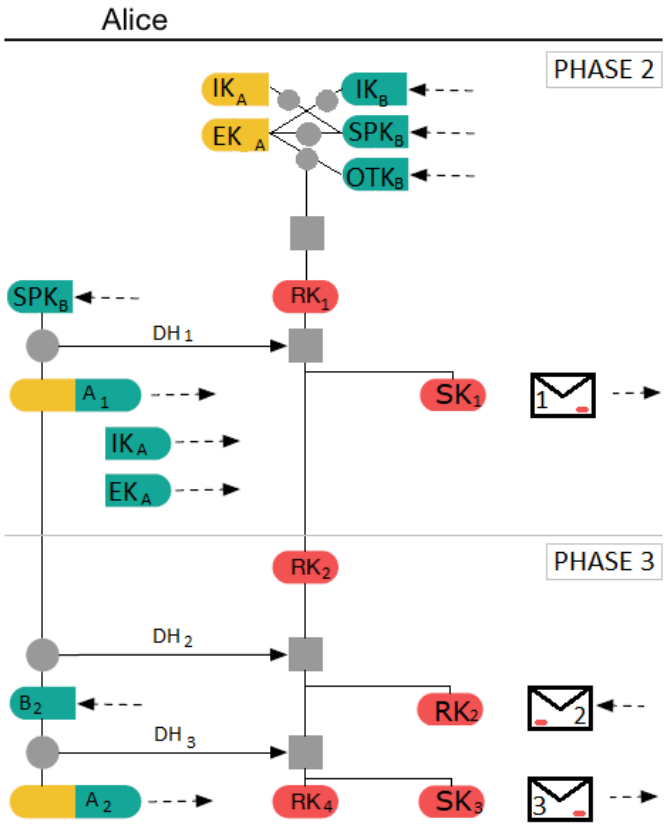

Fig. 3.7.: Phase 2 and 3 of the Signal Protocol. In Phase 2 Alice initiates the first contact between her and Bob, by following the X3DH protocol and sending him a message. Phase 3 of the Signal Protocol in which Bob replies on the Alice initial message and they continue chatting using the Double Ratchet of the Signal Protocol. The figure is an adaptation of an image from [PM16a].

- Bob sends Alice both the encrypted message and his new public key B_1.

Alice receives the message from Bob and his public key. She can decrypt the message by taking the following steps, as can be seen in Figure 3.7:

- Alice creates the same shared secret with a_1 and B_1.
- She put both that shared secret and the RK_1 into the KDF.
- The KDF outputs the decryption key K_2 and the new root key RK_2.

Alice does not have to reply immediately, but when she does she can take the following steps to send a rely to Bob, as can be seen in Figure 3.7:

- Alice generates a new DH key pair (A_2, a_2).
- She create a shared secret between her a_2 and Bob's B_1
- She puts both the shared secret and the RK_2 in the KDF.
- The KDF outputs the encryption key K_3 and chain key RK_3.
- She encrypts the message with K_3.

- Alice sends both the encrypted message and A_2 to Bob

3.4 More implementation choices

In this section, we describe two other implementation choices in the Signal Protocol that make it a better chat application. In Section 3.4.1 we explain how the Signal Protocol handles multiple messages and how the key can be updated, even when no new public key is received. In Section 3.4.2 we describe the Sesame protocol, a part of the Signal Protocol that acts on situations which could happen during chatting like: out of order messages, accidentally deleted encrypt/decrypt keys and multiple devices.

3.4.1 Sending multiple message

People usually send multiple messages to a user before that user replies. However, for simplicity we assume these multiple messages to be one message, as can be seen in Figure 3.8. The Signal Protocol will, however, create a new key for each messages, but not with a Double Ratchet step.

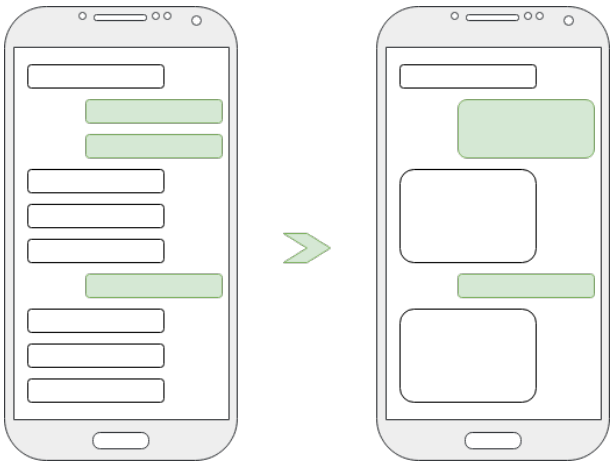

Fig. 3.8.: While we see multiple messages in chats, for simplicity we can assume those messages are one and encrypted with the same key until the other party responses.

The Signal Protocol will generate a new key for each of these messages, but because key renewal requires interaction the Double Ratchet cannot be used for this. Instead the Signal Protocol will use KDF functions to update the key for each message. Actually three chains are used in the Signal Protocol: a root chain, a sending chain and a receiving chain. The encryption and decryption keys (called sending and receiving keys), K_x, from the root chain as described

in Section 3.2.5, are used as input keys for the sending chain and the receiving chain. Those sending and receiving chain keys will be put again through a KDF function, creating a respectively sending chain key and sending key or receiving chain key and receiving key.

Simply only using a KDF instead of three chains will result in forward secrecy, but splitting the output of the KDF into a chain key and a sending/receiving key will prevent an adversary to directly use an obtained sending/receiving key to generate all the future encryption keys for that message series himself. As soon as the receiving party replies, the Signal Protocol will start a new Double Ratchet step, and a new DH key will be used to update the root chain key.

3.4.2 Out of order messages in the sesame algorithm

Syncing keys and messages during a chat could be a problem; a lot of things could go wrong, luckily the Sesame algorithm, which is also a part of the Signal Protocol, manages message encryption sessions in an asynchronous setting, to handle the problems that may occur in the practical context of using the Signal protocol [PM17]. Some problems that could occur are:

- Users have multiple devices and add and remove devices. These devices need new keys or their keys need to become invalid.
- Users delete all their keys or sessions. What happens to already encrypted messages?
- Users initiate new sessions at the same time, something the Double Ratchet does not cover. Which key is chosen as the main key?
- Messages get lost or arrive out-of-order. How to make sure which encryption key belongs to which decryption key, and vice versa.
- Adversaries compromise devices or interfere with the communication.

The Sesame algorithm makes it possible that everything still works, by making sure that every device is keeping track of "active" sessions for every device it is communication with and making sure that these active sessions are kept up-to-date. The working of this algorithm is out of scope for this paper, for more details about this algorithms refer to the paper by Perrin et al. about the Sesame algorithm [PM17].

4 A Post-Quantum Signal Protocol

In the previous section we analysed how the Signal Protocol works and which security properties it has. In this section, we analyse which part of the Signal Protocol is not quantum-safe and we discuss how to create a post-quantum Signal Protocol theoretically.

In Section 4.1 we explain which part of the Signal Protocol should be substituted for a post-quantum alternative, to create a post-quantum Signal Protocol. In Section 4.2 we see the challenges which we might face with the post-quantum substitutes. In Section 4.3 we introduce the hybrid post-quantum Signal Protocol, which deals with part of these challenges and in Section 4.4 we elaborate on possible partially hybrid post-quantum Signal Protocol.

4.1 The Post-Quantum Signal Protocol

In this section, we discuss how to create a post-quantum Signal Protocol, by analysing which parts of the Signal Protocol need to be substitute by a post-quantum alternative. We can see the not quantum-safe Signal Protocol in Figure 4.1.

The Signal Protocol consist of two major parts: the initial X3DH key exchange and the Double Ratchet for message exchange and the key update. The cryptographic primitives used in the protocol are: key exchanges, key derivation functions (KDFs), signature schemes and symmetric encryption. Each primitive uses a specific algorithm in the Signal Protocol, these algorithms are shown in Table 4.1.

As explained in Section 2.3, only *ECDH* Curve25519 is directly threatened by quantum computers, while *AES* and *SHA* only need to adjust their parameter sets. Every *ECDH* key exchange in the Signal Protocol will need a post-quantum substitute, so those the *Curve25519* based signature scheme for the signed-pre-key. The symmetric encryption scheme *AES* could need a larger key, and the KDF function based on SHA-512 might need a higher standard as well. However, this is out of the scope for this thesis, as explained in Section 2.4.1. In this thesis we

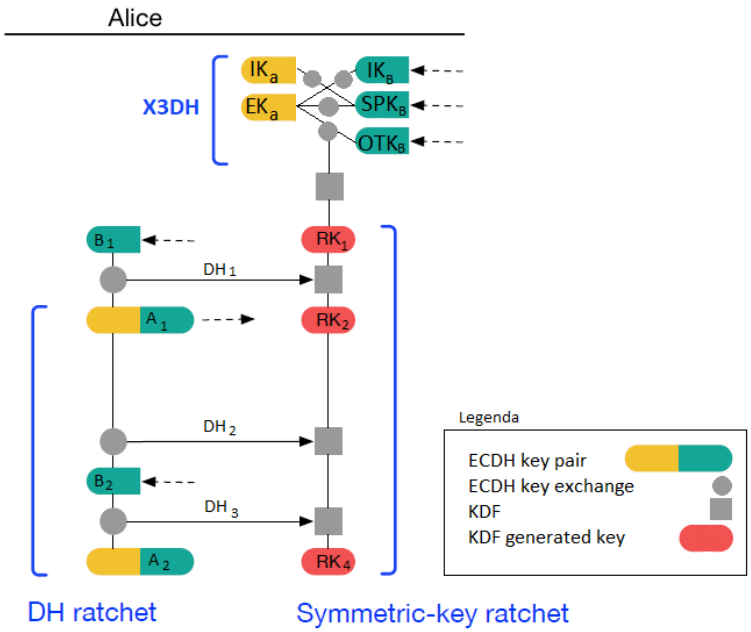

Fig. 4.1.: The Signal Protocol with the X3DH and Double Ratchet algorithm, which both use the ECDH, which is not a quantum-safe solution. The figure is an adaptation from [PM16a; PM16b].

Part in Signal Protocol	Cryptographic primitives	Algorithm used
X3DH	Key exchange	ECDH, with Curve25519
	KDF	HKDF, with SHA-512
	Signature	XEdDSA, with Curve25519
Double Ratchet	Key exchange	ECDH, with Curve25519
	KDF	HKDF, with SHA-512
	Encryption	AES-256 in CBC mode

Tab. 4.1.: The different cryptographic primitives in each part of the Signal Protocol and the algorithm they use [PM16a; PM16b].

focus solely on finding a post-quantum substitute for the *ECDH* key exchanges in the Signal Protocol. The post-quantum Signal Protocol, with an alternative for every ECDH key exchange, can be seen in Figure 4.2.

The post-quantum Signal Protocol has all the security properties: end-to-end encryption, authentication, forward secrecy, backward secrecy and deniability, in a quantum world. However, as we will see in the next two sections, implementing it like this might be not so easy and unwise to do because of security reasons.

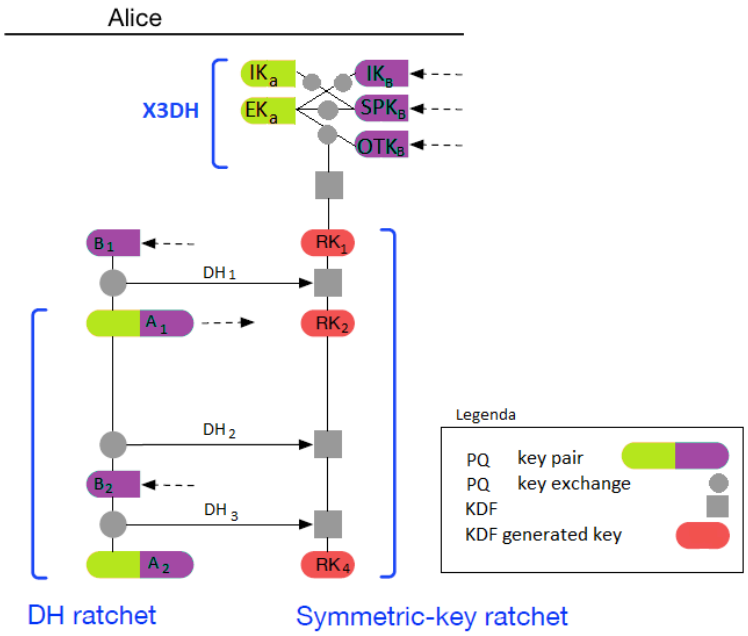

Fig. 4.2.: The post-quantum Signal Protocol with the X3DH and Double Ratchet algorithm, both with a post-quantum DH alternative.

4.2 Challenges with Post-Quantum cryptography

For the post-quantum Signal Protocol an alternative post-quantum ECDH is needed. Unfortunately, there are a lot of different possible post-quantum algorithms and challenges when implementing them, such as:

- Their novelty
- Their larger key size
- Their computational difficulty
- A different way of key exchange

These challenges do not limit themselves to applications in only the Signal Protocol, however, in this thesis we focus on the Signal Protocol. Not all challenges apply to all post-quantum algorithms.

Novel algorithms Most post-quantum algorithms are still in development and thus have a higher chance of containing undiscovered vulnerabilities. The National Institute of Standards and Technology (NIST) is currently in the process of standardising (multiple) post-quantum algorithms (see Section 2.4.1). In this standardisation process 69 post-quantum algorithms are analysed, vulnerabilities

are found and the algorithms are improved, and this will lead to less undiscovered vulnerabilities.

Large key sizes Most post-quantum algorithms use keys with a larger size than the current used keys for a Elliptic Curve Diffie-Hellman key. An increased key size might leaves challenges for storing the private keys on the mobile phone or the public keys on the server, and overhead when sending the keys to other users and the server.

For some algorithms the public keys are small, but the private keys are relatively large, or vise versa. Somehow, either the user has to store large private keys, or the large public keys have to be send to and stored on the server.

Computational difficulty Most post-quantum algorithms require more difficult computational. This computational difficulty results in longer tun times, higher CPU usages and more energy consumption.

The Signal Protocol mostly is used on a mobile phone, and the process capacity of a mobile phone might be different than the capacity on which the post-quantum algorithms were tested. The CPU capacity might be lower then on a computer, resulting in a longer run time. A longer run time might cause a higher energy consumption, which influences the battery life of a mobile phone. A longer run time might also increase the time a user has to wait, which influence the user experience. In a chat application the delay should be kept as small as possible. In Section 5.5 we will look into how people chat and which delays are noticeable by the user.

Different way of key exchange Some post-quantum public key encryption schemes, like some lattice-based schemes, require interaction during key exchanges. This means that having the public key of Bob, will not allow Alice to create a shared secret. Using a interactive key exchange, Alice generates a shared secret with an error. To resolve the error, she has to have an interaction with Bob to make sure they both created the same shared secret. This would mean that both chatting parties should be online when they agree on a shared secret or key. These post-quantum algorithms are not optimal for the non-inter-activeness of the protocol and thus should either not be used, or should be worked around. The last option is possible for the Double Ratchet, in which new key material could be send one message before the key is actually needed. For the X3DH protocol this is not a solution, and a non-interactive key exchange is required.

A lot of post-quantum submissions in the standardisation process of NIST (Section 5.3.4) are key encapsulation mechanisms (KEMs) (Section 2.1.4). However, these KEMs are not a perfect plug-and-play for *ECDH*, as we will explain in Section 5.3.1. Because the shared secret is not created by using both parties key material, one user decides on the shared secret and the other one receives this secret, they are not suitable to implement in a post-quantum X3DH protocol.

4.3 Hybrid Post-Quantum Signal Protocol

In this section, we explain how a hybrid post-quantum Signal Protocol will look, and that it solves the first challenge named in the previous section: the novelty of post-quantum algorithms. A hybrid post-quantum Signal Protocol has both ECDH and a post-quantum algorithm, as explained in Section 2.4.3. In this way there are less security risks, if the post-quantum algorithm holds undiscovered vulnerabilities. In the Hybrid post-quantum Signal Protocol, when the post-quantum algorithm turns out to be not secure against a classical or quantum computer, your data is at least as secure as it was with ECDH.

In a fully hybrid Signal Protocol we run a X3DH protocol and a post-quantum X3DH protocol simultaneously and put both outputs in a KDF function to generate one key. For the Double Ratchet we run the Double Ratchet as it is with ECDH, together with a post-quantum Double Ratchet, the output is again put in a KDF function to get one key, which can be put in the root chain. This hybrid post-quantum Signal Protocol can be seen in Figure 4.3.

The Signal Protocol is secure in a classical world, however, adding post-quantum building blocks might deteriorate the security of the ECDH Signal Protocol. The security of the Signal Protocol can be described by using the Universally Composability Security Framework, see Section 2.4.4. The cryptographic primitives that create the Signal Protocol are:

- ECDH key exchanges
- Key derivation functions
- Symmetric encryption
- Signing of the SPK

Ralf Küsters and Daniel Raush [KR17] proved that all these primitives fall in the Universally Composability framework, and thus that they remain secure when combined in the Signal Protocol. Adding post-quantum blocks to the Signal Protocol, will again not remove the security the initial blocks had in the UC

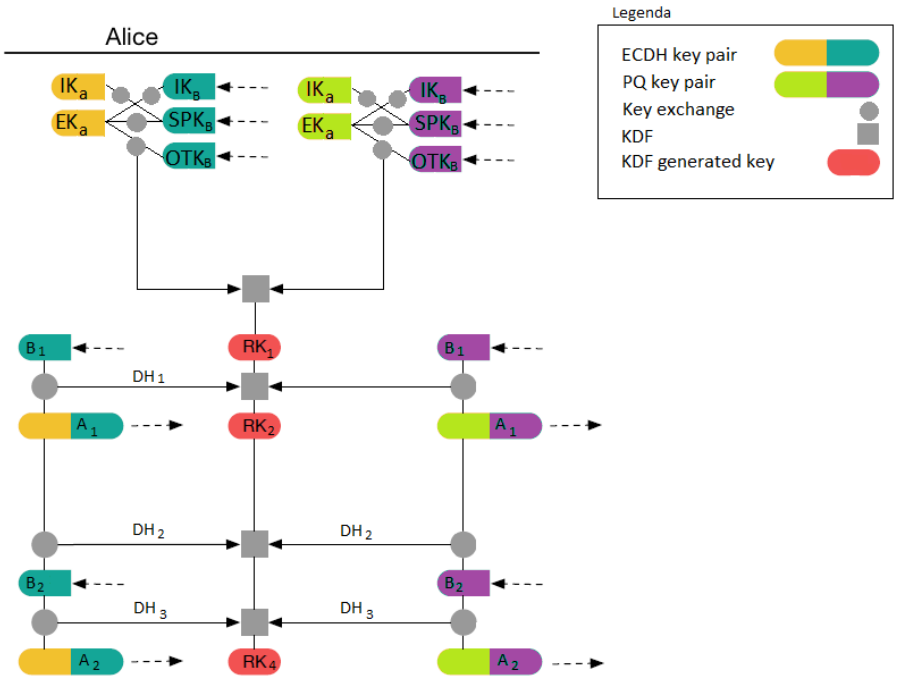

Fig. 4.3.: The hybrid Signal Protocol, which both uses a post-quantum algorithm and the ECDH. The shared secrets created by both are combined with in the symmetric ratchet.

framework. More research is done about fitting post-quantum protocols in the UC framework, however, analysing and explaining that research is out of the scope of this thesis. See the work of Vajda [Vaj17] and Unruh [Unr10] for more details about the UC framework.

4.4 Partially hybrid post-quantum Signal Protocol

In this section, we look at how to create a partially hybrid post-quantum Signal Protocol. This protocol might solve two challenges post-quantum algorithms have, namely: large key sized and computational difficultly, as discussed in Section 4.2. Because of the challenges it might not yet be possible to implement the hybrid post-quantum Signal Protocols. Implementing a partially hybrid Signal Protocol is a solution for the transitional period, to have security against a passive quantum attack right away (see Section 2.4.3).

There are different partially hybrid post-quantum Signal Protocols possible. In these partially hybrid post-quantum Signal Protocols only some ECDH key exchanges are substituted for the post-quantum alternative. Roughly, we see two mayor parts which could be made post-quantum: the X3DH protocol and the Dou-

ble Ratchet. As an alternative for a post-quantum X3DH, we will evaluate a third option in which one building block is added: a post-quantum key exchange.

We elaborate on the influence these three options have on the security properties of the Signal Protocol (Section 3.1) in Section 4.4.2, 4.4.3 and 4.4.4. We then combine the three options in other possible partially hybrid post-quantum Signal Protocols, and again look at the impact on the security properties (Section 4.4.5). To understand the impact of the partially hybrid post-quantum Signal Protocols on the security properties, we have to define the current key, which we do first.

4.4.1 Current key

The current keys are the keys the user has on his phone at a certain moment x. For example, in Figure 4.1, if Alice's current keys are leaked right after the second root key, RK_2, then she leaks:

- her private identity key IK_a,
- her current private Double Ratchet key a_1,
- the root key RK_2,
- the key which which messages are encrypted (not shown in the image).

The keys Alice generates after moment x are not leaked.

With this definition we can look at the impact the partially hybrid post-quantum Signal Protocol have on the forward and backward secrecy.

4.4.2 Post-quantum X3DH

The Signal Protocol with a post-quantum X3DH can be seen in Figure 4.4. This partially post-quantum Signal Protocol will have, in a quantum world, authentication, forward secrecy and deniability. Authentication and deniability due to of structure of the X3DH, and forward secrecy because of the KDFs applied to every used key (see Section 3.2.3). Even though, an adversary can break all the *ECDH* key exchanges, there is still end-to-end encryption due to the post-quantum keys.

There is no backward secrecy in a quantum world. Once the current keys are leaked, the adversary can use the root chain key and a quantum computer to compute all *ECDH* shared secret to obtain all future encryption keys.

