University of Geneva

Master thesis

SECURE GROUP MESSAGING PROTOCOL BASED ON POST-QUANTUM CRYPTOGRAPHY

Quentin RIVOLLAT

Tutor: Dr. Eduardo Solana

April 17, 2021

Abstract

In the past few years messaging applications providing secure communication have become ubiquitous. The widespread use of well designed protocols such as Signal with strong security properties enables message exchanges with a robust end-to-end cryptographic protection. However, these security assumptions do not hold when groups are involved in the communications. Consequently, an attacker who successfully compromises a single group member can inject and read messages without being spotted by others.

Furthermore, the generalized belief that a fully fledged quantum computer will be built in the years to come has threaten the intractability of the underlying algorithms and lead the crypto community and the standardization institutions to come up with secure post-quantum candidate algorithms.

This work addresses these two issues by combining the SIKE protocol, a post quantum candidate in the ongoing NIST standardization contest and the recently published ART protocol featuring strong security guarantees for group messaging frameworks. We first describe from a theoretical perspective the mathematical and cryptographic properties of SIKE and ART and then introduce for the first time a combination of both protocols that results in a quantum resistant and secure group messaging protocol.

Finally, we have developed a prototype implementation together with a benchmark analysis showing that such a combination is acceptable in terms of performance overhead when compared to the existing counterpart. By measuring time and memory requirements of the main phases of the combined protocol we demonstrate that this strong security solution can potentially be adopted by users in their daily group message exchanges.

Contents

Acknowledgements

Throughout the writing of this thesis, I have received a great deal of support and assistance.

I would first like to express my warmest and deepest appreciation to my supervisor, Dr. Eduardo Solana, whose knowledge and advices made this paper possible. His continuous support has ceaselessly encouraged me throughout all this work.

I would also like to thank my family, always present for me, in particular during this unusual year of work at home.

Finally, some special words of gratitude go to my friends, Mélanie, Barbara, Clément, Laura, Laurane, Romain and Benjamin, who made all these university years such a pleasant time, and whose support and encouragement was worth so much more than I can express on paper.

Chapter 1

Introduction

Cryptography is the art of encoding messages in order to make them impossible to be read by unauthorized users. The strength of a cryptographic protocol relies on the secret key length and the mathematical strength of the algorithm, except when backdoors are added to softwares. With Edward Snowden revelations in 2013, the public rediscovered the importance of privacy and encryption. Even though it has always been present in the cryptographic community, this event enlightened the importance of open-source protocols. After this event, the cryptography community renewed their efforts to create next-generation secure system, and one of the major field was instant messaging systems, as always more and more persons are using everyday this type of application to communicate. One of the first security protocol for instant messaging was Off-the-Record messaging, or OTR. It provided encryption and authentication, based on ephemeral Diffie-Hellman keys, making it impossible to derive earlier keys based on a leaked one. Nowadays, some of the biggest messaging application, that are Whatsapp, Facebook messenger and Signal have one thing in common : they all used the Signal protocol. This protocol relies on two important parts : X3DH and Double Ratchet. The former allows authentication, while the latter key establishment and encryption, with a regular renewal of the keys. This protocol is today considered as a reference in the cryptographic world, even though it is still officially unstandardized.

But like everything, the Signal protocol is not perfect. As if backdoors were not enough, we are now facing quantum computers. For the last three decades, research into quantum computing has importantly increased, encouraged by Shor's and Grover's algorithms. A major event was in October 2019 when Google announced having acheived quantum supremacy, with a 53-qubits computer. This type of machine, although still very new, represents a future threat upon many cryptographic primitives, that should be taken seriously. The first International Workshop on Post-Quantum cryptography dates back to 2006, where scientists already talked about the possible risks existing at the time. Indeed, even if quantum algorithms are not yet able to break most encryption schemes, there is the possibility that attackers are recording vast volumes of communication, protected by RSA, ECC, or other cryptographic systems that will break one day or the other against quantum computers. To face the issue, the US National Institute of Standards and Technology (NIST) decided to initiate a process whose objective is to "solicit, evaluate and standardize one or more quantum-resistant

public-key cryptographic algorithms". Among the 69 schemes proposed in the first round, only 4 encryption / key encryption mechanism (KEM) schemes and 3 signature schemes were still studied at the third round, along with 5 alternate candidates, among which the supersingular elliptic curves based protocol SIKE. This protocol, as its name indicates, relies on elliptic curves and isogenies to generate a shared secret between two parties. Its goal is to replace Elliptic Curve Diffie-Hellman (ECDH), presents in many cryptographic protocol but that is threatened by quantum computers.
The second issue of the Signal protocol is the difficulty to scale up to group communication. Various messaging app implemented their own protocol, but most of the time, either asynchronicity or Post-Compromise secrecy (PCS) was unachievable. The former property refers to the ability for the protocol to initiate a group conversation without requiring every user to be online, while the second is related to the capacity to endure key leak, with minimum consequences on the whole protocol. The solution proposed by Open Whisper Systems, the software development group responsible for the Signal protocol, is simply to apply this protocol between each member of the group, to keep hold of the security properties making its success. But this solution becomes inefficient as soon as the number of members grows. In 2018, Katriel Cohn-Gordon and al. proposed a system designed to solve the issues of asynchronicity and PCS, with the Asynchronous Ratchet Tree protocol, or ART. It is based on the idea to construct a binary tree, on which each node and leave possesses a key pair, that can be used to compute a root key, the former being the input of an encryption algorithm. Moreover, it aims to facilitate the group modification, through its corresponding tree, and in particular the possibility to regularly update the user's keys, in order to maximize the security.

In this work, we propose an implementation of a protocol based on the combination of ART and SIKE, in order to face both the issue of post-quantum cryptography and the reaching of asynchronicity and PCS. The first part of this new protocol is strongly based on the authentication made in the Signal protocol, through X3DH, while the rest follow the guidances given in the ART paper. The details of the implementation are followed by an analysis of the protocol, with first a look at the computational impact, between the use of ECDH, of our SIKE implementation, and the optimized one proposed by the sike organisation. Eventually, we study the feasibility of the whole messaging application, by measuring the delay necessary for each step of the protocol. The implementation will have many improvable aspects, but it will act as a proof of work, showing such a protocol can be developed and used by an average user.

Chapter 2

Signal Protocol

As indicated in the introduction, the goal of this work is to create a post-quantum group messaging application. The concept of a group messaging app can be seen as an extension and adaptation of a two-party communication, and the reference is the Signal protocol. In Section 2.1, we present the basic knowledge related to cryptography, and a brief introduction about the Signal Protocol. Section 2.2 is about the methods to encrypt messages and keep them secure. In Section 2.3, we describe how authentication is made, through the X3DH protocol. In Section 2.4, we continue on the Signal protocol to explain how messages are encypted and the method to regularly update the keys.

2.1 Introduction

In this section, we first define the keywords that will occur multiple times throughout this paper, and that are important, although it mainly involves encryption and authentication. We then start talking about the Signal protocol, by presenting the three main steps, that any user would have to follow in order to reach the properties presented.

2.1.1 Main concepts

In the very first chapter of this paper, we talked about the wishes to keep the communication as private as possible. This concept of privacy in the communication world is deeply linked to the domain of encryption.

Encryption: set of methods whose goal is to keep a message unreadable from any person who does not have the necessary information. The initial clear message is called a *plaintext*, and to obtain a message impossible to read, one needs to transform it in an alternate form, known as *ciphertext*. The process of going from a *plaintext* to a *ciphertext* is the *encryption*, while going in the opposite direction is the *decryption*.

Most conversations are made between two persons, more commonly called parties. Then, how does one send a message to another party? The basic answer would be to know their address and take care of the transmission themselves. As long as there are few parties, this might be feasible, but as soon as the number increases, it might quickly become a mess. Instead, an easier solution would be to follow the real-life method, where one gives a letter along the address to a post office, the latter being in charge of delivering the document. With messaging app, this is similar: the sender gives its message to a server, which takes care to transmit it to the correct party. But much as we trust the postman not to look into the envelop, there usually exists not much trust when it comes to the Internet. Two persons conversing, who wants to keep private their communication from any outsider party, need an End-to-End encryption protocol.

End-to-end encryption: system of communication where only the communicating parties, meaning the sender and the recipient, are able to read the messages sent [Yel16]

There are mainly two methods to establish an End-to-end encryption scheme between two parties: Symmetric Key Encryption, also called Symmetric Key Cryptography, and Asymmetric Encryption, also known as Public-Key Cryptography.

Symmetric Key Encryption: system providing secrecy between two or more parties by using the same key k, a shared secret, for both encryption and decryption [DK07]

Public Key Encryption (PKE): system providing secrecy using pairs of keys, where each party owns a public key, used to encrypt, and a private, used to decrypt or sign [SBBB12]. The public key is accessible to anyone, while the private one is kept secret by its owner.
If Alice wants to send a message to Bob, she uses his public key to encode the *plaintext*, and Bob will decrypt the *ciphertext* by using his private key.

In the case where it is the Symmetric Key Encryption method that is chosen, the parties will first need to agree on a shared key before starting to communicate. This is done with a Key-Exchange Protocol.

Key-Exchange Protocol (KEP): mechanisms by which two parties that communicate over an adversarially-controlled network can generate and/or exchange a common secret key [CK01].

With the same key, both parties are able to encrypt and decrypt their communication. These processes are made by using a specific algorithm, such as AES (Advanced Encryption Standard). But these kinds of algorithm need as input a key with a specific format, format that a KEP may not always supply. In order to obtain a correct key from a shared secret, a KDF is necessary.

Key Derivation Function (KDF): algorithm that generates a cryptographic key from a password, a master key or a randomly generated password [Spa16].
It can also be used to stretch keys or derive multiple keys at once.

Now that we know how a message can be kept secret with an encryption, we need know who sent it.

Authentication: method used to provide the identity of an entity [ATAa14]

Similarly to the real-life model, one could add at the end of the message a signature, to prove they are the source.

Digital Signature: mathematical scheme for demonstrating the authenticity of digital messages or documents, by computing a unique fingerprint. [Pau17]
Using asymmetric keys is a good way to sign data and let the recipient check the signature.

Eventually, there is one last concept to take into consideration. Let's say Alice and Bob are in a two-party communication, which is encrypted, and they are able to authenticate one another's messages through a digital signature. They thus reach the main requirement to make a communication channel secure. These properties stand as long as both parties comply. But what if Bob decides to reveal all the communication to a third party? Since he knows the keys, he can show anybody Alice's messages and any information coming along, such as her signature. Most of the time, we try to to avoid this kind of situation.

Deniability: property enabling any party implied in a communication to deny that they have sent or generated a message to a third party. [FMB+16]

The presentation all these concepts could be summed up in the following way : we want to be able to send encrypted messages only concerned parties can read, meaning only them can derive the correct key to decrypt, and we also want to be sure of the origin of the messages, as long as we can keep non-concerned party outside the whole process, in any aspect.

2.1.2 Phases of the protocol

The previous section showed that the core concept in cryptography is the idea of key. With keys, a party can encrypt, decrypt and even authenticate the sender. That is why the Signal Protocol's main phases are linked to keys as well.

The Signal protocol can be is divided into three parts:

1. Key generation
2. Key agreement
3. Key renewal

The first part, Key generation, happens before the beginning of any communication. The idea is to create several pairs of keys, each one made up of a private key and its corresponding public key, in order to use them when a party wants to initiate a secure communication. The idea of using public and private keys signifies it is asymmetric cryptography that is applied here.

Then, the second part, Key agreement, is done through the first message sent from a party to another, where both parties agree on an initial key, necessary for the rest of the Signal protocol. The protocol applied is call eXtended Triple Diffie-Hellman, or X3DH, and as one can guess, it uses Diffie-Hellman, meaning this part is based on asymmetric cryptography as well.

Finally, the last step, Key renewal, takes place each time a message is sent, leading to an update of the key. Here, it is the Double Ratchet algorithm that is used.

2.2 Encryption and Security

We presented in the previous Section the important notions about cryptography. These properties are usually set up at the beginning of a protocol, but what if something happens during the process? It is quite difficult to have a perfect system, and so we need to limit the consequences of any leak of key(s). The properties we want to achieve all through a protocol in order to keep it secure are *End-to-end encryption*, *Forward Secrecy* and *Post-compromise*. In particular, the two last ones protect a system before and after this kind of damage.

2.2.1 End-to-end encryption

The first objective of a secure messaging app is to provide absolute secrecy about the transmitted messages, along with authentication from every parties involved in a session.
As mentioned in the first part of this chapter, an end-to-encryption allows only the concerned parties to encrypt and decrypt a message. No one else, not even an intermediate server, should have the possibility to obtain the plaintext corresponding to a ciphertext.

There are essentially two ways to encrypt a message: either with a symmetric key or with a pair of asymmetric keys. Both methods have their advantages and their inconveniences.

Indeed, on one hand, symmetric key encryption:

- is faster: since the assymetric decryption of a message should be harder than encrypting it, the former step is much slower than the latter, but both are slower than the symmetric method. See the benchmark for comparison [Doe19],
- uses shorter keys: usually, the longer a key, the more secure a system. But given a level of security, public keys encryption needs larger keys in order to reach the same level as symmetric cryptography.

And on the other hand, asymmetric key encryption:

- enables message authentication: since the key pair private-public is supposed to be unique, a message signed with a user's private key can only be verified with their public key.
- allows the computation of a shared secret: combining Alice's private key with Bob's public key should give a similar result to a combination between Alice's public key and Bob's private key. This is the core concept of Diffie-Hellman.

Instead of having to choose only one method, why not mixing both ? For example, the symmetric system could be used to encrypt a message, quickly and in a secure way, while the asymmetric one

could be in charge of deriving the key used in the encryption and authenticate the users at the same time.

All these concepts established up to now enable any party to set a secure communication with anyone, and this security aspect is supposed to stay as long as no information is leaked. But it would be unwise to hope such thing. A good messaging protocol should be able to face this kind eventuality.

Single point of failure

There is one major single point of failure that comes right in mind in this kind of scenario: what if the shared key used to encrypt and decrypt is revealed or obtained by an adversary ? The repercussion would be immediate: all messages encrypted with this specific key will lose the property of secrecy, meaning the new possessor of the symmetric key will be able to decrypt them all.

This eventuality should be avoided at all cost. If a key is leaked, as few messages' encryption as possible should be threatened, the "best" situation being only the ciphertext encoded by the key in question is jeopardized.

Nevertheless, public key cryptography can be very useful, in particular for key exchange. That is why it is in the two first steps of the Signal protocol.

2.2.2 Forward Secrecy

If we want to sum up what was said up to now, we can say that :

1. A ciphertext is vulnerable if the key used to encrypt the corresponding plaintext has been leaked
2. A good secure communication system is one where a single point of failure would put at risk the security of as few ciphertexts as possible

So, the logical solution would be that at each encryption stage, one uses a new key. This new key cannot be totally random, since both parties implicated in a conversation must be able to derive the same shared key in order to encrypt and decrypt. Instead, we derive the key at time t based on the previous one, by using a Key-Derivation Function:

$$k_t = KDF(k_{t-1})$$

Then, by starting with a common shared secret key k_0, the parties implicated in the conversation will be able to obtain the same keys. We thus obtain Forward Secrecy (see Figure 2.1).

Forward secrecy: A protocol provides Forward secrecy if the exposure of long-term keying material that was used in a protocol to generate or agree on session keys doesn't compromise the secrecy of session keys established before the exposure [Kra05].

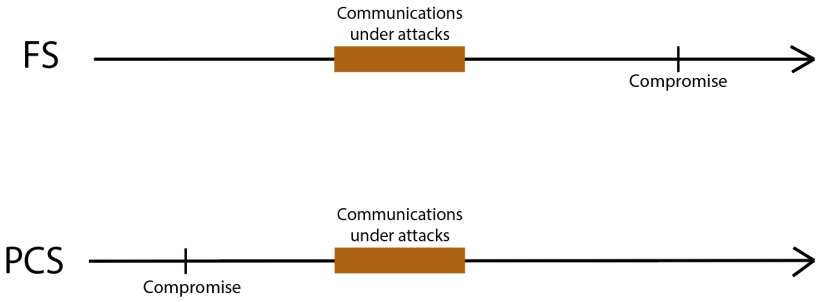


Figure 2.1: Forward Secrecy (*top*) and Post-Compromise Secrecy (*bottom*)

2.2.3 Post-Compromise Secrecy

Nevertheless, it is not enough. If an adversary obtains the key at time t, he will still be able recalculate all the following keys simply by applying the KDF to this leaked key, and thus to read every message encrypted and sent after this moment t. That is why future, or post-compromise, secrecy is necessary as well.

Post-Compromise Secrecy: A protocol provides Post-Compromise secrecy if the leak of long-term keying information used to generate session keys does not compromise these later keys established after the exposure.

In order to obtain such a property, we could add as input to the KDF another secret value, independent to the key k_t, but shared between the parties. We would then obtain what we call one-time keys, i.e. keys used only once, and then deleted. (see Figure 2.2)

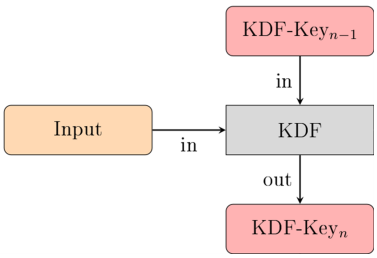


Figure 2.2: Key Derivation Function

An example of an algorithm that has both Forward and Post-compromise secrecy is the Diffie-Hellman Ratchet (see Section 2.4.3)

If a system is able to achieve Post-Compromise secrecy, along with End-to-end encryption and Forward secrecy, then it can proceed without worrying much about a third party cracking a key. Most properties presented in Section 2.1 should stand.

2.3 Authentication X3DH

When we talked about communication, we said it was between two parties. It is not incorrect, though to be precise, these two parties never communicates directly, since their messages go through a server. Moreover, neither of these parties should totally trust this server, and that's why end-to-encryption is important: even if the server in the middle was hacked or had malicious intention, it should not matter, as long as the messages going through can only be decoded by the parties involved in the communication. In the case where the server is not absolutely secure, how can one be sure that the person who sent them a message is the one they pretend to be ? If it is Alice that initiates the exchange of messages, she must be certain that she's speaking with Bob, during the whole session. Thus, we need protection against a man-in-the-middle attack, where an attacker could impersonate Bob. That is why end-to-end authentication is a key phase.

We present in this section Extended Triple Diffie-Hellman, or X3DH, a Key Agreement Protocol (KAP) whose goal is to create a shared secret key between two parties but also to authenticate each other with the help of public keys. [PM16b]

Key Agreement Protocol: protocol in which two parties or more cooperate to establish a shared key. This key must depends on the data given from all the parties involved. [BMS20]

One big advantage of X3DH is that it is designed to work asynchronously, meaning that if Alice wants to start the protocol, then Bob doesn't need to be online. He only needs to have previously published a set of public keys, that Alice will fetch and use.

Table 2.1 lists all the keys necessary in order to apply X3DH, with Alice the initiator. Here, Alice's keys are her private ones, while Bob's keys are public. Of course, each public key has a corresponding private key, and vice versa. Moreover, these public keys used here are based on elliptic curves (see Elliptic Curve Diffie-Hellman, or ECDH, [Bro09]).

Name	Definition	Duration	
IK_A	Alice's identity key	Long-term	bound to identity
EK_A	Alice's ephemeral	One-time	never reused
IK_B	Bob's identity key	Long-term	bound to identity
SPK_B	Bob's signed prekey	Medium-term	reused across sessions
OPK_B	Bob's one-time prekey	One-time	never reused

Table 2.1: Keys used in X3DH

The X3DH protocol is divided into three phases:

1. Bob publishes to a server his identity key and prekeys
2. Alice sends an initial message to Bob, using Bob's prekeys to compute a shared key
3. Bob receives the initial message that Alice sent, and processes it

After a successful protocol run, Alice and Bob share a secret key (SK), that will be used later in the second and third part of the Signal protocol.

2.3.1 Publishing the keys

The very first thing that is done when a party decides to use an application based on the Signal protocol is publishing several specific keys onto a server. Everyone needs to publish these keys in order to enable authentication and privacy in future message exchanges. This set of keys of commonly called a *prekey bundle*.

The keys that any new party places into this common server are the following public ones:

- IK: An identity key, whose link with its owner is shown by what is called a *certificate*. It proves that the party is really who they pretend to be, since only them possess this key.
- SPK: A signed prekey, with its signature using IK, to indicate the owner of the one-time prekeys made available into the server.
- $\{OPK_i\}_{i=1}^{n}$: A set of n one-time prekeys, randomly generated and deleted as soon as they are used, in order to reach forward secrecy.

Obviously, the owner of these public keys possesses all the corresponding private keys, but doesn't show them; they are kept secret. The identity key is a long-term key and it should stay the same as long as possible. And since it is used multiple times, it is very important to keep the private key secret, to avoid any impersonation from a malicious party. The signed prekey is a medium-term key, which Bob updates at some previously defined interval. And the one-time keys are, as their names indicates, used only once, and then deleted. Once all the keys are used (or shortly before), Bob should upload a new set of public one-time keys.

2.3.2 Sending the initial message

In order to perform an X3DH key agreement with Bob, Alice first needs to fetch his prekey bundle, by making a request to the server. Therefore, Alice will be in possession of the following public keys:

- Bob's identity key (IK_B)
- Bob's signed prekey (SPK_B) and its signature
- One of the Bob's one-time prekey (OPK_B)

First, Alice verifies the prekey signature. It is important to include the signature, because without it, the system would allow a "weak forward secrecy" attack [PM16b]: a malicious server could give Alice a prekey bundle containing forged prekeys, and later compromise Bob's IK_B, by calculating the SPK.

Next, she can calculate the following Diffie-Hellman[1] values (she uses her privates keys, and Bob's public keys):

- $DH_1 = DH\left(ik_A, SPK_B\right)$
- $DH_2 = DH\left(ek_A, IK_B\right)$
- $DH_3 = DH\left(ek_A, SPK_B\right)$
- $DH_4 = DH\left(ek_A, OPK_B\right)$

where DH is the function applying ECDH, in order to obtain a shared secret. Finally, Alice can compute the shared secret SK:

$$SK = KDF\left(DH_1||DH_2||DH_3\right)$$

where || represents the concatenation operation, and KDF is the Key Derivation Function[2].

Figure 2.3 sums up the 4 combinations of private-public keys.

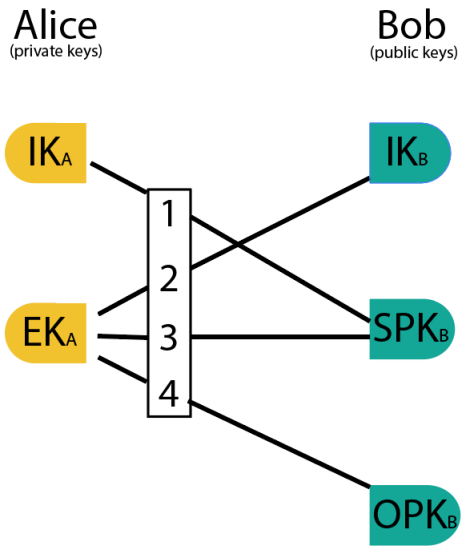


Figure 2.3: Four shared secrets between Alice and Bob

These combinations of keys have the following goals: DH_1 and DH_2 use identity keys to provide mutual authentication, while DH_3 and DH_4 use one-time and medium-term keys to provide forward secrecy.
Now, Alice can send Bob the initial message we talked about, containing:

[1]Diffie-Hellman is a method to securely exchange cryptographic keys: given the key pairs (a, g^a) and (b, g^b), respectively owned by Alice and Bob, the shared secret is g^{ab}, which is also called Diffie-Hellman. g is a public parameter

[2]see Section 2.1.1

- IK_A: Alice's identity key
- EK_A: Alice's ephemeral key
- i: identifiers indicating which Bob's prekeys that Alice used
- C: an initial ciphertext, encrypted with the shared key SK

2.3.3 Receiving the initial message

The initial message that Bob received contains (IK_A, EK_A, i, C), that is Alice's identity key and ephemeral key, along with the ciphertext and the indication i. With this information, Bob is able to repeat the DH and KDF computations, by using his private keys, as did Alice, and can derive SK. And finally, he can decrypt the secret message by using his shared secret. At the same time, he authenticates Alice as being the sender if the ciphertext is successfully decrypted.

The Figure 2.4 sums up the different phases in the X3DH protocol.

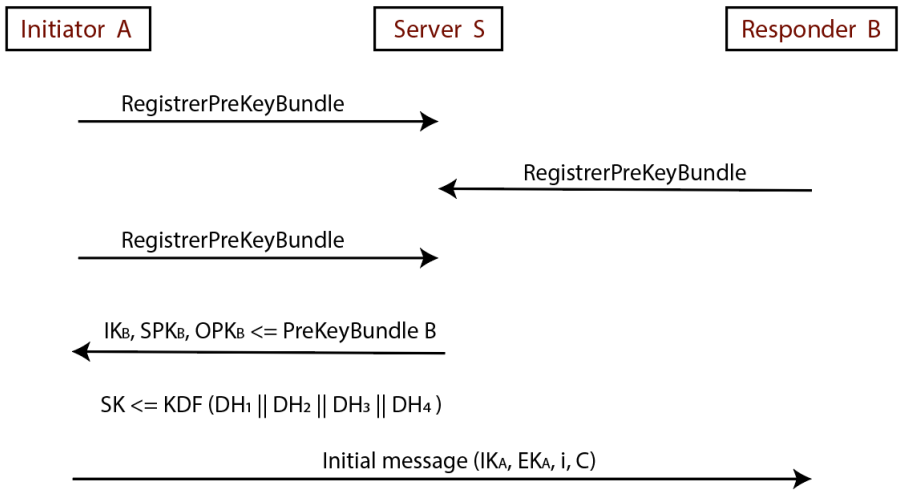


Figure 2.4: Example of message flows during the execution of X3DH between two clients A and B via a server S, initiated by A

2.4 Double Ratchet

We now have method to initiate a conversation, along with the authentication of the involved parties. The next step is : how do we send a message securely?

The Double Ratchet algorithm is the second main part of the Signal Protocol. It was developed by Trever Perrin and Moxie Marlinspike in 2013, and is used in order to provide end-to-end encryption

for instant messaging. The major difference with another usual symmetric encryption algorithm is that Double Ratchet manages the ongoing renewal of session keys, in order to provide Forward and Post-compromise secrecies.

The algorithm is a combination of the Diffie-Hellman Ratchet and the Symmetric-key Ratchet, the latter being based on Key Derivation Functions. All of these new concepts are detailed in this Section.

2.4.1 KDF chains

As defined earlier, a KDF is a cryptographic function that takes a secret and a random value as input, to return data [PM16a]. The reason of the random value is to obtain an indistinguishable output from random data. KDF chain is an important concept employed in the Double Ratchet Algorithm. This term designs the process of using a part of the output from a KDF as input of the following KDF step. The other part is used as output key.

In each Double Ratchet session, Alice and Bob stores a KDF key for three chains: a root chain, a sending chain and a receiving chain (Alice's receiving chain matches Bob's sending chain, and vice versa).
As Alice and Bob exchange messages, they also communicate their newly generated Diffie-Hellman key (the public one), and the DH secrets calculated become the inputs to the root chain. Then, the output keys from the root chain become the new KDF keys for the sending and receiving chains. This process is called the Diffie-Hellman ratchet.

The sending and receiving chains both rely on the root chain, and each one of them can be updated. Indeed, the sending and receiving chains are rebooted when the root chain is updated, and the latter event happens when a party receives a new Diffie-Hellman public key. With this new public key, and by generating a new private key, one can then recompute a DH shared secret, apply KDF, and eventually update the root chain value. Nevertheless, as long as the root key is kept the same, the sending and receiving chains are also updated each time a message is sent or received. The output message keys obtained are used for the encryption or decryption of the messages. This process is called the Symmetric-key Ratchet.

2.4.2 Symmetric-key ratchet

The Double Ratchet algorithm is used by two parties to exchange encrypted messages, based on an initial shared secret key, like the one Alice and Bob derived with the X3DH protocol.

For every message sent, the parties derive new keys, so that earlier keys cannot be computed by using later ones (to reach Forward Secrecy). Moreover, the parties send Diffie-Hellman public values as well, attached to their messages.

This latter value is used in the KDF (with the previous key) as input, to generate a receiving chain key and a sending chain key. On both chain, Alice applies a symmetric-key ratchet, resulting in

a new chain key (receiving or sending) and a message key, which is the key used to encrypt (or decrypt) a message. The result can be in seen in the Figure 2.5.

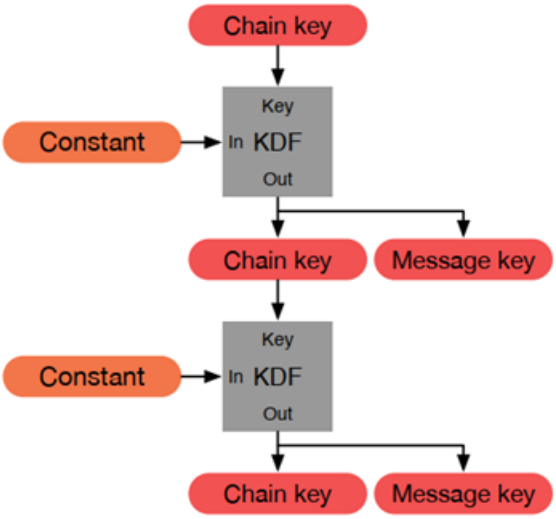


Figure 2.5: Symmetric Ratchet

2.4.3 Diffie-Hellman Ratchet

A possible algorithm that one could use in order to achieve Forward and Post-Compromise Secrecy is the Diffie-Hellman Ratchet algorithm. The method enables both Alice and Bob to encrypt every message with a different symmetric key. As one would expect, it relies on the DH key exchange.

The basic idea of DH Ratchet is that each party generates a DH key pair (k, K), where k is the private key and K the public one, to update the chains. We call this pair their current ratchet key pair. When a new ratchet public key is received by a party, a DH ratchet step is performed, consisting of updating the receiving party's current ratchet key pair with a new pair of keys. Then, the party computes the DH output, based on its private key and the public key received.

If Alice sends, after that, her public key, Bob should be able to derive the same DH output, as illustrated in Figure 2.6.

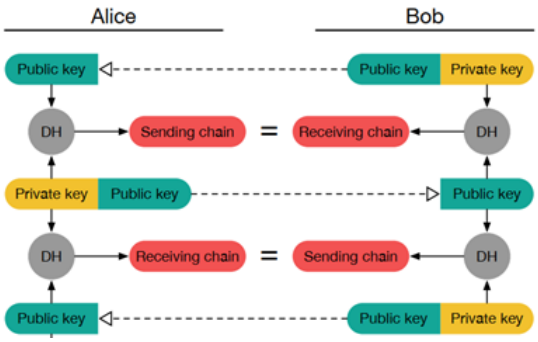


Figure 2.6: Diffie-Hellman Ratchet chain

Let's say Alice wants to send the first message to Bob. Then, she first needs Bob's public key B_1, and follows the next steps:

1. Generate a new DH key pair: (A_1, a_1)
2. Create a shared secret DH$(B_1, a_1) = ss_1$
3. Derive a symmetric key: $S_1 = KDF\,(ss_1)$
4. Encrypt the message with the key: $CT_1 = encrypt\,(M_1, S_1)$
5. Send the tuple (ciphertext, A's public key) = (CT_1, A_1) to B

Then, to decrypt the message, Bob follows the next steps:

1. Create the shared secret: $DH\,(A_1, b_1) = ss_1 = (DH\,(B_1, a_1))$
2. Derive the symmetric key
3. Decrypt the message

These steps describe the example in Figure 2.7.

This means that we could say the shared secret derived by Alice is a sending chain, and the one used by Bob to decrypt is a receiving chain (as shown in Figure 2.6)

If Bob wants to send Alice an answer, he follows the same steps Alice did, but beforehand he needs to generate a new DH key pair (B_2, b_2) and a new symmetric secret $DH\,(A_1, b_2)$, with A's public key. He will eventually send the ciphertext alongside his public key B_2

With such an algorithm, the keys are updated after every message, meaning that if a private key a_i leaks, only the ciphertext CT_i and CT_{i+1} won't be protected anymore, since the symmetric key used to encrypt CT_i is based on (a_i, B_i) and for CT_{i+1}, it is based on (a_i, B_{i+1}).

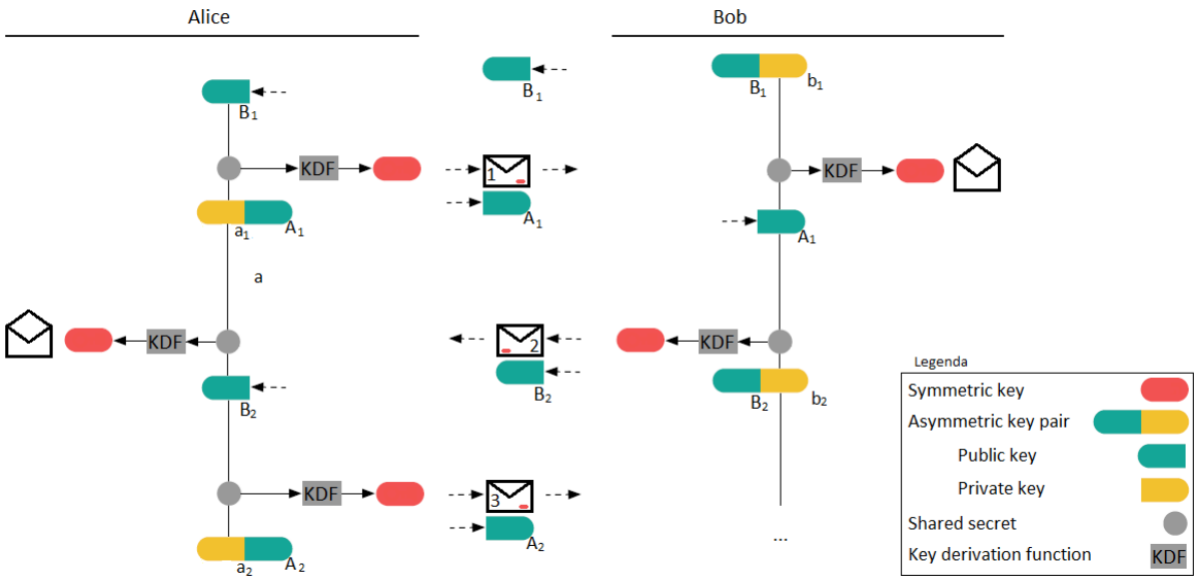


Figure 2.7: Diffie-Hellman Ratchet chain

2.4.4 The Double Ratchet

If we want to be accurate, the above drawings and explanations are a simplification of what's really happening, because in reality, after calculating the DH output, Alice and Bob apply the symmetric ratcheting in order to derive the receiving chain key and sending chain key, as shown in Figure 2.8.

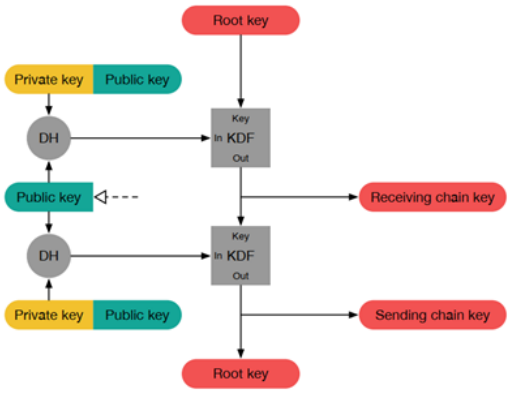


Figure 2.8: Double Ratchet

Then, each time Alice receives a message, her receiving chain key advances (she just applies KDF on it), so that she gets a new receiving chain key and a message key, the latter one being used to decrypt the message. And similarly, when she wants to send something, she updates her sending

chain key. By following these steps, Alice's receiving chain key should be equal to Bob's sending chain key.

Hence, by combining the symmetric-key and DH ratchets, we obtain the **Double Ratchet**:

1. When a message is sent or received, a symmetric-key ratchet step is performed to the sending or receiving chain to derive the message key
2. When a new ratchet public key is received, a DH ratchet step is applied before the symmetric-key ratchet, to replace the chain keys.

A last example is shown in Figure 2.9.

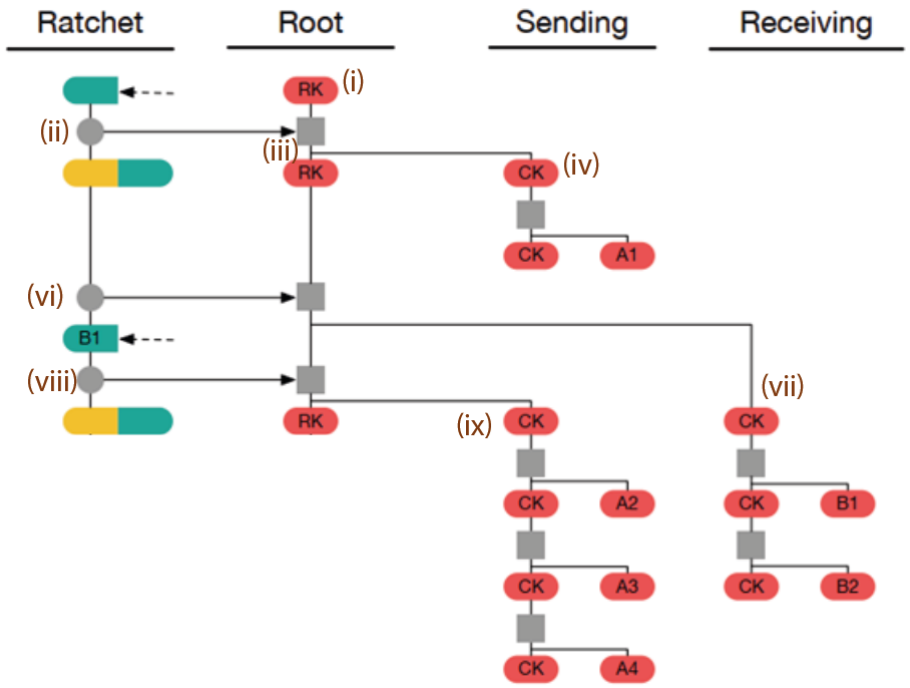


Figure 2.9: Example of the Diffie-Hellman Ratchet: Alice is the sender, Bob is the recipient

Here is how it goes:

(i) An initial root key (RK) is obtained after X3DH

(ii) Alice receives Bob's DH public key, creates a new pair for her, and computes the DH value.

(iii) Alice applies the KDF chain, with the DH value and RK as inputs, and obtains a sending chain key.

(iv) She wants to send a message $M_1\,(A)$. She then applies KDF on her sending chain key, and gets a symmetric key KA_1.

(v) Alice sends the encrypted message, with her public DH key, to Bob.

(vi) Alice receives two messages from Bob, with his public DH key. She derives a new DH value.

(vii) With this value, advances the KDF chain, to obtain a receiving chain key, and apply twice the receiving chain ratchet, to derive two message keys, used to decrypt the corresponding ciphertext.

(viii) Alice creates a new DH pair, computes the DH shared secret, and apply a KDF chain ratchet, to derive a new sending chain key.

(ix) She applies three times the sending chain ratchet, to get multiple message keys, and encrypt three messages with different keys, and then sends them.

Thus, we obtain Forward secrecy and Post-compromise secrecy, but also resilience (the output keys appear random to an adversary without knowledge of the KDF keys) and break-in recovery (if an adversary learns the KDF key at a time t, the future output keys will still appear random [PM16a]).

Moreover, if an attacker, Eve, manages to impersonate Alice (meaning Bob thinks he is speaking with Alice), the process should fail since Eve does not know Alice's private identity key, which was necessary for the initialisation of the ratchet key. Nevertheless, if Eve acts as a man-in-the-middle, there might be bigger risks because she would decrypt the messages received from Bob, read them and re-encrypt them with her keys, without constraint with FS and PCS since she can imitate Alice's behaviour and capacity to generate new keys. That is why the signature is an important aspect, along with the certificate, to ensure the entity is really who they claim to be.

Chapter 3

Quantum Cryptography

The previous chapter explained the functioning of the Signal protocol. Against today's computer, it is quite secure, but with the arrival of more developed quantum computers, it might soon not be the case anymore. Section 3.1 introduces what a quantum computer is and the main developments that concern the cryptographic community. In Section 3.2, we go into more details on the two major algorithms, one by Shor and another one by Grover, threatening cryptographic primitives. And in Section 3.3, we present three fields of study to resist quantum algorithms.

3.1 Introduction

Today's computers, both in theory (with Turing machines) and practice (PCs, laptops, smartphones, etc...), are based on classical physics. So, they are limited by the ability to be in only one state at the time. With quantum physics, things are different. Indeed, a quantum system can be in a superposition of many different states at the same time, enabling much more efficient computation, and thus shorter time for some algorithms, compared to classical computers. The main threat that represents this kind of computers relies on their capacity to undermine common cryptographic defences. Even though the machines aren't powerful enough to do this today, they are evolving rapidly. That is why a new defence mechanism is necessary, as soon as possible.

But what do we mean, by quantum computer ?
A quantum computer could simply be described as a computer using quantum mechanics so that it can perform certain kinds of computation more efficiently than a regular computer can [Sug18].

To be more precise, while a classic computer uses bits to store information, a quantum computer uses qubits, elements that can be set to 0 or 1, but also to 0 and 1, and thus can compute algorithm once on multiple different inputs simultaneously, enabling faster computations, even though the risk of errors increases with the size of the input.

The idea of a quantum computer appears first with Richard Feynman, in 1981, who mentions that simulating quantum mechanical systems on a new kind of machine following the quantum principles could result in great computation performance [Str11].

But the first really major development in quantum computation was brought by Peter Shor, with his publication in 1994 of a quantum algorithm that could factorize prime integers in polynomial time. This paper provided a way to reach exponential speedup over the fastest classical algorithms, meaning that a lot of public-key cryptography systems were threatened by this, such as RSA which depends on the fact that there is no known efficient classical algorithm that factors integers into prime numbers.

The second breakthrough that impacted some security algorithms is the Lov Grover's quantum database search. This algorithm searches a specific entry in an unordered database, using amplitude amplification to achieve a polynomial speedup over the best existing classical algorithms [Sug18]. Grover's discovery can solve some NP problem, such as the SAT problem and graph colouring. To sum up, because of Grover's algorithm, one need to double the key sizes of symmetric key cryptography in order to obtain the same security level than with classical computers.

3.2 Risks

We briefly introduced quantum computers and the two main algorithms we should have concerns about. But we didn't specify which algorithms were threatened, and how big the danger was. Indeed, for some algorithms like hash functions or symmetric encryption protocols, it is more an inconvenience than a real menace. But for some asymmetric protocol, the danger is much bigger.

3.2.1 Quantum computer consequences

With the revolution provoked by quantum algorithms, some cryptographic primitives are not secure anymore. Table 3.1 lists the main algorithms used in encryption, and the consequence of quantum computers.

Cryptographic primitive	Type	Purpose	Impact from quantum computer
AES-256	Symmetric key	Encryption	Larger key sizes needed
SHA-256	One-way	Hash functions	Larger key sizes needed
RSA	Public key	KEP, Signature	No longer secure
ECDSA, ECDH	Public key	KEP, Signature	No longer secure
DSA	Public key	KEP, Signature	No longer secure

Table 3.1: NIST impact analysis of quantum computing on encryption schemes [BW17]

RSA: Breaking the RSA encryption scheme is possible on a classical computer, through factoring, since inverting the RSA function is categorized as hard as a factorization. The ways in which RSA

might be attacked are well studied, since quite a long time [BW17], and the attacks are well documented. But the problem for each method is the efficiency. This is what makes RSA secure to classical computer.

Peter Shor's algorithm is designed to factor integers into their prime number components, using Euler's method, i.e. determining the period of the function. This method runs on a quantum computer in polynomial time, whereas the General Number Field Sieve, which is the fastest factoring algorithm we currently have on conventional computers, runs "only" in a super-polynomial time [BW17] [LL93].

Diffie-Hellman: DH key-exchange protocol is at the foundation of public-key cryptography. The objective of such a protocol is to enable two parties to derive a shared secret, from each other's public key and their own private keys. Diffie-Hellman problem is said to be as hard as computing discrete logs, since an adversary's goal is to find a from $A = g^a$. The best discrete logarithm algorithm effective today is from Barbulescu, Joux and Thomé, but it runs in quasi-polynomial time [BGJT14], which is slow enough to keep DH secure from classical computing.

Shor's quantum algorithm is able to solve the Discrete Logarithm Problem (DLP), and thus breaks Diffie-Hellman, in polynomial time. This means that every protocol relying on Diffie-Hellman is jeopardized, in particular ECDH, or Elliptic Curve Diffie-Hellman, for two reasons: the first is that give $A = g^a$, one is now able to find a and thus compute the shared secret by taking Bob's public key $B = g^b$ and apply a, giving $B^a = g^{ab}$ which is equal to what Bob has ($A^b = g^{ab}$); the second that Eve can pretend to be Alice to Bob, and vice-versa, by doing a *man-in-the-middle* attack, meaning intercepting both parties' messages, decrypt them and encrypt the same message or another one with Eve's keys. This is why the threat upon Diffie-Hellman should be taken seriously.

DSA: since a lot of Digital Signature Algorithm are based on using public keys pair (private key to sign, public key to check), it becomes easy for an adversary using quantum algorithm to forge a signature into a message, i.e. fabricating a digital signature for a message without having access to the respective signer's private signing key. The security property of non-repudiation is also lost.

3.2.2 Risks with Signal

As we have seen, some important cryptographic protocols are more or less impacted by quantum computers. We then need to adapt the Signal protocol to those new risks, in order to get a resistant protocol, by checking the primitives at risk.

Table 3.2 lists the cryptographic primitives used either in X3DH or in Double Ratchet, with the corresponding algorithm ([PM16a] [PM16b]).

The hash function SHA-512 and the symmetric encryption primitive AES are not directly threatened by quantum computers. Nevertheless, it is necessary to take larger keys in order to keep an acceptable level of security.
The most concerning aspect here is ECDH Curve, which is no longer secure. It is absolutely neces-

Signal Protocol's part	Primitives	Algorithm
X3DH	Key Exchange	ECDH (Curve25519)
	KDF	HKDF (SHA-512)
	Signature	XEdDSA (Curve 25519)
Double Ratchet	Key Exchange	ECDH (Curve25519)
	KDF	HKDF (SHA-512)
	Encryption	AES-256 (CBC mode)

Table 3.2: Cryptographic primitives used in the Signal Protocol

sary to trade it for a post-quantum key exchange protocol. SIKE is one of them, and is detailed in Chapter 5.

Overall, it seems that quantum computers coerce us to get rid of every asymmetric protocol present in the Signal protocol and change them with quantum-resistant ones, and strengthen the level of security of the symmetric primitives and the hash functions.

3.3 Solutions

As briefly explained in Section 3.1, with quantum computers:

1. Shor's algorithm could break all classical factorization and discrete log-based public key crypto
2. Grover's algorithm could force doubling the key sizes of symmetric key cryptography.

Let's explore the two main alternatives existing thus far [Mat19]:

- from physicists: quantum technology is used to break old cryptographic systems, so one should use this same quantum technology, but to create more resistant protocols based on it.
  $\Rightarrow$ Quantum cryptography.
- from mathematicians: in the same way there exist protocol said to be as hard as mathematical problems, we should develop problem quantum computers cannot solve.
  $\Rightarrow$ Quantum resistant or Post-Quantum cryptography.

Quantum cryptography

The main purpose of Quantum cryptography has been to reduce as much as possible the breaking of public key cryptography from quantum computers, with Quantum Key Distribution, or QKD. Based on fundamentals in quantum physics, QKD enjoys a secure way to distribute keys through insecure channels. [QQL10]
But Quantum cryptography has several drawbacks:

- mainly limited to quantum key distribution
- provides no authentication
- has problem with large distances
- does not scale well

For these reason, we prefer focusing here on the second form of cryptography, even though QKD is being significantly deployed lately.

Post-quantum cryptography

In order to be efficient and easily used by as many systems as possible, PQ cryptographic systems should respect some basic criteria. First, they should run efficiently on classical computer (time, memory, ...). Then, they must be hard to break by both quantum and classical algorithms (since it is *Post-Quantum*, by definition). Eventually, they need to rely on different mathematical problems than integer factorization or discrete logarithms, since these kinds of problem are pointless against quantum computers.

There are many Post-Quantum cryptographic algorithms, but we can categorize them, and the types we are interested in are:

- Code-based
- Lattice-based
- Isogeny-based

There are other types, like Multivariate and Hash-based, but are designed for multivariate statistics or signature scheme, which we won't focus on it in this work.

3.3.1 Code-based

Code-based cryptography is an old method, based on coding theory, used for the last 40 years. Even though it was considered unusable for a long time due to the large key size necessary, in particular compared to number-theoretical cryptosystems, like RSA or ECDSA, code-based systems are nowadays reconsidered, because of their capacity to resist to quantum computations.

Code-based cryptography is based on error correcting codes. These codes were originally developed to improve communication accuracy, since transmitting information over an unreliable channel would often provoke noise and errors. In order to correct an inaccurate received message, an error-correcting code is added to the message before sending it, so that the recipient can retrieve the exact message, or at least notice that the message received is incorrect and ask for a retransmission. The method used to do so is based on linear codes, that is we can use matrix-vector multiplication to encode and decode messages (see [MLS19] for more explanations).

This concept has been developed in order to be resistant to quantum computers: the message is, as before, converted into a code, but then, one deliberately adds a secret error to it. And because the recipient knows the parameters used (it's a shared secret between the two parties), he is able to retrieve the original code, and thus the message. The security relies on the fact that an adversary should not be able to distinguish the exact code from a randomly generated one. This is made possible by having a public key which is a scrambled version of a generator matrix, the latter being used to encrypt the message.

McEliece cryptosystem:

In general, public key cryptography relies on the idea that messages should be easy to encrypt, but hard to decrypt without private information. Let's assume that G is the generator matrix of a code, meaning that by multiplying it with a vector, we obtain a code, i.e. an encrypted message. So, we can multiply the message M by the generator matrix G, to get a code word $c = M \cdot G$ and then add some arbitrary error vector e. Hence, the ciphertext, denoted as T_{Mc}, is:

$$T_{Mc} = M \cdot G + e$$

If one knows the generator matrix G, with the fast decoding algorithm, they can easily decrypt the message M, by finding the closest vector to $c + e$.

Nevertheless, with such a method, there is no security, since an adversary could apply the same algorithm, and retrieve the plaintext M. The McEliece Cryptosystem makes the fast decoding algorithm undoable for someone who doesn't know the correct coding parameters.

The idea of the McEliece cryptographic system is to scramble G, by multiplying it by some matrices. The result is published as the public key [MLS19]. These matrices are S_{Mc}, random and invertible, and P_{Mc}, a permutation matrix, and represent the shared secret. Thus, we get:

$$G' = S_{Mc} \cdot G \cdot P_{Mc}$$

By constructing G' this way, G is very well hidden, and one can eventually compute the new ciphertext:

$$T_{Mc} = M \cdot G' + e$$

The decryption is done by following the next rules:

1. Since P_{Mc} and S_{Mc} are known from both parties, the recipient can inverse the former matrix and multiply it with T_{Mc}:

$$T_{Mc} \cdot P_{Mc}^{-1} = (M \cdot S_{Mc} \cdot G \cdot P_{Mc} + e) \cdot P_{Mc}^{-1} = MS_{Mc}G + eP_{Mc}^{-1}$$

2. A fast decoding algorithm Δ retrieves the nearest code word c from a vector of the form $w = c + e$, using Hamming distance.
   If the ciphertext was exactly $MG + e$, this algorithm would give us:

$$\Delta\left(MG + e\right) = M$$

3. One finally multiplies it with the inverse ofS_{Mc}:

$$\Delta\left(T_{Mc} \cdot P_{Mc}^{-1}\right) \cdot S_{Mc}^{-1} = \Delta\left(MS_{Mc} \cdot G + eP_{Mc}^{-1}\right) \cdot S_{Mc}^{-1} = MS_{Mc} \cdot S_{Mc}^{-1} = M$$

Figure 3.1 sums up these operations.

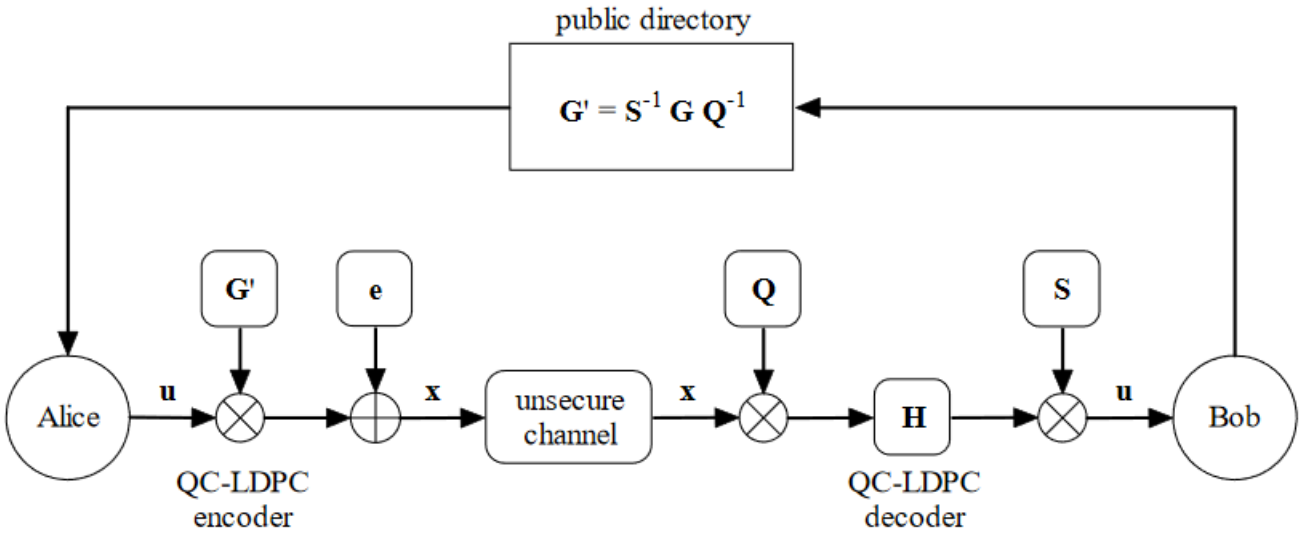


Figure 3.1: McEliece process [BBC13]

The main advantage of McEliece cryptosystem is its efficiency and robustness, although no concrete proof of its security against quantum computers exists. On top of that, the main problem is the large key sizes necessary, even though attempts have been made to reduce it, like introducing more structure into the codes [LXY19].

3.3.2 Lattice-based

This topic is probably the most active one among all the research made in post-quantum cryptography. Indeed, it only involves simple linear operations, like additions and multiplications with modulo, similarly to RSA, and enjoy strong security proofs, thanks to the worst-case hardness of lattice problems, which has been conjectured to be hard to solve in both classical and quantum configurations.

The ideas behind lattice-based cryptographic systems are similar to the ones in code-based.

Shortest Vector Problem

Lattice: Let $\{b_i\}_{i=1}^{m} \in \mathbb{R}^n$ be m independent vectors. Then, an n-dimensional lattice L is the span of all these vectors b_i, i.e.:

$$L = \left\{ \sum_{i=1}^{m} x_i b_i \| x_i \in \mathbb{Z} \right\} (= L(B))$$

So, intuitively, a lattice is a set of points, with a base B that can be considered as a matrix whose columns are the vectors b_i.