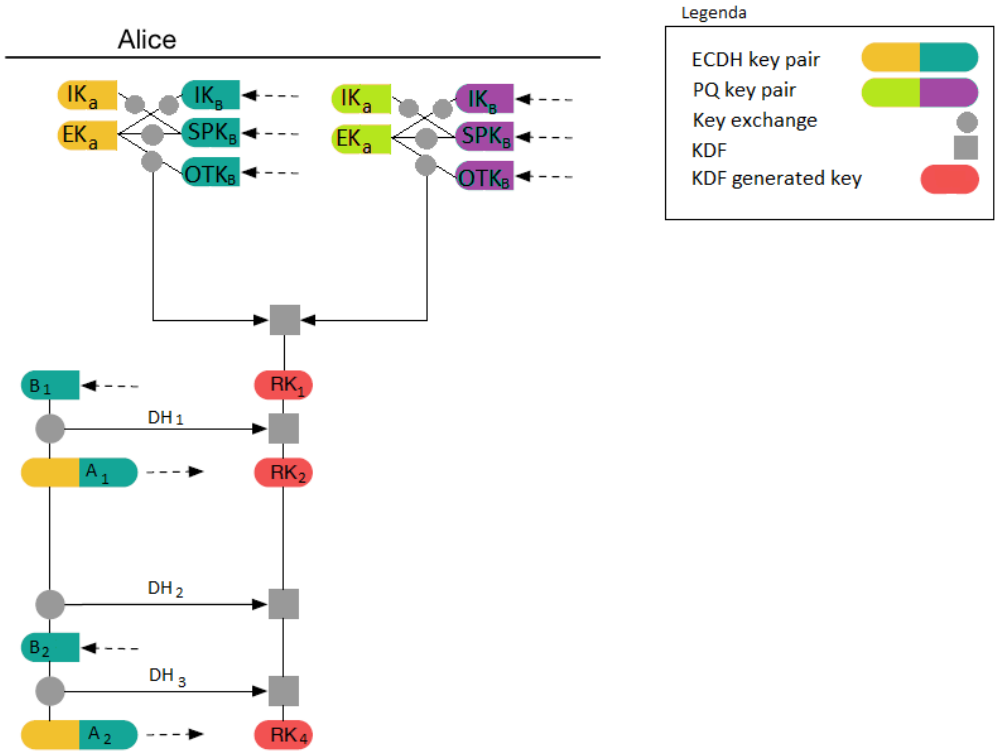

Fig. 4.4.: The Signal Protocol with a Hybrid X3DH protocol.

4.4.3 Post-quantum Double Ratchet

The Signal Protocol with a post-quantum Double Ratchet algorithm can be seen in Figure 4.5. A post-quantum Double Ratchet will have backward secrecy and forward secrecy. Backward secrecy, because you get a new post-quantum key every message. Even when the current keys are leaked, the new post-quantum keys can not be derived from those leaked keys.

Because of the *ECDH* X3DH, some properties are not present in this version of a partially hybrid post-quantum Signal Protocol. The root chain is initialised with an by *ECDH* keys created initial key, which result in no authentication and no deniability. There is also no forward secrecy until the first post-quantum Double Ratchet key is created.

4.4.4 Extra key exchange

The Signal Protocol with an extra key exchange can be seen in Figure 4.6. This exchange can either be an identity key (PQ ID) or an one-time key (PQ OTK), both with their own benefit.

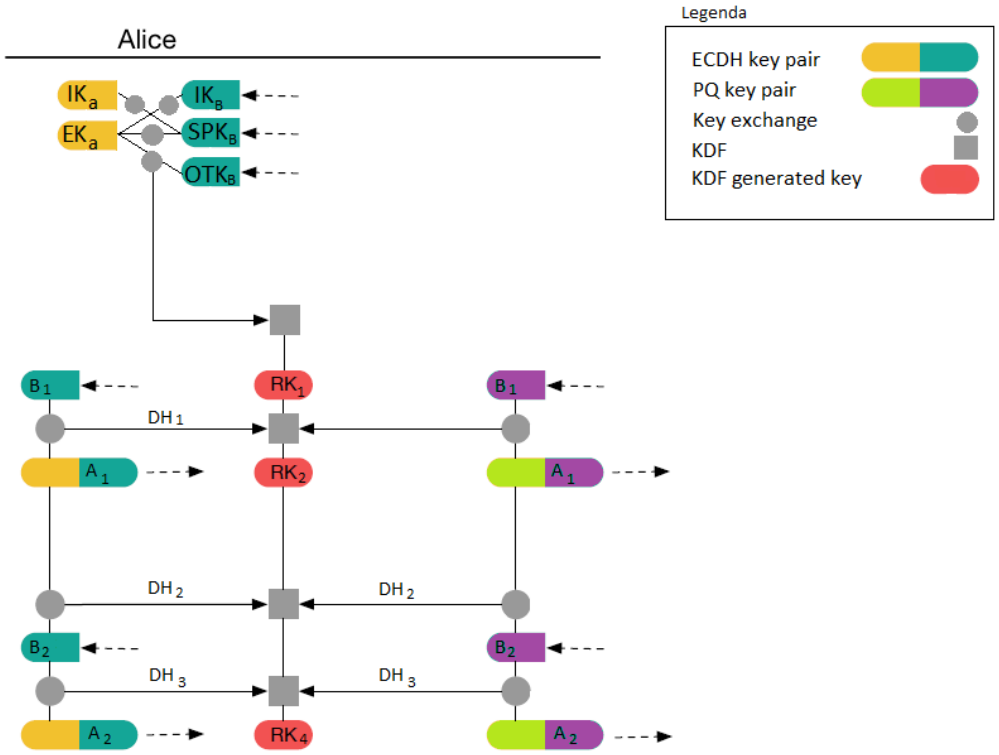

Fig. 4.5.: The Signal Protocol with a hybrid Double Ratchet algorithm.

Extra PQ DH - ID Only adding a post-quantum identity key, instead of a complete post-quantum X3DH protocol, will result in a partially hyrbid post-quantum Signal Protocol with authentication. All the other keys (in the X3DH and Double Ratchet) could be compromised, and there is no guarantee that the other keys are created by the authenticated user. There is no forward or backward secrecy if the current keys are leaked. The post-quantum identity key will be used long term, and thus if a phone is compromised the post-quantum identity key is obtained and an adversary with a quantum computer can obtain all the other keys in the protocol.

Extra PQ DH - OTK Adding a post-quantum one-time key will give forward secrecy and deniability. Deniability because there is no authentication and thus you can deny any involvement in the communication. If the phone is compromised and the current keys are leaked, there is still forward secrecy because of the key derivation function, which is still secure in a quantum world. There is no way to go from the current key to a previous key. However, because the ECDH keys can be obtained by an adversary with a quantum computer, there is no backward secrecy after the current keys are leaked.

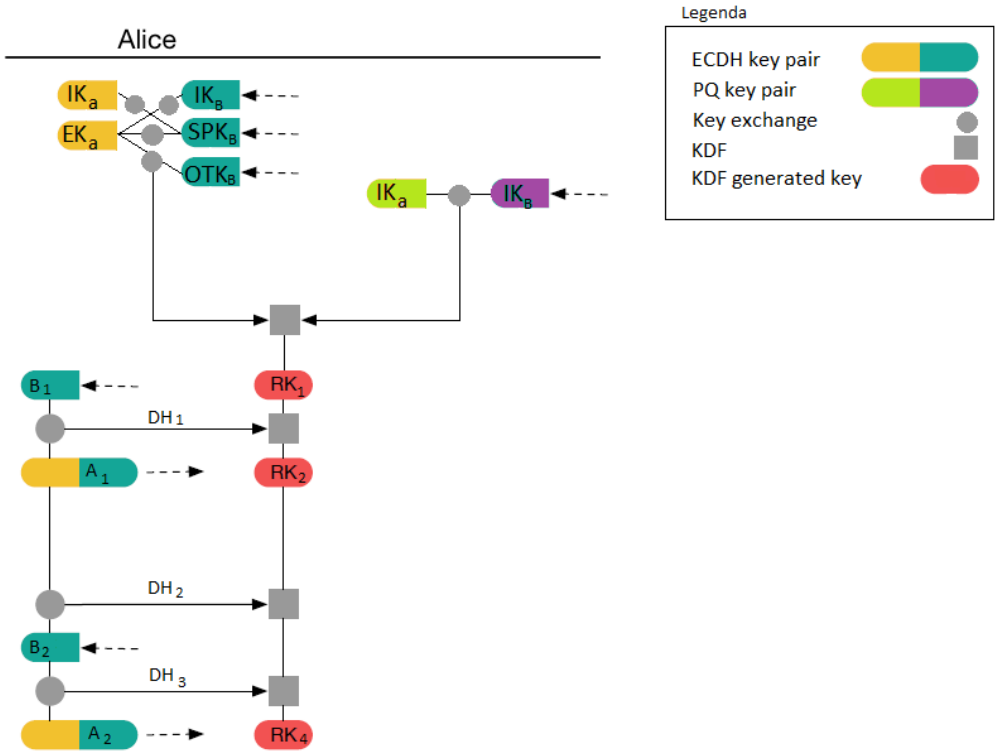

Fig. 4.6.: The Signal Protocol with an extra post-quantum key exchange.

4.4.5 Combining the different hybrid blocks

These four partially Signal Protocols can be combined into other partially Hybrid Signal Protocols. We create a total of nine different partially hybrid post-quantum protocols and their security properties in a quantum world, as seen in Table 4.2. Protocol 1 is the trivial hybrid post-quantum Signal Protocol as elaborated in Section 4.3.

Not all possible combinations are shown, because some combinations are not useful. For example, having a post-quantum X3DH and an extra PQ Diffie-Hellman key exchange would add no extra security benefit, so this combination is not shown in the table.

All partially hybrid post-quantum Signal Protocols, from Table 4.2, are not a perfect alternative for the Signal Protocol with *ECDH* in a Quantum world. Each partially protocol miss a certain security property. There is no authentication in a quantum world if there is not a post-quantum identity key involved in the protocol. The protocol has no forward secrecy if there is not at least one post-quantum one-time key. There is no backward secrecy secrecy if there are no new post-quantum keys introduced after a current key leak. Adding an extra post-quantum identity key, will result in no deniability in a quantum world. All

		Hybrid X3DH	Hybrid DR	Extra ID	Extra OTK	Authentication	Forward secrecy	Backward secrecy	Deniability
1	Hybrid p. p.q. Signal Protocol	√	√			√	√	√	√
2	Hybrid X3DH	√				√	√		√
3	Hybrid DR		√				√	√	√
4	Extra ID			√		√			
5	Extra OTK				√		√		√
6	Extra ID and OTK			√	√	√	√		
7	Hybrid DR and extra ID		√	√		√	√	√	
8	Hybrid DR and extra OTK		√		√		√	√	√
9	Hybrid DR and extra ID and OTK		√	√	√	√	√	√	

Tab. 4.2.: Different versions of a partially post-quantum Signal Protocol. Each version can have a hybrid post-quantum X3DH, a hybrid post-quantum Double Ratchet or/and an extra post-quantum DH key exchange, either ID or OTK. The nine partially protocols either have a certain building block and a certain security property, indicated by √, or not, indicated by the lack of the symbol.

the partially hybrid post-quantum Signal Protocols have end-to-end encryption, because no adversary or server can read the messages send between Alice and Bob. At least one post-quantum key will prevent an adversary, with a quantum computer, from doing that.

Although these partially hybrid post-quantum Signal Protocols miss some security properties in a quantum world, they might be useful for the transitional period.

A simple and easy to implement solution for the transitional period would be protocol 3 or 4, in which we only add one post-quantum key exchange in the beginning of the protocol. A passive quantum attacker can never use a quantum computer to decrypt all the messages in such a protocol, unless he steals the key or the post-quantum algorithm turns out to be not secure. Wire, a chat application which has the Signal Protocol implemented, created a partially hybrid post-quantum Signal Protocol like protocol 4, with NewHope [RA18].

The protocol with a hybrid post-quantum X3DH would be also a nice option for the transitional period. The extra computational work is only needed in the beginning of the protocol and only the 102 private post-quantum keys need storage: one identity key, one signed-pre-key and hundred one-time-keys.

The choice for a hybrid post-quantum Double Ratchet, and no hybrid post-quantum X3DH, is rather odd. This will result in a protocol in which you might

have to do an expensive post-quantum calculation every message, but have no authentication. As we will see in Section 6, the computational power to create a new shared secret each message, has more impact then the few secrets you might have to do when initiate the first contact with a new user.

We conclude that the best solution for the transitional period would be a partially hybrid post-quantum Signal Protocol in which at least one initial key exchange is done with a post-quantum algorithm.

The Signal Protocol with only ECDH was already vulnerable in a Quantum world. Making a partially hybrid protocol from it did not deteriorate the security of the Signal Protocol, even when the post-quantum algorithm turns out to be not secure. In a classical world the ECDH key exchanges were secure in the UC framework and still are when the protocol is made hybrid (See Section 4.3).

5 Method

Section 3 explained how the Signal Protocol works and the reason behind the design choices, which mainly have to do with security properties. In Section 4 the parts of the protocol that should and could be substituted with a post-quantum algorithm, and the possible challenges which are faced doing that, were described.

In this section, the previous obtained information is used to create a method to test if a post-quantum Signal Protocol is possible. In Section 5.1 the research questions are restated. In Section 5.2 three scenario's, for each phase of the Signal Protocol one, are created to test the research questions. Section 5.3 describes the post-quantum algorithms that will be tested and compared in the Signal Protocol. In Section 5.4 the created code and the machine which is used for testing is described. Section 5.5 defines the average chat user with his minimal phone. This information is used to place the results from the tested method in context for an average user in 2018.

5.1 Research questions

The main question in this thesis is: "Considering the status of 2018, is it possible to have a usable post-quantum Signal Protocol implemented in a chat application?" This main question can be divided in the following sub questions, which can be tested:

1. How many CPU cycles does [algorithm] need to add a contact (doing one X3DH key exchange).
2. How many CPU cycles does [algorithm] need to send and receive a message (doing one Double Ratchet step).
3. How much storage space does [algorithm] require?
4. How much does the bandwidth increase when sending the keys of [algorithm]?
5. How much energy does the algorithm use?

And with every question the sub part: "And is this feasible for an average user, with a minimal phone and bandwidth?", should be added.

5.2 The scenarios

To answer the research questions the Signal Protocol is tested with different PQ algorithms. Three different scenarios of the protocol, one scenario per phase in the Signal Protocol: the initial scenario, in which users upload keys; the X3DH Scenario, used to add a new contact; and the Double Ratchet Scenario, used during *normal* chatting, are tested. The initial scenario is based on the event in which a user has to create and upload keys to the server. The X3DH scenario is based on the use case "Alice wants to make contact with Bob, but they never had contact before", which translates to a scenario in which "Alice's phone follows the X3DH steps to initiate contact with Bob's phone". The DR scenario is based on the case "Alice is sending message to and receiving message from Bob, after the initial contact was made", which translate to the scenario in which "Alice's phones encrypts and decrypts ciphertext to send and receive shared secrets which can be used to encrypt and decrypt messages to and from Bob". Both scenarios are done for the initiating user, Alice, and explained in the following sub-sections.

The initial scenario In the initial scenario Alice will create her 100 pre-key bundles and uploads them to the server. She will:

- Create her IK
- Create her SPK
- Create 100 OTK's
- Send the bundles to the server

Note that Alice only has to create one identity key; however, for this scenario we assume she will create a new IK every time she follow this scenario.

The X3DH scenario In the X3DH scenarios Alice wants to initiate contact with one or more contacts. She will take the following steps:

- Receive the pre-key bundle from the contact
- Create one EK per contact
- Create the DH_1, DH_2 and DH_3 shared secret per contact
- Send her public EK and IK to her contact.

The prerequisites are that Alice uploaded her own pre-key bundles and has the pre-key bundle for every user she wants initiate contact with. The variable in the X3DH scenario is the number of contacts, which may vary.

The Double Ratchet scenario In the Double Ratchet scenario Alice is chatting with one contact (Bob), in which they send each other x messages. With *ECDH* and SIDH she takes the following steps, sending message i and receiving message $i+1$:

- Send message i
  Create a new key pair A_i
  Generate s shared secret SS_i, between B_i and A_i.
  Send the public A_i key to Bob
- Receive message $i+1$, with B_{1+1}
  Generate s shared secret SS_{i+1}, between B_{1+1} and A_i.

If a key encapsulation mechanism (KEM) is used the steps are defined different:

- Send message i
  Encapsulate a shared secret SS_i, with B_i, in a ciphertext C_i
  Create a new key pair A_{i+1}
  Sends the public A_{i+1} key to Bob, together with the ciphertext C_i
- Receive message $i+1$, with C_{i+1} (and B_{1+1})
  Decapsulate new shared secret SS_{i+1} from received ciphertext C_{i+1}

For more details about why a KEM works different and how the algorithm should work will be explained in Section 5.3.1.

A prerequisite is that Alice already has received a public key from Bob B_i. Note that it is assumed that for every message Alice sends, she will receive a message back from Bob, see Section 3.4.1 for more details on how multiple message from the same user are encrypted with the Signal Protocol. The Double Ratchet scenario only depends on the number of messages, which may vary.

5.3 The post-quantum cryptographic algorithms

This section explains which post-quantum algorithms are tested in the Signal Protocol and why. There are 11 different post-quantum algorithms tested: *SIDH*, and 10 key encapsulation mechanism (KEMs): *Big Quake, BIKE, Frodo, Crystal-Kyber, Leda, Lima, New Hope, Saber, SIKE and Titanium*. The parameters of most

post-quantum algorithms can be changed to change the security level of that algorithm. All post-quantum algorithms are compared with each other and with *ECDH Curve25519*, which is currenlty used in the Signal Protocol.

First, it is explain why mostly post-quantum KEMs were chosen to substitute *ECDH* and why KEMs are not the perfect solution (Section 5.3.1). Then both *sihd* (Section 5.3.2) and the KEMs (Section 5.3.3) are described in more detail. In Section 5.3.4 the security level of the post-quantum algorithms is given.

5.3.1 Substitutes for *ECDH*

For a post-quantum Signal Protocol a perfect post-quantum plug and play substitute for *ECDH* is needed. Supersingular Isogeny Diffie-Hellman, *SIDH*, is such an algorithm (as we will see in Section 5.3.2). Unfortunately there are not a lot of other algorithms who can substitute Diffie-Hellman directly like that. Instead, a KEMs in used in this thesis, and this section explains how KEMs work and why they are a good substitute for *ECDH* in the Double Ratchet algorithm but not for *ECDH* in the X3DH protocol.

Key encapsulation mechanisms (KEMs) are not perfect plug and play substitutes for *ECDH*; however, they can do most of the things *ECDH* can do. KEMs are currently well studied and researched because NIST [Che+16] is working on finding post-quantum standard KEMs. Where *SIDH* and likewise algorithms are more scarce or require interaction.

In a KEM both parties have a public/private key pair, like in *ECDH*. However, Alice will not create the shared secret using Bob's public key and her private key. Alice generates a shared secret herself, not using either her private or Bob's public key, and will use Bob's public key to encapsulate the shared secret in a ciphertext. She will then send Bob the ciphertext. Bob can decapsulate the shared secret with his private key.

KEMs in the Double Ratchet algorithm The basics of the Double Ratchet with a KEM can be seen in Figure 5.1, the symmetric chains are excluded in the image, for simplicity.

Instead of sending the public key and the encrypted message (as we saw in Section 3.2.2 in Figure 3.1), Bob and Alice also have to send the ciphertext which encapsulates the shared secret when using a KEM. In Figure 5.1, Bob does not need Alice's public key, A_1, to get the shared secret, he only needs his private key to encapsulate it. He needs Alice's A_1 to encapsulate a new shared secret that he

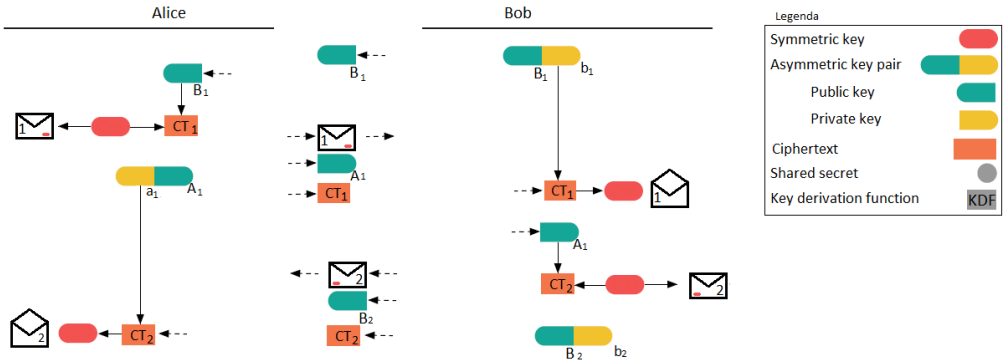

Fig. 5.1.: The simplified Double Ratchet working with a KEM. The symmetric chains are not shown for simplicity. Alice encapsulates the shared secret she used to encrypt the message, in a ciphertext, and sends both the encrypted message, the ciphertext and her new public key to Bob.

used to encrypt message 2, so that Alice on her term can decapsulate the shared secret.

KEMs in the X3DH protocol No interaction is needed to use a KEM. Bob can upload his public keys to a server, so Alice can request them when she needs his key, like with *ECDH*. However, because of the way the X3DH generates three or four shared secrets there is both authentication and deniability. Using a KEM to try to do the same won't result in authentication or deniability.

When using a KEM in the X3DH how can Bob be sure that the key he received from Alice is from Alice? If she just send the ciphertext, there is no way of authentication: Alice can cryptographically deny that it was her who sent the ciphertext, anyone could have sent Bob a ciphertext. Alice could sign the ciphertext, with her private identity key, so Bob can check that it is hers. However, then Bob can proof that the ciphertext came from Alice: and there will be no deniability. Other users can do the same, when the intercept the ciphertext with signature. Alice could sign the shared secret directly with her private identity key; however, that would still allow Bob to proof to others that is was indeed Alice who made contact with him. Again this will lose the deniability. Or Alice could not sign anything, but then this would result in no authentication. There is another solution, Alice and Bob could use deniable MAC signatures. However, for the Signal protocol they specifically did chose to not use deniable MAC signatures *ECDH* [Mar13]. Analysing this further is out of the scope for this thesis.

Interactive key exchanges Another option, instead of KEMs, could be to use an interactive key exchange (IKE). In an IKE, there are again public/private key pairs involved and the key is created based on one party's public and the other parties private key. However, another interaction step is needed before both

shared secrets are the same. This makes the key exchange interactive. Dodis et al. [Dod+09] showed that it is impossible to have a deniable key exchange if the key exchange is interactive. We can thus conclude that the post-quantum Signal Protocol should have a non-interactive key exchange (NIKE) for the X3DH protocol. For the Double Ratchet an IKE could be possible because there is no direct authentication needed; the Double Ratchet gets its authentication and its deniability form the X3DH. To implement an IKE in the Double Ratchet, the shared secret for message i should be done during message $i-1$, as is now done for the KEM public key. Analysing this further is out of the scope of this thesis.

We will not change how the X3DH protocol works, because this is out of the scope of this thesis, and this results in only one post-quantum algorithm suitable to substitute *ECDH*: *SIDH*.