The idea behind lattice-based cryptography is that one can use this well-formed space as a secret key, and a scrambled version of it as a public key (like we scrambled the matrix G in the code-based section). The sender can now map the message to a point on the lattice, and then add an arbitrary error, such that the new point is still closer to the original one than to any other point. Since the recipient knows the correct well-formed base, he can decrypt it by finding the closest vector to the received point. The security relies on the idea that it is hard for an attacker to find the closest point using only a kind of random base, as shown in Figure 3.2.
This can be rephrased as the following problem:

Shortest Vector Problem (SVP): Given a basis B of a lattice L, find the shortest nonzero vector in L(B).

For the approximation variant of SVP is reduced at finding a vector v whose length is lower than γ the length of the shortest nonzero vector, i.e.: $\|v\| \leq \gamma \cdot \lambda_1(L)$, where λ_1 designs the shortest vector in L.

What makes SVP important is that they are classified as hard problems, and quantum algorithms are not much more efficient at solving this kind of problem. Hence, a polynomial time quantum algorithm that can approximate these problems in polynomial time does not exist.

But the reason we are talking about finding a shortest vector in a given lattice is that the decoding process of a received message consists in finding the closest code c from a given non-exact code $w = c + e$. This problem is called Closest Vector Problem (CVP).

CVP is a generalization of the Shortest Vector Problem. That is why it is important that SVP is NP-hard: quantum cannot solve easily this kind of problem neither without complementary information.

3.3.3 Isogeny-based

While the classic Elliptic Curve Cryptography (ECC) could be summarized as finding a hidden relation between two points P and P' on a given elliptic curve E, Isogeny-based cryptography is more about discovering connection and operations between two elliptic curves E and E' inside a certain large pool. These operations that map a curve onto another one with certain properties are called isogenies. The difficulty is that it is supposed to be hard to find the isogeny between two specific EC without more information about them. This information is kept secret into the private key, while the public information is the two elliptic curves.

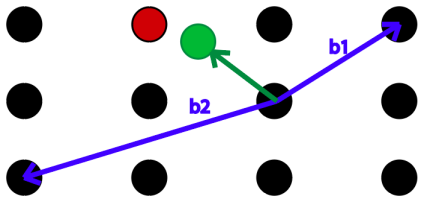


Figure 3.2: Closest Vector Problem: the basis is given by $\{b_1, b_2\}$, and the recipient receive the green vector ; they need to find the closest vector, or closest point, which is the red one here

Hence, the problem that a quantum computer would have to solve is the following:

Problem: Given an elliptic curve E' obtained by applying an isogeny into a base curve E, find how to get from E to E', using the same map [Mat19].

This means that the security of the corresponding protocol, namely Supersingular Isogeny Diffie-Hellman or SIDH, is linked to the problem of finding the isogeny mapping an elliptic curve to another one. According to De Feo, Jao and Plut, the most efficient attack existing is solving the related *claw finding* problem, where given two functions f and g, one has to find a pair (x, y) called a claw such as $f(x) = g(y)$. The attacks against this kind of problem has a complexity of $O(p^{\frac{1}{4}})$ for classical computers and $O(p^{\frac{1}{6}})$ for quantum computers. Thus, to achieve a 128-bit security level[1], one should take p as a 768-bit prime [JDF11].

Supersingular Isogeny Diffie-Hellman exchange uses conventional elliptic curve operations, provides perfect forward secrecy, and can be used for computing a shared secret between two parties [JDF11]. More information in Chapter 5.

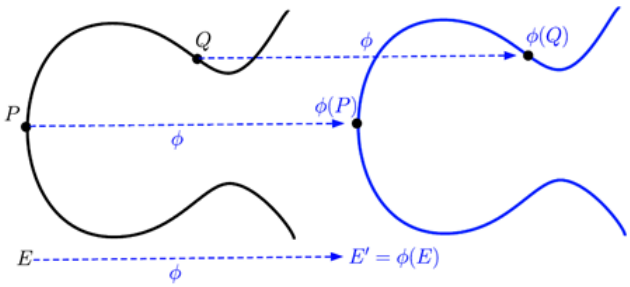


Figure 3.3: Example of Isogeny: here the isogeny is just a translation [Aza]

[1]Security level is a measure, usually expressed in bits, where having n-bit security is equivalent for an attacker to perform 2^n operations in order to break it

Chapter 4

Group messaging and ART

The Signal protocol given in Chapter 2 has two flaws, the first one being its resistance against quantum algorithms, which was explained in Chapter 3. The second one is the difficulty to scale up to group communication. This Chapter details the problem and aims to present a solution.
In Section 4.1, we introduce the issue by comparing a two-party exchange with a group conversation. Section 4.2 displays existing method to create secure group communication system, while Section 4.3.1 presents the ART protocol [CGCG+18], a system based on tree structure, along with its advantages and flaws.

4.1 Introduction

Until now, we only talked about two-party end-to-end encryption communication, and the security properties obtained using the Signal Protocol. But what is true with a two-members conversation may not be with three or more entities.

In a two-party messaging protocol like Signal, one can create a session with his future partner offline. It doesn't prevent them to prepare everything and then send an initial message with all the important information to the other party. We can have asynchronicity. Moreover, we also have the following properties: privacy, Forward secrecy, Post-compromise secrecy, deniability and authentication.

But what about a n-parties group communication with $n > 2$? Can we still have the same properties ?

From now on, we must consider that there are more than two members in a group that would like to exchange messages. How can it be done ?

Well, a very basic solution would be to create pairwise channels between all the members, in which it is possible to apply the exact Signal Protocol. This means that in a group conversation, if Alice wants to send a message, she has to send it to everybody using each common secret she shares with a member. An obvious advantage is that everybody keeps the security properties from the Signal Protocol, since it is the same approach, but with more people.

But the major problem is that more members means more messages to send each time, and for voluminous messages, it can become problematic, especially if the bandwidth is limited.

Another solution would be to use a similar protocol in order to create a unique shared group key, used by everyone to encrypt and decrypt, but of course kept private to members outside the group. Then, when the message is encrypted, one party just needs to send it to the server, which will re-send it to everyone else. This is what we call *server-side fanout*.

Let's explore some already existing solutions.

4.2 Existing solutions

As group messaging already exists in most messenger app, this means that each one choosed a method to implement it. We first present here some approaches of the problem [CGCG+18].

Current group messaging systems

- **OTR-style:** Off-The-Record protocol relies on the idea of deniability, and this was maintained for group communication. The idea is the following: first, members of the group conduct some interactive rounds of communication to derive a group key; then, parties communicate online, with additional cryptographic operations, using the mpOTR protocol (multiparty OTR) that still enables deniability.
- **Physical approaches:** physical constraints, such as verifying visually security code (like Whatsapp in two-party messaging), may be a way to restrict malicious group members, since it forces to clearly know all the persons involved. This allows to derive strong security properties.
- **n-party DH:** if one wants to keep the Diffie-Hellman concept and generalize it, this is the method to consider: given n elements $\{g^{x_i}\}_{i=0}^{n}$ and a single x_i, derive a shared DH secret. As the basic DH, it is hard to compute without knowing one of the x_j. However, with large size groups, it quickly becomes expensive to calculate every value.
- **Tree-based group DH:** this kind of protocol creates a binary balanced tree, where the leaves are associated with parties and contain a private DH key, such as (i) g^{xy} is the secret key of a parent whose children's secret keys are x and y; and (ii) $g^{g^{xy}}$ is the public key. By computing recursively these secret keys up to the root, one obtains a tree key, used either directly to encrypt, or as an input for a KDF in order to obtain a message key.

The problem of most of these solutions is the lack of PCS (Post-Compromise Security), since they don't allow key updates.

Whatsapp group messaging

A last example, which doesn't support keys updating, is the Whatsapp group messaging system, explained here [Whi16]:

Just before a member, let's say Charlie, joins a group. He applies the following steps:

1. Charlie generates a random Chain Key
2. He generates a random Signature Key pair
3. He combines both the Chain Key and the public key from the Signature Key pair into a Sender Key message
4. Finally, Charlie individually sends to each member his Sender Key, using the pairwise channels with the Signal Protocol

Then, for all subsequent messages sent to the group:

1. The sender derives a Message Key from the Chain key, and updates the Chain key, similarly to the symmetric ratchet in the Signal protocol
2. He encrypts the message with the AES256 algorithm
3. He signs the ciphertext using the private key of the Signature Key pair
4. And eventually sends to the server the message, which will transmit to the other members of the group

In order to be able to decrypt the received ciphertext, the other parties will have to update the sender's Chain key, in a ratchet way. By updating, the protocol provides Forward secrecy, and when a group member leaves, the other parties clear their Sender Key and start over.

Although this protocol seems efficient, it still has some drawbacks. The biggest one is the lost of the self-healing property: a sending key is directly linked to the following one, meaning that a leaked key allows an adversary to compute the next keys and thus directly threaten the security of the ciphertexts and impersonate the party whose key has leaked.
Moreover, when a member leaves the group (or is removed by someone else), they may be able to still decrypt the messages if they intercept them, since the key used to encrypt it depends on the one the removed member had, and so this former member will be able to compute this key, even after being removed.

4.3 Asynchronous Ratchet Tree protocol

To address the problems presented above, a protocol was proposed in 2018 by Cohn-Gordon and al. to solve these issues, and in particular the possibility to combine asynchronicity with Post-Compromise secrecy, which has not been the case in the system presented above. In this section,

we first introduce the idea of the protocol and what it aims to achieve, then we give the notations that will be used later, in the construction of a group and the methods to update the group, such as adding or removing a member, or updating the group key.

4.3.1 Introduction

As indicated at the beginning of the chapter, up to now, only two-party communications were studied, and with the Signal protocol, one can get end-to-end encryption, with numerous security properties: privacy, forward secrecy, post-compromise secrecy, deniability, authentication, and non-repudiation.

But for group messaging, some of these properties are not there; and even worse, for some systems an adversary who compromised a single group member can read and write messages with no limit. The major property missing is Post-Compromise Secrecy (PCS), and that is what we will try to get, in this chapter with the ART protocol, alongside the asynchronous aspect.

The ART protocol, or Asynchronous Ratchet Tree protocol [CGCG+18], was developed by Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican and Kevin Milner. The objective of such a protocol is to fix the absence of these two properties missing (PCS and asynchronicity) for group messaging application. The protocol is based on the Signal Protocol and tree-based Diffie-Hellman key exchange.

The Asynchronous Ratchet Trees can be split in two main phases:

1. ART Construction
2. ART Update

In the first part, we describe how the initiator, or creator, of the group derives a tree-based group Diffie-Hellman, and then apply a Key Exchange Protocol (KEP) in order to send all the information necessary to the other members of the group, such as the initial shared secret key.

In the second part, we detail the method allowing an ART group member to update their personal information and establish a new shared group key. It is because of this capacity to update that we obtain PCS.

The main cryptographic primitives proposed are:

1. Diffie-Hellman group operations, using Curve25519
2. Symmetric encryption, using AES-256, with GCM mode
3. Key Derivation Function (KDF), using SHA256
4. Public signature schemes, using Curve25519

4.3.2 Notations

Before going more into details about both phases, let's mention some cryptographic indications and notations used here.

Assumptions

We use lower-case values k to represent DH secret keys and upper-case values $K = g^k$ for their corresponding public keys.

We assume that the Pseudo-Random Function – Oracle Diffie-Hellman problem (PRF-ODH) is hard: the probability to distinguish the tuple (g^x, g^y, g^{xy}) from (g^x, g^y, g^z), with z randomly generated, is $\frac{1}{2} + \epsilon$, with $\epsilon > 0$ very small and bound.

DH groups

In this protocol, we work in a DH group G with a mapping $\iota(\cdot) : G \to \mathbb{Z} \Big/ |G|\mathbb{Z}$, from a group element to integers.

Signature

Two signature schemes are used: one to authenticate the initial group setup message, and a MAC to authenticate subsequent updates.

Trees

We call a binary balanced tree a binary tree in which the depth of the two subtrees of every node never differs by more than 1, with a binary tree being a tree where a node has at most two children [LP07]. All nodes have a parent node and a sibling, except the root.
The path of a node is the set containing the node, its parent, the parent of this parent, and so on, until the root.
The copath of a node is a set of nodes which are the sibling of each element in the path from the node to the root (see Figure 4.1).

Derived keys

ART contains four different kind of keys:

- Leaf keys λ_j: secret DH keys, included in each leaf
- Node keys nk: secret DH keys, included in each non-leaf nodes

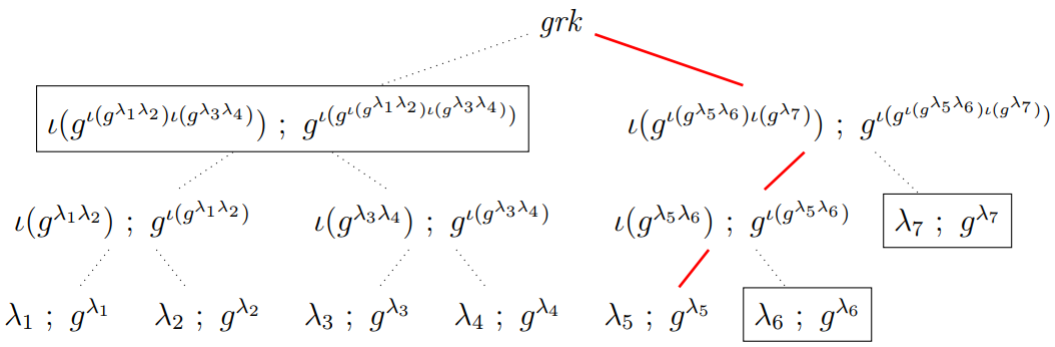


Figure 4.1: ART tree example: In red is the path, and the boxed nodes constitute the copath

- Tree key tk: secret value derived at the root (also called root key)
- Stage key sk: key derived on a combination of the latest tk and the previous sk, with a hash chain

We now have everything necessary to go in details first about how a group is constructed, through the creation of the corresponding tree, and then how the state of the group changes, once again through its tree modification.

4.3.3 ART Construction

During the rest of this chapter, we consider that Alice is the party that creates the group, and that she wants to add n other members: Bob, Charlie, and so on...

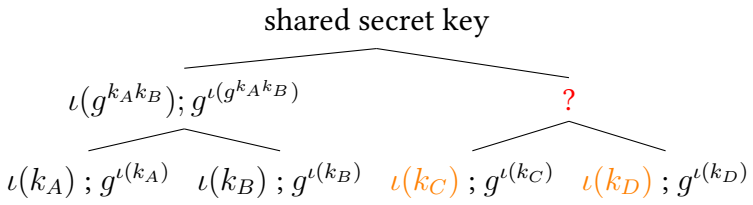
We have compared the ART tree to the tree-based DH protocol, but in reality, to allow the creation of a group asynchronously, the creator Alice would need the private DH keys of every other members, which is not possible since they are not online.

Indeed, in a four-persons group, Alice cannot compute the parent node of C and D, because she would either need to know the public key of C and the private key of D (K_C and k_D) or the inverse (k_c and K_D). See Graph 1.

Instead, it is directly Alice that will derive initial private keys for C and then for D, and for each member, she will send the necessary information allowing them to derive the secret key as well. With this knowledge, Alice is now able to calculate the parent node keys, and thus the shared group key. Let's show how it works.

Cryptographic primitives

The cryptographic primitives necessary in the ART protocol are:



Graph 1: Tree-based Diffie-Hellman: Alice cannot compute the keys of the parent of C and D, since she doesn't know neither C's or D's private key, coloured in orange (based on [CGCG+18], p.12)

- Key Encapsulation Mechanism (KEM): used for Hybrid Public Key Encryption (HPKE) in group operations into the tree
- Authenticated Encryption with Associated Data (AEAD): used for HPKE and message protection
- Hash function: used for HPKE as well
- Signature algorithm: used for message authentication

Some concepts are new here, so it may be good to explain them before going on.

Hybrid Public Key Encryption: combination of a public-key cryptographic system with a symmetric one, the goal being to enjoy the advantages of both systems [CS03]. A common example is that the public-key system could be used to generate a shared key, while the symmetric-key system could encrypt and decrypt data.

Key Encapsulation Mechanism: class of encryption techniques used to encode a key that will be used in a HPKE. The message is then encrypted through a symmetric encryption scheme, and this part is called data encapsulation.

Authenticated Encryption with Associated Data: system that enables a recipient to check the integrity of the encrypted and unencrypted information in a message [Bou18].

The paper [BMO+19] proposes several possibilities of sets of cryptographic primitives, or ciphersuites as it is called in it, with different possibilities.

Table 4.1 lists all the combination presented.

Everything being said, we are almost ready to create the group. Just before, we need to make available for everybody the keys that are going to be used.

Level of security	KEM	AEAD	Hash	Signature
128 bits	DH KEM with Curve25519	AES-128	SHA-256	Ed25519
128 bits	DH KEM with P256	AES-128	SHA-256	P256
128 bits	DH KEM with Curve25519	chacha20poly1305	SHA-256	P256
256 bits	DH KEM with Curve448	AES-256	SHA-512	Ed448
256 bits	DH KEM with P521	AES-256	SHA-512	P521
256 bits	DH KEM with Curve448	chacha20poly1305	SHA-512	Ed448

Table 4.1: Possibilites of Ciphersuite ([BMO+19], Section 15.1)

Initialisation

For our initial authenticated key exchange, we use the X3DH algorithm ([CGCG+18]).

Similarly to the Signal Protocol, a shared secret is calculated based on the information the members have published on a common, non-trusted server.
This PreKey Bundle is called a *KeyPackage*, and it specifies a ciphersuite that the client supports, and a pack of keys:

1. IK: identity key
2. $\{EK^i\}_{i=1}$: a set of n one-time ephemeral keys

This set of keys is almost the same as the one indicated in Section 2.3.1. Only the SPK (Signed PreKey) is missing, but it could easily be added.

Calculating the leaf keys

Let's say Alice wants to create a group with n members, i.e. herself and $n-1$ other persons. After generating an ephemeral pair of keys suk_a and SUK_a (also called *setup key*), she requests to the server the set of public keys of each expected new members of the group.
With Bob's public information, and with her secret keys, she derives the $n-1$ leaf keys $\lambda_1, \lambda_2, \ldots, \lambda_{n-1}$.
To do so, Alice computes:

1. $DH_1 = DH\,(ik_A, EK_B)$
2. $DH_2 = DH\,(suk_A, IK_B)$
3. $DH_3 = DH\,(suk_A, EK_B)$

as in the Signal protocol, and calculates a shared secret:

$$\lambda_i = SK_i = KDF\,(\,DH_1 \,||\, DH_2 \,||\, DH_3)$$

This shared secret key corresponds to the secret key of the leaf i. She thus needs to do so with the $n-1$ members. Once all keys have been calculated, Alice is able to derive the keys of the parent

nodes (which, by the way, solves the problem in Graph 1), then to keys of parent's parents, and so on, until the tree key. Alice can then calculate all the nodes keys and the group key (key of the root). Once it's done, she delete every leaf's keys, since she doesn't need them anymore.

Sending the initial messages

Once again, this step is similar to the one in the Signal protocol : after calculating all these values, Alice sends the following information to each group member i:

1. the public prekeys EK_i (or just an index indicating the one used),
2. Alice's public identity key IK_A and her public key SUK_A corresponding to the private key SUK_A used by Alice,
3. the tree T containing all the public keys,
4. a signature of 1), 2) and 3), under her identity key.

After receiving such a message and verifying the signature, each party is able to reproduce the same computations Alice did, and derive the same leaf keys. But what interests each member at the end of the day is to get the tree key, so that they can communicate safely.

Calculating the node's keys

How can a leaf calculate the tree key, only knowing its pair of keys and the public keys of the nodes in its copath ?

As described earlier, a leaf owns a pair of keys: a public one and a private one. The corresponding member, let's say Bob, also knows the public keys of all the other leaves. In particular, Bob knows the public key $SK_C = Y = g^y$ of his sibling Charlie (y being Charlie's private key). Then, with his private key $sk_B = x$, Bob can calculate a shared DH value:

$$S = DH(x, Y) = Y^x = g^{xy} = g^z$$

with $z = xy$.

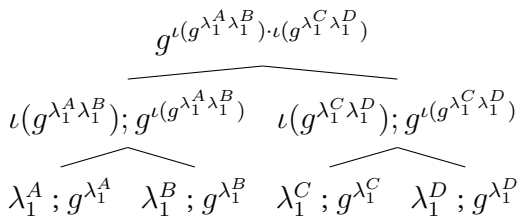
This value represents the new private key of the parent of Bob and Charlie. And the corresponding public key is:

$$g^{\iota(g^z)}$$

with $\iota(\cdot)$ as defined in Section 4.1

Hence, Bob knows the pair of keys of his parent. Since he knows the public key of his parent's sibling as well, he can also compute the key pair of his parent's parent, by applying the same operation he did for his parent. Iteratively, he is able to derive the tree key.

The result of an ART construction, with Alice the initiator and three other members is shown in Graph 2



Graph 2: Tree-based Diffie-Hellman: Alice computes the tree key, since she knows the secret leaf keys of the other members

4.3.4 ART Update

With asynchronicity, the goal of this protocol is achieving PCS. To do so, any group member must be able to update stage keys, based both on state from previous stages and on newly exchanged messages.
Such a change could be provoked by one or a combination of these group operations:

- Adding a member
- Updating the secret leaf of a member
- Removing a member

Once one or several of these operations are done, a *Commit message* is broadcasted, to provide the group this new information.

Adding a member

An Add proposal requests that a user can be added to the group. A new member is characterized by a KeyPackage (or Key bundle), that they should have previously published onto the server. With this set of keys, the initiator of the request can compute a shared leaf key, which will correspond to the secret key of the member until they decides to change it. This phase is quite similar to the tree creation, because the initiator has to update the tree (here, by adding a leaf), compute a shared secret, and send these details to the new member. They must be sent to the other members as well, so that they can keep the tree up to date.

The added member's position in the tree, indicated by an index, should be decided in order to keep the tree left-balanced as much as possible ; if necessary, the tree is expanded. Moreover, a binary tree always has an odd number of nodes, meaning that when a leaf is added, a node has to be created and inserted in the tree. This new node will be the parent of the new leaf.

Updating a key

An Update proposal is similar to the previous one, with the difference that it is the sender's leaf KeyPackage which will be updated, through a new KeyPackage.

Once the leaf's key is updated, the corresponding member calculates the keys of the nodes in its path, and sends these new information to the other nodes in its copath.

If Charlie decides to update his leaf key, every other group member should be able to derive all the intermediate values in the new updated tree, by using only:

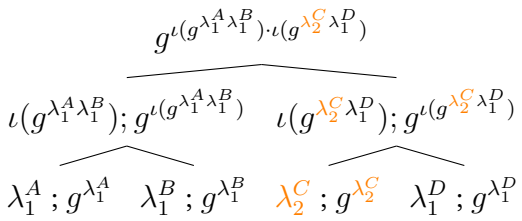
1. their view of the tree before the change
2. the list of updated public DH keys of nodes along the path from Charlie's leaf node to the root of the tree

For example, let's say that Charlie has decided to change his leaf key from λ_C to λ'_C. This means that he can now derive all the values of the node in its path until the root. After having done it, he broadcasts to the group his public key with the intermediate nodes' public key he just computed. He also needs to authenticate the message with a MAC under a key derived from the previous stage key.

At the end, if everything was done correctly, every member of the group should have derived the same tree key, and thus the same stage key.

Moreover, to enforce Forward and Post-Compromise Secrecy, each member should periodically update their leaf secret.

An example is given in Graph 3



Graph 3: ART Update: Charlie changes his leaf key, from λ_1^C to λ_2^C

Removing a member

A Remove proposal requests that a member at a given index in the tree should be removed from the group. The position in the tree is replaced with a blank node. The keys of the parent is then only calculated based on the sibling of the former member, meaning this node gets a new pair of key, and thus the tree key changes as well.

Commit

A *Commit message* is based on a collection of Proposals. This message instructs to all the members of the group to update their view of the tree, by applying the proposals and deriving the tree key, through the computation of the intermediate nodes. As long as a regular member does not receive any Commit message, its representation of the state of the tree should not change.

For simplicity, it is usually considered that a Commit is made after each operation.

To perform an update for a path, i.e. a Commit, the "commiter" sends to each member of the group these informations:

- The sender's public key,
- Encrypted copies of the path secret of the node, i.e. the new keys of the nodes in the path of the sender.

4.3.5 Complementary information

The message keys used in the symmetric-key encryption algorithm (usually AES) are taken directly from the stage keys of each stages, instead of computing chain keys, like in the Double Ratchet algorithm. A signature of the ciphertext is also sent, signed by the send private identity key.

In the ART paper [CGCG+18], it is mentioned the possibility to only use the basic Diffie-Hellman algorithm, but we decided not to use it, since X3DH provides much more security. Also, the choice to add or remove a member is let to the user's freedom, as well as the method to manage the new keys.

For the rest of this paper, we consider that a commit is done after each operation, for simplicity.

4.3.6 Quantum risks

Although ART is a good protocol if one wants to achieve post-compromise secrecy and asynchronicity, it doesn't get away from the quantum threats. We look at each part of the protocol in order to detect all the possible elements at risk.

Initialisation

Even before any tree construction or update, the initialisation phase poses a problem. If the key pairs used to create a tree are based on ECDH, it is then possible to derive a private key k by studying the public key $K = g^k$. The one-time keys are concerned about this as much as the identity key, meaning an attacker could easily impersonate someone else, and there would be no way to correctly authenticate them. The signature algorithms, also based on elliptic curves, won't help us.

Just with this problem, even before beginning to communicate, the whole protocol is jeopardized.

Tree creation

The very first step to create a group is constructing the corresponding tree structure, that is a graph with nodes, leaves and vertices acting like parent-child relation. Since this part doesn't require any cryptographic computation, nothing is threatened by quantum computers (neither by classical computers, by the way).

Then, the creator (Alice) of the tree has to compute a shared secret between her and each new member. This is done with X3DH, but this algorithm goes through ECDH to compute 3 secret values. So, each of these secrets is at risk.
After that, a KDF function is applied, to obtain a leaf key. SHA-256 and SHA-512 are proposed here. Even though none of them are broken by quantum algorithms, Grover's force to increase the number of bits, that is using SHA-512 would be preferred.

Once all the leaf keys are computed, Alice can calculate the key pair of all the parents. For each node, the private key is computed by using the private key of a child and the public key of the other, through ECDH. Then, the corresponding private key is computed. So, that means that a leaf could use the public key of its parent and its own public key to derive the sibling's private key.

Eventually, once the tree is created, the creator must send the details to the other members, and to authenticate the messages, she signs them by using elliptic curves, that is the signature is threatened as well.

Tree update

When a member update their leaf key, they then must update their corresponding public key, and the key pair of every node in their path. Since all these computations are based on ECDH, once again this process is at risk.
And to inform the other members of this update, the member sign their message, by using elliptic curves.

The process of adding a member is quite similar to the tree creation: a member adds a leaf in the tree structure, computes for them a leaf key, and finally calculate all the key pairs above them, before sending the details to everyone.
Thus, the problems are the same: risk when creating the shared key, when computing the key pairs and when sending the data.

Eventually, for the removal of a member, there is only one computation to do, which is to update a leaf key, in order to change the root key. But the problems are exactly the same as before, against quantum computers.

Sending message

To encrypt a message, a user needs a key as input of AES. Even though we consider that the key was securely obtain (which is unrealistic since dervied through DH), there is still the problem of

symmetric cryptography impacted by Grover's algorithm, which forces the user to increase the size of the key, and use AES-256 for minimal security, but the risk is not as important as with elliptic curves.
Nevertheless, there is once again the signature issue, always present.

Conclusion:

With the cryptographic primitives presented in the original ART paper, there is no way one could resist to quantum algorithms. We thus need to combine a solution presented in Chapter 3 with ART if we want to hope having a chance against quantum computer.

Chapter 5

Supersingular Isogeny Key Exchange, or SIKE

To solve the issue of group communication, ART protocol has been chosen. Concerning the quantum computers, we decided to go with SIKE. This fifth chapter presents the whole SIKE protocol, based on SIDH.
Section 5.1 presents the origins of the protocol and its evolution. In Section 5.2, we explain how the core element of the protocol, that is the elliptic curves, is working. In Section 5.3, we broach the concept of isogeny graph, structure on which parties move around until reaching a shared secret. The SIKE's steps are given in Section 5.4, along with an example to enlighten about the protocol with something more concrete. Eventually, in Section 5.5, we give subsidiary information, to go further on the process.

5.1 Introduction

Among the three Post-Quantum protocols studied as replacement to the current ones, isogeny-based cryptography is probably the most recent. Its security relies on the fact that, given two elliptic curves, it is hard to find an isogeny of a given degree to go from one curve to the other. And until now, quantum computers were not shown to able to solve this problem in subexponential-time, making this protocol interesting for a possible Post-quantum Signal protocol, presumably, but also a Post-quantum group protocol.

With Shor's algorithm, the use of Diffie-Hellman to create a shared secret became obsolete in quantum systems. This is why it could be swapped with a protocol with a similar idea, based on supersingular isogeny, meaning working with a specific kind of elliptic curve.
The idea of using isogeny in cryptography first appears with Couveignes, in 1997 [Cou06]. In 2010, Jao, Childs and Soukharev demonstrated that it was possible to break this scheme with quantum algorithms [CJS14]. The main problem of Couveignes' protocol is that the group containing the isogeny is commutative, hence making the attack much easier.

This is why Jao and De Feo [JDF11] decided to use supersingular elliptic curves, whose ring of endomorphims is non-commutative. This led to the key-agreement scheme, called SIDH, for Supersingular Isogeny Diffie-Hellman. The current state-of-the-art implementation is SIKE, created in 2017, and was submitted to the NIST competition, and is now at the Round 3. Most of the following explanation about this protocol comes from the corresponding paper [ACC+17]. The proof won't be given, because it is not the subject of this paper (and is quite complex), but for more details about it, see [JDF11], [DF17] and [Ren18].

5.2 Elliptic curve

Let's start with the general definition of an elliptic curve.

An elliptic curve E over a field K is defined as the equation

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in K$.
This general form is also called the *generalized Weiestrass equation.*

There exist several specific forms of elliptic curve, such as:

- Weierstrass form[1]: the most common, often used in papers talking about Elliptic Curve Theory.
  Its equation is
  $$E : y^2 = x^3 + Ax + B$$

- Edwards form: a recent form studied for Elliptic Curve Diffie-Hellman cryptography, used to simplify formulas in the theory of elliptic curves and functions [EH18].
  Its equation is
  $$x^2 + y^2 = 1 + dx^2y^2$$

- Montgomery form: this is the one we are going to use through the rest of the chapter. It is the most used in Isogeny-based cryptography.
  Its equation is
  $$By^2 = x^3 + Ax^2 + x$$
  Unless specific indication, we consider $B = 1$.

A couple of examples are given in Figure 5.1.

[1] also called *short Weiestrass form*, in order to differentiate with the *generalized eiestrass equation*

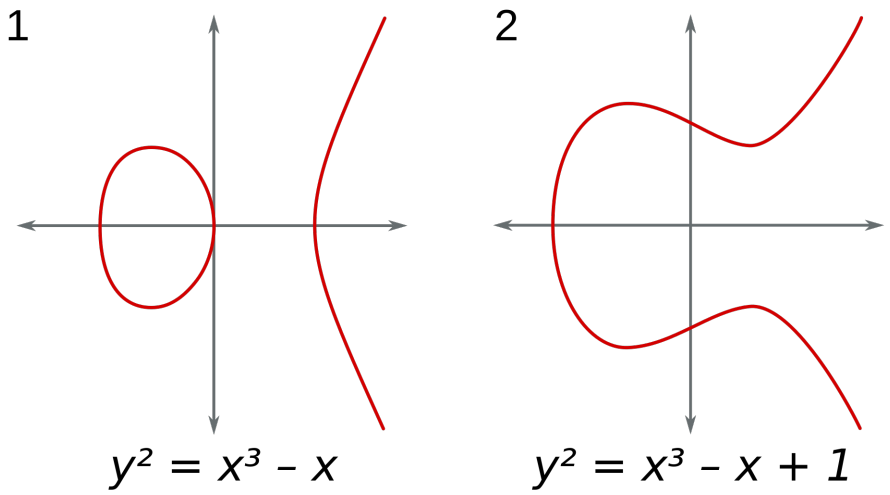


Figure 5.1: Two examples of elliptic curves, in the Weierstrass form

5.2.1 Fundamental theory

Before going more in detail about Elliptic Curve Theory, it could be good to first express mathematics basics, necessary for the rest of the chapter. Every notion presented below are concerning elliptic curves in general, not only the Mongtomery curve.

We first describe the two operations used to operate the points on a curve. Then, we restrain the number of points, to avoid working with sets of unlimited size. After that, we expend this restrained set in order to work with complex numbers.

Finite field

A field, often denoted K, is a set associated with two operations $(+, \cdot)$, called addition and multiplication, respecting the following conditions, $\forall x, y, z \in K$:

Addition:

1. Commutativity: $x + y = y + x$
2. Associativity: $(x + y) + z = x + (y + z)$
3. Existence of an additive identity: $\exists\, 0 \in K$, called zero, such that $x + 0 = x$
4. Existence of additive inverses: $\forall x, \exists - x \in K$ such that $x + (-x) = 0$

Multiplication:

1. Commutativity: $x \cdot y = y \cdot x$
2. Associativity: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

3. Distributivity: $(x + y) \cdot z = x \cdot z + y \cdot z$
4. Existence of a multiplicative identity: $\exists\ 1 \in K$, such that $x \cdot 1 = x$
5. Existence of multiplicative inverses: if $x \neq 0$, then $\exists x^{-1} \in K$ such that $x \cdot x^{-1} = 1$

For more details about fields, and field theory in general, see [Mar03]

The finite field $\mathbb{F}_p$

The elements of the finite field $\mathbb{F}_p$ are all the positive integers up to p:

$$\mathbb{F}_p = \{0, 1, ..., p - 1\}$$

All the field operations (+ and $\cdot$) are defined modulo p, meaning $a+b \equiv r[p]$, where r is the remainder of $a + b$ divided by p. Same goes for the multiplication.

Moreover, all elements in $\mathbb{F}_p$ have additive and multiplicative inverse in $\mathbb{F}_p$:

- Addition: finding the additive inverse of an element $a \in \mathbb{F}_p$ is easy: one just need to add p to $-a$
- Multiplication: finding the multiplicative inverse of a is harder with modulo. The default algorithm used is the Extended Euclidean algorithm, but with high numbers, it can quickly become expensive. That is why, in a lot of optimized algorithms, programmers tend to avoid computing them, with some equivalent calculations.

Finite field $\mathbb{F}_{p^2}$

The elements used in SIDH come from the finite field $\mathbb{F}_{p^2}$, which is an extension[2] of $\mathbb{F}_p$.

An element of the finite field $\mathbb{F}_{p^2}$ is written in the following form:

$$c = c_0 + c_1 \cdot i \qquad \text{where } c_0, c_i \in \mathbb{F}_p \text{ and } i^2 = -1$$

This writing can remind the complex numbers. Indeed, we have $\mathbb{F}_{p^2} = \mathbb{F}_p(i) := \{a + i \cdot b \quad | a, b \in \mathbb{F}_p\}$, similarly to the complex set: $\mathbb{C} = \mathbb{R}(i)$. Moreover, the arithmetic operations (+, $\cdot$) are defined similarly to the complex operations (for details, see [Byl90]).

[2] A field H is an extension of a field G if $G \subset H$ and H has the same field operations than G

5.2.2 Montgomery curve

Now that we have laid the foundations to work with elliptic curves in the cryptographic domain, we have to restrain on the type of elliptic curves we will be working on. We have defined the general form of an elliptic curve (EC) in the introduction of this section. Nevertheless, we use a specific form of elliptic curve in SIKE: the Montgomery curves.

Let $A, B \in \mathbb{F}_q$ with $B(A^2 - 4) \neq 0$, with q a power of a prime number ($q = p^n$, $n \in \mathbb{N}$).

A **Montgomery curve** $E_{A,B}$, defined over $\mathbb{F}_q$, often denoted $E_{A,B}/\mathbb{F}_q$, is the set of point $P = (x, y)$ satisfying the equation

$$By^2 = x^3 + Ax^2 + x \tag{5.1}$$

with an extra point $\mathcal{O}$, called the point at infinity, or neutral point.

To be more general, we denote $E(K)$ as the set of points in curve E over the field K

$$E(K) = \{P = (x_P, y_P) \in K^2 \quad | \quad B \cdot y_P^2 = x_P^3 + A \cdot x_P^2 + x_P\}$$

As indicated at the beginning of this section, we set $B = 1$ for the rest of this paper.

We will explain later why we prefer Montgomery-form elliptic curves.

5.2.3 Group law

In Section 5.2.1, we talked about the two operations that one can use in order to manipulate the points within elliptic curves, along with the properties, but we didn't give the details of these operations. This is what we are doing here, with Point addition, Point Doubling, Point Tripling and Multiplication.

Addition

In any elliptic curve, starting with two points, it is possible to create a new one. It is even possible to do this with only one point.

We start with two points, let's say:

$$P = (x_P, y_P), \quad Q = (x_Q, x_Q)$$

on an elliptic curve $E : By^2 = x^3 + Ax^2 + x$. For now, we assume that $P \neq Q$ and $x_P \neq x_Q$ in particular.

To find a new point R based on P and Q, we do as follow:

1. Draw the line L going through P and Q

2. Set R' the intersection between the line L and the curve E
3. Reflect R' across the x-axis, changing the y-coordinate, to obtain R

We thus write for now $P \oplus Q = R$.

An visual example is given Figure 5.2.

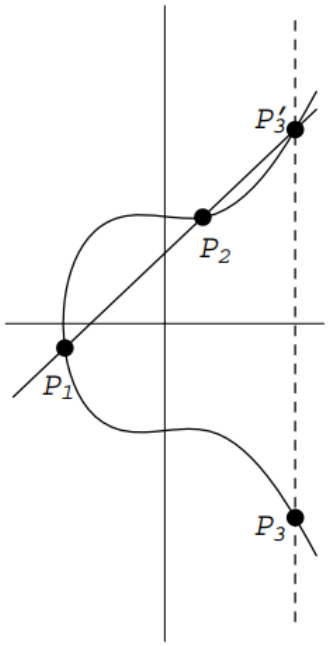


Figure 5.2: Adding points on an Elliptic curve

So, given the previous steps, how does one compute a points addition ?
To explain the process, we take over these steps, and "translate" them in mathematics calculations.

1. L is the line going through P and Q. Its slope is then:

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P}$$

The equation of L is then given by the equation:

$$y = \lambda \cdot (x - x_P) + y_P$$

which is also equivalent to $y = \lambda \cdot (x - x_Q) + y_Q$

2. To find the intersection with E, we substitute y in (5.1), and we get:

$$B(\lambda \cdot (x - x_P) + y_P)^2 = x^3 + Ax^2 + x$$

If we develop the left side and move everything to the right side, we can rearrange (with $B = 1$) the equation to:

$$0 = x^3 + (\lambda^2 - A) \cdot x^2 + \dots$$

The three roots of this equation of degree 3 correspond to the three points of intersection between L and E. Usually, it is not easy to solve a cubic equation, but here, we already know two of the roots, namely x_p and x_Q. Moreover, an equation of degree 3 which has 3 distinct solutions r, s, t, can be factorized:

$$x^3 + ax^2 + bx + c = (x - r)(x - s)(x - t) = x^3 - (r + s + t)x^2 + \dots$$

This means:

$$r + s + t = -a$$

If two roots are known, such as r and s, we can recover the third:

$$t = -a - r - s$$

In our case, we have: $r = x_P$, $s = x_Q$, $-a = (\lambda^2 - A)$, and $t = x_{R'}$ the point we want to find. Thus, we obtain:

$$x_{R'} = \lambda^2 - A - x_P - x_Q$$

and

$$y_{R'} = \lambda \cdot (x_{R'} - x_P) + y_P$$

3. Eventually, we just have to reflect across the x-axis the point R' in order to obtain R, which is made by changing the sign of $y_{R'}$, giving us:

$$x_R = \lambda^2 - A - (x_P + x_Q), \quad y_R = \lambda \cdot (x_P - x_R) - y_P$$

To sum up, we can rewrite the addition map as

$$\begin{array}{rccc} \oplus \quad : & E_A \times E_A & \to & E_A \\ & (x_P, y_P) \oplus (x_Q, y_Q) & \mapsto & \big(\lambda^2 - A - (x_P + x_Q) \quad ; \quad \lambda^2 \cdot (x_P - x_R) - y_P\big) \end{array} \tag{5.2}$$

where

$$\lambda = \frac{y_Q - y_P}{x_Q - x_P}$$

In the situation where $x_P = x_Q$ but $y_P \neq y_Q$, it means that the line L going through P and Q is vertical. We then say that the line intersects E in ∞.
Therefore, in this case:

$$P \oplus Q = \infty$$

Point doubling

The case where $P = Q$ means that we are calculating $P \oplus Q = [2]P$, which is called *Point doubling*.

When two points on a curve are really close one to another, the line through them approximates a tangent line. Therefore, when the two points coincide, we consider that the line L is the tangent line to the point P.

To find the slope at this point, we apply implicit differentiation on equation (5.1):

$$2y \cdot \frac{dy}{dx} = 3x^2 + 2Ax + 1$$

$$\Rightarrow \lambda = \frac{dy}{dx} = \frac{3x_P^2 + 2Ax_P + 1}{y_P}$$

As before, the equation of the line L is

$$y = \lambda(x - x_P) + y_P$$

We obtain the third degree equation

$$0 = x^3 - (\lambda^2 - A)x^2 + ...$$

This time, we know only one root, namely x_P, but it is a double root. Thus, we can proceed as previous, and we obtain:

$$x_R = \lambda^2 - A - 2x_P, \quad y_R = \lambda \cdot (x_P - x_R) - y_P$$

To sum up, we can rewrite the doubling map as

$$\begin{array}{rccc} [2] \quad : & E_A & \to & E_A \\ & (x_P, y_P) & \mapsto & \left(\lambda^2 - A - 2x_P \quad , \quad \lambda^2 \cdot (x_P - x_R) - y_P\right) \end{array} \tag{5.3}$$

with

$$\lambda = \frac{3x_P^2 + 2Ax_P + 1}{y_P}$$

Point tripling

The last operation is point tripling, meaning we want to compute $P \oplus P \oplus P = [3]P$.

This time, we are not going to explain the calculations made, but just show the result:

$$x_{[3]P} = \frac{(x_P^4 - 4Ax_P - 6x_P^2 - 3)^2 \cdot x_P}{(4Ax_P^3 + 3x_P^4 + 6x_P^2 - 1)^2}$$

and

$$y_{[3]P} = y_P \cdot \frac{(x_P^4 - 4Ax_p - 6x_P^2 - 3)(x_P^8 + 4A(x_P^7 + x_P) + 28(Ax_P^6 + x_P^5 + Ax_P^3 + x_P^2) + (16A^2 + 6)x_P^4 + 1)}{(4Ax_P^3 + 3x_P^4 + 6x_P^2 - 1)^3} \tag{5.4}$$

For more details about these computations, see [DIK06].

Nevertheless, one can remark that $x_{[3]P}$ only depends on x_P and A. We could have decided to make it depends on y_P as well, like in Point Addition and Point Doubling, but the reason behind this choice is that it is possible to optimize the computations of elliptic curves operations. We explain more in detail in the next Chapter.

Multiplication

Finally, if one wants to compute $[n]P$, with n higher than 3, they just need to decompose n in power of 2, and then apply the square computation.

For example, to calculate $[11]P$, we first compute

$$P, \quad [2]P, \quad [4]P = [2]([2][P]) \quad \text{and} \quad [8]P = [2]([4]P)$$

and we then apply Point addition:

$$[11]P = P \oplus [2]P \oplus [8]P$$

which is much faster than doing $P \oplus P \oplus \cdots \oplus P$ eleven times.

This is the same concept applied in Elliptic Curve Diffie-Hellman (ECDH) : it is possible to quickly compute $[k]P$ given k and P, but much harder to derive k given P and $[k]P$.

We now have everything needed to start studying more in detail SIDH :

1. $E_{AB} : By^2 = x^3 + Ax^2 + x$: the form of the curve used, called Montgomery elliptic curve,
2. $\mathbb{F}_{p^2}$: the set in which the points on an elliptic curve are defined,
3. the four main operations, to manipulate these points.

5.3 Isogeny graph

As explained in the introduction of this chapter and in Section 3.3.3, the goal for Alice and Bob is to generate a shared secret by using isogenies and elliptic curves. This shared secret is found after a specific number of repetitive steps. At the end of each step, a party gets a new elliptic curve and the one obtained at the last step is the secret. To go from one step to the other, isogenies are used.

This process could be represented by a graph. This is why the central concept in SIDH is Isogeny graph. It is this kind of graph that is used between two parties to exchange keys.

An **isogeny graph** is a graph, whose nodes consist of all elliptic curves in $\mathbb{F}_{p^2}$, up to $\mathbb{F}_p$-isomorphism, represented by their j-invariants, and the edges represent isogenies between these curves.

A lot of concepts are new here, and we are going to introduce all of them in the following parts.

5.3.1 The j-invariant nodes

As indicated, SIDH works in the quadratic extension $\mathbb{F}_{p^2}$. Such a group contains p^2 elements (since $\mathbb{F}_{p^2} = \{a + ib | a, b \in \mathbb{F}_p\}$), but we are only interested in a subset of it, of size $\lfloor \frac{p}{12} \rfloor + r$, where $r \in \{0, 1, 2\}$. The value of r depends on p mod 12. For more information, see [Sil09] (Theorem V.4.1 (c)).

This subset is the set of supersingular j-invariants in $\mathbb{F}_{p^2}$.

Supersingular elliptic curve

Characteristic: the characteristic of a field F is the smallest integer n such as $n \cdot x = 0, \forall x \in F$, with $\cdot$ being the multiplication operation of the field.

Based on this definition, the characteristic of an elliptic curve E over a field K is simply the characteristic of K.

The set of all the elliptic curves can be separated into two groups: the subset of the *supersingular* EC, and the set of the *ordinary* EC. This classification depends on the characteristic of the elliptic curve ([Gal01]).

The criteria to say that a curve is supersingular is : $\#E(\mathbb{F}_p) = p + 1$, where $\#$ is the cardinality. Moreover, the supersingular case offers a couple of advantages:

1. an efficient method is set up to construct and instantiate them easily,
2. and must of all, attacks against this kind of curve have exponential complexity.

j-invariant

As we have said, in the field $\mathbb{F}_{p^2}$, there exist p^2 elliptic curves. In practice, p can sometimes measure hundreds of bytes, meaning that $|\mathbb{F}_{p^2}|$ nodes would be clearly too much to consider in a single graph[3].

Instead, we are going to regroup the supersingular elliptic curves together into groups. But to do so, we need a criteria to compare EC. This is where the j-invariant function arrives.

Given an elliptic curve E_a in the Montgomery form

$$E_a : y^2 = x^3 + ax^2 + x$$

[3] $|G|$ gives the order (or cardinality) of the group G

we define the j-**invariant** of E_a as

$$j(E_a) = \frac{256(a^2-3)^3}{a^2-4} \in \mathbb{F}_{p^2}$$

With this application, we are going to group together the isomorphic curves.

Isomorphism: an isomorphism from G to H is a bijection (ie. injective and surjective application) $\phi : G \to H$ with the property that $\phi(ab) = \phi(a)\phi(b), \forall a, b \in G$. This means that ϕ preserves the group (or field) operations [Vin03].

Therefore, G and H are said to be isomorphic if there exists an isomorphism going from one group to the other.

Maybe the most important thing to say about j-invariant is the following theorem.

Theorem 1 *Two curves are isomorphic over a field $K = \overline{K}$ if and only if they have the same j-invariant.*

Proof: See [DF17], Section 2.

This is why, in the introduction of the section, we said that a node contains all the curves up to $\mathbb{F}_{p^2}$-isomorphism: we said we only work with the supersingular EC, and among this subset, we regroup the curves into the same group if they have the same j-invariant, and therefore in the same node in the graph.

Let's illustrate this part with an example.
Let $p := 434$. This means that we have $\lfloor \frac{p}{12} \rfloor + 2 = 37$ different j-invariants in $\mathbb{F}_{p^2}$ (we say there are 37 supersingular j-invariants).
They are represented in Figure 5.3

If we take $a_1 = 161 + 208i$ and $a_2 = 162 + 172i$, then we obtain:

$$j(E_{a_1}) = 304 + 364i = j(E_{a_2})$$

meaning that $E_{a_1} : y^2 = x^3 + (161 + 208i)x^2 + x$ and $E_{a_2} : y^2 = x^3 + (162 + 172i)x^2 + x$ are isomorphic, and both correspond to the node $304 + 364i$ in Figure 5.3.

5.3.2 SIDH in a nutshell

Before continuing talking about the isogeny graph and its edges, the isogenies, we could first sum up the Supersingular Isogeny Diffie-Hellman algorithm with our current knowledge.

As its name indicates, there is a link between SIDH and the traditional Diffie-Hellman protocol.

Let's just remind Diffie-Hellman:
In a cyclic group G, let g be a public parameter, accessible by anyone. Let Alice and Bob be the two

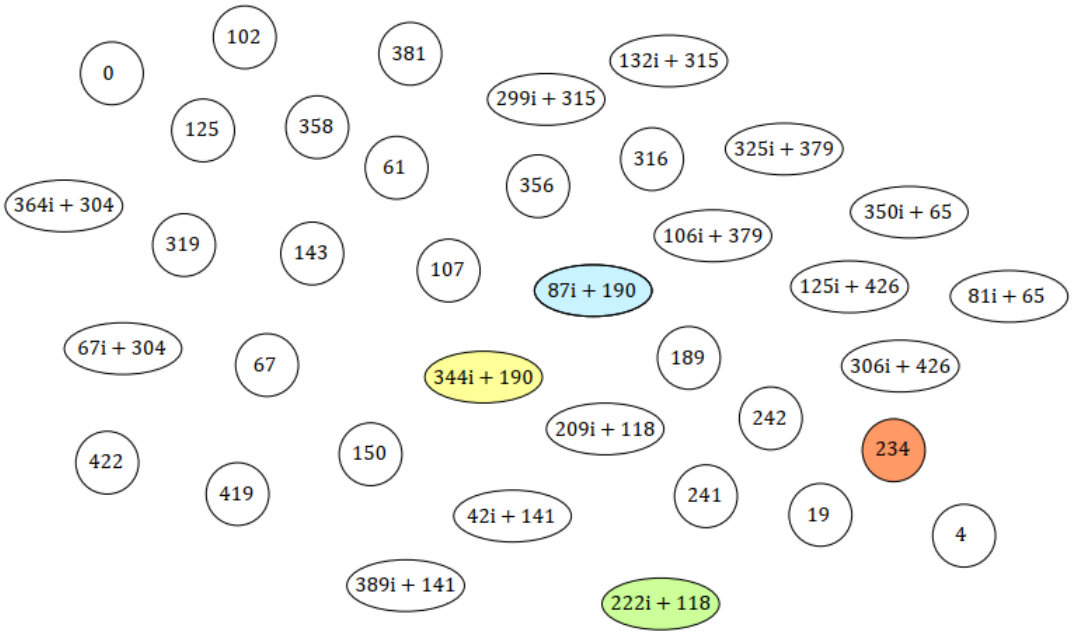


Figure 5.3: Set of the 37 supersingular j-invariants in $\mathbb{F}_{434^2}$ [Cos19]

parties implied in the protocol, each one possessing a secret key, a and b respectively. First, both parties compute g^{sk_a}, g to the power of their secret key. Alice obtains g^a, while Bob gets g^b.
They exchange their value, and apply again their secret key. The secret key obtained is $g^{ab} = g^{ba}$.

SIDH is, in a way, similar. Referring back to Figure 5.3:

1. Alice and Bob start both from the same (blue) node, corresponding to the j-invariant $190+87i$ (g in Diffie-Hellman),
2. Alice chooses a secret integer that, after some calculations, make her move from a node to another (starting with the blue one), until she reaches the (green) node, with j-invariant equal to $118 + 222i$ (g^a in DH),
3. She then sends to Bob this value, that corresponds to her public key,
4. Similarly, Bob also chooses a secret value, making him move until the yellow node, of j-invariance $190 + 344i$ (g^b in DH),
5. He sends this value to Alice, his public key,
6. With her secret key and Bob's public key, Alice can perform another sequence of moves, to end up at the (orange) j-invariant 234. This final value is the secret shared secret, g^{ab}, and Bob does the same, with his private key and Alice's public key.

To be accurate, the public keys contain more information necessary to ensure that Alice and Bob

land at the same node, but we will explain it later.

Each party is able to move from one node to another, ie. between j-invariants, by using the edges which are isogenies.

5.3.3 Isogeny

Presentation of Isogenies

Now, let's talk about the vertex of the graph, namely the isogenies.

Isogeny: Let E_1 and E_2 be elliptic curves. An *isogeny* from E_1 to E_2 is a non-constant morphism

$$\phi : E_1 \to E_2$$

satisfying $\phi(\mathcal{O}) = \mathcal{O}$.[4] ([Sil09])
An isogeny is not necessary invertible.

Similarly to the relation between isomorphism and isomorphic curves, two curves are said to be *isogenous* if there exists an isogeny going from one to the other.

We have already seen two examples of isogeny: the *multiplication-by-2* (or Point Doubling) and the *multiplication-by-3* (or Point Tripling). We also saw the translation as a first example, much earlier, in Figure 3.3.

The isogenies going from a curve to itself are called *endomorphism*. The basic example of it is the *multiplication-by-m*, as seen earlier

$$[m] : P \mapsto [m]P = \underbrace{P \oplus P \oplus ... \oplus P}_{\text{m times}}$$

In most examples cited before, we had endomorphisms. But in general, an isogeny is a map

$$\phi : E \to E'$$

going from one elliptic curve to another.

A question is still unanswered: how does one move from an elliptic curve to another one, ie. how is an isogeny computed ?

[4] To be accurate, we should write $\phi(\mathcal{O}_{E_1}) = \mathcal{O}_{E_2}$ since the neutral element in each elliptic curve may be different. But in the Montgomery curves, we say that in both cases, it corresponds to the point at infinity

Finding the Isogenies

An isogeny $\phi : E_1 \to E_2$ can be expressed with two rational functions f and g over $\mathbb{F}_p$, such that

$$\phi\big((x, y)\big) = \big(f(x), y \cdot g(x)\big)$$

Since $f(x)$ is rational, it can written as $f(x) = \frac{q(x)}{r(x)}$, with $q(x), r(x) \in \mathbb{F}_p[x]$ with no common factor. Same thing goes for $g(x)$.

We define the degree of ϕ as

$$deg(\phi) = max\{deg(q), deg(r)\}$$

Thus, a **ℓ-isogeny** is simply an isogeny ϕ such as $deg(\phi) = \ell$.

We also define the kernel of the same isogeny as

$$ker(\phi) = \{P \in E_1 : \phi(P) = \mathcal{O} \in E_2\}$$

where $\mathcal{O}$ is the neutral element.

For isogeny computations, Vélu's formula [Vél71] is used most of the time. However, this formula was originally computed for Weierstrass-form EC; if the elliptic curve is of another form, this formula may not preserve its form.
Nevertheless, in the original SIDH paper, De Feo, Jao and Plût applied Vélu's formula and composed with the appropriate isomorphisms to return to Montgomery form. The problem is that this approach is very expensive for producing 2-isogenies, and even worse for 3-isogenies.

Instead, Renes [Ren18] adapted a Theorem from [[CH17], Theorem 1] for the Montgomery curves.