5.3.2 Supersingular isogeny based Diffie-Hellman and *ECDH*

Supersingular isogeny Diffie-Hellman (*SIDH*) is a perfect substitute for *ECDH* (as explained in Section 5.3.1). There are two different versions of *SIDH*, *SIDH503* and *SIDH751*. They differ in security level for each given in the NIST competition (see Section 5.3.4). The *SIDH* algorithms are tested in the X3DH scenario and Double Ratchet scenario.

In Table 5.1 information about the security level and the length of the public key, secret key and shared secret of *ECDH* and its post-quantum substituted *SIDH* is given. The *SIDH* implementation used in this thesis is created by Microsoft[1].

	Public Key (B)	Secret Key (B)	Shared Secret (B)	Q-Security level
ECDH	32	32	32	0
SIDH503	378	32	126	1
SIDH751	564	48	188	3

Tab. 5.1.: The different algorithms used in the X3DH key exchange and their key lengths in Bytes

To compare the post-quantum algorithms to the *normal* Signal Protocol, both scenarios are run with *Curve25519* which is the default *ECDH* cryptography in the Signal Protocol for both X3DH and Double Ratchet. The original open source code of the Signal Protocol was used [Sig18], to have an accurate baseline measurement.

[1] https://github.com/Microsoft/PQCrypto-SIDH

5.3.3 The post-quantum KEMs

There are 10 different post-quantum KEMs tested for the Double Ratchet, these 10 algorithms can have different security levels (see Section 5.3.4) resulting in 42 different post-quantum algorithms and the *ECDH*. In Table 5.2 those 43 algorithms and their type: lattices-based, code-based and isogeny-based, are shown.

The Double Ratchet was tested with different post-quantum Key encapsulation mechanisms (KEMs). The Open Quantum Safe[2] library was used for this because it contains ten different post-quantum KEM's: *Big Quake, BIKE, Frodo, Crystal-Kyber, Leda, Lima, New Hope, Saber, SIKE and Titanium*. The OQS library has different versions available for each KEM, which differ mainly in security level and results in a total of 45 different versions of these algorithms. The Open Quantum Safe library was chosen because it provides a nice set of different post-quantum algorithms in one implementation.

Most algorithms have one version for each security level; however, *Bike* [Ara+17], *LEDAkem* [Bal+17], *LIMA* [Sma17], and *Frodo* [Bos+16a] have multiple. BIKE has 3 different versions, version 1, *bike1*, is based on a variation of McEliece, version 2, *bike2*, on Niederreiter's framework and version 3 follows the work of Ouroboros. Each version has its own advantages and disadvantages. *LEDAkem* has 3 versions with different parameter values, which result in a different balance between performance and public key size. The two different versions of *LIMA* are either based on a "two power" or a "safe prime". Frodo has two different versions because they use either AES or cSHAKE, the AES version is quicker on hardware with specific AES acceleration, while cSHAKE is faster when that hardware acceleration is not available. For more details about each of these four algorithms refer to their documentation (*Bike* [Ara+17], *LEDAkem* [Bal+17], *LIMA* [Sma17], and *Frodo* [Bos+16a]).

5.3.4 The security level of post-quantum cryptography

The post-quantum cryptographic algorithms can be categorised on their security strength, see Section 2.4.2). *ECDH* Curve25517, is not secure against a quantum computer; however, *ECDH* Curve25517, has an almost 128 bit level security against classical computers [PM16b]. The post-quantum cryptography used are submissions for the standardisation process of NIST (see Section 2.4.1). The post-quantum algorithms with NIST security level 1, have a 128 bit level security

[2] https://openquantumsafe.org/

Algorithm name	Public key Length (B)	Secret key Length (B)	Security Level	Type
ECDH	32	32	0	DH
SIKE503	378	434	1	Isogeny
light_saber	672	1568	1	Lattice
kyper512	736	1632	1	Lattice
newhope512	928	1888	1	Lattice
ledac1n02	3480	24	1	Codes
ledac1n03	4688	24	1	Codes
bike1_l1	2542	2542	1	Codes
bike2_l1	2542	2542	1	Codes
bike3_l1	2758	2758	1	Codes
ledac1n04	6408	24	1	Codes
lima_p_1018	6109	9163	1	Lattice
frodo640aes	9616	19872	1	Lattice
frodo640cshake	9616	19872	1	Lattice
titanium_std	16352	16384	1	Lattice
bigquake1	25482	14772	1	Codes
SIKE751	564	644	3	Isogeny
saber_saber	992	2304	3	Lattice
kyper768	1088	2400	3	Lattice
ledac3n02	7200	32	3	Codes
bike1_l3	4964	4964	3	Codes
bike2_l3	4964	4964	3	Codes
ledac3n03	10384	32	3	Codes
bike3_	5422	5422	3	Codes
ledac3n04	13152	32	3	Codes
lima_p_1024	6145	9217	3	Lattice
lima_p_1822	14577	21865	3	Lattice
titanium_med	18272	18304	3	Lattice
frodo976aes	15632	31272	3	Lattice
frodo976cshake	15632	31272	3	Lattice
bigquake3	84132	30860	3	Codes
fire_saber	1312	3040	5	Lattice
kyper1024	1440	3168	5	Lattice
newhope1024	1824	3680	5	Lattice
ledac5n02	12384	40	5	Codes
bike1_l5	8188	8188	5	Codes
bike2_l5	8188	8188	5	Codes
bike3_l5	9034	9034	5	Codes
ledac5n03	18016	40	5	Codes
ledac5n04	22704	40	5	Codes
titanium_hi	20512	20544	5	Lattice
titanium_uper	26912	26944	5	Lattice
bigquake5	149800	41804	5	Codes

Tab. 5.2.: The different tested post-quantum KEMs and their public and private key length, security level and type.

against a quantum computer, and are relatively as strong against a quantum computer as *ECDH* Curve25517 is against a classical computer. The level 3 (198 bits) and 5 (256 bits) PQ algorithms have a relatively higher level of security, and although they may be slower than *ECDH* Curve25517, they are also more secure.

Currently a 256 bit security against quantum computers is assumed high. We evaluate these post-quantum algorithms anyway, to see if they can be implemented in the Signal Protocol, and at what cost.

5.4 Code and test machine

In this section, we explain on which machine and how the scenarios were tested. To determine the effect post-quantum cryptography has on the Signal Protocol, we created a C program which implements the basics of the three scenarios: the Extended Triple Diffie-Hellman protocol (X3DH) and the Double Ratchet algorithm (DR), and test them against the different post-quantum algorithms. The pseudo code can be found in the Appendix B.

The test machine The machine used for testing has the following specifications:

- Type: Dell latitude 7280
- RAM: 8GB (7,89 GB usable)
- Processor: Inter Core i5-73000 CPU @ 2.60 GHz 2.71 GHz, dual core
- Operating system: Windows 64x
- Windows subsystem: Linux Ubuntu

The code was ran on the Windows subsystem, Linux Ubuntu, and cannot be put on GitHub because this thesis was written in collaboration with TNO.

The code was tested on only one core, and the time an algorithm takes is directly related to the number of CPU cycles the algorithm needs. On the used machine an average 2713460729 CPU cycles take 1 second.

How the scenarios were tested We kept track of the number of CPU cycles in each scenario and the values shown in this theses are the average of that. We did multiple iterations, trying to have minimal background noise. We also tested the CPU cycles for each number of messages or contacts (depending on the scenario)

to check if the algorithms would perform linearly, which they did. In Section 6 we explain more about this linearity.

5.5 An average WhatsApp user

In this section, we describe the average user and his minimal phone, because the Signal Protocol is used in chat applications on mobile phones. This information is useful to put the results, that we will get after testing the three scenarios with different post-quantum algorithms, in context.

In the following paragraphs we look into how people chat; how many messages people send, how quick they respond and how many contacts they have, which phones are used in the year 2018 and at the current available bandwidth. We end with a paragraph in which we summarise the average user and his minimal phone.

Average WhatsApp user WhatsApp is the most used chat application in 2018 with 1500 million active monthly users in October 2018[3]. A study on WhatsApp user patterns by Rosenfeld et al. [Ros+18] showed the following data:

- Users send 145 messages a day (we round this up to 150).
- Users have 97 groups (we round this up to 100).
- Messages are on average 5.66 words long.
- Almost 60% of the messages get a response within 1 minute.

Users, on average, send and receive 145 messages a day, 38 of which are sent, 107 are received. The number between sent and received differs because people are in group chats. Whatsapp users, on average, are in 97 groups in which they chat, including two person groups. Because 71.5% of these groups are two person groups, we assume for simplicity that all groups are with two users. Most messages get a quick response, 58.8% of the responses are within the minute, and another 9.8% of the responses within 5 minutes. While 20.8% of the messages do not receive a response within an hour. The study also found that 99% of the message contained only text, and the remaining 1% could consists of files or links.

Type speed Different studies were done on the type speed of an average user on a normal keyboard or phone, to name a few: Ratatype [Rat], Anderson [And+09] and Roebers [Roe+03]. The type speed differs, reaching from 28 word per minute

[3] https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

(WPM) to 65 WPM. If the average user has the highest type speed, he would be able, in combination with the study by Rosenfeld et al. [Ros+18] to type almost 12 messages per minute, one message every 5 seconds.

Average phones To see how the CPU cycles need for the post-quantum algorithms in Signal will react to the nowadays used mobile phones, we picked a few phones, both more expensive and less expensive phones to give an overview of what is on the current market. An overview of the specifications of these phones can be seen in Table 5.3.

	Storage (GB)		CPU (GHz)		Battery (mAh)
Phone name	Min	Max	Min	Max	
Samsung Galaxy S9	64	256	1.7	2.8	3000
Apple iPhone X	64	256	2.4	2.4	2716
Huawei Mate 20	128	128	1.8	2.6	4000
SAMSUNG Galaxy J4 Plus	32	32+	1.4	1.4	3300
MOTOROLA E5 Play 16GB	16	16+	1.4	1.4	2100
HUAWEI Y6 2018 16GB	16	16+	1.4	1.4	3000
Minimal Phone	16	128+	1.4	1.4	2100

Tab. 5.3.: Common used smart phones used in 2018 and their specifications. The difference in min and max value of storage space and CPU speed is because of different versions.

We define the *minimal phone* as the combination of those 6 phones but with the lowest specifications: 16GB storage and 1.4 GHz CPU, which means that the processor will do 1400000000 CPU cycles per second, where the tested computer had 2.6 GHz. All the times we will see in Section 6 are directly dependent on the number of CPU cycles.

All phones with a low storage space could extend it with a at least 32GB card for 30 euros, and we could thus assume 48GB available storage space. The minimal Phone would than cost 130 euro, which is a low cost for a smart phone.

The battery life of the minimal phone is 2100 mAh. Battery life depends on usage, battery type and other aspects we cannot explicate in this thesis. For mobile phones and other mobile devices the voltage is typically 5V. We will assume that a our user has the minimal phone, which will have one hour of 2100 mA and 5V.

Average bandwidth To compare the increase in bandwidth for the current Signal Protocol (as described in Section 3) and a post-quantum version it is useful to to see how much bandwidth is available. A study by Open Signal [Ope17] showed

that the average mobile networks speed differs per country, but for example in Ireland it is 12 Mbps and the Netherlands 26 Mbps, while the lowest measured average speed is Costa Rica with 3 Mbps. We assume a lower than average maximum of 8 Mbps, which is 1 MBps, that is available.

Acceptable delay time In a chat conversation users might have to wait for a message to be encrypted, decrypted or sent. How much delay is acceptable? Considering no other delay, the average user will type a message in 5 seconds, wait for another 5 seconds while the other user responds and then starts typing a new replay. There are, of course, less optimal cases in which users type very short messages: with one character or emoticon. We might not be able to answer where this specific threshold is in this thesis, finding this value is out of the scope of this paper. We can, however, make an assumption.

In *Website Response Times* [Nie10] Nielsen explains that a delay of 0.1 seconds will give the feeling of instantaneous response, while a delay of 1 second will give users a sense of delay but is not experienced as annoying by users. Waiting between 1 and 10 seconds keeps the users attention; however, gives them a strong sense of delay and annoyance. We will assume that for an active chat a delay of maximum 1 second should be fine, and for things that should be done not that often even, up until 10 seconds are acceptable.

Average user Concluding from the rest of the data we define the average user with a minimal phone. The average user:

- has 100 contacts,
- sends 150 messages per day,
- has a phone with 1400000000 CPU cycles per second and 48 GB storage space,
- has a bandwidth of 1 MBps,
- does not want to wait more than 1 second.

6 Experimental results

The Signal Protocol consists of three phases and for each phase a scenario was created (Section 5). In this section, we evaluate different post-quantum algorithms in each scenario and look at the impact they have on the number CPU cycles, storage space, bandwidth, network utilisation and energy usages for the average user. Recall that the average user has a minimal phone, 100 contacts, sends 150 message a day and is fine with a delay of 0.1 second during chatting (see Section 5.5). Different post-quantum algorithms are compared to *ECDH Curve 25519* which is the default algorithm in the current Signal Protocol.

The initial scenario is evaluated in Section 6.1, the X3DH scenario in 6.2 and the Double Ratchet scenario in 6.3. We combine the results obtained in these three scenarios in Section 6.4, to create different possible post-quantum Signal Protocols. We explain which post-quantum Signal Protocols are best for a real world implementation.

6.1 The initial scenario

In this section, we look at the initial scenario and how much CPU cycles (Section 6.1.1), key storage (Section 6.1.2) and network load (Section 6.1.3) the initial Scenario uses with *SIDH503* and *SIDH751* in comparison to *ECDH*.

In the initial scenario Alice has to create 100 pre-key bundles, so that if she is offline other users can still make contact with her, as was described in 5.2. Creating the 100 pre-key bundles results in creating 100 OTK, one IK and one SPK key pairs at the same time, storing the private keys locally and sending the public keys to the server. The initial scenario is tested with *ECDH*, *SIDH503* and *SIDH751*, as was explained in Section 5.3.

6.1.1 CPU cycles

In this section, we evaluate the average CPU cycles the initial scenario needs with the following algorithms *ECDH*, *SIDH503* and *SIDH751*. The initial scenario was executed 1000 times, meaning: we created 1000 key pairs, observed the CPU

cycles per creating, took the average for one key pair creation and calculated the time it would take to do that many CPU cycles on the minimal phone.

In Table 6.1 we see one key exchange in the initial scenario for each algorithm; the average number of CPU cycles, the standard deviation that goes with it and the time it takes. The factor compares the CPU cycles between each algorithm and *ECDH*.

Initial scenario	1 key pair creation			
Algorithm	Average CPU cycles	Standard deviation	Time (s)	Factor
ECDH	166561	49893	0.00012	1
SIDH503	7125660	386574	0.00509	42.8
SIDH751	19162022	1014579	0.01369	115

Tab. 6.1.: The number of CPU cycles and the time it takes to create the 100 pre-key bundles that Alice has to upload to the server. The standard deviation (in CPU cycles) is given, and the factor which compares both *SIDH503* and *SIDH751* to *ECDH*.

We see that *SIDH503* and *SIDH751* take a factor 43 and 115 more than *ECDH*. A pre-key bundle consists of 102 keys, so we will multiply the numbers of CPU cycles for 1 key pair creating from the table with 102. Creating a pre-key bundle for *ECDH* will take 0.012 seconds, for *SIDH503* this will take half a second, while for *SIDH701* this takes 1.4 seconds.

We see that this means a high increase in time and CPU cycles for SIDH, in comparison to *ECDH*. However, both a *SIDH503* and *SIDH751* pre-key bundle is created within no more than 1.5 seconds of extra time than it would have taken with *ECDH*. The key bundle creation is not done that often, only when 100 contacts made contact with a user, the user has to reupload 101 new keys (the identity key only has to be uploaded once). We expect the average user is not annoyed by this delay, because it is within 10 seconds (see Section 5.5).

The standard deviation of CPU cycles of the key creations is small for *SIDH503* and *SIDH751*. However, for the standard deviation f *ECDH* a distortion is seen. It is relatively big as can be seen in Table 6.1. we assume that the big distortion could be the result of other processes still running while we did the test. On a small average number of CPU cycles a small distortion has more influence than with a bigger average number, like with *SIDH*.

6.1.2 Key storage

In this section, we evaluate how much key storage is needed to store the keys in the initial scenario for both *ECDH* and *SIDH*. Storing the keys for the initial

scenario requires storage space for 102 private keys for the user. The number of bytes that need storage for 102 private keys can be seen in Table 6.2.

Algorithm	102 private keys only		Factor	102 public keys		Factor
ECDH	3264 B	3.19 KB	1	3264 B	3.19 KB	1
SIDH503	3264 B	3.19 KB	1	38556 B	37.65 KB	11.813
SIDH751	4896 B	4.78 KB	1.50	57528 B	56.18 KB	17.625

Tab. 6.2.: The different algorithms in the X3DH key exchange and the storage space they require to store the 102 private keys, for the user, and 102 public keys, for the server and as network utilisation.

This results in a maximum storage of 4.78 KB for 102 private keys for *SIDH751*, a factor 1.5 compared to the default *ECDH*, while *SIDH503* requires the same storage space as *ECDH*. On the minimal phone (as described in Section 5.5) this increase of storage space is not a problem.

The private parts of those keys are the only keys a user needs to store, all public keys are available on the server. For the server this would mean that it would need to store 102 public keys per users. Because the public keys are bigger than the private keys, the server needs to store 38 KB for *SIDH503*, a factor 11.8 larger than *ECDH*, and 56 KB for *SIDH751*, a 17.6 factor larger. We will not go into detail about the server and it's storage space in this thesis.

6.1.3 Network load

In this section, we look at the bandwidth and the network utilisation required to send the key bundles created in the initial scenario to the server. Uploading the created 102 public keys to the server, all at once, will require network utilisation as can be seen in Table 6.2. For security level 1 *SIDH503* that will require 38 KB, a factor 11.8 increase in data size when sending 102 keys in comparison to *ECDH*, and with *SIDH751* this will be 56 KB, a factor 17.6 more.

For the bandwidth of the average user this means that will take 0.039 second for Level 1 *SIDH503* and 0.058 second for Level 3 *SIDH751*, in comparison with only 0.003 seconds for ECDH. This is a small increase in bandwidth especially when there is a bandwidth of 1 MBps. The average user will not find this increase in time annoying because it is still below the 0.1 seconds.

6.1.4 The post-quantum initialisation phase

In this section, we give an overview of the previous 3 sections about the CPU cycles needed, the storage space required and the bandwidth needed for the

initial scenario. In Table 6.3 we see how many CPU cycles, time and storage space is needed to create 102 pre-keys.

Initial scenario	Creating and generating			Network load	
	CPU	Time	User storage	Time	Size
ECDH	16989244	0.012 s	3.2 KB	0.003 s	3.2 KB
SIDH503	726817369	0.519 s	3.2 KB	0.039 s	37.7 KB
SIDH751	1954526273	1.369 s	4.8 KB	0.058 s	56.2 KB

Tab. 6.3.: The values for an average user in the initial scenario for *ECDH, SIDH503 and SIDH751*. The number of CPU cycles, time (s), storage space (KB), bandwidth time (s) and network utilisation (KB) are shown.

The creating and uploading of 100 pre-key bundles is not done that often, and it is not crucial to create and upload a new bundle within a millisecond. It should be done of course, but it could be done in the background when the user is not using their phone. Therefore we could argue that the increase in storage and network utilisation is very small when *ECDH* is substituted with either *SIDH503* or *SIDH751* in the initial scenario. The total waiting time, to create and upload 100 pre-key bundles will still be within 1.5 seconds (for *SIDH751*). The average user will experience a small delay. To lower the delay for the user, two things could be done. The first time pre-key bundles are uploaded is should be done during the account set-up. Secondly, the other times 100 new bundles are uploaded, it could be done during a moment the user is inactive.

6.2 The X3DH scenario

In this section, we look at the X3DH scenario and how much CPU cycles (Section 6.2.1), key storage (Section 6.2.2) and network load (Section 6.2.3) the X3DH Scenario uses with *SIDH503* and *SIDH751* instead of *ECDH*.

In the X3DH scenario Alice is going to send the first message to Bob. She has received the pre-key bundle from Bob and will use the X3DH protocol and initiate the Double Ratchet chain to do so. To do that Alice generates one ephemeral key pair and three shared secrets, created as described in Section 5.2. The X3DH scenario is tested with *ECDH*, *SIDH503* and *SIDH751* (as was explained in Section 5.3).

6.2.1 CPU cycles

In this section, we evaluate the average number of CPU cycles the X3DH scenario needs with both *ECDH* and *SIDH*. The X3DH scenario was executed 1000 times,

which means we created one EK key pair and three shared secrets 1000 times. For every iteration we measured the number of CPU cycles. We took the average number of CPU cycles for one X3DH scenario and calculated the time it would take to do one scenario on the minimal phone.

The number of CPU cycles needed for one contact; to create one key pair and three shared secrets, can be seen in Table 6.4.

X3DH scenario	1 contact (creating: 1 key pair and 3 shared secrets)			
Algorithm	Average CPU cycles	Standard deviation CPU cycles	Time (ms)	Factor
ECDH	591010.82	51696	0.0004	1.00
SIDH503	23227464	508919.78	0.0166	39.30
SIDH751	64376866	3915170.33	0.0460	108.93

Tab. 6.4.: The number of CPU cycles needed to generate the shared secrets for one contact, following the X3DH scenario. The factor is the ratio between SIDH and *ECDH*. The time is the time it would take to add that many contacts on the minimal phone.

We see that *SIDH751* needs almost a factor 109 more CPU cycles than *ECDH*, while *SIDH503* needs a factor 39 more. For our average user it would roughly take 0.017 seconds to add one new contact with *SIDH503* and 0.046 seconds with *SIDH751*, as can be seen in Table 6.4. For an average user this delay is acceptable because it is below 0.1 second

To get the number of CPU cycles and time it takes for an average user to initiate contact with 100 contacts at the same time, we multiply the time and CPU cycles with 100. Hundred contacts is the number of contacts an average user had. It would take 1.7 and 4.6 seconds for *SIDH503* and *SIDH751* respectively, where it would only take 0.04 seconds with *ECDH*. The standard deviation is a lot smaller for *ECDH* in this scenario than it was with the initial scenario.

We assumed that both the initial scenario and the X3DH scenario was linear, and when this assumption was tested we found that it was. The complete data table for this linearity can be found in the Appendix C, Table C.1 and figure C.1.

6.2.2 Key storage

In this section, we evaluate how much storage space is required for the X3DH scenario. The key pair, EK, is used to generate the shared secret and then the public part of EK is send to Bob. After that the key does not have to be stored. The created shared secrets are put in a key derivation function, to derive the message key, and are not needed after that. In the initial scenario, therefore, no storage space required for the used keys.

There are, of course, keys that need storage: the message keys and chain keys. In this thesis we focus on the post-quantum algorithms. Therefore, we excluded the storage space needed for the symmetric message keys and the CPU cycles for to create those as well. Both in our post-quantum Signal Protocol, and the Signal Protocol with *ECDH* the same KDF function is used, namely SHA-512. So whether we use a post-quantum algorithm or *ECDH* the shared secrets and the symmetric message key will have the same length. Therefore, we will not take the size of those into account here.

Note that for the first Double Ratchet step a key should be stored, namely the key pair A_1. However, this key is excluded from the X3DH scenario because it is already included in the Double Ratchet scenario.

6.2.3 Bandwidth and network utilisation

In this section, we look at the bandwidth and network utilisation needed for the X3DH scenario, in which the first message is sent. The created public key EK should be send to the other users, together with the public identity key of Alice, IK. The three shared secrets are obviously not shared over the network. In Table 6.5 the size of one public key is shown, together with the network utilisation and the bandwidth for 200 public keys.

Algorithm	1 public key	200 public keys	sec	Factor
ECDH	32 B	6400 B	0.006	1
SIDH503	378 B	75600 B	0.072	11.8
SIDH751	564 B	112800 B	0.108	17.6

Tab. 6.5.: The different algorithms in the X3DH key exchange and the storage space they require to store the 102 private keys, for the user, and 102 public keys, for the server.

If a user would send all his 100 contacts the first message at once, this would result in sending 200 keys. The network utilisation for 200 keys with *ECDH* is 6.3 KB and it would take 0.006 seconds to send that data. *SIDH* will require a factor 11.8 more for *SIDH503* and 17.6 for *SIDH751* in comparison to *ECDH*. This results in a network utilisation of 73.8 KB for *SIDH503* and 110.2 KB for *SIDH751*, as can be seen in Table. Sending this data over a 1 MBps connection will result in a bandwidth of 0.072 and 0.108 seconds in total. Again for an average user this delay is acceptable because it is below 0.1 second

6.2.4 A post-quantum X3DH scenario

In this section, we summarise the previous three sections, to see how much CPU cycles are needed, storage space is required and bandwidth is needed for the X3DH scenario. The overview can be seen in Table 6.6.

X3DH scenario	Creating and generating			Network load	
	CPU	Time	Storage	Time	Size
ECDH	59101082	0.04 s	0 KB	0.006 s	6.3 KB
SIDH503	2322746441	1.66 s	0 KB	0.072 s	73.8 KB
SIDH751	6437686648	4.60 s	0 KB	0.108 s	110.2 KB

Tab. 6.6.: The values for an average user in the X3DH scenario, with *ECDH* and *SIDH*. The number of CPU cycles, time (s), storage space (KB), bandwidth (s) and network utilisation (KB) are shown.

Remember that the average user has 100 contacts, and that the table shows what would happen if the user would send them all a first message at once (follow the X3DH scenario 100 times). In most cases the X3DH scenario would not be done for 100 contacts at the same time. However, there is a case: if you lost your phone on new years eve, instantly bought a new one, and want to send all your contacts a first message wishing them a happy new year. In that specific case the Table shows, for an average user, what the impact will be on CPU cycles, time and network load for all three algorithms.

In that worst case scenario it would take *SIDH503* only 1.7 seconds to create the keys and shared secrets and another 0.07 seconds to send them. For *SIDH751* this will be a bit longer: 4.6 seconds to create the keys and shared secrets, and 0.1 second to send them. It is below 10 seconds which we define as fine, but annoying for the user. Luckily this worst case scenario will not happen daily.

In a better case, in which a user maybe initiate the first message with a few users at the time, the extra delay, using *SIDH* instead of *ECDH*, is not that big. For example it will take less than half a second for *SIDH751* and less than 0.2 seconds for *SIDH503* to send ten contacts a first message. In this case the average user will have to wait less than a 1 second, which we defined as a doable but noticeable delay for an average user. Again this first message is only sent once to each contact the user has, not daily or weekly.

6.3 The Double Ratchet Scenario

In this section, we see the differences between each post-quantum algorithm and *ECDH* in the Double Ratchet scenario in terms of CPU cycles needed (Section

6.3.1), energy consumption (Section 6.3.2), storage space (Section 6.3.3) and network load (Section 6.3.4). In the Double Ratchet Scenario Alice is sending and receiving one message to and from Bob, as described in Section 5.2. To do that there is one new key pair created for Alice, and two shared secrets are generates between Alice and Bob. If a KEM is used, instead of *ECDH* or *SIDH*, Alice will encapsulate one shared secret and decapsulate the other one, as was explained in Section 5.3.1. The Double Ratchet scenario is tested with 10 different KEMs, *SIDH* and *ECDH* as was explained in Section 5.3.2 and 5.3.3.

6.3.1 CPU Cycles

In this section, we evaluate the average number of CPU cycles needed to send and receive one message in the Double Ratchet scenario. We tested the different algorithms 1000 times for one message in the Double Ratchet scenario and took the average number of CPU cycles. The average number of CPU cycles, and the standard deviation for all the algorithms with security level 1 can be seen in Figure 6.1.

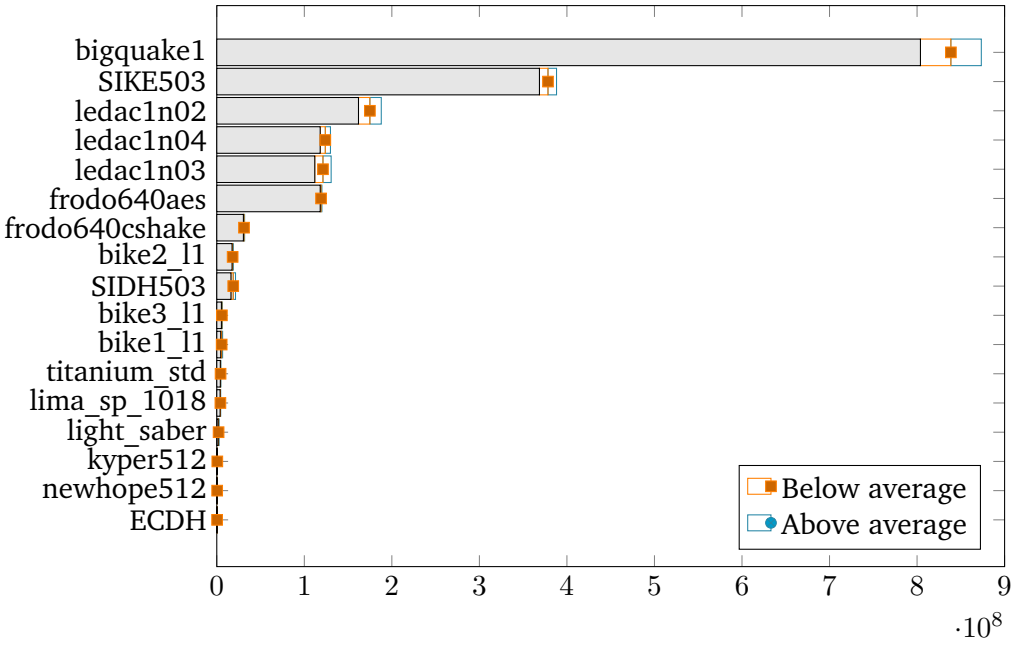

Fig. 6.1.: The different level 1 post-quantum algorithms in the Double Ratchet scenario. The average number of CPU cycles for sending 1 message are indicated with the orange square. The standard deviation is indicated with the two with boxes left and right form the average.

In Table 6.7 the average number of CPU cycles and the time it takes to send 1 and 150 messages is shown for all security levels. The average user sends 150 messages per day, to get the average number of CPU cycles for 150 messages, we multiplied the average of 1 message with 150.

SL	Algorithm	1 message		150 messages		Factor
		CPU cycles	Time (ms)	CPU cycles	Time (s)	
	ECDH	389050	0.28	56669413	0.02	1.00
1	newhope512	455169	0.33	68275377	0.05	1.2
1	kyper512	605986	0.43	90897841	0.06	1.5
1	light_saber	2064936	1.47	309740338	0.22	5.3
1	lima_sp_1018	4134622	2.95	620193249	0.44	10.6
1	titanium_std	4324690	3.09	648703464	0.46	11.1
1	bike1_l1	5582535	3.99	837380225	0.60	14.3
1	bike3_l1	5812725	4.15	871908748	0.62	14.9
1	bike2_l1	18110468	12.94	2716570176	1.94	46.3
1	SIDH503	18801852	13.43	2820277835	2.01	48.1
1	frodo640cshake	31095974	22.21	4664396051	3.33	79.5
1	frodo640aes	119130540	85.09	17869580955	12.76	304.5
1	ledac1n03	121304036	86.65	18195605430	13.00	310.1
1	ledac1n04	123992725	88.57	18598908780	13.28	316.9
1	ledac1n02	174901610	124.93	26235241560	18.74	447.1
1	SIKE503	378372971	270.27	56755945695	40.54	967.1
1	bigquake1	838627044	599.02	125794056540	89.85	2143.5
3	kyper768	851649	0.61	127747344	0.09	2.3
3	lima_2p_1024	1741608	1.24	261241173	0.19	4.6
3	saber_saber	3678476	2.63	551771453	0.39	9.7
3	titanium_med	5194122	3.71	779118233	0.56	13.7
3	lima_sp_1822	8154292	5.82	1223143776	0.87	21.6
3	bike1_l3	17268806	12.33	2590320837	1.85	45.7
3	bike3_l3	19563389	13.97	2934508419	2.10	51.8
3	SIDH751	51986181	37.13	7797927150	5.57	132.9
3	frodo976cshake	63773556	45.55	9566033448	6.83	168.8
3	bike2_l3	64711221	46.22	9706683080	6.93	171.3
3	frodo976aes	274384652	195.99	41157697773	29.40	726.3
3	ledac3n04	354120036	252.94	53118005360	37.94	937.3
3	ledac3n03	431532989	308.24	64729948335	46.24	1142.2
3	ledac3n02	672597927	480.43	100889689026	72.06	1780.3
3	SIKE751	1296655543	926.18	194498331384	138.93	3432.2
3	bigquake3	7639230018	5456.59	1145884502750	818.49	20220.5
5	newhope1024	882603	0.63	132390467	0.09	2.3
5	kyper1024	1277892	0.91	191683860	0.14	3.4
5	titanium_hi	5620445	4.01	843066720	0.60	14.9
5	fire_saber	5782534	4.13	867380136	0.62	15.3
5	titanium_super	7876282	5.63	1181442266	0.84	20.8
5	bike1_l5	39619721	28.30	5942958219	4.24	104.9
5	bike3_l5	44419807	31.73	6662971059	4.76	117.6
5	bike2_l5	191978514	137.13	28796777063	20.57	508.2
5	ledac5n04	998907868	713.51	149836180158	107.03	2644.0
5	ledac5n03	1193232407	852.31	178984860975	127.85	3158.4
5	ledac5n02	1951035846	1393.60	292655376953	209.04	5164.3
5	bigquake5	15324179943	10945.84	2298626991441	1641.88	40562.0

Tab. 6.7.: The number of CPU cycles and the time each algorithm takes on the minimal phone, for both 1 and 150 messages for each algorithm. The algorithms are ordered per security level (SL) and ECDH is at the top of the table. The factor compares the CPU cycles of the post-quantum algorithms to that of ECDH.

There is a large variation between the different algorithms in the average number of CPU cycles. For security level 1, there are seven algorithms which will take less than a factor 15 in comparison to *ECDH*, three take a factor less than 100, and the remaining six algorithms take a factor raining between 300 and 2144 in comparison to *ECDH*.

The quickest level 1 algorithm, newhope512, takes only 0.33 ms per message, resulting in a total delay of 0.05 second per day for the average user. This is only an increase of 1.2 in comparison to *ECDH*. While the slowest level 1 algorithm, bigquake1, takes 0.6 second for each message, resulting in a delay of 1.5 minute per day, an increase of a factor 2144.

For part of the algorithms, both level 1, 3 and 5, the increase in time it takes to create the keys needed to send one message and receive the other are low. Take for example level 3 *kyper768* and level 5 *newhope1024*, they both only take a factor 2.3 more time than *ECDH*. There are also algorithms, like *bigquake5*, that will take more than 10 seconds per message.

The question that remains is: how much delay will go unnoticed and which extra delay is still acceptable. We defined that for our average user a delay of 0.1 seconds feels like like no noticeable delay, while a delay of 1 second will be noticed but not found annoying. Thirteen of the security level 1 algorithms take less than 0.1 seconds to send one message, ten of the level 3 algorithms and seven of the level 5 algorithms. There are of course more factors we should take into account, like bandwidth and the time it takes to do a key derivation function.

In Table 6.7 the CPU cycles for one message and 150 messages in the Double Ratchet scenario are given. Since the number of CPU cycles are linearly dependent on the number of messages, the CPU cycles for 150 messages where calculated by multiplying the value for 1 message by 150. The linear relationship between the number of CPU cycles and the number of messages can be explained by the definition of the Double Ratchet scenario, for each message certain operation should be done (see Section 5.2). However, this assumption was also tested and is shown in Figure E.2 for security level 3. Except for some outliers all the linear expected value was within a 5 percent variance of the measured data as shown in Table 6.7. The complete list of all the tested number of messages, ranging from 1 until 325, and all algorithms, can be found in the Appendix E in Table E.1, E.2 and E.3.

6.3.2 Energy consumption

In this section, we look at the energy consumption the algorithms need in the Double Ratchet scenario.

Banerjee et al. [BH18] collected the run time and energy consumption data in their paper for all the Algorithms in the NIST competition. In Table 6.8 we see an overview of this, in which the energy consumption of the KEMs that we evaluate in this these are shown. The complete overview can be seen in Appendix F.

In the table we see that *bigquake* is the most energy consuming KEMs in each security level. For security level 1, bigquake consumes a factor 524 more energy than the most energy efficient KEM: *light_saber*, which only uses at total 2292 mJ for 150 messages. The energy efficiency of *light_saber* is a close match with *kyper512*, which uses 2520 mJ and *newhope512* with 2811 mJ. For security level 3 kyper and saber are again in the top two of most energy efficient, however, *kyber768* uses only 3983 mJ, while *saber_saber* uses 4172 mJ. *Bigquake3* needs 2760 times more energy than *kyber768*. Looking at the energy efficiency in the level 5 KEMs, we can confirm that the three most efficient KEMs are: *kyper_1024*, *newhope_1024* and *fire_saber* with 5432, 5710 and 6311 mJ, respectively. On this level *bigquake5* even uses 3890 times more energy than the most efficient one.

Unfortunately we do not have test results for *ECDH*, *SIDH503* and *SIDH751*, so we had to make an assumption about these values. We did this by taking the worst energy consumption of all the tested algorithms in [BH18] (which was 83.11 Watts), and multiplied them with the run time we found in Section 6.3. If we compare the run time in Banerjee's *Energy Consumption of Candidate Algorithms for NIST PQC Standards* [BH18], we see that our run time on the minimal phone is sometimes double the number that Banerjee et al. found. However, when assuming the energy values of *ECDH*, *SIDH503* and *SIDH751* we decided to not compensate for that.

Another problem we encountered with the data from Banerjee et al. [BH18] was that the different versions of *bike*, *ledakem*, *lima*, and *frodo*, where not tested and that it was not always clear which version they used. For simplicity we assumed that the energy consumption is the same for all versions of that KEM.

6.3.3 Key Storage

In this section, we evaluate on the number of bytes that need storage in the Double Ratchet scenario. For the Double Ratchet scenario for every contact one

Algorithm	1 DR scenario		150 DR scenario	
	Time (ms)	Energy (mJ)	Time (ms)	Energy (mJ)
L1				
light_saber	0.57	15.30	86	2295
kyper512	0.64	16.80	95	2520
newhope512	0.69	18.74	104	2811
lima	1.26	33.31	189	4997
frodo640	1.42	36.76	212	5514
bike_l1	1.46	37.52	219	5628
titanium_std	1.91	50.07	287	7511
ledac1	44.26	1177.76	6639	176664
SIKE503	115.73	3110.01	17360	466502
bigquake1	303.90	8011.21	45585	1201682
L3				
kyper768	1.00	26.55	149	3983
saber_saber	1.04	27.81	156	4172
titanium_med	2.17	57.32	326	8598
lima	2.64	70.63	396	10595
frodo976	2.80	73.48	420	11022
bike_l3	8.52	223.16	1278	33474
ledac3	126.55	3361.80	18983	504270
SIKE751	378.82	10227.89	56823	1534184
bigquake3	2767.40	73280.46	415110	10992069
L5				
kyper1024	1.40	36.21	209	5432
newhope1024	1.40	38.07	210	5710
fire_saber	1.56	42.07	234	6311
titanium_hi	2.96	78.13	444	11720
bike_l5	7.87	205.78	1181	30867
ledac5	176231.58	9705.73	26434737	1455860
bigquake5	5190.20	137231.25	778530	20584688
Calculated (see text)				
ECDH	0.14	11.64	21	1745
SIDH503	13.43	1116.17	2015	167425
sid751	37.13	3085.87	5570	462881

Tab. 6.8.: The energy consumption (in mJ) and time (in ms) each KEM takes for sending one message and 150 messages in the Double Ratchet Scenario. This means doing one encapsulation, one key creating and one decapsulation. The data is taken from [BH18] and the last 3 rows are calculated using that data as well.

key pair has to be stored. An average user will have 100 contacts and thus have to store 100 key pairs. The number of bytes that need storage for the Double Ratchet scenario on the minimal phone of the average user can be seen in Table 6.9. For more information on which keys need to be stored and until when, see Appendix A.

The level 1 algorithm that requires the least storage space is *SIDH503* and only needs a factor 6.4 more storage space than *ECDH*, to store 0.04 MB. The biggest level 1 algorithm, *bigquake1*, will require 629 more space than *ECDH*, to store almost 4 MB of public key material. The storage also requires storing the 3 chain keys, however, the length of these values only depends on the key derivation function (KDF) used. As already explained in Section 6.2.2, the default KDF used is SHA-512, which means that the lengths for the chain keys are a constant factor. We will not consider them in this thesis.

For security level 3 and 5 the storage requirements are higher, but there are still feasible options. For the smallest level 3 algorithm, *SIDH751*, only 0.06 MB is required, a factor 9.6 more than *ECDH*. While for level 5 algorithm *fire_saber* is the smallest algorithm with a space of 0.42 MB, a factor 68. The worst case, level 5 *bigquake5*, requires a total storage space of 18 MB, that is almost a factor 3000 more than *ECDH*, while the worst case level 3 *bigquake3* only requires 11 MB.

The storage space required for all the algorithms will be no problem for the average user. Especially since two third of the level 1 algorithms, and half of the level 3 algorithms do not require more than one MB more data to be stored. Even *bigquake5*, which requires the most storage space of them all, 18 MB, will be fine for the minimal phone.

6.3.4 Network load

In this section, we evaluate on how much bandwidth and network utilisation is needed for an average user in the Double Ratchet scenario. The average user will send and receive 150 messages a day, resulting in 150 public keys being send from Alice to Bob. If a KEM is used, instead of *ECDH* or *SIDH*, both the public key and the ciphertext, which encapsulates the shared secret, should be included in the total size for the network utilisation (see Section 5.3.3). The total number of bytes for all those 150 public keys (and ciphertexts) can be seen in Table 6.10.

Sending 150 *ECDH* public keys will cost no more than 0.005 MB. The lowest level 1 algorithm, *SIDH503*, will multiply this number with a factor 11.8, resulting in 0.05 MB of data. While the worst case level 1, *bigquake1*, will take almost

	1 key pair	100 key pairs			
Algorithm name	Bytes	Bytes	MB	Factor	SL
ECDH	64	6400	0.01	1.00	0
SIDH503	410	41000	0.04	6.41	1
SIKE503	812	81200	0.08	12.69	1
light_saber	2240	224000	0.21	35.00	1
kyper512	2368	236800	0.23	37.00	1
newhope512	2816	281600	0.27	44.00	1
ledac1n02	3504	350400	0.33	54.75	1
ledac1n03	4712	471200	0.45	73.63	1
bike1_l1	5084	508400	0.48	79.44	1
bike2_l1	5084	508400	0.48	79.44	1
bike3_l1	5516	551600	0.53	86.19	1
ledac1n04	6432	643200	0.61	100.50	1
lima_sp_1018	15272	1527200	1.46	238.63	1
frodo640aes	29488	2948800	2.81	460.75	1
frodo640cshake	29488	2948800	2.81	460.75	1
titanium_std	32736	3273600	3.12	511.50	1
bigquake1	40254	4025400	3.84	628.97	1
SIDH751	612	61200	0.06	9.56	3
SIKE751	1208	120800	0.12	18.88	3
saber_saber	3296	329600	0.31	51.50	3
kyper768	3488	348800	0.33	54.50	3
ledac3n02	7232	723200	0.69	113.00	3
bike1_l3	9928	992800	0.95	155.13	3
bike2_l3	9928	992800	0.95	155.13	3
ledac3n03	10416	1041600	0.99	162.75	3
bike3_l3	10844	1084400	1.03	169.44	3
ledac3n04	13184	1318400	1.26	206.00	3
lima_2p_1024	15362	1536200	1.47	240.03	3
lima_sp_1822	36442	3644200	3.48	569.41	3
titanium_med	36576	3657600	3.49	571.50	3
frodo976aes	46904	4690400	4.47	732.88	3
frodo976cshake	46904	4690400	4.47	732.88	3
bigquake3	114992	11499200	10.97	1796.75	3
fire_saber	4352	435200	0.42	68.00	5
kyper1024	4608	460800	0.44	72.00	5
newhope1024	5504	550400	0.52	86.00	5
ledac5n02	12424	1242400	1.18	194.13	5
bike1_l5	16376	1637600	1.56	255.88	5
bike2_l5	16376	1637600	1.56	255.88	5
bike3_l5	18068	1806800	1.72	282.31	5
ledac5n03	18056	1805600	1.72	282.13	5
ledac5n04	22744	2274400	2.17	355.38	5
titanium_hi	41056	4105600	3.92	641.50	5
titanium_super	53856	5385600	5.14	841.50	5
bigquake5	191604	19160400	18.27	2993.81	5

Tab. 6.9.: The number of bytes needed to store one key pair and hundred key pairs, the number the average user needs, for each post-quantum KEM.