Theorem 2 *Let K be a field, with $a \in K$ such that $a^2 \neq 4$, and $E/K : y^2 = x^3 + ax^2 + x$ a Montgomery curve. Let $G \subset K$ be a finite subgroup such that $(0, 0) \notin G$ and let ϕ be an isogeny such that $ker(\phi) = G$.*
Then, there exists a curve

$$E/G : y^2 = x^3 + Ax^2 + x$$

such that

$$\begin{array}{rccc} \phi\text{:} & E & \to & E/G \\ & (x, y) & \mapsto & \big(f(x), c_0 y \cdot f'(x)\big) \end{array}$$

where

$$f(x) = x \cdot \prod_{Q \in G^*} \frac{x \cdot x_Q - 1}{x - x_Q}$$

Moreover, writing

$$\pi = \prod_{Q \in G^*} x_Q \quad \textit{and} \quad \sigma = \sum_{Q \in G^*} \left(x_Q - \frac{1}{x_Q} \right)$$

we have $A = \pi(a - 3\sigma)$ and $c_0^2 = \pi$

Proof: See [Ren18], Section 4.1 .

2- and 3-isogenies

We need to change a last thing to the previous Theorem to really be able to apply it in the key exchange protocol in SIKE.

Indeed, SIKE prefers simply using 2-isogenies and 3-isogenies, easier to compute, and also giving us what we want: a way to move between the j-invariant nodes.
These choices of the two smallest primes currently give the most efficient instantiation of SIDH.

Moreover, when we take $\ell \neq p$, we get that the number of edges of each node in the ℓ-isogeny graph is $\ell + 1$ ([dF19], Corollary 29).

For Alice, we usually take $\ell = 2$, meaning that every node in Alice's graph will be of degree[5] 3, and for Bob, we take $\ell = 3$, meaning his 3-isogeny graph will be of degree 4. Nevertheless, the nodes are obviously the same. Otherwise, it would be impossible to obtain a shared secret.

A consequence of Theorem 2 is that we obtain a formula for 2-isogenies for points other than $(0, 0)$.

Corollary 2.1 *Let $E/K : y^2 = x^3 + ax^2 + x$ be a Montgomery curve over a field K. Let $P \in E(K)$ such that $P \neq (0, 0)$ and $[2]P = \mathcal{O}_E$[6].*
Then

$$\begin{array}{rccc} \phi : & E & \to & \tilde{E}/K : y^2 = x^3 + Ax^2 + x \\ & (x,\, y) & \mapsto & \big(f(x), yf'(x)\big) \end{array}$$

with

$$A = 2 \cdot (1 - 2x_P^2) \tag{5.5}$$

is a 2-isogeny with $ker(\phi) = \langle P \rangle$, where

$$f(x) = x \cdot \frac{x \cdot x_P - 1}{x - x_P} \tag{5.6}$$

Proof: See [Ren18], Section 4.2 .

Remark: For the y-coordinate, we thus obtain

$$y \cdot f'(x) = y \cdot \frac{x^2 x_P - 2x x_P^2 + x_P}{(x - x_P)^2} \tag{5.7}$$

Similarly, here is what we get for 3-isogenies.

Let $(x_P, y_P) \in E_a : y^2 = x^3 + ax^2 + x$ be a point of order 3 and let $\phi_3 : E_a \to E_A$ be the unique (up to isomorphism) 3-isogeny with kernel $\langle P \rangle$.
Then, E_A can be computed as

$$A = (ax_P - 6x_P^3 + 6)x_P \tag{5.8}$$

[5] a graph is of degree n if all the vertices are of degree n, ie. n edges are connected to each vertex
[6] This means that P is of order 2, or $char(P) = 2$

We then have:

$$\phi(x,y) = \left(\frac{x(xx_P - 1)^2}{(x - x_P)^2} \quad , \quad y \cdot \frac{(xx_P - 1)(x^2x_P - 3xx_P + x+)}{(x - x_P)^3} \right) \tag{5.9}$$

We have now the formulas to compute 2-isogenies and 3-isogenies, based on kernels.
This means that we need a last thing: kernels.

Finding kernels

We have defined earlier the degree of an ℓ-isogeny as the maximal degree of the rational function. But based on [Sil09] (Theorem III.4.10), it is also the number of elements in its kernel:

$$deg(\phi) = \ell = |ker(\phi)|$$

This means that the cardinality of the kernel of an 2-isogeny is 2. Moreover, we know that the neutral element $\mathcal{O}$ is always included in the kernel of an isogeny, since we said in the definition of the isogeny $\phi(\mathcal{O}) = \mathcal{O}$

$$\mathcal{O} \in ker(\phi)$$

Therefore, there is just another point to find to complete the kernel.

Let's go back to the isogeny *mulitplication-by-2*, that we also called *point doubling*:
For a fixed Montgomery curve $E_a : y^2 = x^3 + ax^2 + x$, the isogeny is (for the x-coordinate)

$$\begin{array}{rccc} [2]: & E_a & \to & E_a \\ & x & \mapsto & \frac{(x^2-1)^2}{4x(x^2+ax+1)} \end{array} \tag{5.10}$$

We observe that the doubling map has a denominator that might be equal to 0, meaning it would create "exceptional points".
To find them, we need to find the zeros.

Since $4x(x^2 + ax + 1) = 4y^2$, we look at $4y^2 = 0 \Leftrightarrow y = 0$.

Then, the three points are $(0, 0)$, $(\alpha, 0)$ and $(1/\alpha, 0)$, where $\alpha^2 + a\alpha + 1 = 0$.
The last point comes from the second:

$$\alpha^2 + a\alpha + 1 = 0 \Leftrightarrow \alpha^2(1 + \frac{a}{\alpha} + \frac{1}{\alpha^2}) = 0 \Leftrightarrow 1 + a \cdot \frac{1}{\alpha} + \left(\frac{1}{\alpha}\right)^2 = 0$$

These three points, of order 2 on E_a, with the neutral element $\mathcal{O}$ form the entire kernel of the doubling map.

And from this, we can create 3 subgroups:

$$\{\mathcal{O}, (0,0)\}, \quad \{\mathcal{O}, (\alpha, 0)\} \quad \text{and} \quad \{\mathcal{O}, (1/\alpha, 0)\}$$

Each of these subgroups is used to compute a 2-isogeny, meaning we obtain in total three 2-isogenies, as predicted earlier, when we said that in a 2-isogeny graph, each node would have three edges connected to it.

For the 3-isogenies, the idea is the same :
We take back to the isogeny *mulitplication-by-3*. For a fixed Montgomery curve E_a of the form $y^2 = x^3 + ax^2 + x$, the isogeny is (for the x-coordinate)

$$\begin{array}{rccc} [3]: & E_a & \to & E_a \\ & x & \mapsto & \frac{x(x^4-6x^2-4ax^3-3)^2}{4x(3x^4+4ax^3+6x^2-1)^2} \end{array} \tag{5.11}$$

Again, the denominator gives us four exceptional points. Let's suppose its four roots are β, δ, ψ and τ; each of these values correspond to the x-coordinate of points of order 3 in E_a, but this time, there are two non-zero y-coordinates for each x.
Together with $\mathcal{O}$, we obtain nine points in the kernel of $[3]$:

$$kern([3]) = \{\mathcal{O}, (\beta, y_1), (\beta, -y_1), (\delta, y_2), (\delta, -y_2), (\psi, y_3), (\psi, -y_3), (\tau, y_4), (\tau, -y_4)\}$$

Eventually, we can form four subgroups, of size 3, that will give use the four 3-isogenies.

$$\{\mathcal{O}, (\beta, y_1), (\beta, -y_1)\}$$

$$\{\mathcal{O}, (\delta, y_2), (\delta, -y_2)\}$$

$$\{\mathcal{O}, (\psi, y_3), (\psi, -y_3)\}$$

$$\{\mathcal{O}, (\tau, y_4), (\tau, -y_4)\}$$

We now have everything necessary to use SIKE, that is : the isogeny graph, where one can go from a node, defined as a j-invariant, to another by taking an edge, representing an isogeny. But before going in detail in the SIKE protocol, an example is welcomed, to apply everything seen on something more concrete.

5.3.4 Example

In this part, we are going to generate the edge of isogeny graph, given the nodes, that is the j-invariant of the curves in F_{p^2}.

Let's take back our previous example, with $p = 434$, giving us 37 j-invariants, shown is Figure 5.3.

As said before, both Alice and Bob have their own graph. For Alice, it will be the 2-isogeny graph, whereas for Bob, it will be the 3-isogeny one. But currently, there are only the nodes. And we are going to construct the vertices.

Alice starts from the curve $E_a : y^2 = x^3 + (161 + 208i)x^2 + x$, with $j(E_a) = 304 + 364i$. So, we are going to find the vertex coming from this node. With the 2-isogeny equation in (5.6), we find $E_{a'} : y^2 = x^3 + (423 + 102i)x^2 + x$ with $j(E_{a'}) = 190 + 364i$; the kernel was generated by $(\alpha, 0) \in E_a$ with $\alpha = 68 + 350i$. Therefore, we can draw an edge between these two j-invariants.

Furthemore, there exist two other kernels: one generated by $(1/\alpha, 0)$ and the other by $(0, 0)$. These produce two additional edges, one connecting $j = 304 + 364i$ to $j = 319$ and one connecting $j = 304 + 364i$ to $j = 67$, respectively.

Remark: We cannot use the formula applied on $(\alpha, 0)$ and $(1/\alpha)$ with $(0, 0)$, because we would divide by 0. The equation is thus different, and much more complex. For more information, see [dF19].

By doing the same operations for the other j-invariants, we obtain all the vertex, depicted in Figure 5.4[7].

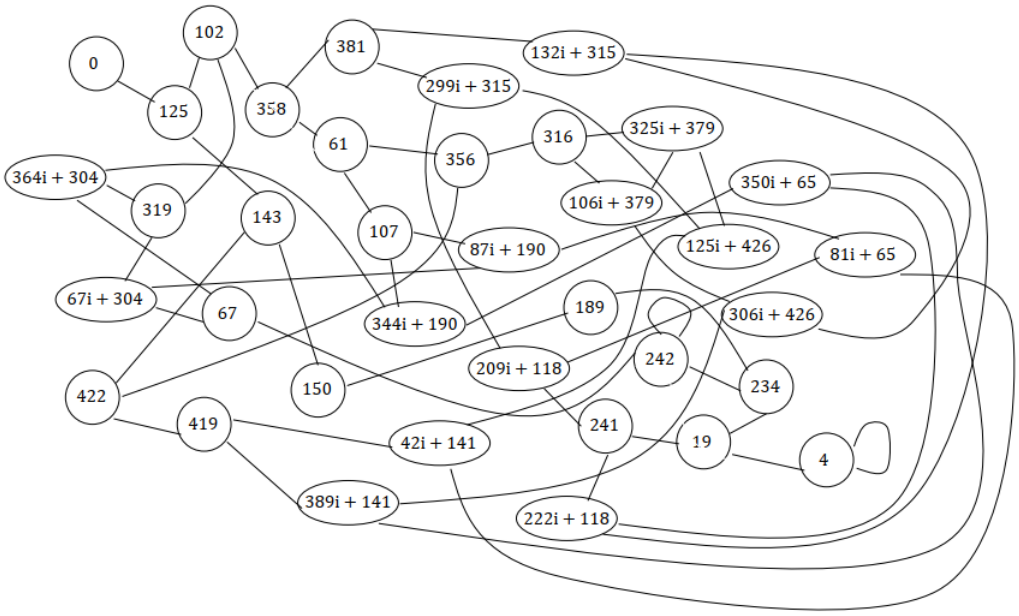


Figure 5.4: The 2-isogeny graph for p = 434. The 37 nodes are the supersingular j-invariants and the edges between them correspond to 2-isogenies

The 3-isogeny graph is similarly constructed, but this time, there are four outgoing edges for each

[7]We remark that for $j \in \{0, 4, 242\}$, there are only 2 edges. These are exceptional points, and the reason of this difference is explained in [Cos19], Section 6

j-invariant[8]

With the same curve E_a as before, and by using the point $(\beta, \gamma) = (56 + 321i, 174 + 303i)$ of order 3 on E_a, we find the new curve $E_{a'} : y^2 = x^3 + 415x^2 + x$, giving $j(E_{a'}) = 189 = j_1$.

And with the three other points, we obtain three curves, giving us the j-invariants $j_2 = 19$, $j_3 = 141 + 42i$ and $j_4 = 379 + 106i$.

The final result is shown in Figure 5.5.

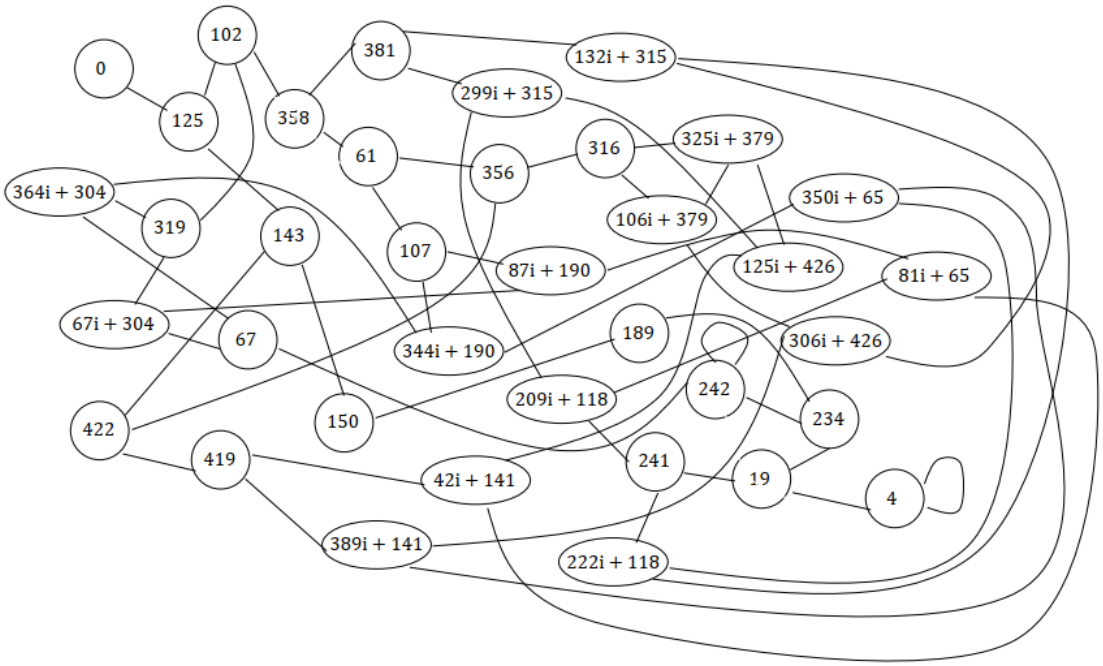


Figure 5.5: The 3-isogeny graph for p = 434. The 37 nodes are the supersingular j-invariants and the edges between them correspond to 3-isogenies

[8]Again, there is a small number of exceptions, for $j \in \{0, 4, 125, 242\}$

5.4 SIKE protocol

Now that we have seen every theoretical aspects concerning SIKE, it's time to actually present the protocol itself, through the main steps one has to follow, and then with another example to illustrate everything that was explained.

But just before, we will clearly recall the difference between SIDH and SIKE. On one hand, SIDH is a post-quantum Diffie-Hellman key exchange system, with a non-abelian structure based on supersingular isogenies. On the other hand, SIKE is the name of the specific submission given to NIST (for standardization), with a set of specific and fixed parameters.
In short, SIKE is a kind of SIDH, but not all SIDH parameters are SIKE.

5.4.1 Protocol's main steps

The SIKE protocol is composed of four steps. The first sets up the public parameters, common to all the party involved in the protocol. They agree on the curve and the set $\mathbb{F}_{p^2}$. After that, each party has to compute the isogeny, based on kernels. The first kernel is based on the basis points decided beforehand. Then, a certain number of loops is made, in which n isogenies are computed, making the user move from one elliptic curve to another, through the corresponding j-invariant. This combination allows the user to construct its private key and some image points, the latter ones being then sent to the other parties. With this public key, another user can compute the shared secret, which is the j-invariant obtained after the same steps made in the calculation of the public key.

(i) Parameters

The first thing to do is choosing the parameters, all of them being public. To be accurate, it is not exactly a step of the process since it is made beforehand, but still it is necessary in order to work with SIKE.

p is set as :

$$p = 2^{e_A}3^{e_B} - 1$$

with $2^{e_A} \simeq 3^{e_B}$

The protocol begins on an initial supersingular curve $E_a : y^2 = x^3 + ax^2 + x$, depending on the choice of a.

Alice's public secret depends on three elements:

- Alice secret key k_A,
- and two basis points of order 2^{e_A}: P_A and Q_A

These two points will be used to generate the kernel for the initial isogeny, with a secret linear combinations.

Bob generates the equivalent set of points for himself.

(ii) Secret key and generator

To compute her public key, Alice first chooses her secret key, namely $k_A \in 0, 1, ..., 2^{e_A} - 1$.
Then, Alice computes a secret generator point S_A:

$$S_A = P_A + [k_A]Q_A$$

P_A and Q_A need to be linearly independent. These points being of order 2^{e_A}, S_A is as well. The reason for the independence is that otherwise, there would exist k_A such that $P_A + [k_A]Q_A = [0, 0]$, which is of order 2, meaning no multiplication is possible on such a point.

(iii) Public key

To reach the node set in her public key, Alice moves e_A times in her 2-isogeny graph, through the 2-isogenies she computes.

Each 2-isogenies are calculated based on the kernel computed each time. To compute it, she follows these steps, with t going from $e_A - 1$ to 0:

1. Compute S'_A as S_A multiplied by 2^t
2. Take α as the x-coordinate of S'_A
3. Update the curve parameter A, by $2(1 - \alpha^2)$
4. Compute the j-invariant: $j = 256\frac{(A^2-3)^3}{A^2-4}$
5. The new 2-isogeny is then:

$$\phi_t([x, y]) = \left(\frac{x(x\alpha - 1)}{x - \alpha}; y \cdot \frac{x^2\alpha - 2x\alpha^2 + \alpha}{(x - \alpha)^2}\right)$$

6. Update S_A, P_B and Q_B with their image: $\phi_t(S_A), \phi_t(P_B)$ and $\phi_t(Q_B)$
7. Start over, with t decremented

After calculating an isogeny, she uses it to compute a new supersingular elliptic curves $E_{a'}$ and then compute its j-invariant $j(E_{a'})$, which correspond to the new node to go.

Her secret isogeny is

$$\phi_1 \circ \dots \phi_{e_A-1} = \phi_A : E \to E_A$$

where $E_A = E/\langle S_A \rangle$ (meaning that ϕ_A is based on the kernel $\langle S_A \rangle$), is a composition of the e_A isogenies of degree 2 found.

Eventually, Alice public key is

$$PK_A = (E_A, P'_B, Q'_B) = (\phi_A(E), \phi_A(P_B), \phi_A(Q_B))$$

where

- $E_A = \phi_A(E)$ is the image curve
- $P'_B = \phi_A(P_B)$ and $Q'_B = \phi_A(Q_B)$ are the images of Bob's public basis points.

This triplet of point composes the public key of Alice, and she will send it to Bob.

Similarly, Bob chooses $k_B \in \{0, 1, ..., 3^{e_B} - 1\}$, calculates S_B and computes his secret isogeny $\phi_B : E \to E_B$ (based on (5.9)), a composition of e_B isogenies of degree 3. His public key is then

$$PK_B = (E_B, P'_A, Q'_A) = (\phi_B(E), \phi_B(P_A), \phi_B(Q_A))$$

(iv) Shared secret

Once Alice received Bob's public key, she uses once again her secret key k_A to compute the new secret generator S'_A

$$S'_A = P'_A + [k_A]Q'_A$$

As before, she computes another secret isogeny

$$\phi'_A : E_B \to E_{AB}$$

Finally, the shared secret is computed

$$j_{AB} = j(E_{AB})$$

Bob does the same: he calculates $S'_B = P'_B + [k_B]Q'_B$, computes his new secret isogeny $\phi'_B : E_A \to E_{BA}$, and can eventually computes the shared secret $j_{BA} = j(E_{BA})$.

Remark: Why the image points ?
In Alice's and Bob's public key, they put the image points of each other. The reason is that it is necessary to apply $\phi_{A'}$ on E_B or $\phi_{B'}$ on E_A, because it would not really make sense to consider a composition of the isogenies $\phi_A : E \to E_A$ and $\phi_B : E \to E_B$, given their domains and codomains. These images of the basis points are used here in order to "describe" an isogeny whose domain is E_B, and an isogeny starting from E_A for Bob. That's why S'_A and S'_B are computed, based on the image points.

5.4.2 Example

As an illustration, let's finish with a full example of the SIKE protocol.

We continue with the same graphs as before. The public parameters are:

- $p = 2^4 3^3 - 1 = 434$, meaning $e_A = 4$ and $e_B = 3$
- $E_{a_0} : y^2 = x^3 + a_0 + x$, with $a_0 = 423 + 329i$ the starting curve
- $j(E_{a_0}) = 190 + 87i$ the starting node
- $P_A = (122 + 163i, 5 + 14i)$ and $Q_A = (252 + 54i, 295 + 136i)$, Alice's basis points
- $P_B = (322 + 136i, 85 + 291i)$ and $Q_B = (74 + 53i, 258 + 401i)$, Bob's basis points

Alice's public key computation

Let's say that Alice decides that her secret value is

$$k_A = 9$$

We indeed have $k_A \in \{0, 1, ..., 2^4 - 1\}$.

The first step is the computation of her secret generator point S_A.

$$\begin{array}{rcccc} S_A & = & P_A & + & [k_A] \quad Q_A \\ & = & (122 + 163i, 5 + 14i) & + & [9] \quad (252 + 54i, 295 + 136i) \\ & = & (224 + 59i, 7 + 312i) & & \end{array}$$

Since P_A's and Q_A's order is 16 on the curve E_{a_0}, so is S_A's.

Alice now computes her public key, only by applying the Point Doubling Operation defined in (5.10), and the 2-isogeny operation in (5.6).

- Compute ϕ_0:
  Three repeated applications of the doubling onto S_A produces the point

  $$R_A^{(1)} = [8]S_A = (37 + 18i, 0)$$

  which is of order 2 on E_{a_0}.
  Inputting $R_A^{(1)}$ into (5.5) gives E_{a_1}, with $a_1 = 132 + 275i$ and $j(E_{a_1}) = 107$. We also obtain the map

  $$\phi_0(x) = \frac{x((37 + 18i)x - 1)}{x - (37 + 18i)}$$

  The basis points are updated as well

  $$P_B^{(1)} = \phi_0(P_B) = (290 + 92i, 299 + 61i)$$
  $$Q_B^{(1)} = \phi_0(Q_B) = (352 + 222i, 257 + 114i)$$
  $$S_A^{(1)} = \phi_0(S_A) = (312 + 296i, 262 + 286i)$$

  $S_A^{(1)}$ is now of order 8 on E_{a_1}.

- Compute ϕ_1:
  Two repeated applications of the doubling onto $S_A^{(1)}$ produces the point
  $$R_A^{(2)} = [4]S_A^{(1)} = (49 + 7i, 0)$$
  which is of order 2 on E_{a_1}.
  Inputting $R_A^{(2)}$ into (5.5) gives E_{a_2}, with $a_2 = 76 + 273i$ and $j(E_{a_2}) = 190 + 344i$. We also obtain the map
  $$\phi_1(x) = \frac{x((49 + 7i)x - 1)}{x - (49 + 7i)}$$
  The basis points are updated once again
  $$P_B^{(2)} = \phi_1(P_B^{(1)}) = (215 + 165i, 229 + 9i)$$
  $$Q_B^{(2)} = \phi_1(Q_B^{(1)}) = (360 + 222i, 67 + 145i)$$
  $$S_A^{(2)} = \phi_1(S_A^{(1)}) = (339 + 375i, 194 + 422i)$$
  $S_A^{(2)}$ is now of order 4 on E_{a_2}.
- Compute ϕ_2:
  One application of the doubling onto $S_A^{(2)}$ produces the point
  $$R_A^{(3)} = [2]S_A^{(2)} = (328 + 344i, 0)$$
  which is of order 2 on E_{a_2}.
  Inputting $R_A^{(3)}$ into (5.5) gives E_{a_3}, with $a_3 = 341 + 289i$ and $j(E_{a_3}) = 304 + 364i$. We also obtain the map
  $$\phi_2(x) = \frac{x((328 + 344i)x - 1)}{x - (328 + 344i)}$$
  The basis points are updated
  $$P_B^{(3)} = \phi_3(P_B^{(2)}) = (333 + 137i, 407 + 383i)$$
  $$Q_B^{(3)} = \phi_3(Q_B^{(2)}) = (260 + 59i, 273 + 137i)$$
  $$S_A^{(3)} = \phi_3(S_A^{(2)}) = (156 + 391i, 0 + 0j)$$
  $S_A^{(3)}$ is now of order 2 on E_{a_3}.
- Compute ϕ_3:
  Since $S_A^{(3)} = (156 + 391i, 0)$ is of order 2, no scalar multiplication is necessary. Inputting $S_A^{(3)} = R_A^{(4)}$ into (5.5) gives E_{a_4}, with $a_4 = 430 + 355i$ $(= a_A)$ and $j(E_{a_4}) = 319$. We also obtain the map
  $$\phi_3(x) = \frac{x((156 + 391i)x - 1)}{x - (156 + 391i)}$$

The basis points are updated

$$P_B^{(4)} = \phi_3(P_B^{(3)}) = (408 + 155i, 1 + 366i)$$

$$Q_B^{(4)} = \phi_3(Q_B^{(3)}) = (367 + 122i, 358 + 189i)$$

And since $S_A^{(3)} \in ker(\phi_3)$, we leave $S_A^{(3)}$ as it is.