	1 public key & ciphertext		150 messages	Factor	SL
Algorithm name	Size (B)	Time (ms)	Size (MB)		
ECDH	32	0.03	0.005	1.0	-
SIDH503	378	0.36	0.054	11.8	1
SIKE503	780	0.74	0.112	24.4	1
light_saber	1408	1.34	0.201	44.0	1
kyper512	1536	1.46	0.220	48.0	1
newhope512	2048	1.95	0.293	64.0	1
bike1_l1	5084	4.85	0.727	158.9	1
bike2_l1	5084	4.85	0.727	158.9	1
bike3_l1	5516	5.26	0.789	172.4	1
ledac1n02	6960	6.64	0.996	217.5	1
ledac1n03	7032	6.71	1.006	219.8	1
ledac1n04	8544	8.15	1.222	267.0	1
lima_sp_1018	10318	9.84	1.476	322.4	1
frodo640aes	19352	18.46	2.768	604.8	1
frodo640cshake	19352	18.46	2.768	604.8	1
titanium_std	19904	18.98	2.847	622.0	1
bigquake1	25683	24.49	3.674	802.6	1
SIDH751	564	0.54	0.081	17.6	3
SIKE751	1160	1.11	0.166	36.3	3
saber_saber	2080	1.98	0.298	65.0	3
kyper768	2240	2.14	0.320	70.0	3
bike1_l3	9928	9.47	1.420	310.3	3
bike2_l3	9928	9.47	1.420	310.3	3
lima_2p_1024	10372	9.89	1.484	324.1	3
bike3_l3	10844	10.34	1.551	338.9	3
ledac3n02	14400	13.73	2.060	450.0	3
ledac3n03	15576	14.85	2.228	486.8	3
ledac3n04	17536	16.72	2.509	548.0	3
titanium_med	22816	21.76	3.264	713.0	3
lima_sp_1822	23404	22.32	3.348	731.4	3
frodo976aes	31400	29.95	4.492	981.3	3
frodo976cshake	31400	29.95	4.492	981.3	3
bigquake3	84538	80.62	12.093	2641.8	3
fire_saber	2784	2.66	0.398	87.0	5
kyper1024	2944	2.81	0.421	92.0	5
newhope1024	4032	3.85	0.577	126.0	5
bike1_l5	16376	15.62	2.343	511.8	5
bike2_l5	16376	15.62	2.343	511.8	5
bike3_l5	18068	17.23	2.585	564.6	5
ledac5n02	24768	23.62	3.543	774.0	5
titanium_hi	26560	25.33	3.799	830.0	5
ledac5n03	27024	25.77	3.866	844.5	5
ledac5n04	30272	28.87	4.330	946.0	5
titanium_super	35264	33.63	5.045	1102.0	5
bigquake5	150292	143.33	21.499	4696.6	5

Tab. 6.10.: The bandwidth and network utilisation needed to send and receive one message. The network utilisation depends on the size of the public key and the ciphertext (if we are using a KEM). The factor compares the algorithm to *ECDH*.

3.7 MB, a factor 802.6 more. For level 3, *SIDH751* is the smallest with 0.08 MB, a factor 17.6 more than *ECDH*, closely followed by *SIKE751* with 83 KB and a factor of almost 36.3, while the worst case *bigquake3* sends 12 MB, a factor 2642 more. The smallest level 5 algorithm, *fire_saber*, will need 0.4 MB, a factor 87 more than *ECDH*. The worst case level 5 KEM is again *bigquake5* with 21.5 MB, and a factor of 4697.

With an bandwidth of 1 MBps there are nine level 1 algorithms, four level 3 and three level 5 that will take less than a second extra delay to send all 150 public keys and ciphertexts at once. Even better; all algorithms, except one, will take less than 0.1 second to send one message. For the average as described in Section 5.5 this was not annoying. However, if we would combine this delay with the delay of creating the keys, it might be a problem.

Another factor that might influence the delay during chatting is the maximum transmission unit (MTU). For most networks a MTU is 1500 bytes. If a message is larger than this, it will be split in multiple packages, resulting in increased data overhead, less efficiency and package delay [Mur+12]. As Table 6.10 shows, only *SIDH* and *SIKE* meet this restriction or the data should be split over multiple packages, with its own disadvantages. Unfortunately, researching this any further is out of the scope of this thesis.

6.3.5 The post-quantum Double Ratchet scenario

In this section, we give an summary of the results we obtained for the Double Ratchet scenario. The Double Ratchet scenario was evaluated with 15 different 128-bit secure post-quantum algorithms. In Table 6.11 we see the overview of the CPU cycles, time, energy consumption, storage space and bandwidth needed for an average user (Section 5.5). The CPU cycles and time are obtained by multiplying the values for 1 Double Ratchet step by 150 (Section 6.3.1), the energy consumption is obtained by multiplying the values for 1 Double Ratchet step by 150 (Section 6.3.2), the storage space is calculated by multiplying the values with 100 (Section 6.3.3) and bandwidth (Section 6.3.4) needed for an average user.

We can conclude that *bigquake1* is the worst post-quantum candidate for the Double Ratchet scenario because it has the highest value for each category. With that said, it is also the worst case code-based KEM. The best code-based KEM is not that easy to determine; while the bike algorithms have a lower number of CPU cycles, time, energy usages and a smaller bandwidth size and time, the *leda* algorithms have a lower storage space. Because the higher storage space of the

KEM Level 1	Creating and generating				Network load		Type
	Average CPU cycles	Time (s)	Energy (mJ)	Storage (KB)	Time (s)	Size (KB)	
newhope512	**68275377**	**0.05**	2811	275.0	0.29	300.00	L
kyper512	90897841	0.06	2520	231.3	0.22	225.00	L
light_saber	309740338	0.22	**2295**	218.8	0.20	206.25	L
lima_sp_1018	620193249	0.44	4997	1491.4	1.48	1511.43	L
titanium_std	648703464	0.46	7511	3196.9	2.85	2915.63	L
frodo640cshake	4664396051	3.33	5514	2879.7	2.77	2834.77	L
frodo640aes	17869580955	12.76	5514	2879.7	2.77	2834.77	L
ledac1n03	18195605430	13.00	176664	460.2	1.01	1030.08	C
ledac1n04	18598908780	13.28	176664	628.1	1.22	1251.56	C
ledac1n02	26235241560	18.74	176664	342.2	1.00	1019.53	C
bike1_l1	837380225	0.60	5628	496.5	0.73	744.73	C
bike3_l1	871908748	0.62	5628	538.7	0.79	808.01	C
bike2_l1	2716570176	1.94	5628	496.5	0.73	744.73	C
bigquake1	125794056540	89.85	1201682	3931.1	3.67	3762.16	C
SIKE503	56755945695	40.54	466502	79.3	0.11	114.26	I
SIDH503	2820277835	2.01	167425	**40.0**	**0.05**	**55.37**	I

Tab. 6.11.: The 16 different level 1 KEMs (and *SIDH503*) and the impact they have on an average user in the Double Ratchet scenario. The types of the KEMs are either lattice-based (L), code-based (C) or isogeny-based (I). The bold numbers indicates the lowest value in that column.

bike algorithm is still very small for the minimal phone, we could conclude that *bike1_l1* is the best option for a code-based KEM in the Double Ratchet scenario case.

There is only one isogeny-based KEM: *SIKE503*. While *SIKE503* is the second worst algorithm in terms of CPU cycles needed, run time and energy efficiency, it is the best level 1 post-quantum algorithm in terms of storage space, bandwidth time and size!

There are seven lattice-based algorithms, the top 3 consist of *newhope512*, *kyper512* and *light_saber*. The *newhope512* algorithms is the overall quickest in terms of CPU cycles and time, *light_saber* is the overall most energy efficient and *kyper512* is the second best in each of these categories. All fail to match the low value for storage space, bandwidth space and bandwidth time of the isogeny-based *SIKE503*. However, these algorithms are the second, third and fourth best in the overall level 1 KEMS. Depending on the most important criteria for the lattice-based KEM in this scenario, select *newhope512* for the best performance and run time, and *light_saber* for the most energy efficient, lowest storage requirement and best bandwidth. The worst case lattice-based algorithms

are *frodo640aes*, with a run time of almost 13 seconds, and *light_saber*, with the second overall slowest bandwidth time of 2.3 seconds.

In Table 6.12 and 6.13 similar results can be seen but for Security level 3 and 5. For Level 3 *kyper768* seems promising looking at the average CPU cycles, time and energy, while *SIDH751* is best for storage space and bandwidth. For level 5 *newhope1024* is best for creation time, while *kyper1024* is most energy efficient. For the most optimal bandwidth and least storage space *fire_saber* is the best choice.

KEM Level 3	Creating and generating				Network load		Type
	Average CPU cycles	Time (s)	Energy (mJ)	Storage (KB)	Time (s)	Size (KB)	
kyper768	**127747344**	**0.31**	**3983**	340.6	0.32	328.13	L
lima_2p_1024	261241173	0.64	10595	1500.2	1.48	1519.34	L
saber_saber	551771453	1.36	4172	321.9	0.30	304.69	L
titanium_med	779118233	1.91	8598	3571.9	3.26	3342.19	L
lima_sp_1822	1223143776	3.01	10595	3558.8	3.35	3428.32	L
frodo976aes	9706683080	101.12	11022	4580.5	4.49	4599.61	L
frodo976cshake	7797927150	23.50	11022	4580.5	4.49	4599.61	L
bike1_l3	2590320837	6.36	33474	969.5	1.42	1454.30	C
bike3_l3	2934508419	7.21	33474	1059.0	1.55	1588.48	C
bike2_l3	9566033448	23.85	33474	969.5	1.42	1454.30	C
bigquake3	1145884502750	2815.31	10992069	11229.7	12.09	12383.50	C
ledac3n04	41157697773	130.50	504270	1287.5	2.51	2568.75	C
ledac3n03	53118005360	159.03	504270	1017.2	2.23	2281.64	C
ledac3n02	64729948335	247.87	504270	706.3	2.06	2109.38	C
SIDH751	100889689026	72.06	462881	**59.8**	**0.08**	**82.62**	I
SIKE751	194498331384	477.86	1534184	118.0	0.17	169.92	I

Tab. 6.12.: The 16 different level 3 KEMs (and *SIDH751*) and the impact they have on an average user in the Double Ratchet scenario. The types of the KEMs are either lattice-based (L), code-based (C) or isogeny-based (I). The bold numbers indicates the lowest value in that column.

6.4 A post-quantum Signal Protocol

This section the three different scenarios are combined together into different post-quantum Signal Protocols. We evaluate the post-quantum Signal Protocols for the average user, as described in Section 5.5.

The three most suitable level 1 post-quantum Signal Protocols, for our analysis, are given and compared with each other (Section 6.4.1). The focus on level 1 post-quantum cryptography was explained in Section 5.3.4. To compare the level 1 post-quantum Signal Protocols with the Signal Protocol using *ECDH*, we

KEM Level 5	Creating and generating				Network load		Type
	Average CPU cycles	Time (s)	Energy (mJ)	Storage (KB)	Time (s)	Size (KB)	
newhope1024	**132390467**	**0.33**	5710	537.5	0.58	590.63	L
kyper1024	191683860	0.47	**5432**	450.0	0.42	431.25	L
titanium_hi	843066720	2.07	11720	4009.4	3.80	3890.63	L
fire_saber	867380136	2.13	6311	**425.0**	**0.40**	**407.81**	L
titanium_super	1181442266	2.90	11720	5259.4	5.04	5165.63	L
bike1_l5	5942958219	14.60	30867	1599.2	2.34	2398.83	C
bike3_l5	6662971059	16.37	30867	1764.5	2.58	2646.68	C
bike2_l5	28796777063	70.75	30867	1599.2	2.34	2398.83	C
ledac5n04	149836180158	368.13	30867	2221.1	4.33	4434.38	C
ledac5n03	178984860975	439.75	30867	1763.3	3.87	3958.59	C
ledac5n02	292655376953	719.02	1455860	1213.3	3.54	3628.13	C
bigquake5	2298626991441	5647.47	20584688	18711.3	21.50	22015.43	C

Tab. 6.13.: The 12 different level 5 KEMs and the impact they have on an average user in the Double Ratchet scenario. The types of the KEMs are either lattice-based (L) or code-based (C). The bold numbers indicates the lowest value in that column.

evaluated its impact as well in Section 6.4.2. We then have a quick look at the level 3 post-quantum Signal Protocol and level 5 post-quantum Signal Protocol (section 6.4.3).

6.4.1 The level 1 post-quantum Signal Protocols

In this section, we look at the three most suitable level 1 post-quantum Signal Protocols according to our analysis. The resulting three most suitable post-quantum Signal Protocols can be seen in Table 6.14.

	Level 1 quantum	Creating and generating				Network load		Total
	-safe Signal	CPU	Time (s)	Energy* (mJ)	Storage (KB)	Time (s)	Size (KB)	Time
L	kyper512 + SIDH503	3140461651	2.24	2701	234.5	0.26	336.5	2.5
C	bike1_l1 + SIDH503	3886944035	2.78	5809	499.7	0.77	856.3	3.5
I	Full SIDH503	5869841645	4.19	167606	43.2	0.09	166.9	4.3

Tab. 6.14.: The resulting best post-quantum Signal Protocol of level 1 and their impact they have in terms of CPU cycles, time (s), energy (mJ), storage space (KB), bandwidth (s) and network utilisation (KB). The total time indicates the total time it takes for both creating and sending the created keys.

These three most suitable post-quantum Signal protocols are created by looking at each scenario individually. For the initial and the X3DH scenario there is only one level 1 option: *SIDH503*, while for the Double Ratchet scenario there are multiple (as explained in Section 5.3.1). For the Double Ratchet scenario we look

at Table 6.11 in Section 6.3, and see that there is not one single best suitable algorithm. Some algorithms perform better, while others require less storage space or bandwidth.

We therefore decided to choose a most suitable per type cryptography, resulting in one full isogeny based post-quantum Signal Protocol, one isogeny-/code-based post-quantum Signal Protocol and one isogeny-/lattice-based post-quantum Signal Protocol.

For the fully isogeny based post-quantum Signal Protocol we compare the only two level 1 isogeny candidates: *SIDH503* and *SIKE503*. We can conclude that *SIDH503* is the most suitable because *SIDH503* requires less storage space and less CPU cycles then *SIKE503*. That is actually a good match, because *SIDH503* is a perfect plug and play substitute for *ECDH* and could directly replace ECDH (without the minor changes that the KEMs need to make the algorithm work with them, as was explained in Section 5.3.1).

For the most suitable isogeny-/code-based Signal Protocol we compared the different code-based algorithms. The level 1 *bike* algorithms are the better options for CPU time, bandwidth and network utilisation. Except *leda1n02* and *leda1n03*, which required less storage space than the *bike* algorithms. However, the difference is small and *leda1n02* and *leda1n03* needs more time to create the keys, 13 seconds in total. We conclude that *Bike1_l1* was the most suitable for our analysis. Compared to *Bike2_l1* it needed a bit less time to create the keys but has a equal storage space and bandwidth requirements, while *bike3_l1* needs a bit more in all three categories.

For the isogeny-/lattice-based Signal Protocol the most suitable algorithm is harder to find, because there are more lattice-based candidates. Especially because *newhope512*, *kyper512* and *light_saber* are on some fields a close match in the Double Ratchet scenario. While *newhope512* is the fasted with creating keys, *light_saber* uses the least energy. In the end, the decisive factor was the total time it would take to both create and send the keys, because the required storage space and energy consumption differ very little. This results in *kyper512* being the most suitable algorithm for the isogeny-/lattice-based Signal Protocol. The energy consumption of the SIDH keys is estimated(*) and therefore is very high in comparison to the other algorithms, see Section 6.3.2 for how this number was estimated.

We see a big difference in run time in the three different level 1 post-quantum Signal Protocols. This difference can be explained because *SIDH503* takes double the CPU cycles and time to generate the keys and shared secrets needed for the

Double Ratchet scenario, compared to *bike1_l1* and *kyber512*. The difference in required storage space is explained by *SIDH503* which require less storage space than their lattice and code based competitor. Because *bike* requests the most bandwidth, we see that the isogeny-/code based version is slow on that front.

We conclude that the initialisation of the quantum-safe Signal Protocols, varying in the post-quantum algorithms used, are feasible for the average user. There is no most suitable combination of post-quantum algorithms. A balance between a low run time and a low network load would be optimal. Especially because the higher storage space requirements of some post-quantum algorithms, although high in comparison to others, are not a problem for the minimal phone. The complete isogeny based post-quantum Signal Protocol was most easy to implement (as explained in Section 5.3.1) and therefore, although being the more slow combination, is a good option.

However, because the isogeny-/lattice-based with *kyper512* is performing fastest, as can be seen in Table 6.14, we could say that that one is the winner for the fastest level 1 post-quantum Signal Protocol. It only needs 2.5 seconds to both create and send the 150 keys an average user needs during the day.

Again it is important to note that these post-quantum Signal Protocols should be implemented in a hybrid form with *ECDH*, as described in Section 4.3. In the next section, we see that *ECDH* has a small impact on the performance, storage space and network load of the Signal Protocol. Thereby, we can conclude that a hybrid Signal Protocol, with both *ECDH* and one of the three mentioned, most suitable, post-quantum Signal Protocols, is feasible.

6.4.2 ECDH in all three scenarios

In this section, we look at the Signal Protocol with *ECDH* and how it scores in the different categories. The current Signal Protocol, with *ECDH*, has a 128-bit security level for classical computers. In Table 6.15 we see what the Signal Protocol, as it is with *ECDH*, requires from an average user with a minimal phone. Table 6.15 shows that currently an average user has to wait 0.047 seconds to create and generate keys and shared secrets, only has to store 12.5 KB on information and has to send 14 KB data over the internet which can be done in 0.0145 second. The estimated energy consumption(*) of the execution of one ECDH key exchange is a total of 6.17 mJ (see Section 6.3.2 for how this number was estimated).

ECDH	Creating and generating				Network load	
	CPU	Time	Energy*	Storage	Time	Size
S. Init	16989244	0.012 s	1 mJ	3.2 KB	0.0033 sec	3.2 KB
S. X3DH	59101082	0.040 s	3.51 mJ	0 KB	0.0064 sec	6.2 KB
S. DR	56669413	0.022 s	1.66 mJ	6.2 KB	0.0048 sec	4.69 KB
Total	132759738	0.047 s	6.17 mJ	9.4 KB	0.0145 sec	14.09 KB

Tab. 6.15.: The three different scenarios with *ECDH* and the impact they have on the Signal Protocol for an average user in terms of CPU cycles, time (s), estimated(*) energy (mJ) (calculated using 83.11 Watt), storage space (KB), bandwidth (s) and network utilisation (KB).

6.4.3 The post-quantum level 3 and 5 Signal Protocols

For a average user with a minimal phone we can conclude that it also possible, next to a level 1 post-quantum Signal Protocol, to have a level 3 post-quantum signal Protocol. For the best level 3 post-quantum Signal Protocol we need *SIDH768* for both the initial scenario and the X3DH scenario because that was the only option for those scenarios as explained in Section 5.3.1. Again we consider three possible best post-quantum Signal Protocols with a security level of 3: one full isogeny based, one isogeny-/code-based and one isogeny-/lattice-based. The resulting three post-quantum Signal Protocols can be seen in Table 6.16.

	Level 3 quantum	Creating and generating				Network load		Total
	-safe Signal	CPU	Time (s)	Energy (mJ)	Storage (KB)	Time (s)	Size (KB)	Time (s)
L	kyper768 + SIDH751	8519960265	6.28	4479	345.4	0.38	494.5	6.7
C	bike1_l3 + SIDH751	10982533758	12.33	33970	974.3	1.48	1620.7	13.8
I	SIDH751	109281901947	78.03	463377	64.6	0.14	249.0	78.2

Tab. 6.16.: The three most suitable post-quantum Signal Protocol of level 3 and their impact they have in terms of CPU cycles, time (s), energy (mJ), storage space (KB), bandwidth (s) and network utilisation (KB). The total time indicates the total time it takes for both creating and sending the created keys.

We consider *kyper768* most suitable for a level 3 isogeny-/lattice-based Signal Protocol. It needs only 0.63 seconds for the complete Double Ratchet scenario and, in combination with *SIDH503* in the other two scenarios, will only take 6.7 seconds in total. It was also the most energy efficient lattice-based algorithm and second best on storage space, only *saber_saber* requires less storage. For the isogeny-/code-based Signal Protocol the *bike1_l3* easily won as best code-based candidate for the Double Ratchet level 3 categories. Resulting in a protocol that only needs 13.8 seconds to send all 150 messages, and need less than one MB of storage space. The fully isogeny based Signal Protocol is using only *SIDH751*, which will give it the largest run- and bandwidth time of the three winning

protocols, namely 78.2 seconds; however, it requires the least total storage space: only 65 KB is needed to store all the keys for an average user with this protocol.

For a level 5 post-quantum Signal Protocol we did not test an algorithm that was suitable for the initial and X3DH scenarios. Unfortunately *SIDH* was not submitted to the NIST standardisation process and there was no suitable code for *SIDH964*. There is, however, a *SIKE964* version, uploaded to the NIST standardisation process; however, that one was unfortunately not available in the library we used. We decided to not look into possible post-quantum protocols for the initial and X3DH scenario, for security level 5, because of the high security level as mentioned in 2.4.2.

However, we can see looking at Table 6.13 for the Double Ratchet scenario, we see that *kyper1024* only needs 0.89 seconds to create and send 150 keys, while *newhope1024* is a close second.

7 Conclusions

In this section, we conclude on the work in this thesis and look at future work that might be of added value.

7.1 Conclusion

In this thesis we analyse that it is feasible to have a post-quantum Signal Protocol considering the status of 2018. We describe the whole protocol and distinguish two main parts: the extend triple Diffie-Hellman Protocol (X3DH) which is used to send the first message, and the Double Ratchet algorithm, which is used to update the encryption key while users are chatting. The Signal Protocol uses *ECDH* key exchanges which are not secure in a quantum world. Therefore, both the X3DH protocol and the Double Ratchet algorithm need a substitute for the *ECDH* key exchange.

For the transitional period, from classical to quantum computers, we look into different ways to make a partially hybrid post-quantum Signal Protocol. The most simple solution for the transition period would be to add at least one post-quantum initial key exchange, next to the *ECDH* X3DH protocol. This could enable the Signal Protocol to have passive quantum security.

In the case that quantum computers take over the world, the whole Signal Protocol and all its ECDH key exchanges should be substituted for an post-quantum alternative. To create a complete post-quantum Signal Protocol we divided the Signal Protocol in three different phases, the *initial setup* phase, the *first message phase* and the *message exchange and key update* phase. For each phase we created a scenario and those three scenarios were tested with different post-quantum alternatives for the *ECDH*.

For a quantum-safe Signal Protocol with 128-bit security we determined the three best suitable options. The most suitable post-quantum Signal Protocol used a combination between a isogeny-based algorithm, *SIDH503*, for the initial and X3DH scenario, and a lattice-based KEM, *kyper512*, for the Double Ratchet scenario. Based on our analysis it is concluded that this combination allows

an average chat user to create 100 key bundles, upload them, initiate contact with 100 contacts and send 150 messages, all at once, in 2.5 seconds. However, key encapsulation mechanisms are not an easy plug-and-play substitute for *ECDH* key exchanges. The post-quantum Signal Protocol that is most easy to implement, uses only the isogeny- based algorithm *SIDH503* for all three scenarios. This requires the least storage space, 43 KB, for keys but has a run time of 4.5 seconds. The last post-quantum Signal Protocol combines the code-based algorithm *bike1_l1* and the isogeny-based *SIKE503*. It results in a faster code than the fully isogeny based: only 3.5 seconds; however, requires a higher network utilisation.

We showed that it is feasible to create a post-quantum Signal Protocol which is 128-bit secure against quantum computers. We also showed that a 198-bit and 256-bit secure post-quantum is possible; however, this requires a higher amount of CPU cycles, storage space and bandwidth.

All these quantum-safe Signal Protocols should be implemented as a hybrid form between *ECDH* and a post-quantum algorithm, because of the risks that might come with new post-quantum algorithms.

Overall we are positive that a post-quantum Signal Protocol is a feasible for an average user in 2018, and hope that more research could be done to the complete impact a post-quantum algorithm has on the Signal Protocol.

7.2 Future research

There are many interesting topics which could be researched in the future, in this section we name a few.

The different post-quantum Signal Protocols we found give a nice idea on which properties are important for the post-quantum algorithms in a chat application. Quick performance and low bandwidth are important because of the speed with which users use chat applications. We only implemented and tested a simple version of the Signal Protocol (see Section 5.4). An interesting research could be implementing all the post-quantum cryptography into the actual Signal Protocol code. The implementation of these different post-quantum algorithms in the protocol, would provide good insight on the impact of a (hybrid) post-quantum Signal Protocol in the real world. In Section 5.5, we defined an average user and a minimal phone based on a lot of assumptions. Testing the post-quantum Signal Protocols on actual phones and testing them with actual users, could give

insight in how post-quantum algorithms influence the actual performance and user chat experience. In the real world the network aspects could be tested as well. It would give an idea of the influence the post-quantum Signal Protocol has on the network utilisation and bandwidth, and how that influences the chat experience. Another user related process is the use of video chats, they require a fast post-quantum algorithms so that each frame could be encrypted. Looking into how feasible it is to have post-quantum video chat in the Signal Protocol, or just in general, might be an interesting topic as well.

The different post-quantum Signal Protocols created in this thesis were only evaluated with eleven different post-quantum algorithms (see Section 5.3.3). In the future it would be nice to have it tested with all the NIST applications. Maybe even more potential new post-quantum cryptography could be tested. In that way an overview of the advantages and disadvantages of each post-quantum algorithm could be created for the Signal Protocol. Thereby only code-based, isogeny-based and lattice-based KEMs and *SIDH* were tested (Section 5.3.3). Multivariate KEMs and KEMs based on other mathematics were excluded (see Section 2.4). However, testing these KEMs as well might be useful, because if for example, one type of cryptography is not secure at all, post-quantum algorithms based on other mathematical principles would be needed.

In this thesis mostly KEMs were studied; however, we conclude that for the X3DH protocol it was not possible to just simply substitute *ECDH* with a KEM to create the shared secrets in the way *ECDH* did (section 5.3.1). To use a KEM in the Double Ratchet algorithm small but feasible changes were required as was explained in Section 5.3.1. It is interesting to analyse and compare how interactive key exchanges, non-interactive key exchanges and key encapsulation mechanisms are different, to get a clear overview of which security properties can be created using which method. How could certain ways of creating keys be used to get certain properties in the Signal Protocol?

Another thing, which we did not consider in this thesis, is a post-quantum signature scheme to substitute the ECDH signature used in the Signal Protocol. For the transitional period, a post-quantum signature is not yet the highest priority. However, as soon as quantum computers are there and the signatures are still created with *ECDH*, adversaries could fake the signatures and that would be a problem. Future research in post-quantum signatures is, therefore, important.

On the other hand more research could be done to analyse and test the post-quantum algorithms. A lot of the post-quantum algorithms are newly developed and might have flaws. More verification can be done to the post-quantum algorithms, to see if they are indeed quantum safe and secure to use. There could be

ways to improve the mathematical problems on which the cryptography is based. Will these problems still be hard to solve in the future? The implementation of the post-quantum algorithms could be analysed and improved as well. Although the algorithm might be mathematically correct, the implementation might have bugs which makes it easier for an adversary to break the cryptography. In this way, these post-quantum algorithms are improved in time and perform better.

Better post-quantum algorithms are not only an addition for a post-quantum Signal Protocol, but could also be used in other protocols where quantum-safe standards are useful. While there is already research going on about some post-quantum protocols, like quantum-safe VPN and TLS, as we saw in Section 1.2, many more possibilities could be studied. Do all the protocols have the same bottlenecks or do they differ? While in the Signal Protocol speed and high performance are essential to maintain a good chat experience, maybe setting up a VPN does not have to be done within one second? Creating an overview of which properties of certain post-quantum algorithms are useful for certain applications.

On the other hand, more security protocols can be researched. The Signal Protocol is not the only security protocol for chat applications. The Signal Protocol is very strong in secure communication between two parties; however, is very weak in group communication [CG+18]: "*an adversary who compromises a single group member can intercept communications indefinitely*". In the current Signal Protocol there is no backward secrecy at all in group chats. There are other protocols that might be added to the Signal Protocol as a solution to this problem. For example, the *Provably authenticated group Diffie-Hellman key exchange* by Bresson et al. [Bre+01] and the *Asynchronous Ratcheting Tree* by Cohn Gordon et al. [CG+18]. Analysing these other options further and creating post-quantum versions of them could give interesting insights both in how we can create more secure protocols and how we could create new secure protocols with post-quantum cryptography in mind.

Appendices

A

Key Storage in the Signal Protocol

The Figure A.1 shows how long certain keys need to be stored. In Figure A.2 we see how the Signal Protocol handels keys using KEM's.

Alice	Alice	Alice	Store key untill	Phase	Server	Bob	Bob	Bob	Store key untill
Initalization Signal	gen	ID_Aa	Forever	0		Initalization Signal	gen	ID_Bb	Forever
					give keys <	upload pre-key bundle	gen	SPK_Bb, OTK_Bb	
Initalization first message	get	ID_B, SPK_B, OTK_B	1	1	< give keys				
	gen	E_Aa	1	1					
	calc	SK_i = f(ID_a, ID_B, SPK_B, OTK_B, EK_a)	1	1					
	gen	DH_Aa1	4	1					
	calc	DH_1 (SPK_B, DH_a1)	1	1					
	calc	SK_1=(SK_i, DH_1)	4	1					
	enc	encrypt message with SK_1, send keys		1	> message >				
				2		Receive first message	get	ID_A, EK_A, DH_A1	2,2,3
				2			calc	SK_i = f(ID_A, ID_b, SPK_b, OTK_b, EK_A)	2
				2			calc	DH_1 (SPK_B, DH_a1)	2
				2			calc	SK_1 = (SK_i, DH_1)	6
				2			dec	decrypt message with SK_1	
				3		Send message	gen	DH_Bb1	6
				3				DH_2 = (DH_A1, DH_b1)	4
				3				SK_2 = (DH_2, SK_1)	6
				3	< message <		enc	encrypt message with SK_2, send keys	
Receive message	get	DH_B1	5	4					
	calc	DH_2 = (DH_a1, DH_B1)	4	4					
	calc	SK_2 = (DH_2, SK_1)	8	4					
	dec	decrypt message with SK_2		**4**					
Send message	gen	DH_Aa2	8	5					
	calc	DH_3 = (DH_a2, DH_B1)	5	5					
	calc	SK_3 = (DH_3, SK_2)	8	5					
	enc	encrypt message with SK_3, send keys		5	> message >				
				6		receive message	get	DH_A2	7
				6			calc	DH_3 = (DH_A2, DH_b1)	6
				6			calc	SK_3 = (DH_3, SK_2)	10
				6			dec	decrypt message with SK_3	
				7		send message	gen	DH_Bb2	10
				7			calc	DH_4 = (DH_A2, DH_b2)	7
				7			calc	SK_4 = (DH_4, SK_3)	10
				7	< message <		enc	encrypt message with SK_4, send keys	
Receive Message	get	DH_B2	5	8					
	calc	DH_4 = (DH_a2, DH_B2)	4	8					
	calc	SK_4 = (DH_4, SK_3)	8	8					
	dec	decrypt message with SK_4		**8**					
Send Message	gen	DH_Aa3	8	9					
	calc	DH_5 = (DH_a3, DH_B2)	5	9					
	calc	SK_5 = (DH_5, SK_4)	8	9					
	enc	encrypt message with SK_5, send keys		9	> message >				
				10		receive message	get	DH_A3	11
				10			calc	DH_5 = (DH_A3, DH_b2)	10
				10			calc	SK_5 = (DH_5, SK_4)	14
				10			dec	decrypt message with SK_5	

Fig. A.1.: The table shows how the Signal Protocol works and until which phase which key has to be stored.

Alice	Alice	Alice	Store key untill	Phase	Server	Bob	Bob	Bob	Store key untill
Initalization Signal	gen	ID_Aa	Forever	0		Initalization Signal	gen	ID_Bb	Forever
					give keys <	upload pre-key bundle	gen	SPK_Bb, OTK_Bb	
Initalization first message	get	ID_B, SPK_B, OTK_B	1	1	< give keys				
	gen	E_Aa	1	1					
	calc	SK_i = f(ID_a, ID_B, SPK_B, OTK_B, EK_a)	1	1					
	gen	DH_Aa1	4	1					
	calc	DH_1 (SPK_B, DH_a1)	1	1					
	calc	SK_1=(SK_i, DH_1)	4	1					
	enc	encrypt message with SK_1, send keys		1	> message >				
				2		Receive first message	get	ID_A, EK_A, DH_A1	2,2,3
				2			calc	SK_i = f(ID_A, ID_b, SPK_b, OTK_b, EK_A)	2
				2			calc	DH_1 (SPK_B, DH_a1)	2
				2			calc	SK_1 = (SK_i, DH_1)	6
				2			dec	decrypt message with SK_1	
				3		Send message	gen	DH_Bb1	6
				3				DH_2 = (DH_A1, DH_b1)	4
				3				SK_2 = (DH_2, SK_1)	6
				3	< message <		enc	encrypt message with SK_2, send keys	
Receive message	get	DH_B1	5	4					
	calc	DH_2 = (DH_a1, DH_B1)	4	4					
	calc	SK_2 = (DH_2, SK_1)	8	4					
	dec	decrypt message with SK_2		**4**					
Send message	gen	DH_Aa2	8	5					
	calc	DH_3 = (DH_a2, DH_B1)	5	5					
	calc	SK_3 = (DH_3, SK_2)	8	5					
	enc	encrypt message with SK_3, send keys		**5**	> message >				
				6		receive message	get	DH_A2	7
				6			calc	DH_3 = (DH_A2, DH_b1)	6
				6			calc	SK_3 = (DH_3, SK_2)	10
				6			dec	decrypt message with SK_3	
				7		send message	gen	DH_Bb2	10
				7			calc	DH_4 = (DH_A2, DH_b2)	7
				7			calc	SK_4 = (DH_4, SK_3)	10
				7	< message <		enc	encrypt message with SK_4, send keys	
Receive Message	get	DH_B2	5	8					
	calc	DH_4 = (DH_a2, DH_B2)	4	8					
	calc	SK_4 = (DH_4, SK_3)	8	8					
	dec	decrypt message with SK_4		**8**					
Send Message	gen	DH_Aa3	8	9					
	calc	DH_5 = (DH_a3, DH_B2)	5	9					
	calc	SK_5 = (DH_5, SK_4)	8	9					
	enc	encrypt message with SK_5, send keys		**9**	> message >				
				10		receive message	get	DH_A3	11
				10			calc	DH_5 = (DH_A3, DH_b2)	10
				10			calc	SK_5 = (DH_5, SK_4)	14
				10			dec	decrypt message with SK_5	

Fig. A.2.: The table shows how the Signal Protocol works using a KEM.

The pseudocode

B

In Section 5.4 more about the code and the testing machine is explained. In Section 5.2 the tested scenarios are explained.

B.1 Initial scenario

```
void test_initial{int num_iterations, char algorithm_name}{
    long long cpu_cycles1, cpu_cycles2;

    for(int i = 0; i < num_iterations; i++){
        \\Start cpu count
        cpu_cycles1 = get_current_cpu_cycles();

        \\Create 1 key pair
        char public_key;
        char private_key;
        create_keypair(public_key, private_key, algorithm_name);

        \\Stop cpu count
        cpu_cycles2 = get_current_cpu_cycles();
    }

}
```

B.2 The X3DH scenario

```
void test_X3DH{int num_iterations, char algorithm_name}{
    long long cpu_cycles1, cpu_cycles2;

    for(int i = 0; i < num_iterations; i++){
        char public_key_A_ID;
        char private_key_A_ID;
        create_keypair(public_key_A_ID, private_key_A_ID,
```

```
                algorithm_name);

        \\Create Bobs key pair
        char public_key_B_ID;
        char private_key_B_ID;
        create_keypair(public_key_B_ID, private_key_B_ID,
                algorithm_name);

        \\Create Bobs key pair
        char public_key_B_OTK;
        char private_key_B_OTK;
        create_keypair(public_key_B_OTK, private_key_B_OTK,
                algorithm_name);

        \\Start cpu count
        cpu_cycles1 = get_current_cpu_cycles();

        \\Create 1 key pair
        char public_key_EK;
        char private_key_EK;
        create_keypair(public_key_EK, private_key_EK, algorithm_name);

        DH1 = create_sharedsecret(private_key_A_ID, public_key_B_OTK,
                algorithm_name);
        DH2 = create_sharedsecret(private_key_A_EK, public_key_B_ID,
                algorithm_name);
        DH3 = create_sharedsecret(private_key_A_EK, public_key_B_OTK,
                algorithm_name);

        \\Stop cpu count
        cpu_cycles2 = get_current_cpu_cycles();
    }

}
```

B.3 The Double Ratchet scenario

For the Double Ratchet scenario there were two versions, the Double Ratchet was tested with ECDH and SIDH with this version:

```
void test_DoubleRatchet_DH{int num_iterations, char algorithm_name}{
    long long cpu_cycles1, cpu_cycles2;

    for(int i = 0; i < num_iterations; i++){

        \\Create Bobs key pair
        char public_key_B;
        char private_key_B;
        create_keypair(public_key_B, private_key_B, algorithm_name);

        \\Create Bobs key pair
        char public_key_B_2;
        char private_key_B_2;
        create_keypair(public_key_B_2, private_key_B_2,
            algorithm_name);

        \\Start cpu count
        cpu_cycles1 = get_current_cpu_cycles();

        \\Send the message to Bob
        char public_key_A;
        char private_key_A;
        create_keypair(public_key_A, private_key_A,
            algorithm_name);
        create_sharedsecret(private_key_A, public_key_B,
            algorithm_name);

        \\Receive the message from Bob
        create_sharedsecret(private_key_A, public_key_B_2,
            algorithm_name);

        \\
        cpu_cycles2 = get_current_cpu_cycles();
    }

}
```

And the KEMs were tested with this version:

```
void test_DoubleRatchet_KEM{int num_iterations, char algorithm_name}{
    long long cpu_cycles1, cpu_cycles2;

    for(int i = 0; i < num_iterations; i++){
        \\Create Bobs key pair
        char public_key_B;
        char private_key_B;
        create_keypair(public_key_B, private_key_B, algorithm_name);

        \\Create Bobs key pair
        char public_key_B_2;
        char private_key_B_2;
        create_keypair(public_key_B_2, private_key_B_2,
                algorithm_name);

        \\Start cpu count
        cpu_cycles1 = get_current_cpu_cycles();

        char sharedsecret_1;

        char ciphertext = encapsulate_sharedsecret(sharedsecret_1,
                public_key_B, algorithm_name);

        \\Create Alice key pair
        char public_key_A;
        char private_key_A;
        create_keypair(public_key_A, private_key_A, algorithm_name);

        \\Receive the message from Bob, with the ciphertext
        char ciphertext_2;

        char sharedsecret_2 = decapsulate(cipthertext_2,
                private_key_A, algorithm_name);

        \\
        cpu_cycles2 = get_current_cpu_cycles();
    }

}
```

C

X3DH Test Data

See Section 6.2 for the summarised table. Table C.1 shows the average CPU cycles for doing the cryptography needed when adding a certain amount of contacts, for 3 algorithms (SIDH751, SIDH503 and ECDH). The linearity was tested and plotted, and can be seen in figure C.1.

Cont	SIDH751	SIDH503	ECDH	Cont	SIDH751	SIDH503	ECDH
1	83567283	30133473	679939	260	16632655162	5842552021	129683434
10	680366060	232227097	5557722	270	17244126075	6098668864	134737126
20	1332070091	455872654	10438749	280	17890176419	6307419484	139845008
30	1951150754	680095940	15392181	290	18530601899	6563716267	144584805
40	2564098700	906906227	20417156	300	19355518396	6771537925	149875313
50	3205637906	1137790524	25554646	310	19788645041	6983563041	157214630
60	3852971420	1364670170	30649292	320	20425313060	7213726112	161330887
70	4504027573	1613295048	35683448	330	21066868476	7507860165	166620143
80	5122590055	1805749814	40319481	340	21694613310	7779888071	169159411
90	5751838412	2027165389	45466344	350	22325270307	7910077554	174476855
100	6420367724	2260764536	50414970	360	23022970827	8112598505	178895424
110	7100736999	2486478669	55379370	370	23637074328	8330918809	183867674
120	7735735220	2720569616	61426901	380	24268582877	8612243039	188791723
130	8309076003	2928711709	65358253	390	24902459876	8815071000	193874536
140	8963331992	3152432045	70296151	400	25548353800	9023688029	199425809
150	9586021133	3396685999	75208963	410	26158264434	9232482224	205024158
160	10242919351	3606263805	80326056	420	26828444790	9480242685	208869453
170	10856420873	3888985300	85258254	430	27509798028	9712084454	216664835
180	11519303921	4053359292	90143522	440	28101676182	9913199567	219009620
190	12164096661	4322296247	95261916	450	28737468678	10139611376	223473302
200	12773440441	4510613420	100477391	460	29406985758	10361967302	229245124
210	13506417350	4726261361	104965424	470	29999127991	10577559703	233349209
220	14078716723	4984485263	110331562	480	30674354189	10806468465	238363537
230	14688881071	5186619974	114843983	490	31275050111	11056956753	243487883
240	15371179359	5499775724	119753220	500	32221219431	11445783788	249004246
250	15969344054	5649177721	124915078				

Tab. C.1.: The table shows the average number of CPU cycles per algorithm (SIDH751, SIDH503 and ECDH) for the cryptography that needs to be done for the initial scenario and the X3DH scenario for the given number of contacts (ranging from 1 to 500). Ten iterations were done per value and the average value is shown.

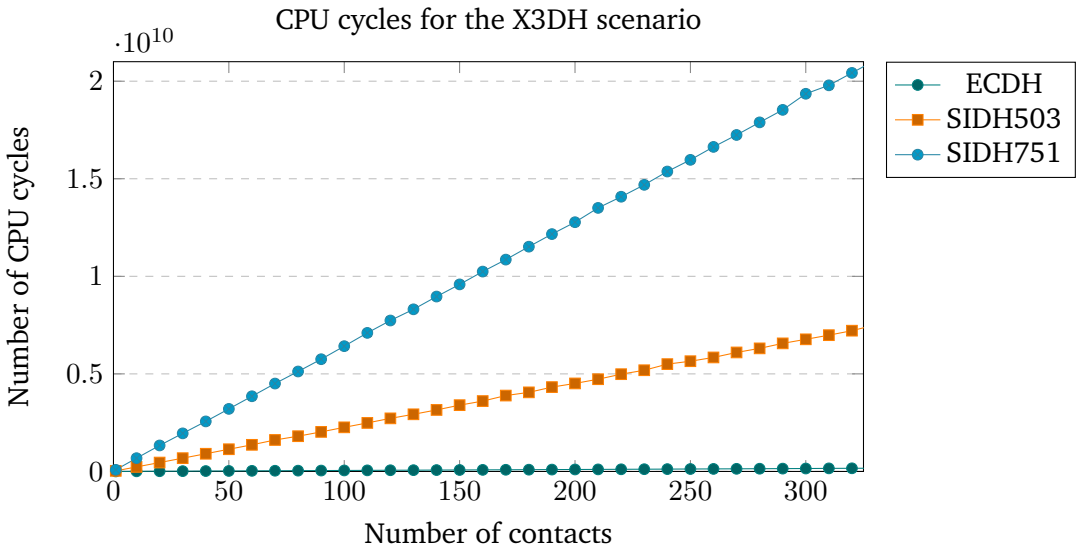

Fig. C.1.: The average number of CPU cycles per X3DH key exchange (and thus contact), shown for the 3 different algorithms: *ECDH*, *SIDH503* and *SIDH751*.

Key Length

D

The ten post-quantum KEM's from the Open Quantum Safe library, have different versions resulting in 45 different algorithms. In Table D.1 we see, per algorihm how much bytes is needed for the keys, shared secrets and the cipher text.