Alice's secret 2^4-isogeny is eventually the composition of the four 2-isogenies computed before.

$$\begin{array}{rccc} \phi_A : & E_{a_0} & \to & E_{a_4} \\ & \phi_A(x) & \mapsto & (\phi_3 \circ \phi_2 \circ \phi_1 \circ \phi_0)(x) \end{array}$$

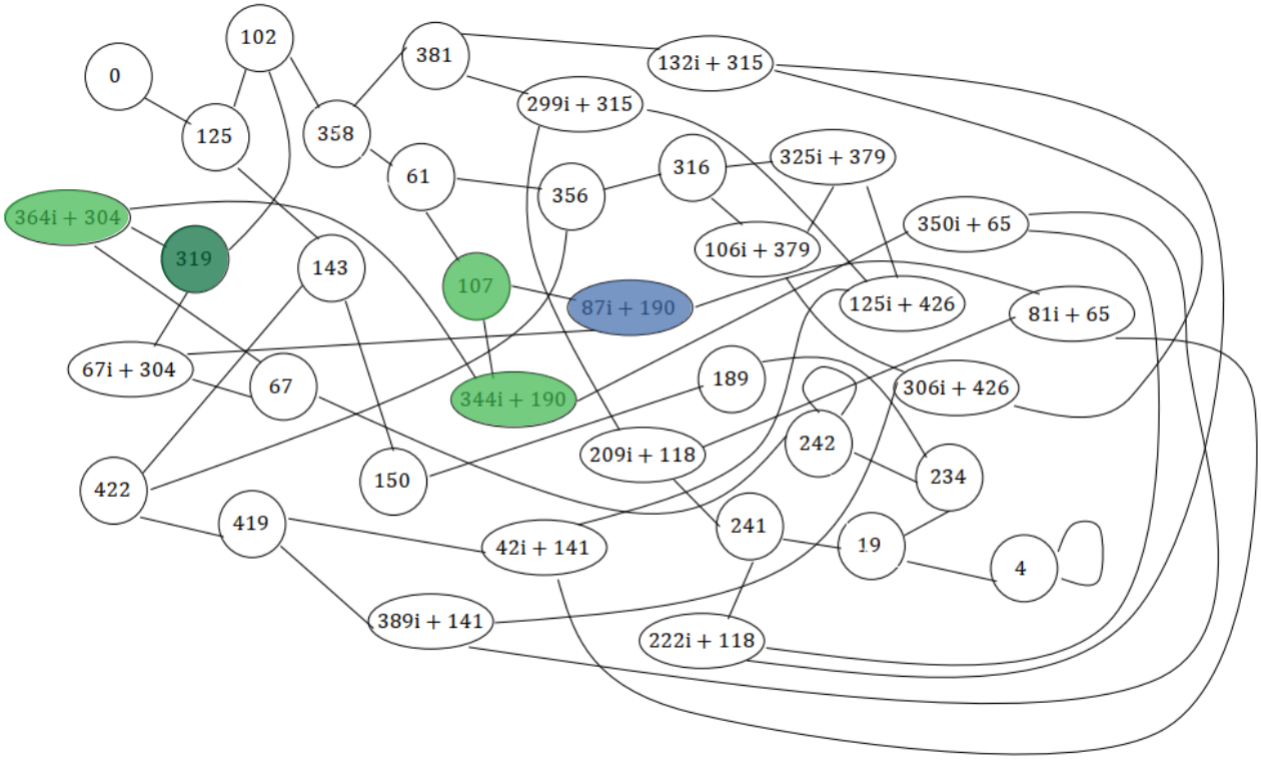


Figure 5.6: Alice's public key generation. The starting j-invariant is $190 + 87i$, and the final is 319

Eventually, her public key is

$$\begin{array}{rcl} PK_A & = & \big(a_A, \phi_A(P_B), \phi_A(Q_B)\big) \\ & = & \big(430 + 355i, (408 + 155i, 1 + 366i), (367 + 122i, 358 + 189i)\big) \end{array}$$

Bob's public key generation
Let's say that Bob decides that his secret value is

$$k_B = 2$$

We indeed have $k_A \in \{0, 1, ..., 3^3 - 1\}$.
The first step is the computation of his secret generator

$$\begin{array}{rcccc} S_B & = & P_B & + & [k_B] \quad Q_B \\ & = & (322 + 136i, 85 + 291i) & + & [2] \quad (74 + 53i, 258 + 401i) \\ & = & (55 + 327i, 238 + 230i) & & \end{array}$$

Since P_B and Q_B are of order 16 on the curve E_{a_0}, so is S_B.
Bob now computes her public key, only by applying the Point Tripling Operation defined in (5.11), and the 3-isogeny operation in (5.9).

- Compute ϕ_0:
  Two repeated applications of the tripling onto S_B produces the point

  $$R_B^{(1)} = [9]S_B = (37 + 23i, 302 + 4i)$$

  which is of order 3 on E_{a_0}.
  Inputting $R_B^{(1)}$ into (5.11) gives E_{a_1}, with $a_1 = 2 + 134i$ and $j(E_{a_1}) = 379 + 106i$. We also obtain the map

  $$\phi_0(x) = \frac{x((37 + 23i)x - 1)^2}{(x - (37 + 23i))^2}$$

  The basis points are updated

  $$P_A^{(1)} = \phi_0(P_A) = (276 + 99i, 320 + 9i)$$

  $$Q_A^{(1)} = \phi_0(Q_A) = (47 + 185i, 114 + 265i)$$

  $$S_B^{(1)} = \phi_0(S_B) = (381 + 430i, 378 + 65i)$$

  $S_B^{(1)}$ is now of order 9 on E_{a_1}.
- Compute ϕ_1:
  One application of the tripling onto $S_B^{(1)}$ produces the point

  $$R_B^{(2)} = [3]S_B^{(1)} = (193 + 58i, 389 + 317i)$$

  which is of order 3 on E_{a_1}.
  Inputting $R_B^{(2)}$ into (5.11) gives E_{a_2}, with $a_2 = 276 + 262i$ and $j(E_{a_2}) = 316$. We also obtain the map

  $$\phi_0(x) = \frac{x((193 + 58i)x - 1)^2}{(x - (193 + 58i))^2}$$

The basis points are updated as well

$$P_A^{(2)} = \phi_1(P_A^{(1)}) = (385 + 410i, 126 + 160i)$$

$$Q_A^{(2)} = \phi_1(Q_A^{(1)}) = (209 + 78i, 65 + 199i)$$

$$S_B^{(2)} = \phi_1(S_B^{(1)}) = (168 + 19i, 88 + 60i)$$

$S_B^{(2)}$ is now of order 3 on E_{a_1}.

- Compute ϕ_2:
  Since $S_B^{(2)} = (168+19i, 88+60i)$ is of order 3, no scalar multiplication is necessary. Inputting $S_B^{(2)} = R_B^{(3)}$ into (5.11) gives E_{a_3}, with $a_3 = 132 + 275i$ and $j(E_{a_3}) = 107$. We also obtain the map

  $$\phi_0(x) = \frac{x((168 + 19i)x - 1)^2}{(x - (168 + 19i))^2}$$

  The basis points are also updated

  $$P_A^{(3)} = \phi_2(P_A^{(2)}) = (139 + 64i, 135 + 66i)$$

  $$Q_A^{(3)} = \phi_2(Q_A^{(2)}) = (148 + 72i, 159 + 378i)$$

  And since $S_A^{(2)} \in ker(\phi_2)$, we leave $S_A^{(2)}$ as it is.

Bob's secret 3^3-isogeny is eventually the composition of the three 3-isogenies computed before.

$$\begin{array}{rccc} \phi_B : & E_{a_0} & \to & E_{a_3} \\ & \phi_B(x) & \mapsto & (\phi_2 \circ \phi_1 \circ \phi_0)(x) \end{array}$$

Eventually, his public key is

$$\begin{array}{rcl} PK_B & = & \big(a_B, \phi_A(P_B), \phi_A(Q_B)\big) \\ & = & \big(132 + 275i, (139 + 64i, 135 + 66i), (148 + 72i, 159 + 378i)\big) \end{array}$$

Alice's shared secret computation
Alice starts over from the curve $E_B : y^2 = x^3 + a_B x^2 + x$, with a_B received within Bob's public key.[9].

Once again, Alice's first step is to compute a secret generator E_{a_0} based on her secret key k_A.

$$\begin{array}{rcccccc} S_A & = & \phi_B(P_A) & + & [k_A] & \phi_B(Q_A) \\ & = & (139 + 64i, 135 + 66i) & + & [9] & (148 + 72i, 159 + 378i) \\ & = & (55 + 86i, 326 + 44i) & & & \end{array}$$

[9]Since we are doing the same process, we write E_0 for E_B and $a_0 = 76 + 273i = a_B$ as the new starting curve

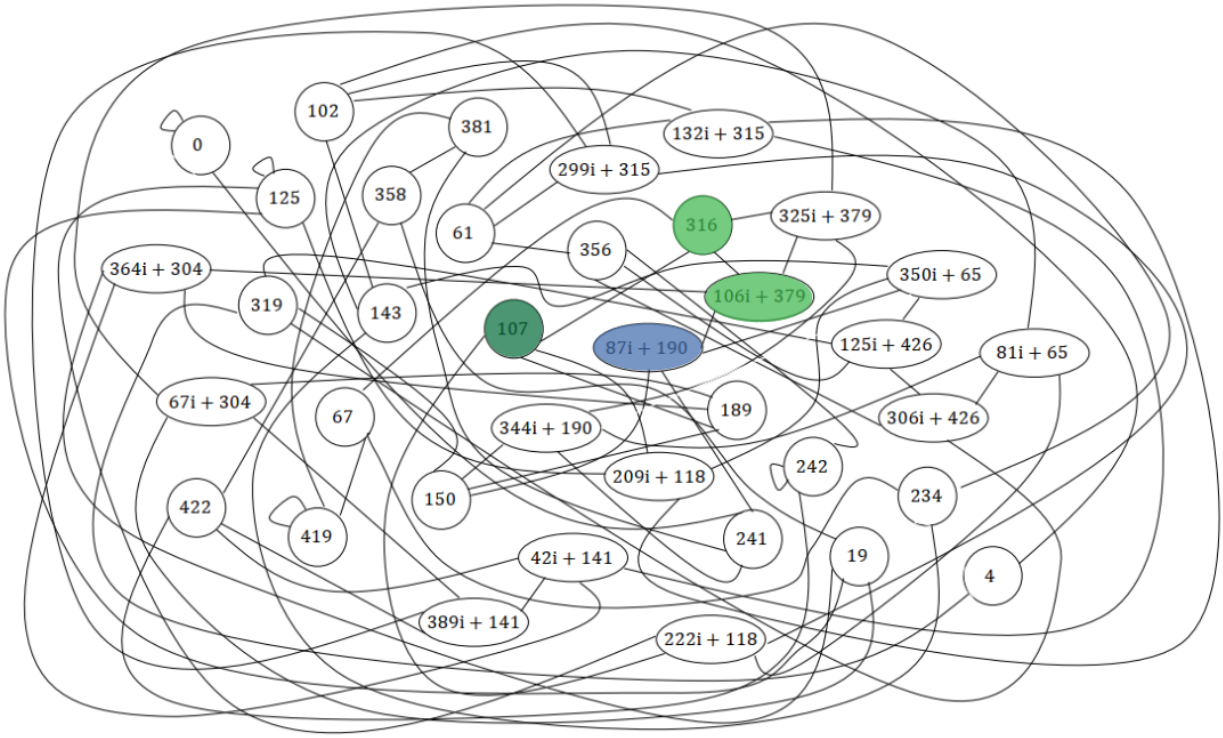


Figure 5.7: Bob's public key generation. The starting j-invariant is $190 + 87i$, and the final is 107

Then, she applies the exact same steps she did before, with the only difference that she doesn't need to compute the image of the basis points anymore, ie. $\phi_i(P_B^{i-1})$ and $\phi_i(Q_B^{i-1})$, because only the final node is wanted.
The four 2-isogenies computations are summarised as

$$\begin{array}{lll} \phi_0 : E_{a_0} \to E_{a_1} & \text{with } a_1 = 76 + 273i & \text{and } j(E_{a_1}) = 190 + 344i \\ \phi_1 : E_{a_1} \to E_{a_2} & \text{with } a_2 = 341 + 289i & \text{and } j(E_{a_2}) = 304 + 364i \\ \phi_2 : E_{a_2} \to E_{a_3} & \text{with } a_3 = 428 + 414i & \text{and } j(E_{a_3}) = 67 \\ \phi_3 : E_{a_2} \to E_{a_4} & \text{with } a_4 = 246i & \text{and } j(E_{a_4}) = 242 \end{array}$$

The shared secret she obtains is $j_{AB} = 242$.

Bob's shared secret computation
Bob starts over from the curve $E_A : y^2 = x^3 + a_a x^2 + x$, with a_A received within Alice's public key.

Once again, Bob's first step is to compute a secret generator E_{a_0} based on his secret key k_B.

$$\begin{array}{rcccc} S_B & = & \phi_A(P_B) & + & [k_B] \quad \phi_A(Q_B) \\ & = & (408 + 155i, 1 + 366i) & + & [2] \quad (367 + 122i, 358 + 189i) \\ & = & (367 + 377i, 430 + 98i) & & \end{array}$$

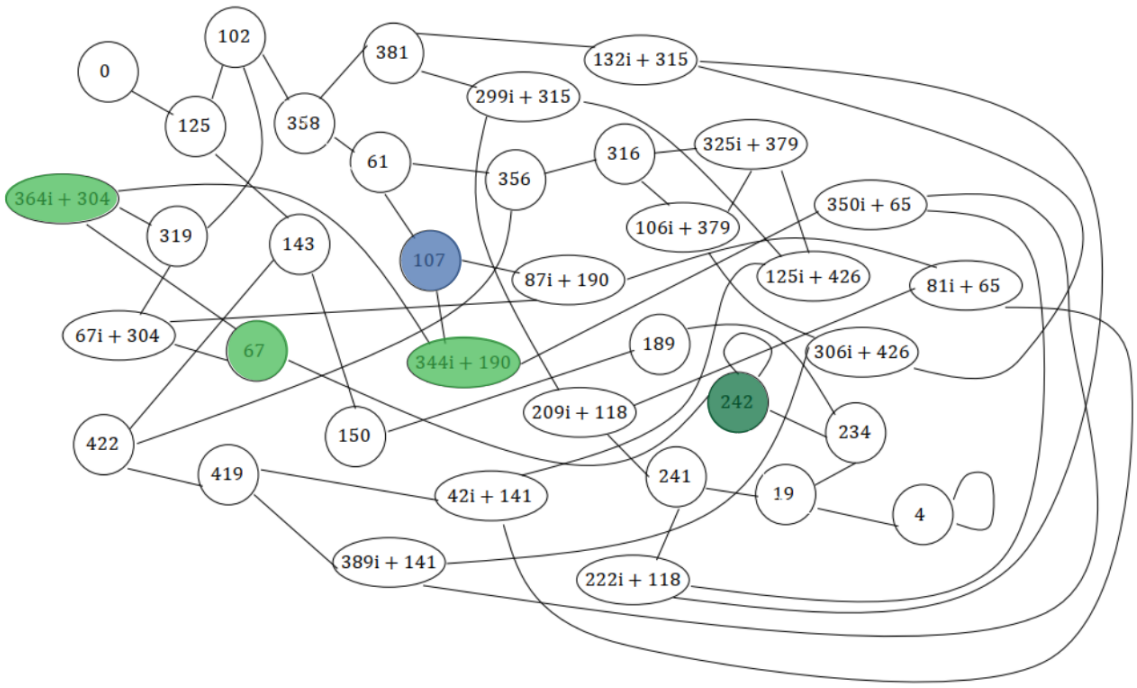


Figure 5.8: Alice's shared secret generation. The starting j-invariant is 107, and the final is 242

Then, he applies the exact same steps he did before, with the only difference that he doesn't need to compute the image of the basis points anymore. The three 3-isogenies computations are summarised as

$$\begin{array}{lll} \phi_0 : E_{a_0} \to E_{a_1} & \text{with } a_1 = 429 + 180i & \text{and } j(E_{a_1}) = 426 + 306i \\ \phi_1 : E_{a_1} \to E_{a_2} & \text{with } a_2 = 220i & \text{and } j(E_{a_2}) = 356 \\ \phi_2 : E_{a_2} \to E_{a_3} & \text{with } a_3 = 246i & \text{and } j(E_{a_3}) = 242 \end{array}$$

The shared secret he obtains is $j_{BA} = 242$.

Hence, both Alice and Bob have obtained the exact same j-invariant, namely $j(E_{AB}) = 242$.

5.5 Additional information

Everything explained up to now are the important elements necessary in order to run the SIKE protocol in a basic way. Nevertheless, there are other information one should know in order to really make the protocol efficient in the long term. The first is the main risk that was on SIDH, that is the possibility for a party to crack his partner's private key. The second presents an alternative to SIKE, based on SIDH as well, but relying on the idea of commutation of the curves, unlike SIKE.

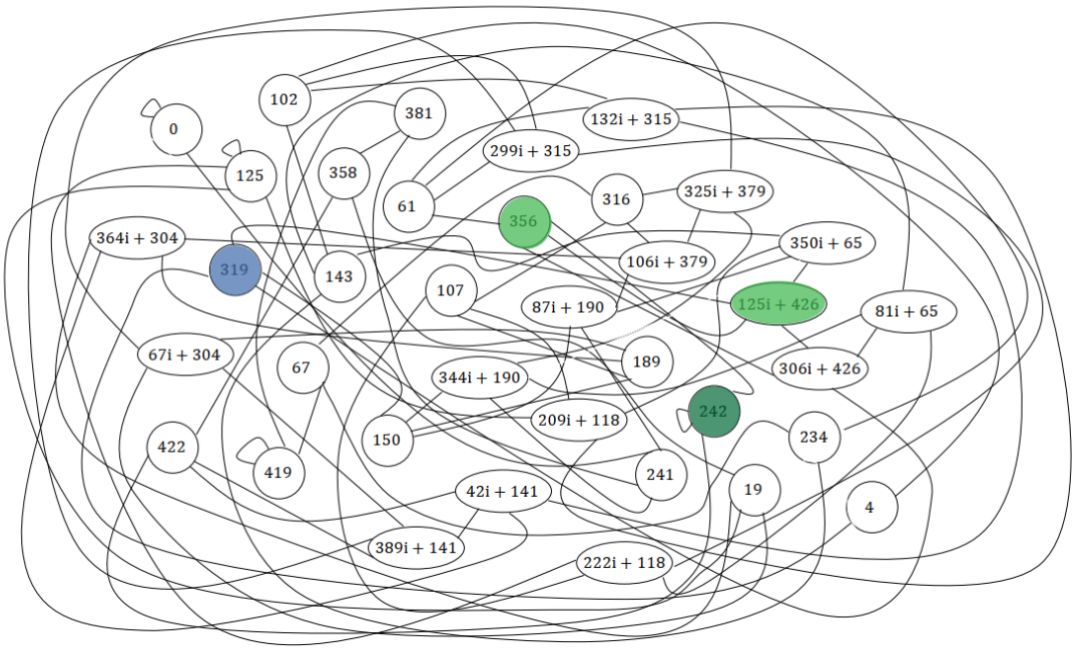


Figure 5.9: Bob's shared secret generation. The starting j-invariant is 319, and the final is 242

5.5.1 Attack

Detail of the attack

Up to now, we talked about the way SIDH was working, but without studying how much secure it was. Many papers following the original SIDH paper [JDF11] introducing the protocol in 2011 analysed it, to assess if some computations could be speed up ([ES19], [FHLOJRH17]), if the keys could be compressed ([CJL+17], [ZSP+18]), and other optimizations, but also studied the security of the model.

In 2016, Galbraith, Petit, Shani and Ti published a paper [GPST16] which brought to light a possible attack, based on the four basis points (P_A, Q_A) and (P_B, Q_B) in the public parameters, and their image points included in the public keys. Indeed, in a two-party communication, if one of the parties reuses the same secret key for multiple protocol instances, a malicious party could learn the entire secret, by performing as many interactions as the length (in bytes) of the key.

To illustrate the problem, we return to the example given in Section 5.4.2. The example comes from Costello's paper [Cos19].

As a recall, Alice key pair was (k_A, PK_A), with

$$k_A = 9$$

and

$$\begin{aligned} PK_A &= (\phi_A(E_{a_0}), \phi_A(P_B), \phi_A(Q_B)) \\ &= (430 + 355i, (408 + 155i, 1 + 366i), (367 + 122i, 358 + 189i)) \end{aligned}$$

while Bob key pair was (k_B, PK_B), with $k_B = 2$ and

$$\begin{aligned} PK_B &= (\phi_B(E_{a_0}), \phi_B(P_A), \phi_B(Q_A)) \\ &= (132 + 275i, (139 + 64i, 135 + 66i), (148 + 72i, 159 + 378i)) \end{aligned}$$

To compute her shared secret, Alice had first to get a point S_A in the basis of $E[2^{e_A}]$, computed as follow

$$\begin{aligned} S_A &= (\phi_B(P_A) + [k_A] \cdot \phi_B(Q_A)) \\ &= (139 + 64i, 135 + 66i) + [9](148 + 72i, 159 + 378i) \\ &= (55 + 86i, 326 + 44i) \end{aligned}$$

Now, let's suppose Bob is the malicious party, and he wants to know Alice's secret k_A.
Once Bob has computed his secret key, he has $PK_B = (\phi_B(E_{a_0}), \phi_B(P_A), \phi_B(Q_A))$ that he should send to Alice. But he decides instead to send his second image point $\phi_B(Q_A) = (139+64i, 135+66i)$ to another point, of order 2, in the curve $E_{76+273j}$, such as $R = (27 + 245i, 0)$. He then obtains:

$$(139 + 64i, 135 + 66i) + (27 + 245i, 0) = (219 + 198i, 279 + 379i)$$

The fake public Bob sends is thus:

$$\begin{aligned} PK'_B &= (\phi_B(E_{a_0}), \phi_B(P_A), \phi_B(Q_A) + R) \\ &= (76 + 273i, (226 + 187i, 360 + 43i), (219 + 198i, 279 + 379i)) \end{aligned}$$

After that, Bob goes on on his side normally, combining Alice's public key PK_A and his secret k_B to arrive at the shared secret j-invariant $j_B = 234$.

Receiving Bob's public key, Alice computes $S'A$:

$$\begin{aligned} S'_A &= (\phi_B(P_A) + [k_A] \cdot (\phi_B(Q_A) + R) \\ &= (226 + 187i, 360 + 43i) + [11](219 + 198i, 279 + 379i) \\ &= (125 + 120i, 167 + 4i) \end{aligned}$$

Even though the obtained point is of order 16, the point is still different from S_A, and so is the kernel computed, that is $\langle S'_A \rangle \neq \langle S_A \rangle$.
As Alice finishes the protocol, she obtains the j-invariant $j_A = 234$.

The SIDH protocol will fail since $j_A \neq j_B$, and with this information, Bob knows immediately the final bit of Alice's secret: indeed, if Alice's secret key was even, then we would have $[k_A] \cdot (\phi_B(Q_A) + R) = [k_A] \cdot \phi_B(Q_A)$

$$\begin{aligned} [k_A] \cdot (\phi_B(Q_A) + R) &= [2n] \cdot (\phi_B(Q_A) + R) \\ &= [2n] \cdot \phi_B(Q_A) + [n]([2]R) \\ &= [2n] \cdot \phi_B(Q_A) + [n]0 \\ &= [k_A] \cdot \phi_B(Q_A) \end{aligned}$$

and so Alice's would have obtained $S'_A = S_A$, meaning the protocol would have succeeded. Hence, whether the protocol succeeds or not, Bob can guess one bit of Alice's secret.

By continuing bit by bit, Bob is able to reconstructed all but the last two bits of Alice's secret, but it can be brute forced since there are only 4 possibilities left.

Solution

Confronted with this flaw, two solutions are possible:

1. We could either oblige both parties to use only one-time keys, meaning that no key should be reused
2. Or use a generic transformation to allow one of the two parties to reuse a long-term secret

Jao and al. decided that the SIKE protocol would head for the second solution, as its name indicated it (Supersingular isogeny **key encapsulation**). The idea is that Alice should not directly send a message encrypted, but rather encapsulate the plaintext with a hash of the shared secret, by XOR-ing them together. She then can send it along with her public key to Bob. When the latter receives the dack, he uses Alice's public key to derive the shared key, apply the hash function and eventually derive the message.

The pseudo-code of both encapsulation and decapsulation processes described above are given, in Algorithm 1 and Algorithm 2 respectively. Nevertheless, as one can see in the first Algorithm, at the fourth line, the usage of the XOR makes impossible to used twice the same h, that is the same shared secret, since it would be possible to derive h as soon as two messages are encrypted with this method.

Algorithm 1: Enc

Input : sk, PK, m

Output: (c_0, c_1)

1 $c_0 \leftarrow \text{compute\_pk}(sk)$;

2 $j \leftarrow \text{shared\_secret}(PK, sk)$;

3 $h \leftarrow \text{hash}(j)$;

4 $c_1 \leftarrow h \oplus m$;

Return: (c_0, c_1)

5.5.2 Alternative

SIKE is one the main instantiation of the SIDH protocol, but there is another one quite popular: CSIDH, or Commutative Supersingular Isogenies Diffie-Hellman. Unlike the former, CSIDH core

Algorithm 2: Dec

Input : $sk, (c_0, c_1)$

Output: m

1 $j \leftarrow \text{shared\_secret}(c_0, sk)$;

2 $h \leftarrow \text{hash}(j)$;

3 $m \leftarrow h \oplus c_1$;

Return: m

concept is the commutativity of the group action.

This difference with SIKE leads to a couple of other interesting aspects:

1. The commutativity removes the constraint to distinguish the initiator from the responder. Like the original Diffie-Hellman, any party can initiate, without specific preparation (except the public keys, of course)
2. The curves are defined over $\mathbb{F}_p$, instead of $\mathbb{F}_{p^2}$

The constraint to defined over $\mathbb{F}_p$ makes the protocol analog to ordinary curves protocols, in particular about the hardness to attack : as the is a commutative group action on them, a subexponential quantum attack applies, an example being Kuperberg's algorithm [CJS14]. But even with such possible attacks, the structure of CSIDH enables the possibility to lower the computation cost of the process, obtaining an improved balance efficiency against quantum security [BS20]

Nevertheless, we won't go into details about this protocol, because even though it follows SIDH idea at first, it rapidly explores new aspects, and the comparison with it gets harder. For more information, see the official paper [CLM+18a], and other publications about the subject such as a graphical illustration of CSIDH concepts [CLM+18b], and a security analysis of the protocol [BS20].

Chapter 6

SIKE and ART

With the ART and SIKE protocols, we are ready to implement a post-quantum group messaging application, by combining both. Similarly to the Signal protocol, this one can be split up into three parts, each one of them being explained.
Section 6.1 is about the initialisation phase, where the public parameters are set and the key bundle is published onto the server. In Section 6.2, we describe the method to create a group, through the construction of a tree, and how to fill it. And in Section 6.3, we present the three ways to update the tree, that are updating a key, adding a member and removing one. Eventually, 6.4 gives details about the implementation that we wrote.

6.1 Before starting

At the start of this paper, we had two objectives: creating a messaging group in an asynchronous way along with the PCS property and making it resistant to quantum algorithms. The first was resolved with the Asynchronous Ratchet protocol, while for the second one, a solution presented was the SIKE protocol. Hence, we eventually have to combine these two protocols into one.

In ART, Diffie-Hellman is used within several steps of the protocol:

- In the initialization, when the creator calculates the leaf keys: Diffie-Hellman is used three times in X3DH, with the identity keys and some ephemeral keys.
- Still in the initialization, the parents' key pair is computed with DH, based on the public and private key of their children.
- In the update of the keys: when a key leaf is modified, DH is used to re-calculate the key pair of the leaf, and the key pair of each of the nodes in its path. The other nodes also need to update some of their parents node.

We will thus try to mix SIKE with each of these steps, in order to make them quantum-resistant. The signature aspect won't be studied here, as it is not the objective of this paper.

But before all of this, there is as usual an initialisation phase, where the public parameters are set, and where each party publishes their keys on a common server. This phase is similar to the one in the Signal protocol, but nonetheless important.

6.1.1 Public parameters

As a remainder, the following parameters are public, and thus seen by every party:

- e_A and e_B: the number of 2- and 3-isogenies to compute, respectively, in order to compute a secret key and a shared secret,
- $p = 2^{e_A} \cdot 3^{e_B} - 1$, defining the field $\mathbb{F}_{p^2}$,
- A and B, defining the initial curve $E_{AB} : By^2 = x^3 + Ax^2 + x$,
- $\{(P_i, Q_i)\}_i$ a set of pair of basis points, used to compute a kernel $\langle S \rangle = \langle P_i + [k]Q_i \rangle$, based on a secret key k

In our implementation, we decide in advance two pairs of points $\{(P_A, Q_A)\}_i$ and $\{(P_B, Q_B)\}$, one for each party involved in a SIKE stage. The reason is that computing a new pair before each exchange is computationally expensive, especially when p is big. Even though two differents methods ([Pie17], [CJL+17]) were given to compute such points, the idea is the same: finding a point on the curve, multiplying it by 3^{e_B} (respectively 2^{e_A}) and testing if it becomes a point of order 2^{e_A} (respectively 3^{e_B}) in order to obtain a first point for Alice (respectively Bob). After finding a second point, one has to check that they are linearly independent. If one of these steps fails, we start over. That is why we prefer setting the pairs before starting the whole protocol.

Initiator - Responder

The objective of SIKE is to substitute Diffie-Hellman. But while DH key agreement does not depend on who initiated the exchange, SIKE needs to distinguish the party who initiates from the one who responds to it. As presented in the previous chapter, given parameters e_A and e_B, a party will have to compute either e_A 2-isogenies or e_B 3-isogenies. For simplicity, in the rest of this chapter, we will call the party in the first situation the *initiator*, and the *responder* in the second one. And as one can guess, the initiator will always be the party that initiate the SIKE protocol with the responder.
Moreover, this means that, given a private key, one can obtain two different public keys, according to if they initiate or respond to the exchange.