Algorithm name	Public key	Secret key	SS	Ciphertext	SL	Alternative name
ECDH Curve25519	32	32	32	32	-	-
bigquake1	25482	14772	32	201	1	BIG_QUAKE_1
bike1_l1	2542	2542	32	2542	1	BIKE_1_64
bike2_l1	2542	2542	32	2542	1	BIKE_2_64
bike3_l1	2758	2758	32	2758	1	BIKE_3_64
frodo640aes	9616	19872	16	9736	1	FrodoKEM-640
frodo640cshake	9616	19872	16	9736	1	FrodoKEM-640
kyper512	736	1632	32	800	1	kyber512
ledac1n02	3480	24	32	3480	1	128SL_N02
ledac1n03	4688	24	32	2344	1	128SL_N03
ledac1n04	6408	24	32	2136	1	128SL_N04
light_saber	672	1568	32	736	1	light_saber
lima_sp_1018	6109	9163	32	4209	1	CPA.LIMA-sp1018
newhope512	928	1888	32	1120	1	newhope512cca
SIKE503	378	434	16	402	1	SIKEp503
titanium_std	16352	16384	32	3552	1	Titanium_CCA_std
lima_sp_1306	10449	15673	32	6763	2	CPA.LIMA-sp1306
bigquake3	84132	30860	32	406	3	S
bike1_l3	4964	4964	32	4964	3	BIKE_1_96
bike2_l3	4964	4964	32	4964	3	BIKE_2_96
bike3_l3	5422	5422	32	5422	3	BIKE_3_96
frodo976aes	15632	31272	24	15768	3	FrodoKEM-976
frodo976cshake	15632	31272	24	15768	3	FrodoKEM-976
kyper768	1088	2400	32	1152	3	kyber768
ledac3n02	7200	32	48	7200	3	192SL_N02
ledac3n03	10384	32	48	5192	3	192SL_N03
ledac3n04	13152	32	48	4384	3	256SL_N02
lima_2p_1024	6145	9217	32	4227	3	CCA.LIMA-2p1024
lima_sp_1822	14577	21865	32	8827	3	CCA.LIMA-sp1822
saber_saber	992	2304	32	1088	3	saber
SIKE751	564	644	24	596	3	SIKEp751
titanium_med	18272	18304	32	4544	3	Titanium_CCA_med
lima_2p_2048	12289	18433	32	7299	4	CCA.LIMA-2p2048
lima_sp_2062	16497	24745	32	9787	4	CPA.LIMA-sp2062
bigquake5	149800	41804	32	492	5	BIG_QUAKE_5
bike1_l5	8188	8188	32	8188	5	BIKE_1_128
bike2_l5	8188	8188	32	8188	5	BIKE_2_128
bike3_l5	9034	9034	32	9034	5	BIKE_3_128
fire_saber	1312	3040	32	1472	5	fire_saber
kyper1024	1440	3168	32	1504	5	kyber1024
ledac5n02	12384	40	64	12384	5	192SL_N04
ledac5n03	18016	40	64	9008	5	256SL_N03
ledac5n04	22704	40	64	7568	5	256SL_N04
newhope1024	1824	3680	32	2208	5	newhope1024cca
titanium_hi	20512	20544	32	6048	5	Titanium_CCA_hi
titanium_super	26912	26944	32	8352	5	Titanium_CCA_super

Tab. D.1.: The different algorithms present in the OQS library, and their security level (SL).

E

Double Ratchet Test Data

See Section 6.3.1 for the summary of this data. In Table E.1, E.2, E.3 and E.4 the average number of CPU cycles are show, for 1 until 325 message (increasing per 25) being encrypted and then decrypted again for the different KEMs. The KEM's are ordered on security level and their type is given, either coded-based, lattice-based or isogeny-based. For the lower numbers of messages more then 100 iterations were done, while for 325 message, only 4 iterations were done, after which the obtained values were averaged.

In Figure E.1, E.2 and E.3 the data from Table E.2, E.3 and E.4 is plotted to show the linearity of the KEMs. There is one figure for each security level.

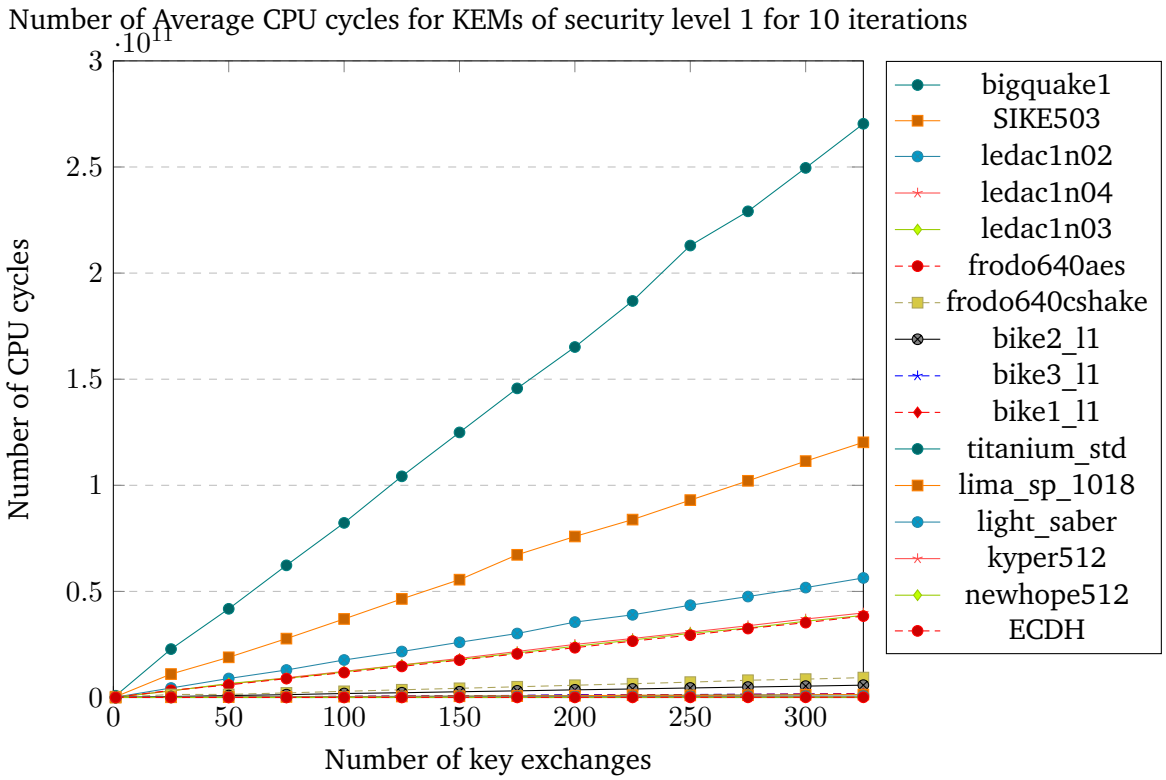

Fig. E.1.: The number of CPU cycles per number of key exchanges (and thus messages), for each KEM of level 1.

		1	25	50	75
	ECDH	389050	13362355	19334261	29014648
C	bigquake1	834231666	22823222149	41796855935	62285400232
I	SIKE503	372782930	11055943102	18944663056	27756562552
C	ledac1n02	177123815	4482333117	8962827601	12966031353
C	ledac1n04	123756915	3206962031	6304551854	9299127105
C	ledac1n03	120279046	3096878653	6605683749	9014544366
L	frodo640aes	119080004	3373293602	5931194493	8904175804
L	frodo640cshake	29389299	1434437605	1460993564	2180946233
C	bike2_l1	18025702	469436897	913150690	1342954537
C	bike3_l1	5940347	148479057	294630539	424136655
C	bike1_l1	5460934	167047688	278444245	415363185
L	titanium_std	4288115	112981473	221548306	321061833
L	lima_sp_1018	3905126	122268237	202018538	294615804
L	light_saber	1962361	73123720	121182536	148183651
L	kyper512	583665	23746115	29159819	47965206
L	newhope512	442904	20179302	22197515	33348406
C	bigquake3	8417432756	202270636364	377925876612	562020487666
I	SIKE751	1276956863	35076394418	65034543091	95435292172
C	ledac3n02	654206583	22088545529	33204889154	49311717864
C	ledac3n03	422621959	13207302620	21471913253	31652596130
C	ledac3n04	348409429	10610768271	17632410215	26189554916
L	frodo976aes	271680254	7731228917	13687032617	20354744190
C	bike2_l3	64570687	1666033437	3267103153	4809670342
L	frodo976cshake	62843060	1910481240	3108297526	4632574867
C	bike3_l3	19787679	508901124	995113294	1460409576
C	bike1_l3	16870580	443832473	877448639	1282483418
L	lima_sp_1822	7976238	278536033	405268040	593104888
L	titanium_med	5184965	136832692	263201275	387703343
L	saber_saber	3592772	114557231	197781939	269506867
L	lima_2p_1024	1720591	57787441	86862571	127899858
L	kyper768	822522	27236761	41665516	62794261
C	bigquake5	18047996001	405917090909	755861946279	1134781808250
C	ledac5n02	1920286857	57906208425	97712032108	143417102225
C	ledac5n03	1156916049	39062485551	58498497083	86570031118
C	ledac5n04	979677004	30790867442	50325425678	73398209884
C	bike2_l5	191981230	4942877634	9737754390	14323301914
C	bike3_l5	45179547	1178254298	2230835600	3328946029
C	bike1_l5	39239959	1038799637	2026972901	2947181565
L	titanium_super	7850349	205872617	396707554	585597005
L	fire_saber	5768648	185423920	290891627	424419431
L	titanium_hi	5656494	147433194	285308423	419717244
L	kyper1024	1229283	39879330	61186928	90346480
L	newhope1024	903408	26955226	44814413	65227557

Tab. E.1.: The Table shows the average number of CPU cycles, per number of messages (1 till 75), for each KEM's implemented in the Double Ratchet. The algorithms are ordered on Security Level (level I first), and the type of KEM is given: C for code-based, L for lattice-based and I for isogeny-based.

		100	125	150	175
	ECDH	37950933	47761483	56669413	66133716
C	bigquake1	82288100894	104229333326	124928183606	145675072028
I	SIKE503	37019856979	46465790332	55532065583	67246553662
C	ledac1n02	17684586280	21673744621	26033470745	30160237673
C	ledac1n04	12309552030	15376002672	18416112069	21692315982
C	ledac1n03	12046775772	15253547276	17895325430	20998566337
L	frodo640aes	11760238239	14650327227	17619079184	20562764958
L	frodo640cshake	2914464745	3647892101	4336706720	5061698120
C	bike2_l1	1885932971	2247602336	2717081857	3136309895
C	bike3_l1	564864950	689378004	834723586	997202310
C	bike1_l1	553565597	689378004	827883240	957511120
L	titanium_std	455236568	537350542	639030656	746297768
L	lima_sp_1018	390683315	487491148	582855403	692388494
L	light_saber	195733336	243401284	292061032	366101307
L	kyper512	62299215	71268431	85256138	99326652
L	newhope512	42040117	53119280	63689198	74439358
C	bigquake3	740433797153	978980137022	1114068459117	1308287872878
I	SIKE751	127976882723	177400051323	190706106565	223905870950
C	ledac3n02	65314202800	82156717362	98392659249	114608111055
C	ledac3n03	43126664133	52876892550	63508441926	73810919035
C	ledac3n04	34786349619	43392441087	52174877853	60939589625
L	frodo976aes	27375357327	33757436986	40635712987	47418520395
C	bike2_l3	6475597557	8041308462	9604373468	11255983292
L	frodo976cshake	7341678253	7692982633	9223701264	10812556657
C	bike3_l3	1951586624	2437351737	2895108517	3427402954
C	bike1_l3	1751187472	2142157528	2603001492	2987227851
L	lima_sp_1822	790514277	986558243	1192900925	1379318644
L	titanium_med	524505037	647686562	770360112	904647270
L	saber_saber	358167844	447922642	536599800	625951933
L	lima_2p_1024	170714644	211359468	253517794	294952550
L	kyper768	103900272	102248768	122481387	141759505
C	bigquake5	1486588880556	1880906275798	2235068400000	2652761719137
C	ledac5n02	193859771062	239122756584	286297867702	335001435021
C	ledac5n03	115605878231	144342118864	173002543694	202020646560
C	ledac5n04	97892615693	121961938821	146616807162	170678634738
C	bike2_l5	19143759888	23843421747	28573912710	33556163120
C	bike3_l5	4492464165	5542571888	6544074120	7597846053
C	bike1_l5	4023711470	4939826109	5871789625	6824088508
L	titanium_super	812283260	977925722	1192019933	1364789322
L	fire_saber	562174190	702705890	847850570	983540969
L	titanium_hi	568901535	698507263	833725243	972340501
L	kyper1024	123443296	150597900	186149137	209766773
L	newhope1024	87590044	107308182	129216145	151651279

Tab. E.2.: The Table shows the average number of CPU cycles, per number of messages (100 till 175), for each KEM's implemented in the Double Ratchet. The algorithms are ordered on Security Level (level I first), and the type of KEM is given: C for code-based, L for lattice-based and I for isogeny-based.

		200	225	250
	ECDH	75164538	85906621	93656910
C	bigquake1	165165003465	186870945657	212981595848
I	SIKE503	75887664491	83839679266	93019602117
C	ledac1n02	35540192967	38968101164	43499384482
C	ledac1n04	25025912519	27811541457	30827227554
C	ledac1n03	24150352404	27098706586	30217164973
L	frodo640aes	23457039911	26564079300	29384814202
L	frodo640cshake	5778193189	6504182448	7242020447
C	bike2_l1	3619249633	4025129535	4499146066
C	bike3_l1	1401286212	1301216174	1424319125
C	bike1_l1	1091442981	1231373610	1374062369
L	titanium_std	856907681	957724727	1084474047
L	lima_sp_1018	818710702	1086410305	997291145
L	light_saber	388976115	438030908	487250194
L	kyper512	113823057	126994690	141145479
L	newhope512	84507813	95367002	105967956
C	bigquake3	1531120660558	1703233884129	1859835174001
I	SIKE751	255149228447	287383562769	319136665311
C	ledac3n02	132059141235	147779124047	162512978983
C	ledac3n03	85348151066	95239723310	105705570673
C	ledac3n04	69683276175	78378042519	87020755930
L	frodo976aes	54222048538	61526621684	67673270299
C	bike2_l3	13132905115	14359715725	16022315769
L	frodo976cshake	12352416340	13867785566	15637868706
C	bike3_l3	3933495339	4333180957	4849085888
C	bike1_l3	3401973749	3897751474	4447763335
L	lima_sp_1822	1578031697	1775868964	1966810717
L	titanium_med	1028301032	1158589649	1285182048
L	saber_saber	714298948	805861883	894090280
L	lima_2p_1024	339336187	379510279	421303722
L	kyper768	161711276	182419039	202078351
C	bigquake5	2982893226304	3424594217878	3748770729140
C	ledac5n02	382499256850	430041237687	478850163104
C	ledac5n03	230805579658	259188059712	289338851021
C	ledac5n04	195951213014	220762255797	244493385055
C	bike2_l5	38511988405	42837890971	47909920720
C	bike3_l5	8900109370	9882973530	10917118419
C	bike1_l5	8011124757	8894841684	9733810196
L	titanium_super	1556600692	1751969440	1945774702
L	fire_saber	1125274788	1263462459	1413714503
L	titanium_hi	1115932856	1250792845	1398350979
L	kyper1024	238858490	267661795	299174863
L	newhope1024	171264995	194632458	213841034

Tab. E.3.: The Table shows the average number of CPU cycles, per number of messages (200 till 275), for each KEM's implemented in the Double Ratchet. The algorithms are ordered on Security Level (level I first), and the type of KEM is given: C for code-based, L for lattice-based and I for isogeny-based.

		275	300	325
	ECDH	104234159	113016745	121830106
C	bigquake1	229110075962	249572675522	270362402987
I	SIKE503	102119512840	111393332464	120257533285
C	ledac1n02	47579398950	51796865554	56345136730
C	ledac1n04	33827015792	36992332328	39841495134
C	ledac1n03	32749426426	35944787289	38688228225
L	frodo640aes	32500766850	35273290866	38367308639
L	frodo640cshake	8142468295	8660004161	9383510723
C	bike2_l1	4930803530	5363911508	5819548347
C	bike3_l1	1568061027	1703174710	1838816956
C	bike1_l1	1508117523	1638879236	1784562391
L	titanium_std	1172974910	1275394867	1382839459
L	lima_sp_1018	1077968261	1315059550	1383418029
L	light_saber	534960038	583804829	632684106
L	kyper512	156720674	169870363	183370135
L	newhope512	116903774	126549810	136882631
C	bigquake3	2107901634661	2244037086541	2417841508290
I	SIKE751	349381800551	381747825977	415243731587
C	ledac3n02	180763604772	196721394057	213063821223
C	ledac3n03	116235252051	126596135136	137187494537
C	ledac3n04	95881553486	104490510679	113791646540
L	frodo976aes	74893813184	81119747087	88283781788
C	bike2_l3	18100208289	19315047613	20819948176
L	frodo976cshake	17031262746	18509171912	20091037476
C	bike3_l3	5337290762	5758410568	6351584647
C	bike1_l3	4706450273	5101499431	5581768120
L	lima_sp_1822	2195935899	2365892921	2559438250
L	titanium_med	1417340130	1546952370	1674511484
L	saber_saber	983682874	1072177763	1161831261
L	lima_2p_1024	462906528	506071082	551019888
L	kyper768	222152759	242980846	263753947
C	bigquake5	4131849087495	4468917949161	4869369826887
C	ledac5n02	526775455898	574366191522	639495819162
C	ledac5n03	327913937979	364024680799	381514679178
C	ledac5n04	270186576469	293441275010	318936861707
C	bike2_l5	52704700169	57222188187	62029125850
C	bike3_l5	12036617646	13434310643	14174023075
C	bike1_l5	10785734396	11705781217	12801602848
L	titanium_super	2156599905	2332051708	2528638334
L	fire_saber	1550773815	1686262766	1826656626
L	titanium_hi	1534313270	1669677662	1807478768
L	kyper1024	328069491	404540317	531944996
L	newhope1024	235820480	257132636	277904840

Tab. E.4.: The Table shows the average number of CPU cycles, per number of messages (300 till 350), for each KEM's implemented in the Double Ratchet. The algorithms are ordered on Security Level (level I first), and the type of KEM is given: C for code-based, L for lattice-based and I for isogeny-based.

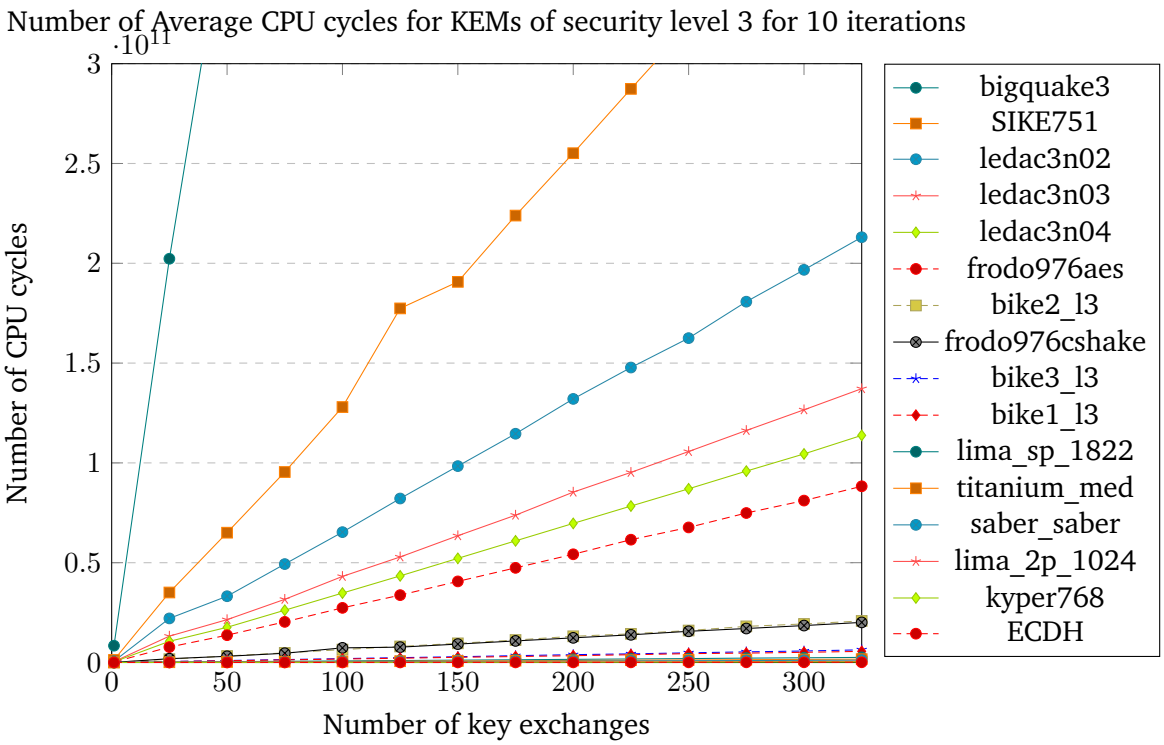

Fig. E.2.: The number of CPU cycles per number of key exchanges (and thus messages), for each KEM of level 3. We plotted the slowest algorithms on purpose out of the graph, so that some other algorithms could be better distinguish. The complete data set can be found in the Appendix.

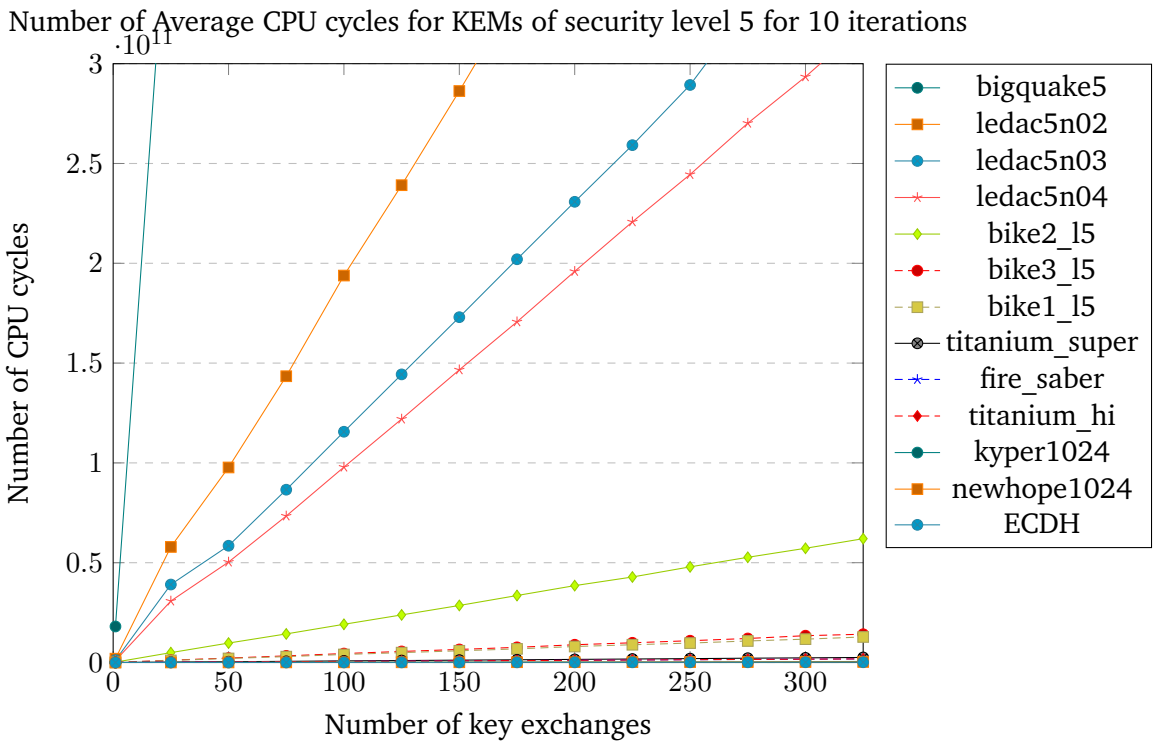

Fig. E.3.: The number of CPU cycles per number of key exchanges (and thus messages), for each KEM of level 5.

Energy consumption

F

In Table F.1, F.2 and F.3 the run time, in milliseconds, an their energy consumption, in milli Joule, are shown for different post-quantum KEM algorithms for one such senario in security Level I, III and V. The data is taken from [BH18]. The tables show only the NIST PQC candidates for Key Encapsulation Mechanisms. A big difference between runtime and energy consumption can be seen. The worst case PQ KEM for a security level I is CFPKM, and runs for half a second, with an energy consumption of 14,6 Joule. While the best level I case PQ KEM, Lepton, only runs 0.05 millisecond with an energy consumption of 1.3 milli Joule. For Security Level III and V the worst case KEM PQ is BIGQUAKE, having relatively 2.8 and 5.2 seconds, and 73 and 137 Joule. While the best case PQ KEM still is Lepton with 0.15 milliseconds and 4.2 Joule for both security levels.

Scheme	Type	Time (mili s)	Energy (mili J)
Lepton	?	0.0484	1.31
Rlizard KEM	?	0.09	2.43
NewHope CPA	Lattices	0.414	11.05
NTRUEncrypt KEM	Lattices	0.47	12.71
SABER	Lattices	0.57	15.3
Round2-u KEM	Lattices	0.61	16.44
CRYSTALSKyber	Lattices	0.636	16.8
Quroboros-R	Codes	0.69	18.23
NewHope CCA	Lattices	0.69	18.74
LIMA CPA	Lattices	0.925	24.38
EMBLEM	Lattices	1.927	27.17
LAKE	Codes	1.2	31.55
LIMA CCA	Lattices	1.26	33.31
FRODO	Lattices	1.415	36.76
BIKE	Code	1.459	37.52
HQC	Codes	1.48	38.62
Titanium CCA	Lattices	1.91	50.07
RQC	Codes	2.39	63.99
Lizard KEM	Lattices	3.17	83.68
DING	Lattices	4.76	128.12
LOCKER	Codes	5.16	138.43
LOTUS KEM	Codes	10.22	273.63
Round2-n KEM	Lattices	10.56	293.66
Ramstake	Lattices	15.61	422.97
NTS-KEM	Codes	16.76	451.24
LEDA KEM	Codes	44.263	1177.76
NTRU-HRSS-KEM	Lattices	58.55	1585.64
SIKE	Isogeny	115.73	3110.01
Old Manhattan	Lattices	148.47	3997.76
BIGQUAKE	Code	303.9	8011.21
RLCE-KEM	Codes?	395.32	10613.23
CFPKM	?	547	14557.49

Tab. F.1.: The runtime, in milliseconds, and the energy consumption in milli Joule, for one key creation, encapsulation and decapsulation of the post-quantum KEM algorithm for Security Level I are shown. All algorithms are from the NIST competition. The table is created using table 7, 8 and 9 (table with key generation, key encapsulation and key decapsulation representatively) from [BH18].

Scheme	Type	Time (mili s)	Energy (mili J)
Lepton	?	0.1546	4.14
Rlizard KEM	?	0.21	5.71
KINDI-KEM	Lattices	0.28	7.55
CRYSTALSKyber	Lattices	0.995	26.55
SABER	Lattices	1.04	27.81
Quroboros-R	Codes	1.11	29.01
NTRUEncrypt KEM	Lattices	1.12	29.32
LAKE	Codes	1.61	42.15
LIMA CPA	Lattices	1.98	52.16
Titanium CCA	Lattices	2.17	57.32
LIMA CCA	Lattices	2.64	70.63
FRODO	Lattices	2.803	73.48
HQC	Codes	3.17	84.17
Round2-u KEM	Lattices	5.29	142.95
LOCKER	Codes	5.61	149.69
RQC	Codes	5.86	155.82
BIKE	Code	8.52	223.16
DAGS	Code	11.4056	301.446
Lizard KEM	Lattices	11.66	310.02
Round2-n KEM	Lattices	15.54	425.28
LOTUS KEM	Codes	18.47	500.92
DME	Multivariance	26.5	681.98
NTS-KEM	Codes	45.46	1208.33
Ramstake	Lattices	69.16	1869.06
LEDA KEM	Codes	126.552	3361.8
QC-MDPC	Codes	164.88	4429.46
Old Manhattan	Lattices	285.8	7762.45
SIKE	Isogeny	378.82	10227.89
CFPKM	?	1484	39675.46
BIGQUAKE	Code	2767.4	73280.46

Tab. F.2.: The runtime, in milliseconds, and the energy consumption in milli Joule, per post-quantum KEM algorithm for Security **Level III** are shown. All algorithms are from the NIST competition. The table is created using table 7, 8 and 9 (table with key generation, key encapsulation and key decapsulation representatively) from [BH18].

Scheme	Type	Time (mili s)	Energy (mili J)
Lepton	?	0.155	4.15
Rlizard KEM		0.297	7.76
NewHope CPA	Lattices	0.78	21.03
Round2-u KEM	Lattices	1.16	31.18
Three Bears	Lattices	1.2	31.89
CRYSTALSKyber	Lattices	1.395	36.21
NewHope CCA	Lattices	1.4	38.065
Quroboros-R	Codes	1.52	40.64
SABER	Lattices	1.56	42.07
LAKE	Codes	1.84	48.81
Hila5	Lattices	2.54	68.24
Titanium CCA	Lattices	2.96	78.13
LIMA CPA	Lattices	3.44	91.79
HQC	Codes	4.54	119.66
LIMA CCA	Lattices	5.02	134.11
LOCKER	Codes	6.51	174.43
Lizard KEM	Lattices	7.28	194.98
RQC	Codes	7.36	198.96
BIKE	Code	7.87	205.78
DING	Lattices	9.37	252.71
NTRU Prime	Lattices	18.64	506.45
Round2-n KEM	Lattices	21.93	599.68
LOTUS KEM	Codes	27.06	725.7
Mersenne-756839	Lattices	33.43	901.68
NTS-KEM	Codes	88.9	2394.98
DME	Multivariance	100.547	2598.96
DAGS	Code	107.926	2894.493
NTRUEncrypt KEM	Lattices	210.72	5724.48
LEDA KEM	Codes	365.619	9705.73
Old Manhattan	Lattices	548.86	14857.18
Classic McEliece	Code	1019.23	28091.42
RLCE-KEM	Codes?	3891.64	104212.54
BIGQUAKE	Code	5190.2	137231.25

Tab. F.3.: The runtime, in milliseconds, and the energy consumption in milli Joule, per post-quantum KEM algorithm for Security **Level V** are shown. All algorithms are from the NIST competition. The table is created using table 7, 8 and 9 (table with key generation, key encapsulation and key decapsulation representatively) from [BH18].

Bibliography

[Alk+15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. *Post-quantum key exchange - a new hope*. Cryptology ePrint Archive, Report 2015/1092. https://eprint.iacr.org/2015/1092. 2015 (cit. on pp. 5, 17).

[And+09] Allison M Anderson, Gary A Mirka, Sharon MB Joines, and David B Kaber. „Analysis of alternative keyboards using learning curves“. In: *Human factors* 51.1 (2009), pp. 35–45 (cit. on p. 56).

[Ara+17] N Aragon, PSLM Barreto, S Bettaieb, et al. *BIKE—bit flipping key encapsulation*. 2017 (cit. on p. 53).

[Bal+17] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. „Design of LEDAkem and LEDApkc instances with tight parameters and bounded decryption failure rate (V1.0)“. In: *NIST submission* (2017) (cit. on p. 53).

[Bar16] Elaine Barker. *NIST Special Publication (SP) 800-57 Part 1 Revision 4*. https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-4/final. 2016 (cit. on p. 19).

[Ber+08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post Quantum Cryptography*. 1st. Physical copy from Andreas Peter. Springer Publishing Company, Incorporated, 2008. ISBN: 3540887016, 9783540887010 (cit. on pp. 16, 17).

[Ber09] Daniel J. Bernstein. *Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?* http://cr.yp.to/hash/collisioncost-20090823.pdf. 2009 (cit. on p. 16).

[BH18] Tanushree Banerjee and M. Anwar Hasan. „Energy Consumption of Candidate Algorithms for NIST PQC Standards“. In: online, 2018 (cit. on pp. 69, 70, 107–110).

[Bin+17] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. *Transitioning to a Quantum-Resistant Public Key Infrastructure*. Cryptology ePrint Archive, Report 2017/460. https://eprint.iacr.org/2017/460. 2017 (cit. on p. 4).

[BK04] Mihir Bellare and Tadayoshi Kohno. „Hash Function Balance and Its Impact on Birthday Attacks“. In: *Advances in Cryptology - EUROCRYPT 2004*. Ed. by Christian Cachin and Jan L. Camenisch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 401–418. ISBN: 978-3-540-24676-3 (cit. on pp. 13, 15).

[Bor+04] Nikita Borisov, Ian Goldberg, and Eric Brewer. „Off-the-record Communication, or, Why Not to Use PGP“. In: *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*. WPES '04. Washington DC, USA, 2004, pp. 77–84. ISBN: 1-58113-968-3. URL: https://otr.cypherpunks.ca/otr-wpes.pdf (cit. on p. 14).

[Bos+16a] Joppe Bos, Craig Costello, Léo Ducas, et al. *Frodo: Take off the ring! Practical, Quantum-Secure Key Exchange from LWE*. Cryptology ePrint Archive, Report 2016/659. https://eprint.iacr.org/2016/659. 2016 (cit. on pp. 5, 17, 53).

[Bos+16b] Joppe Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. *Post-quantum key exchange for the TLS protocol from the ring learning with errors problem*. https://s3.amazonaws.com/files.douglas.stebila.ca/files/research/papers/SP-BCNS15-full.pdf. 2016 (cit. on p. 4).

[BR05] Mihir Bellare and Phillip Rogaway. „Introduction to modern cryptography“. In: *Ucsd Cse* 207 (2005), p. 207 (cit. on p. 1).

[Bra+98] Gilles Brassard, Peter HØyer, and Alain Tapp. „Quantum cryptanalysis of hash and claw-free functions“. In: *LATIN'98: Theoretical Informatics*. Ed. by Cláudio L. Lucchesi and Arnaldo V. Moura. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 163–169. ISBN: 978-3-540-69715-2 (cit. on p. 15).

[Bre+01] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. „Provably authenticated group Diffie-Hellman key exchange“. In: *Proceedings of the 8th ACM conference on Computer and Communications Security*. ACM. 2001, pp. 255–264 (cit. on p. 86).

[Cam+15] Matthew Campagna, Lidong Chen, O Dagdelen, et al. „Quantum safe cryptography and security: an introduction, benefits, enablers and challengers“. In: *European Telecommunications Standards Institute (ETSI), ISBN* 979-10 (2015), pp. 92620–03 (cit. on pp. 13, 15).

[Can01] Ran Canetti. „Universally composable security: A new paradigm for cryptographic protocols“. In: *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*. IEEE. 2001, pp. 136–145 (cit. on p. 20).

[CG+16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. „On Post-Compromise Security“. In: (2016). https://eprint.iacr.org/2016/221 (cit. on p. 28).

[CG+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling†, Luke Garratt, and Douglas Stebila. „A Formal Security Analysis of the Signal Messaging Protocol“. In: (2017). https://eprint.iacr.org/2016/1013.pdf (cit. on p. 14).

[CG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. „On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees“. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 1802–1819 (cit. on p. 86).

[Cha+17] Olive Chakraborty, Jean-Charles Faugère, and Ludovic Perret. „CFPKM: A Key Encapsulation Mechanism based on Solving System of non-linear multivariate Polynomials 20171129“. PhD thesis. UPMC-Paris 6 Sorbonne Universités; INRIA Paris; CNRS, 2017 (cit. on p. 17).

[Cha18] Dave Chaffey. *Global social media research summary 2018*. Visited January 2019 https://www.smartinsights.com/social-media-marketing/social-media-strategy/new-global-social-media-research/. Visitied jan 2019. 2018 (cit. on p. 1).

[Che+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, et al. *Report on Post-Quantum Cryptography*. NISTIR 8105. https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf. 2016 (cit. on pp. 5, 16, 50).

[Cos+16] Craig Costello, Patrick Longa, and Michael Naehrig. *Efficient algorithms for supersingular isogeny Diffie-Hellman*. Cryptology ePrint Archive, Report 2016/413. https://eprint.iacr.org/2016/413. 2016 (cit. on pp. 5, 18).

[Cry] Cryptocat. *Chat with your friends, securely.* Visited September 2018 https://crypto.cat/ (cit. on p. 4).

[Din+11] Hang Dinh, Cristopher Moore, and Alexander Russell. „McEliece and Niederreiter Cryptosystems That Resist Quantum Fourier Sampling Attacks“. In: (2011). URL: https://www.iacr.org/archive/crypto2011/68410758/68410758.pdf (cit. on p. 18).

[Dod+09] Yevgeniy Dodis, Jonathan Katz, Adam Smith, and Shabsi Walfish. „Composability and On-Line Deniability of Authentication“. In: *Theory of Cryptography*. Ed. by Omer Reingold. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 146–162. ISBN: 978-3-642-00457-5 (cit. on p. 52).

[Erm+16] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. „End-to-End Encrypted Messaging Protocols: An Overview“. In: *Internet Science*. Ed. by Franco Bagnoli, Anna Satsiou, Ioannis Stavrakakis, et al. Cham: Springer International Publishing, 2016, pp. 244–254. ISBN: 978-3-319-45982-0 (cit. on p. 14).

[Fro+14] Tilman Frosch, Christian Mainka, Christoph Bader, et al. *How Secure is TextSecure?* Cryptology ePrint Archive, Report 2014/904. https://eprint.iacr.org/2014/904. 2014 (cit. on pp. 14, 22).

[Gro96] Lov K. Grover. „A Fast Quantum Mechanical Algorithm for Database Search“. In: *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*. STOC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 212–219. ISBN: 0-89791-785-5. URL: http://doi.acm.org/10.1145/237814.237866 (cit. on pp. 2, 15).

[Kam+18] Panos Kampanakis, Peter Panburana, Ellie Daw, and Daniel Van Geest. *The Viability of Post-quantum X.509 Certificates*. Cryptology ePrint Archive, Report 2018/063. https://eprint.iacr.org/2018/063. 2018 (cit. on p. 4).

[KR17] Ralf Küsters and Daniel Rausch. „A Framework for Universally Composable Diffie-Hellman Key Exchange“. In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 881–900 (cit. on pp. 20, 39).

[Kra10] Hugo Krawczyk. „Cryptographic extraction and key derivation: The HKDF scheme“. In: *Annual Cryptology Conference*. Springer. 2010, pp. 631–648 (cit. on p. 11).

[Lan+16] A. Langley, M. Hamburg, and S. Turner. *Elliptic Curves for Security*. http://www.ietf.org/rfc/rfc7748.txt. 2016 (cit. on pp. 13, 15).

[Len04] Arjen K Lenstra. „Key Length. Contribution to The Handbook of Information Security“. In: (2004) (cit. on p. 12).

[Lue+17] I Luengo, M Avendaño, and M Marco. „DME: A public key, signature and KEM system based on double exponentiation with matrix exponents“. In: *preprint* (2017) (cit. on p. 17).

[Lun18] Joshua Lund. *Signal partners with Microsoft to bring end-to-end encryption to Skype*. Visited July 2018 https://signal.org/blog/skype-partnership/. 2018 (cit. on p. 1).

[Mar13] Moxie Marlinspike. *Simplifying OTR deniability*. https://signal.org/blog/simplifying-otr-deniability/. Blog. 2013 (cit. on pp. 26, 51).

[Mar16a] Moxie Marlinspike. *Facebook Messenger deploys Signal Protocol for end-to-end encryption*. Visited July 2018 https://signal.org/blog/facebook-messenger/. 2016 (cit. on pp. 1, 4).

[Mar16b] Moxie Marlinspike. *Open Whisper Systems partners with Google on end-to-end encryption for Allo*. Visited July 2018 https://signal.org/blog/allo/. 2016 (cit. on pp. 1, 4).

[Mar16c] Moxie Marlinspike. *WhatsApp's Signal Protocol integration is now complete*. Visited July 2018 https://signal.org/blog/whatsapp-complete/. 2016 (cit. on pp. 1, 4).

[McE78] R. J. McEliece. „A Public-Key Cryptosystem Based On Algebraic Coding Theory“. In: *The Deep Space Network Progress Report* (Feb. 1978), pp. 114–116. URL: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19780016269.pdf\#page=123 (cit. on p. 18).

[Men+96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN: 0849385237 (cit. on p. 7).

[Mos15] Michelle Mosca. „Cybersecurity in an era with quantum computers: will we be ready?“ In: (2015) (cit. on p. 2).

[Mur+12] David Murray, Terry Koziniec, Kevin Lee, and Michael Dixon. „Large MTUs and internet performance“. In: *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*. IEEE. 2012, pp. 82–87 (cit. on p. 74).

[Nie10] Jakob Nielsen. *Website Response Times*. Visited January 2019 https://www.nngroup.com/articles/website-response-times/. Visitied jan 2019. 2010 (cit. on p. 58).

[NIS16] NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. NISTIR. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf. 2016 (cit. on pp. 5, 16, 18).

[Ope17] OpenSignal. *Global State of Mobile Networks (February 2017)*. Visited in December 2018, https://opensignal.com/reports/2017/02/global-state-of-the-mobile-network. 2017 (cit. on p. 57).

[Par13] Gómez J.L. Pardo. „Chapter 7: Introduction to Public-Key Cryptography: The Diffie–Hellman Protocol“. In: (2013). URL: https://link.springer.com/chapter/10.1007/978-3-642-32166-5_7 (cit. on p. 9).

[PM16a] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorihm*. Internet-Draft. Revision 1. Signal, Nov. 2016. URL: https://signal.org/docs/specifications/doubleratchet/ (cit. on pp. 23, 28, 29, 32, 36).

[PM16b] Trevor Perrin and Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Internet-Draft. Revision 1. Signal, Nov. 2016. URL: https://signal.org/docs/specifications/x3dh/ (cit. on pp. 24, 36, 53).

[PM17] Trevor Perrin and Moxie Marlinspike. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. Internet-Draft. Revision 2. Signal, Apr. 2017. URL: https://signal.org/docs/specifications/sesame/ (cit. on p. 34).

[PS96] David Pointcheval and Jacques Stern. „Security proofs for signature schemes“. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1996, pp. 387–398 (cit. on p. 11).

[RA18] Raphael Robert and J. P. Aumason. *Wire and post-quantum resistance*. https://blog.wire.com/blog/post-quantum-resistance-wire/. 2018 (cit. on pp. 4, 45).

[Rat] Ratatype. *Average typing speed infographic*. Visited in December 2018, https://www.ratatype.com/learn/average-typing-speed/ (cit. on p. 56).

[RF18] Marco Baldi Vincent Dupaquis Jacob Alperin-Sheiff Ryo Fujita Daniel Bernstein. *Total number of PQ NIST proposals by field*. Visited October 2018, https://groups.google.com/a/list.nist.gov/forum/#!topic/pqc-forum/1lDNio0sKq4. 2018 (cit. on p. 16).

[Roe+03] Helena Roeber, John Bacus, and Carlo Tomasi. „Typing in thin air: the canesta projection keyboard-a new method of interaction with electronic devices“. In: *CHI'03 extended abstracts on Human factors in computing systems*. ACM. 2003, pp. 712–713 (cit. on p. 56).

[Ros+18] Avi Rosenfeld, Sigal Sina, David Sarne, Or Avidov, and Sarit Kraus. *A Study of WhatsApp Usage Patterns and Prediction Models without Message Content*. https://arxiv.org/abs/1802.03393. 2018 (cit. on pp. 56, 57).

[RP00] Eleanor Rieffel and Wolfgang Polak. „An Introduction to Quantum Computing for Non-physicists“. In: *ACM Comput. Surv.* 32.3 (Sept. 2000), pp. 300–335. ISSN: 0360-0300. URL: http://doi.acm.org/10.1145/367701.367709 (cit. on p. 15).

[RS06] Alexander Rostovtsev and Anton Stolbunov. *PUBLIC-KEY CRYPTOSYSTEM BASED ON ISOGENIES*. Cryptology ePrint Archive, Report 2006/145. https://eprint.iacr.org/2006/145. 2006 (cit. on pp. 5, 18).

[Sho94] Peter W. Shor. *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.47.3862&rep=rep1&type=pdf. 1994 (cit. on pp. 2, 15).

[Sig] Signal. *TECHNICAL INFORMATION*. Visited September 2018 https://signal.org/docs/ (cit. on p. 1).

[Sig18] Signal. *Curve25519*. https://github.com/signalapp/libsignal-protocol-c/tree/master/src/curve25519. Commit e5eb4017c1f45d50fca77d2396283a01104860fa. 2018 (cit. on p. 52).

[SM17] Douglas Stebila and Michele Mosca. „Post-Quantum Key Exchange for the Internet and the Open Quantum Safe Project“. In: (2017). https://eprint.iacr.org/2016/1017.pdf (cit. on p. 4).

[Sma17] Nigel P. Smart. *LIMA 1.1 A PQC Encryption Scheme*. 2017 (cit. on p. 53).

[Tel] Telegram. *Telegram, a new era of messaging*. Visited September 2018 https://telegram.org/ (cit. on p. 4).

[Thr] Threema. *The messenger that puts security and privacy first.* Visited September 2018 https://threema.ch/en/ (cit. on p. 4).

[Unr10] Dominique Unruh. „Universally composable quantum multi-party computation“. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2010, pp. 486–505 (cit. on pp. 20, 40).

[Vaj17] István Vajda. „On Classical Cryptographic Protocols in Post-Quantum World“. In: *International Journal of Computer Network and Information Security* 9.8 (2017), p. 1 (cit. on pp. 20, 40).

[Vri16] Simon and de Vries. *Achieving 128-bit Security against Quantum Attacks in OpenVPN*. 2016 (cit. on p. 4).

[Wic] WickrMe. *The messenger that puts security and privacy first.* Visited September 2018 https://wickr.com/ (cit. on p. 4).

[Wir] Wire. *The most secure collaboration platform.* Visited September 2018 https://wire.com/en/ (cit. on p. 4).