6.1.2 Key Bundle

Similarly to the Signal protocol, any new user registering in the messaging application has to put a key bundle first into a server, made up of an idendity key and one-time keys.
To be exact, a user only generates the "*responder-side*" public keys corresponding to either the private

identity key or the private one-time keys. The reason is simple: if Alice initiates an exchange with Bob, she will use her private key and Bob's public key. This means that Bob will act as the responder and will need to have previously pushed his "responder" public key on a server. That is why one only needs to compute the "responder" public keys and upload them onto the server, and not its "initiator" public keys. Nevertheless, it's still necessary for Alice to compute the "initiator" public key of the private key she used, and to send it to Bob.

Once the public keys are computed, their owner push them into the server in order to make them available for everyone, while they keep the corresponding private keys on its own database. It is also important to store them using id's, in order to find the corresponding public key of a private key into the server, and vice-versa.

6.2 Group creation

The creation of a group can be divided in two parts: the computation of the shared secret, or leaf key, that will be the shared secret between the creator and a member; and the computation of the key pair of each node above, the parents, until obtaining the root key. Both these parts are done in succession, but here, we explain them separately and how each party does it.

6.2.1 Leaf key

When Alice decides to create a group, she needs three things:

- the names of the future members
- their public identity key and a public one-time key, both calculated in *response* mode

She can get that by fetching the Key bundle of each party (let's say Bob for now) from the server, by a simple request. For the one-time key, it is important that she keeps in memory its id, in order to indicate to Bob which one she used.

Similarly, she has to select two of her keys: her private identity key and an private one-time key.

X3DH

In the Signal protocol, X3DH is made up of three different Diffie-Hellman operations. All of them are here replaced with SIKE, but the principle stays the same. The goal is still to obtain authentication and forward secrecy, from both Alice and Bob.

Table 3 recalls the key combination applied in order to achieve these properties, and Algorithm 3 presents the associated pseudo-code.

Secret	Alice	Bob	Main purpose
SS_1	ik_A	EK_B	Authentication of Alice, forward secrecy for Bob
SS_2	suk_A	IK_B	Authentication of Bob, forward secrecy for Alice
SS_3	suk_A	EK_B	Forward secrecy for both Alice and Bob

Table 6.1: The three shared secrets generated in X3DH (if initiated by Alice)

Algorithm 3: X3DH

Input : id_A, otk_A, ID_B, OTK_B

Output: $(ss1, ss2, ss3)$

1 $ss_1 \leftarrow$ shared_secret(id_A, OTK_B)

2 $ss_2 \leftarrow$ shared_secret(otk_A, ID_B)

3 $ss_3 \leftarrow$ shared_secret(otk_A, otk_B)

Return: (ss_1, ss_2, ss_3)

Each party then obtains three shared secrets. But the order is different according to who initiated X3DH and who was the responder.[1] Alice would get the keys in this order: SS_1, SS_2 and SS_3; while Bob would have: SS_2, SS_1 and SS_3. Thus, it is important to order the keys for Bob before applying the KDF, to be sure both parties get the same result.
Once the three shared secrets are computed, one can apply the KDF, to obtain the leaf key. Algorithm 4 gives the associated pseudo-code.

Algorithm 4: computing_leaf_key

Input : $initiator, responder$

Output: $leaf\_key$

1 $(id_A, otk_A) \leftarrow$ fetch_keys$(initiator,$'$private$')

2 $(ID_B, OTK_B) \leftarrow$ fetch_keys$(responder,$'$public$')

3 $(ss_1, ss_2, ss_3) \leftarrow$ X3DH$(id_A, otk_A, ID_B, OTK_B)$

4 $leaf\_key \leftarrow$ KDF$(ss_1||ss_2||ss_3)$

Return: $leaf\_key$

[1] See Section 2.3

Broadcast the information

Once Alice (the creator of the tree) has computed the shared secret key of Bob, she can calculate his public key (we will see in the next section whether it is in *initiator* or *responder*), and send it to him, along with the index of Bob's public OTK used (EK_B), and the corresponding public key of her OTK (ik_A). When Bob receives these informations, he re-computes the shared secret, calculate the public key, and verify that it is the same to that one sent by Alice.

If Bob arrives to the same result than Alice, the process can go on.

6.2.2 Root key

As a recall, the root key, also designated as the tree key, corresponds to the public key of the root of the tree. With the information computed by Alice, Bob, and the other members, each one of them can calculate that key.

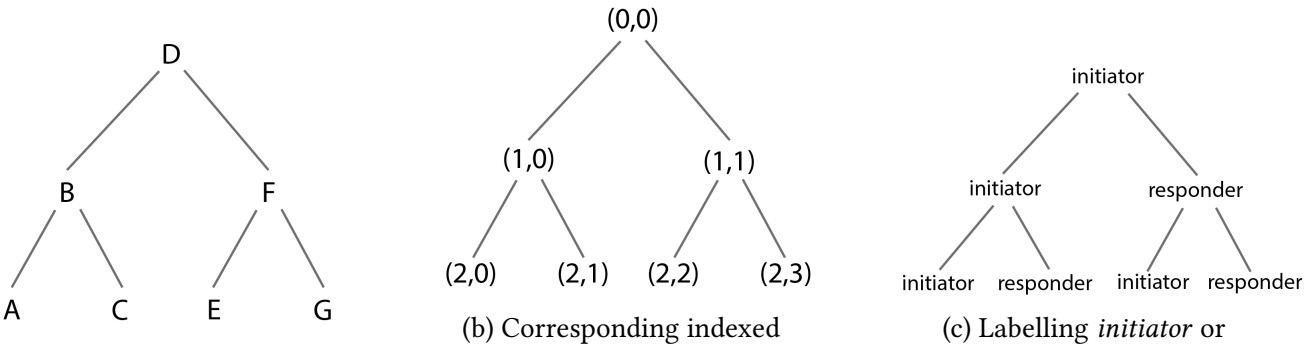
Computing the public keys

Once Alice obtained the private and public keys of every leaf, she is ready to compute the key pair of all the nodes above. For her parent, she takes her private key and the public key of her sibling (the second child of her parent), and computes the shared secret, which correspond the her parent's private key. And with this latter key, she can compute the corresponding public key. Nevertheless, before doing that, there is one concern to resolve: we said earlier that a private key was linked to two public keys: an initiator public key and a responder one. So, for Alice's parent, which one should we calculate ?

To answer to this question, we need to agree on a method so that every member of the tree has the same view about this.
A binary tree is easy to index, by using two integers x and y: x corresponds to the depth where the node is, while y is the position among the other nodes at the same depth. Both indexes start at 0. An example is given Figure 6.1.

Each node knows its index, and the indexes of the other ones.
Then, for a given node and its index (x, y), if y is even (meaning $y \equiv 0[2]$), then we decide that this node will compute the *initiator* public keys. On the contrary, if y is odd ($y \equiv 1[2]$), then it will be the *responder* public key. To go back to the previous example, this means we would have the attribution of the "side" (initiator or responder) given in Figure 6.1 (c). That is how we know which kind of public key Alice computes for her parent, given the private key.

Since the creator knows the private key for each leaves, they can do the same for each parent of these leaves. They can then go on, until computing the key pair of the root, and thus obtaining the root key. Eventually, by combining it with the stage key, they obtain the group key, used for encryption.



(a) Left-balanced binary tree

(b) Corresponding indexed tree

(c) Labelling *initiator* or *responder*

Figure 6.1: Indexation of a tree. The "even" nodes correspond to 2-isogenies, and the "odd" nodes represent the 3-isogenies

Algorithm 5 gives the method to compute the key pair of all the nodes. The idea is to iteratively compute each node, starting from the root and applying the function to both children, until we reach a leaf. Once we are at a leaf, we put the leaf key computed previously and calculate the corresponding public key. Then, we go up to the parent, use the private key of one of the children and the public key of the other one, in order to compute the key pair, and so on until coming back to the root and computing the root key pair.

Algorithm 5: compute_nodes_keys

Input : $node, leaf\_keys$

1 **if** $node.is\_leaf$ **then**
2 $\quad node.private\_key \leftarrow leaf\_keys[node.name]$;
3 $\quad node.public\_key \leftarrow$ compute_pk$(node.private_key, node.side)$;
4 **else**
5 $\quad$ compute_nodes_keys$(node.left_child)$;
6 $\quad$ compute_nodes_keys$(node.right_child)$;
7
8 $\quad (sk, PK) \leftarrow$ get_children_keys$(node)$;
9 $\quad node.private\_key \leftarrow$ shared_secret(sk, PK) ;
10 $\quad node.public\_key \leftarrow$ compute_pk$(node.private_key, node.side)$;
11 **end**

Broadcasting the keys

Once Alice computed all the key pair for each node, she sends the tree she obtained, but removes beforehand every private keys on the tree, not to let the other members knowing the secret of one another. Along with the tree is present all the information concerning the group created, such as the name of the group, the member, their position in the tree (the index), and so on...
The RFC[2] of ART [BMO+19] suggested in Section 5.5 to encrypt these informations with the public key of each member, but we decided not to do that, because if an attacker is able to derive the private key from the public keys that was broadcasted, then they are able to break the SIDH protocol, and thus they are also able to decrypt the encrypted public keys, meaning that the encryption does not add more protection or security on the keys.

With these informations, the members receiving them can reproduce the creation of the tree. For instance, Bob will take the public keys sent by Alice, by only taking the one of the nodes situated in his copath. He shouldn't take the one of the nodes in his path since he must be able to compute them himself (along the private key). Nevertheless, he can check if each public key he computes is the same to the one sent by Alice, for a given parent.

Similarly to Alice, once Bob gets the private key of a parent, he needs to know the "side" of this node (initiator or responder) in order to compute the correct public key. If every step is correctly done, Bob obtains the same root key than Alice sent him in the tree.

The very last step for the creation of the tree for Bob is the necessity to immediately update his private key (method detailed in the following section), to secure the possibility that Alice did not get rid of it. And with this new leaf key, Bob has to re-compute all the keys in its path, before sending his update notification to the other members.

6.3 Group update

We have presented in Chapter 4 the different methods to update an Asynchronous Ratchet Tree, namely updating a key, adding a member and removing one. We develop here each of these processes, in the context of using SIKE. These explanations could all the same have been given in the fourth chapter, but the two main papers developping ART ([BMO+19] and [CGCG+18]) did not give any precise instruction, so we were left with the liberty to decide the way to do it (in particular concerning the addition and the removal of a member). Obviously, this means that other techniques could have applied in this situation, but the ones used here seem the most appropriate.

6.3.1 Key updating

Among the three methods resulting in the update of the tree, the update of a leaf key is central. It is by this process that Post-Compromise security is achieved.

[2] Request For Comment

When a member of a tree, let's say Bob, decides to update his leaf key, he changes the private key of his corresponding leaf. This modification leads to the update of the keys in his path. After his private key is updated, Bob has to compute his new public key, according to his side (initiator or responder). When it is done, Bob takes the public key of his sibling and computes the shared key (or private key) of his parent, and then the public key of the latter, and so on until the root. A new root key is obtained, and by combining it with the previous stage key, a new stage key appears. At the end of this process, Bob can gather all the public keys in his path, and broadcast them to every other members.

Algorithm 6 presents the pseudo-code of the process.

Algorithm 6: update_key

Input : $leaf, new\_key$

1 $leaf.private\_key \leftarrow new\_key$

2 $leaf.public\_key \leftarrow$ compute_pk$(leaf.private_key, node.side)$

3 $path \leftarrow$ get_path$(node)$

4 **foreach** $node \in path$ **do**

5 $\quad (sk, PK) \leftarrow$ get_children_keys$(node)$

6 $\quad node.private\_key \leftarrow$ shared_secret(sk, PK)

7 $\quad node.public\_key \leftarrow$ compute_pk$(node.private_key, node.side)$

8 **end**

Remark: even though the root of a tree is in every path given any node, it should not be included in the keys Bob broadcasts, because anyone receiving it could just take this key, and not do the computation to update their view of the tree, which is necessary.

When Charlie receives these informations, he updates the public key of each node mentioned in Bob's broadcast. Eventually, with these new values, Charlie just needs to update the key of each node in his path in order to obtain the same root key and thus, the same stage key.

A way to be more efficient in the update is for Charlie to only select the node (among the list of public keys sent by Bob) that is in Charlie's copath. Indeed, by definition, if a node is in the copath of a leaf, then its sibling is in the path, and so is the parent of these two nodes. So, among all the nodes in common in Charlie's path and Bob's path (the list of nodes Bob sent, to be exact), it is sufficient to keep only the lowest[3], and update it in Charlie's tree. The rest of the nodes will be updated once Charlie re-computes the key pair of the node in his path. Nevertheless, these remaining nodes could still be used to compare with Charlie's result, to make sure the calculations are correct (only the root key is not compared, since it is not broadcasted by Bob, for security reasons).

[3]the lowest node is the one whose the x-index is the biggest, meaning the node is the deepest, or at least one of the deepest

6.3.2 Adding a member

We detail here the process to add a member into the tree. We consider that one person is added at a time. Let's say that Alice wants to add a new member into the group. Here, Alice doesn't need to have been the creator of the group to do this.

The process can be decomposed into three steps:

1. Adding a new node to the tree
2. Computing the key
3. Informing the other members

New member in the tree

At each group is associated a tree T, meaning that adding a member into a group is made through inserting a leaf whithin the corresponding tree. To do so, it is necessary to know the position of this future leaf. If T is composed of L leaves, corresponding to L members. The number of nodes is then

$$N = 2L - 1$$

This means that the new leaf will be the $N + 1 = 2L = N'$-th node of the tree. Its index will then be:

$$x = \lfloor log_2(N') \rfloor \quad , \quad y = N' - 2^x$$

Once this information is known, the tree is updated, in order to add this new node, and link it to its parent. This parent can be found through the indexes: if (x, y) is the index of the new leaf, then the node acting as its parent has the index $(x-1, \lfloor \frac{y-1}{2} \rfloor)$. But since we are in a binary tree, the new leaf also needs a sibling. And this sibling we are looking for is simply the leaf's parent. Once the former parent becomes a leaf, one needs to create a new node, to play the role of new parent (parent of the new leaf and its sibling).

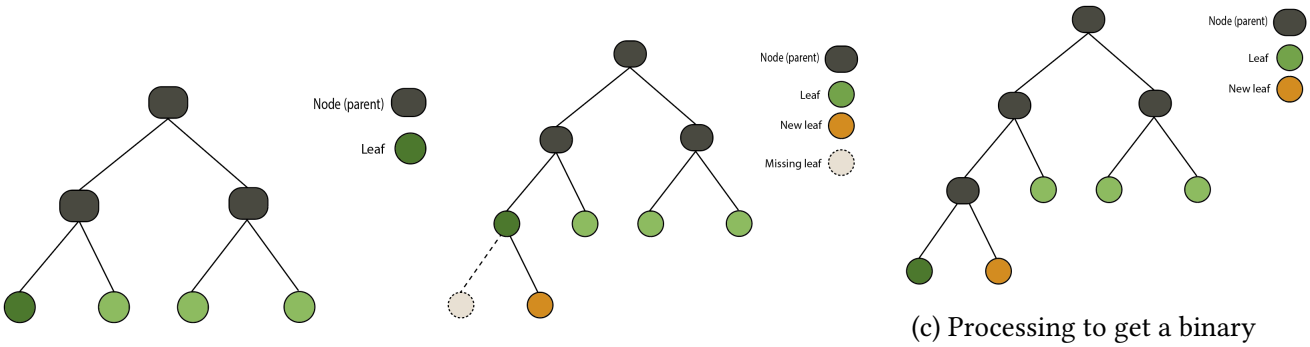
Once all of these nodes are added or moved, the entity in charge of this process has to update their side. A node which had the role of *initiator* might become *responder*, and vice-versa.

Figure 6.2 illustrates the operation of adding a member:

(a) The tree T is made up of $L = 4$ leaves, meaning there are $N = 2 \cdot L - 1 = 9$ nodes. Then, the new leaf is be the 10th, and its index is $(3, 1)$

(b) The new leaf is added, but to have a binary tree, a sibling is necessary at the index $(3, 0)$.

(c) The parent of the new leaf is designated as the sibling, and a parent node is created and put into the tree



(a) Existing tree (b) Adding a new leaf (c) Processing to get a binary tree

Figure 6.2: Insertion of a new leaf in a tree

Algorithm 7: add_member

Input : $tree, new\_member, creator$

Output: new_tree

1 $leaf_key \leftarrow$computing_leaf_key$(creator, new_member)$

2 $new\_tree \leftarrow$ add_node$(tree, new_member)$

3 $leaf \leftarrow$ get_node$(tree, new_member)$

4 update_key$(leaf, leaf_key)$

5 **return** new_tree

Computing the key

This step is very similar to the creation of the tree: Alice calculates a shared key between her and Zoe, that will be used as a starting leaf key, then she computes the key pairs onto Zoe's path, and finally Alice sends her a copy of the tree, in order to allow Zoe to create a similar tree. When Zoe receives these informations, she can create the tree (its structure), then she compute the shared key, check with the one Alice sent her, and finally calculates the private and public keys of the nodes in her path, by using the public keys on her copath.

Eventually, she modifies her leaf key, to secure her private information, and recompute the root key. Of course, after updating her key, she has to notify everybody else.

Inform the other members

What about the other members of the group?
In the same way as the creation of the tree, Alice then broadcasts the information to the other members so that they can apply these modifications onto their tree. These information is the name of the new member, their position, and the public keys onto their path (except the root key).

Algorithm 8: add_node

Input : $tree, new\_member$

Output: new_tree

$N \leftarrow$ length(get_leaves($tree$)) + 1

$x \leftarrow \lfloor \log_2(n) \rfloor$

$y = N - 2^x$

$new\_index \leftarrow [x, y]$

$index\_sibling \leftarrow [x - 1, \frac{y-1}{2}]$

$new\_sibling \leftarrow$ get_node($tree, index\_sibling$)

$new\_leaf \leftarrow$ create_node(new_member)

$new\_parent \leftarrow$ create_node($index\_sibling$)

$new\_parent.left\_child \leftarrow new\_leaf$

$new\_parent.right\_child \leftarrow new\_sibling$

$new\_tree \leftarrow tree.append(new\_parent)$

Return: new_tree

When Zoe comes online, creates the tree on her side and updates her leaf key, these other members will be informed of this modification, and will update the root key, once again.

6.3.3 Remove a member

The process of removing a user is easier. By taking out a member, we also remove their corresponding leaf in the tree. And to avoid their sibling to stay alone (which would result in a non-binary tree), this latter node takes the place of their parent. And by taking the place, we mean taking their index, their side, and in particular its key pairs.

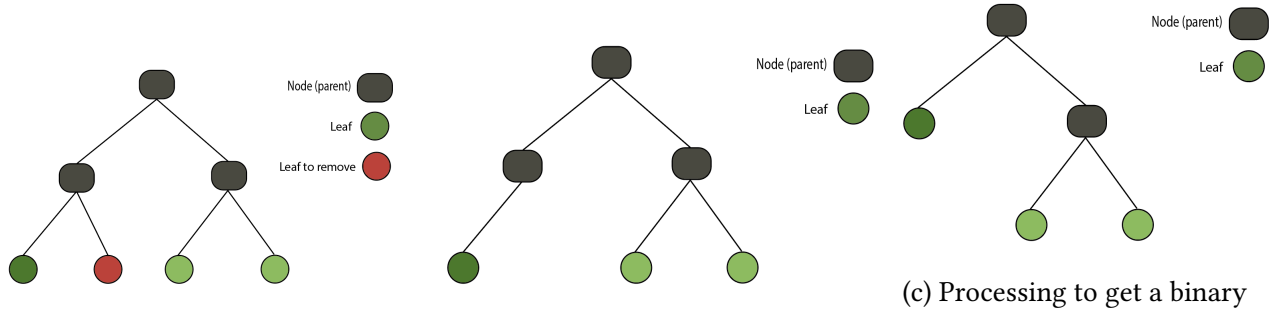
Indeed, if the sibling keeps their keys, everybody will have to wait until the sibling re-computes the keys in their node and broadcast it to the other members, because the party who removed the member does not possess enough information to calculate the keys in the sibling's path.

That is why, instead, the sibling inherits the key pair of their parent. Nevertheless, it is still important for the initiator of the process to immediately update their key, so that the removed member won't be able to read the communications anymore.

Figure 6.3 illustrates the process:

1. (a) Let's say that it is the node at position $(2, 1)$ that is going to be removed.
2. (b) After the separation, its sibling at $(2, 0)$ is left alone. This is not a binary tree anymore.

3. (c) To solve this, the sibling takes the place if its parent



(a) Existing group (b) Removing an existing leaf (c) Processing to get a binary tree

Figure 6.3: Removal of a leaf, located at the index (2, 1)

The algorithm for removing a member was not given, since it is only about deleting a leaf from the tree structure.

6.4 Implementation

After giving the theoretical details about a combination of the protocols ART and SIKE, a good way to demonstrate its feasibility is to implement it, as a proof of work.

In this part, we provide the details about the implementation, the choices of the parameters and since it is certainly not a perfect execution, we will see what can be improved. Moreover, as one will have seen in the previous sections, more liberty was taken about the protocol ART, since SIKE is already well-defined.

6.4.1 Programming language

The language used in this proof of concept is Python. This choice has several reasons:

1. A fast implementation possible
2. A lot of mathematical libraries available freely
3. The possibility to work with HTML pages
4. The framework Flask

In other words, Python enables the possibility to both make mathematical computations easily done (for the SIKE aspect) and directly use the Internet (for the messaging part), through the web framework Flask. The latter allows to make HTTP requests, useful for sending or receiving the messages, but also for navigating between web pages, that act as the user interface.

And, obviously, the implementation will follow the steps given in the previous chapter.

6.4.2 Public parameters

Before starting, the public parameters need to be chosen, and as indicated before, most of them will be constant throughout the protocol.

Prime number p

The very first element to set is the prime number p, since it defines F_{p^2}, meaning the curves, their points, the j-invariant and the isogenies.

p has two conditions to respect:

1. Being equal to 3 mod 4, ie. $p \simeq 3[4]$
2. Being written $p = 2^{e_A}3^{e_B} - 1$

The first condition is fortunately a consequence of the second if $e_A \geq 2$, which should always be the case. So, one can simply chose two integers e_A and e_B, greater than 1, to set p, as long as $2^{e_A}3^{e_B} - 1$ is prime of course ($2^5 3^5 - 1 = 7775$ is not, for example).

It was decided to set $p = 2^{15}3^8 - 1$, which is around 27-bits long. The reason is that the following prime number in the form of $2^{e_A}3^{e_B} - 1$ with $e_A > e_B$ is $p = 2^{13}3^{12} - 1$, which is slightly higher than 2^{32}, meaning that multiplying two numbers of 32-bits would give us a 64+ bits number, more than what Python can handle at best, causing rounded number and thus errors.
Nevertheless, there exist libraries that help to go over this limitation, like GMPpy, but we won't use it, since the goal of this implementation is simply a proof of work.

Starting curve E_A

Once p has been set, one has to choose a starting elliptic curve $E_A : y^2 = x^3 + Ax^2 + x$, through the choice of A as long as E_A is supersingular. There is no method to deduce a correct A given p, so it is necessary to test multiple A until the condition on supersingular curves given by the following theorem is met.

Theorem 3 *Let E/k be an elliptic curve of a field k. Then, we have:*

$$\begin{array}{rcl} E \text{ is ordinary} & \Leftrightarrow & E[p] \simeq \mathbb{Z}/p\mathbb{Z} \\ E \text{ is supersingular} & \Leftrightarrow & E[p] \simeq \{0\} \end{array}$$

Proof:
See [Sut15b], Section 7.1, and [Sut15a], Section 14.

As a recall, $E[p]$ represents the kernel of the multiplication by p. This means, that given any point Q not null in E_A, if $[p]Q = \mathcal{O}$, then E_A is not supersingular. But $[p]Q \neq \mathcal{O}$ doesn't mean that

E_A is supersingular, since this condition needs to be verified for *every* points in the curve, which obviously would be too long.

Instead, the original SIKE paper proposes to use as starting curve $E_A : y^2 = x^3 + 6x^2 + x$, ie. taking $A = 6$ regardless the value of p. ([ACC+17], Section 1.3.2).
This parameter will evolve throughout the process.

Public generator points

The last thing to know before being able to use SIKE is the public generator points pairs (P_A, Q_A) and (P_B, Q_B), necessary to compute the kernels of $E[2^{e_A}]$ and $E[3^{e_B}]$.

Once again, there is no formula to directly find correct points. And there are no default values that could be used, regardless of p, unlike with $A = 6$. One has to generate points P_A and Q_A of order 2^{e_A}, and linearly independent as well (respectively (P_B, Q_B) of order 3^{e_B} and independent).

As indicated earlier, two methods were given ([Pie17] and [CJL+17]) to do so, but both are essentially the same:

1. Find a point P in the curve E_A
   (a) Choose $x \in F_{p^2}$
   (b) Compute $x^3 + Ax^2 + x$
   (c) If it exists, find a square root y of the previous result
   (d) If not, start again
   (e) If found, the point is then $P = (x, y)$
2. Multiply P by 3^{e_B} (respectively 2^{e_A}), to obtain a point Q
3. Check the order of Q
   (a) If Q is of order 2^{e_A} (respectively 3^{e_B}), then output Q
   (b) Otherwise, start over the whole process

This process must executed done twice, in order to find P and Q of order 2^{e_A} (respectively 3^{e_B}). According to Pierrakea, after multiplying by 3^{e_B} in step, P should be of order 2^{e_A} "with high probability" ([Pie17], Section 3.4), even though no proof or indication was given to show this statement. Obviously, the bigger p is, the longer it takes to generate P and Q.

An improvement was brought Costello, Jao and Longa in [CJL+17], to compute more efficiently the first step (obtaining a point in the curve), but the problem in steps 2 and 3 stays the same.

This is why it was decided to set (P_A, Q_A) and (P_B, Q_B) as constant value, and not re-compute them for each shared secret.

Key Derivation Function

When a tree is created, we said that the stage key is a combination of the public key of the tree combined with the previous stage key, by applying a KDF. We decided to set this function as SHA256, in order to use then the result with AES256. The level of security of such a hash function should be enough, although against quantum algorithms, it may be useful to later swap it with SHA512.

Sending message

To send a message to a group, the user just needs to take the root key, combine it with the previous stage key to obtain the new stage key, pass it through the KDF, and eventually encrypt the plaintext with AES.
For more security, we also add the following elements each time:

1. a nonce: random value, used each time the encryption function takes the same key as input, like when sending a group message,
2. a salt: also random value, used for each run of an encryption, so that it makes impossible for an attacker to use precomputed hashes in an attempt to decrypt the cipher,
3. an authentication tag: value used to authenticate the ciphertext when using AES in GCM mode; it also ensures that nobody can change the data without going undetected.

6.4.3 Possible improvements

As stated in at the beginning of this section, a couple of aspects could be improved.

The first element is the overall optimization of the protocol. The implementation we made follows the details given in Chapter 4, but it is possible to make it more efficient.
As one can see when looking at each isogenies presented in this latter chapter, the x-coordinate image of each isogeny shown only depends on A and on the x-value of the antecedent. That is one of the big advantage of the Montgomery curves. Moreover, even though it was presented in this paper, it is also possible to apply the addition only through x-coordinate.
In the end, the possibility to compute every operation only with A and the x-coordinate enables to only calculate the x-value of the image, since the y-value becomes not necessary. Thus, each computation is at least twice faster. And we say "at least", because with the optimized implementation presented in [ACC+17], computations are even faster.

Secondly, another way to optimize each function is by using the language C (or C++) instead of Python, the former being known to be much faster, in particular for these kinds of operations.

A third possibility is to use external libraries that help to handle large numbers, in particular higher than 64 bits. In their implementation in C, Jao, Costello and De Feo used GMP [gmp21], a free library for arbitrary-precision arithmetic, in order to go until $p \sim 500$-bits.

Chapter 7

Protocol analysis

In the previous Chapter, we showed how an implementation of a combination between ART and SIKE could be done. In this Chapter, we analyse this same implementation, mainly through two aspects: the computational impact of changing Diffie-Hellman with SIKE in the whole ART protocol, and the feasibility of the app. Obviously, the goal is to show, on one hand, that either with our implementation or the official implementation of SIKE, the impact is minimum compared to the original protocol, and in other hand that the app could be deployed and used as any other messaging app already existing.
Hence, Section 7.1 is about the computational difference between Diffie-Hellman and SIKE, while Section 7.2 goes more into details about testing the three parts in the implementation, and seeing if each one of them could be made by any user, on a basic device.

7.1 Computational impact

The first aspect to measure is: what is the cost of swapping Diffie-Hellman with SIKE, for the authentication and the computing of secret keys. Ideally, such exchange and the waiting time should not be perceived by a user, meaning the app should be as efficient and smooth as any messaging app based on a tree structure like ART.

7.1.1 What to measure

As indicated, in this section, we want to see the impact of using SIKE instead of Diffie-Hellman. But for now, we only measure the time it takes to compute public and shared keys, and leave for later the measure of the computations related to ART.

The protocol ART+SIKE could be divided into several parts, such as initialisation, tree creation / updates and sending messages. In each of these steps, a computation of either a shared secret or a public key (or several) is necessary. But it is not necessary to make the tests for every parts, and so for two reasons:

- Computing a shared key with DH or SIKE is similar to computing a public key: the same operations, in the same order, with the only difference of the basis points P and Q
- The method to calculate the keys in each part does not change through the time (if we use SIDH once, we use it for the rest of the protocol)

That is why it enough to make these selected measurements in order to have a global view of the impact.

The test that will be done in the following parts are computing one public key, and computing 102 key pairs.

7.1.2 Characteristics of the machine

Before doing any test on the Key Agreement protocols, we describe on which device these tests were made.

Since the app was developed in a computer, all the examinations are made into a computer as well.

The machine used for testing has the following specifications:

- Type: Acer Aspire VN7-791G
- Operation system: Windows 10
- RAM: 8 GB
- Processor: Intel Core i7-4720 CPU @ 2.60 GHz, dual core

For more concrete data, the measurements of the tests made for the reference and the optimized implementation from [ACC+17] (by Costello, Jao and Feo) are added. Nevertheless, these values should be consider with careful attention, due to the many differences with our implementation (language, additional security algorithm, architecture, ...)

Only one core was used for the tests, and $2.6 \cdot 10^9$ CPU cycles take one second. The measurements given of the following tables are the average number of CPU cycles, the standard deviation and the mean time in seconds it took, with 100 executions each time. Nevertheless, only the CPU cycles should be considered, since the time depends on the processor, different between the machines. It was still decided to indicate the time it took to give an idea of how long each step takes.

7.1.3 Tests and comparison

For simplicity, it was decided to divide the ART+SIKE protocol into three parts for the rest of the analysis: initialisation, tree operations (creation and updates), and sending messages. Each one of them is tested in the following parts.

Initialisation

In this phase, a new user has registered in the messaging application. As indicated, the very first thing to do then is generating one-time key pairs. The new user must make available 100 public keys, corresponding to the 100 private one-time keys stored, secretly. Since these public keys are in *responder* mode, the tests of the generation for these keys are in the same way, and not in the *initiator* mode.

The tests for sike27 can be found on the Github folder [Riv21], while the measures for both SIKE434 come from [ACC+17], and the ones for ECDH from [Dui19]. It should be noted that even though it was possible to take the measures of SIKE751, we preferred to stay with p the closest to 27-bits, in order to still be able to compare with sike27, and SIKE434 was the smallest choice of p given. Moreover, here, ECDH corresponds to Curve25519, and the "optimized" SIKE implementation is actually the optimized implementation along with an additional x64 assembly implementation, to optimize every calculation.

Table 7.1 shows the number of CPU cycles and the time it takes to compute a key pair, while Table 7.2 is about the computation of 100 private and public keys.
Only the value for sike27 is different (other than proportionally), since it was the only test that could be correctly done.

Algorithm	Mean CPU cycles	Variance	Time	Factor
ECDH	166	49	0.000064	1
sike27	18,654	646	0.0071	110
SIKE434 (ref)	1,047,991	n/a	0.403	6296
SIKE434 (opt)	6,487	n/a	0.0024	38

Table 7.1: Performance (in thousands of cycles) and the time (in second) it takes for the computation of one public key

Algorithm	Mean CPU cycles	Variance	Time	Factor
ECDH	16,656	4,989	0.0064	1
sike27	1,865,481	64,600	0.71	110
SIKE434 (ref)	104,799,100	n/a	40.3	6296
SIKE434 (opt)	648,700	n/a	0.24	38

Table 7.2: Performance (in thousands of cycles) and the time (in second) it takes for the computation of 100 private keys and 100 corresponding public keys

The first and most obvious observation that one can make is that Elliptic-Curve Diffie-Hellman is the fastest protocol. We then see that the optimized SIKE434 is the second fastest, and even though

it is 38 times slower than ECDH, SIKE434 optimized still gives an acceptable time to compute a hundred of keys through SIDH.
Then comes our implementation with 27-bits p, and eventually the SIKE434 with the reference implementation. The difference of factors between these two protocols may seem important, but it is important to recall that sike27 has p with 27 bits, whereas SIKE434 has 434-bits p. And with a non-optimized method, it should not be a surprise to see such a slow implementation when p is so high.

From these measurements, an implementation of the ART protocol should not cause any problem if it with Elliptic Curve DH, because the speed of computation is the last thing to worry about. But with SIKE, one has to take more precautions: with the reference implementation, the delay would be too significant, and make the protocol unusable. If one decides to decrease the value of p in order to speed up the process, an acceptable time is reachable, but it comes at the price of the security. So, the only viable solution is the optimized implementation, using an x64 assembly structure.

Overall, with the optimized implementation of SIKE, it is possible to swap Diffie-Hellman with SIDH without suffering strong delay effects.

But what about the whole ART+SIKE protocol ?

7.2 Faisability

This section is dedicated to the study of the feasibility of such a messaging application. The analysis is made through two devices: a computer, since the implementation was totally written in Python, and a smartphone, because messaging applications are mostly used on phones. What we take into consideration here is in first the time each step of the protocol takes, and then the size of the key. We could also study the bandwidth consumption, but the format of details sent to the server is different according to the step applied, which makes any measurement uneasy and uncertain.
For both these criteria, a threshold is given to indicate what an average user is ready to accept. This methodology for the feasibility analysis is inspired from the one made in [Dui19]. And as a recall, the main steps for each steps will be stated, to give an idea about the time and the number of CPU cycles one could except.

7.2.1 Real-life scenario

Average user

If correctly done, this messaging app should be used by an average user in a satisfying way for them. But what is in this situation an average user ?

In January 2021, the number one messaging app was Whatsapp, with over 2 billion users worldwide. The main paper describing the behaviour of Whatsapps users was made by Rosenfeld and al.

[RSS+18]. The information to retain are that an average user sends 145 messages a day and is member of 64 groups. Among these 145 messages, 38 are sent while 107 are received. This difference is due to the fact that people are often in group chats. And among the 64 groups, over 70% (71.5%) are two-person groups, 11.4% are groups with 3 to 5 people, 6.9% for 6-10 people groups, and the rest for 10+ persons. We will consider for the rest of the chapter that a group is made up of 10 members.

Average threshold

As shown in the Computational impact section, changing ECDH with SIKE will make each computation slower, by a certain factor. A user can accept this difference in time, as long as it is not too important.

A first threshold to consider is the acceptable waiting time for an average user, when a step from protocol is done. According to Nielson in *Website Respond Times* [Nie10], a delay of 0.1 second gives the feeling of instantaneous response, and a delay of 1 second felt by the user, but not as an annoying delay. But even if a delay between 1 and 10 seconds still keeps the user's attention, a feeling of annoyance and clear delay will be present.
We thus aim at staying under 10 seconds for the initialisation (when a member uses for the very first time the app, by signing up) and under 1 second for the tree updates and the messages conversation.

And from what we have seen in the previous section, we can already forget the reference implementation of SIKE434. The three Key Agreement Protocol tested are ECDH (Curve25519), sike27 and SIKE434. But the fact that the optimized implementation of SIKE434 is in C, and using C libraries, we cannot plug-and-play it, like with ECDH. Instead, we use ECDH for the key computation and add a delay, obtained from the difference of time between these two protocols.

Average device

Now that we know what is an average user, we have to think about the device they might use.
To choose the correct smartphone, we select several phones according to their popularity, and that could be classified into two categories: the high and low value smartphone, high and low being defines by the prices and characteristics. They are presented in Table 7.3. The three first phones are the high-end models, the best selling of 2019 (one per brand), while the tree last could be qualified as affordable phones, under 200 $. All the models presented are the basic ones.

Hence, the device used for the feasibility into a smartphone will have the lowest characteristics between each smartphone presented, ie. 16 GB of storage, a CPU of 1.7 GHz and 2 GB of RAM (which, by the way, corresponds to the Archos model). By doing so, we can expect that if the message app is feasible into this device, it should be feasible for most other smartphones as well.

Without going into detail but based on the same reasoning, an entry-level model for computers that are less than 5 years-old has the following basic specifications: 64 GB of memory, 2.0 GHz and 4 GB of RAM.

We are now ready to test the feasibility of the app, according to the key agreement protocol used.

Phone name	Storage (GB)	CPU (GHz)	RAM (GB)	Price
Samsung Galaxy S10	128	2.73	6	$899
Apple iPhone 11	64	2.65	4	$699
Huawei P30 Pro	128	2.6	8	$609
Xiaomi Redmi Note 9	64	2.3	4	$189
Honor 8A	32	2.0	2	$139
Archos Access 50	16	1.7	2	$119

Table 7.3: Common types of smartphone used, ranked by price, along with their main features

7.2.2 Initialisation

Immediately after a user installed the application, the following steps are set up:

1. A private identity key ik is created
2. The corresponding public keys IK are computed (*initiator* and *responder*)
3. 100 one-time key pairs (otk_i, OTK_i) are computed
4. All the public keys are sent to the server while the private keys are stored in a database

This part is finished when the client receives from the server a confirmation of success.

In Table 7.4, we can observe how many CPU cycles and time is needed for this process.

Algorithm	**Mean CPU cycles**	**Variance**	**Time (Phone)**	**Time (Computer)**	**Factor**
ECDH	4,243,109	591,595	4.99	4.24	1
SIKE434	5,181,895	397,899	6.09	5.18	1.22
sike27	5,526,300	495,179	6.50	5.52	1.3

Table 7.4: Performance (in thousands of cycles) and time (in second) necessary for the initialisation phase.

We see that overall, it takes around 6 seconds to register in the app, compute the 102 keys and store them into the databases, on a phone, and around 5 seconds on a computer. Moreover, the times between the protocol are very close: only a factor of 1.3 between the slowest and the quickest algorithm. This light difference is observed on all the other measures. The reason is simply that the majority of the time spent on the phase is spent when the keys are sent to the server. This step takes more time than the others because the client has to wait for an answer from the server, to state that the keys are correctly received and stored, even though the keys are sent all at once, in a single request (instead of 102 requests). The difference in CPU cycles is then due to the protocol

to compute the keys. As seen in Section 7.1.3, computing 102 keys takes less than a second in the worst case (reference implementation of SIKE434 excepted).

These measures show that the threshold of 10 seconds, where a user could become impatient, is not reached, meaning that the initialisation has an acceptable delay. Besides, this operation is done once when the user signs up in the messaging app database, and then from time to time when the number of one-time key runs out.

7.2.3 Creation of the tree

The process for creating a tree takes as input a list of names and a name for the group, and follows these steps:

1. Create the tree structure (nodes, relations parent-children, ...)
2. For each member, calculate the leaf key pair:
   (a) Fetch the public ID key and a public OTK key
   (b) Compute the three shared key, with X3DH
   (c) Apply the KDF function
   (d) Compute the corresponding public key
3. Calculate the key pair of the parents, and all the nodes above
4. Calculate the stage key
5. Send the details to each members (group name, leaf key, tree structure, ...)

Once a member receives the details, they simply take the tree, compute their leaf key, and compute all the key pairs in their path. To authenticate the creator, the member must check the obtained keys are equal. Eventually, the new member of the group has to update their key, to ensure the creator doesn't know the previous key anymore.
But these tests are not done here, since it requires less operations, and thus is faster than the creation of the group.

With Table 7.4 we can observe how many CPU cycles and time is needed for this process.

Algorithm	Mean CPU cycles	Variance	Time (Phone)	Time (Computer)	Factor
ECDH	1,721,263	126,195	2.02	1.72	1
SIKE434	2,362,095	452,539	2.77	2.36	1.37
sike27	2,516,805	176,814	2.96	2.51	1.46

Table 7.5: Performance (in thousands of cycles) and time (in second) necessary to create a tree and computing the nodes' keys.

We see that the creation of a group and its associated tree takes a couple of seconds. This delay is above one second, but is still acceptable by an average user, in particular since creating a group is not very common.
Moreover, the difference of factors is light: less than the twice the number of CPU cycles for both sike27 and SIKE434 in comparison with ECDH, which is not a lot. Most of the time spent in this step is for sending the public keys to the server, which has to receive and take care of the information, so that it will correctly transfer it when the new members get online.

7.2.4 Tree update

This part concerns key updating, adding a member, and removing one. It was decided to consider these three processes together because they altogether are quite similar: a modification in the tree leading to an update of the root key.

Key update:

In order to keep the system as secure as possible, a user should change their private key, in order to modify the root key and thus the stage key, through the following steps:

1. Change the private key (randomly or not)
2. Compute the public key
3. Calculate the key pair of the nodes in the path
4. Collect the public keys in the copath and send them to the other members
5. Update the stage key

Algorithm	Mean CPU cycles	Variance	Time (Phone)	Time (Computer)	Factor
ECDH	435,630	8,868	0.512	0.435	1
SIKE434	466,765	10,088	0.549	0.466	1.07
sike27	485,004	2,405	0.57	0.485	1.11

Table 7.6: Performance (in thousands of cycles) and time (in second) necessary to update a leaf secret

From the Table 7.6, one can see that updating a key is quite fast: under 0.4 second. And the difference between ECDH, SIKE434 and sike27 is low (around 10% of CPU cycles in more with sike27).
Few computations need to be done: once a member has updated his key pair, they then need to re-compute the keys of the nodes in their path, and send the new keys to the server. If there are m members in the group, there will be $\lfloor \log_2(m) \rfloor$ key pair to calculate, which is low, even with m big.

Moreover, since the keys should often be updated (ideally after each message), it is a good news that it requires not so much CPU cycles to be done.

Adding a member

When a member of a group decides to add a user to the group, they must:

1. Fetch the key bundle of the new member
2. Derive a shared secret with X3DH and the KDF
3. Create a new node and put it in the tree
4. Associate the private key and compute the public key
5. Update the key pair in the path of the new node
6. Send the details of the tree update to the new user and to the former members

The measurements obtained are presented in Table 7.7.

Algorithm	Mean CPU cycles	Variance	Time (Phone)	Time (Computer)	Factor
ECDH	918,034	56,394	1.08	0.918	1
SIKE434	1,067,950	47,190	1.25	1.06	1.15
sike27	1,146,730	16,250	1.34	1.14	1.24

Table 7.7: Performance (in thousands of cycles) and time (in second) necessary to add a member in a group

We see that it takes around 1.5 second to add a member to the group, along with the other operations. Once again, the light difference of factors is due to the network operation, that is warning the former members that a new one is joining the group and send the new member the details necessary for them to create the tree on his side and fill the nodes with the key pairs computed.

To create the shared key, SIKE (or ECDH) is used 3 times, in X3DH. And with m members in the group, there are $2 \cdot \log_2(m)$ keys to compute (the private and the public ones each time). Hence, even when m is high, there are few calculation to do. This means that we should go higher than a couple of seconds for this step. It is not perfect in matter of time, but still acceptable, since adding a user is not as frequent as sending a message.

Removing a member:

For the removal of a member, a user must first remove the corresponding leaf in the node, and then change the root key. This is made with the following steps:

1. Delete the leaf in the tree of the former member
2. Its sibling takes the parent's place in the tree
3. Send the details of the update the other other members

Algorithm	Mean CPU cycles	Variance	Time (Phone)	Time (Computer)	Factor
ECDH	954,980	12,116	1.12	0.95	1
SIKE434	1,077,440	44,460	1.26	1.07	1.12
sike27	1,105,520	11,594	1.30	1.11	1.16

Table 7.8: Performance (in thousands of cycles) and time (in second) necessary to remove a member of a group

4. Change the initiator's private key, to update the stage key

This step is actually quite similar to the previous one: someone modifies the group structure (by adding or removing a leaf), re-computes all the keys of the nodes in a leaf's path (the initiator's or the concerned member's), and eventually sends the information to the server.
The only noticeable difference is that here, one doesn't need to compute a shared secret with X3DH.

In term of CPU cycles and time, as one can observe from Table 7.8 we logically obtain similar results to the ones when adding a member: it takes less than a second and the factors are all very close. It is less than a second, but still a user would notice this delay. Fortunately, this step is not so much common through the use of the messaging app.

7.2.5 Sending messages

This part of the test is slightly different than before: all the tests made until now are about ART+SIKE. But when a message is sent to the group, there is no calculation involving SIDH, since one just has to take the stage key and apply AES-256 into the plaintext, with the key, to get the ciphertext, and similarly to decrypt the message.
Instead, we test the communication directly between two members, because in this situation, both parties have to derive a shared key for AES, based on their identity key and a private key, along with one-time keys.

Algorithm	Mean CPU cycles	Variance	Time (Phone)	Time (Computer)	Factor
ECDH	135,460	672	0.159	0.135	1
SIKE434	150,041	841	0.176	0.151	1.12
sike27	154,960	1,495	0.182	0.154	1.14

Table 7.9: Performance (in thousands of cycles) and time (in second) necessary to send a message

Among all the tests, these measures are the lowest in term of CPU cycles and time, which is a good news since sending a message is the most common action done by a user. We had said that below 0.1 second, an average user would not remark the delay. Although we did not reach this value for

any of the protocols, we are still very close to it, probably close enough to have the right to say that the user won't notice a waiting time. And since there is only one computation done here (the computation of the shared key, given a private and a public key), we obtain a difference of 0.015 second between ECDH and sike27.

When we talked about the average user, we said that they sent or received around 150 messages per day. By multiplying the time obtained here by this value, we get 15.63 sec, 17,36 sec and 17,88 sec to take care of 150 messages. It goes over the 10 seconds we wished not to exceed, but sending 150 messages at once is not usual, or at least rare enough so that the user would accept this price to send so many messages, even more so if these messages are in majority received (107 out of 145, as stated in Section 7.2.1).

7.2.6 Key storage

One of the main advantage of elliptic curves in general is the size of their keys, much smaller than the other equivalent protocols. Table 7.10 shows the space necessary in the memory a device would need to store a key pair.

Algorithm	1 private key	Factor
ECDH	32	1
sike27	2	0.06
SIKE434	44	1.38
SIKE751	80	2.5

Algorithm	1 public key	Factor
ECDH	32	1
sike27	10	0.31
SIKE434	330	10
SIKE751	564	18

Table 7.10: Memory space necessary to store one key, private (*left*) or public (*right*), according to the KAP

And for 102 key pairs (to be exact, there are 101 private keys for 102 public keys, since there are two public keys corresponding to the private key, one for each side, that is *initiator* and *responder*)

Algorithm	101 private key	Factor
ECDH	3,232	1
sike27	202	0.06
SIKE434	4,444	1.38
SIKE751	8,080	2.5

Algorithm	102 public key	Factor
ECDH	3,264	1
sike27	1020	0.03
SIKE434	33,660	10.3
SIKE751	57,528	17.6

Table 7.11: Memory space necessary to store one hundred and two keys, private (*left*) or public (*right*), according to the KAP

Initialisation phase

As indicated earlier in the chapter, a new user who just registered in the messaging app has to compute 102 key pairs, and obviously, these keys take space. Overall, the good news with working with elliptic curves is the size of the keys, that are rather small. With ECDH, it takes 3.19 KB[1] of memory for the 102 private keys. And the maximum storage necessary is for SIKE751, with 7.96 KB, which is juste 2.5 times more than ECDH.
On a phone with a memory of 16 GB, and a computer with 64 GB, the space asked for each protocol is always small enough not to worry.

For the public keys, the size is more important: from 0.99 KB for sike27, up to 56 KB. And the difference of factors with ECDH is more important as well. Nevertheless, it should not be worrisome since all public keys are stored onto the server, so that they stay accessible for anyone. And for the server capacity, we make the assumption that memory is not the priority, especially when the key bundles to store are individually higher than 100 KB, like here.

Tree creation

When a user decides to create a group, the first step is construct the tree structure. Once this is done, it is stored both on the user side and on the server. It was decided that the simplest way to do this in Python was through a PKL file. PKL files are files created by Pickle [cPy20], a Python module used to make objects persistent and store them on disk. Each tree has a corresponding file, and these files never go over 2 KB.

To compute the shared secret, the initiator of the process fetches the recipient's key for X3DH. But once it's done, there is no need to keep them, and so they are immediately deleted. Hence, no storage space is necessary for the keys.
Only space on the disk is needed for the PKL files, and 2KB per file is not a problem for both a smartphone and a computer.

Tree updates

The tree updates can either be updating a leaf key, adding a member or removing one.
For each of these process, there is no need to store keys, or at least on the long-term. Only the addition of a member needs to temporary have the keys from the key bundle in order to compute the shared secret, but as in the creation of the tree, these keys are deleted as soon as the process is finished.

Each of these processes requires (or at least recommends) to update a leaf key, in order to change the root key. Once it is done, the updated key obtained is immediately placed in tree that is stored in the PKL file, while the previous leaf key is deleted. So, we remove one to add another, meaning the size of the file stays the same, and the memory necessary as well.

[1]The conversion from Bytes to Kilobytes is made in binary: 1 B $\simeq$ 0.000976 KB

Still, we could consider that adding a node into the tree increases its size, and so the PKL file's, (as well as decreasing when a member is removed), but these modifications are too small to be considered here.

Sending a message

A message sent is first encrypted with AES-256. If the message's recipient is a whole group, the key for AES is the stage key, which is simply computed by a combination of the root key and the previous stage key, through a KDF. So, there aren't any computation to do with ECDH or SIKE. This means no storage space is necessary, except when a message is received since it must be placed in the memory, but this is independent to the key agreement protocols.
If the message is directly sent to another user, the sender has to compute a shared key. This key is obtained with X3DH, by using the identity keys and one-time keys, meaning that the sender has to store these keys while the computations are in process. When it is finished, the identity keys (the sender's private one and the recipient's public one) are deleted, and the public identity of the recipient is kept, since it should change over time. This means that a member should have the public key of each member, and as shown in Table 7.10, up to 0.54 KB of space per member is necessary, which is small enough to store many keys like this.

7.2.7 Conclusion

Among the two aspects studied here, that are the execution time and the memory space necessary, the latter hasn't been a problem, in any situation. Any device, phone or computer with a minimal of memory, is totally capable of storing many keys, even though we usually don't exceed the 100 keys. Nevertheless, there exists a compressed SIKE implementation ([ACC+17], Section 2.3) with a compression of the public keys. In this situation, the public keys' size is reduced by 41 %, and the size of the ciphertext by 39 %. But the trade-off is the performance, increasing of around 150 %.

For the waiting time, the worst situation is the initialisation, where an average user would feel a delay but it is before starting any communication, we expect them to understand it. Sending a message or updating asks a very short time, short enough not to make the user feel it. For adding or removing a member, the time is around a second, which should still be acceptable. Maybe a phase that could be improved is when creating a tree, taking around 2.5 seconds. Even though it is not much, the delay is well and truly here. The fact that this process is not that common could compensate it, but an improvement is still welcome.
Either way, anyone who would want a feasible protocol has to go with the optimized implementation, and in particular the x64 assembly structure.

Overall, we reach acceptable times, and the improvement mentioned in Section 6.4.3 can only make the whole protocol faster.

7.3 SIKE alternative

In the introduction of this paper, we talked about the NIST competition, with 4 public key-establishment algorithms. Among them, 3 are lattice-based (SABER, NTRU and Crystals-Kyber) and 1 is code-based (Classic McEliece). It could be interesting to compare them with SIKE, in particular with concerning the time of execution.

Banerjee and Hasan measured in 2018 the energy consumption of the candidates [BH18]. The tests were executed on a 64-bits Intel 6700 Skylake Processor, 3.4 GHz. Along with these energy tests was measured the execution time, to create a public key, for key encapsulation and decapsulation, and so on... We won't present all the tests made, but only the key pair generation measures of the protocols within security level 3. A protocol in such level indicates that it should be as hard as breaking AES-192, or more generally, any block cipher using 192-bits keys.

Category	Algorithm	Time	Factor
Lattice	SABER	0.18	1
	NTRU	0.83	4.6
	Crystals-Kyber	0.255	1.41
Code	Classic McEliece	646	3,588
Isogeny	SIKE	85.99	477

Table 7.12: Execution time for key pair generation. Time is in milliseconds

As one can see from Table 7.12, SIKE is one of the slowest post-quantum algorithm : 477 slower than SABER. And overall, the lattice-based protocols seem to be more efficient than the isogeny-based or code-based. With the data given in the same paper, these 3 protocols consume much less energy. The only positive aspect is the key size, which is the smallest with SIKE, because of the elliptic curve aspect ([Dui19], Table 6.12).

These other post-quantum algorithms could be interesting alternatives, in particular if the main objective is speed. For more detail, see Duit's paper, who swapped ECDH in the Signal protocol with various PQ algorithms [Dui19].

Chapter 8

Conclusion and Furture Works

In this Section, we conclude on the whole work presented in this paper, and look at future work that could be added or combined here.

8.1 Conclusion

In this thesis, we first described the Signal protocol, the cryptographic scheme that allows end-to-end encryption, forward and post-compromise secrect, along with authentication. The latter properties is achieved through X3DH, while the former ones are made possible with the Double Ratchet.

Although it is an efficient protocol for two-party communication, we brought two issues to light. The first is the lack of resistance against quantum algorithms, future threat in particular upon ECDH, because of Shor's algorithm, making the whole system soon vulnerable with the rise of the quantum computers.
The second is the difficulty to scale up to group conversation in an efficient way. Most implementations from other applications loose either the capacity to initiate a group conversation with offline user, that is asynchronicity, or the self-healing property with Post-Compromise secrecy.

We then presented a proposition from Cohn-Gordon and al., which describes the tree-based protocol ART, for Asynchronous Ratchet Tree. The protocol being based on Diffie-Hellman to compute a shared group key, we decided to swap the former primitive with a post-quantum scheme.
This scheme is SIKE, an instantiation of SIDH, based on supersingular elliptic curves, said to be quantum-resistant, although concrete proofs are awaited.

To prove the combination of both protocol was possible, we gave the details of a possible implementation of a group messaging application, based on ART and SIKE, before actually implementing it on Python.
Eventually, to show that this implementation is not only doable, but also could be feasible in a real-life scenario, we analysed it. The tests were made in two steps : the first was to only measure the

computational impact of swapping ECDH with SIKE when computing private and public keys; the second was testing the feasibility of the app regarding the time and delay necessary in part of the protocol, that are the initialisation, the group creation, and the group update. On one hand, the first tests showed that although ECDH is the faster, we can achieve correct time of computation, in particular when limited to around 100 key computations, as long as we used the optimized implementation, along with the x64 assembly structure. On the other hand, the second tests showed that it was indeed feasible to create a post-quantum group instant messaging protocol, with acceptable delay from an average user, with once again the optimized implementation. As a recall, with the optimized SIKE434, it takes around 0.3 seconds to compute the 100 public keys that are published onto the server. For the protocol, one can expect more than 6 seconds for the initialisation phase, which is acceptable but clearly can be better; up to 3 seconds to create a group, i.e. creating the tree structure, computing all the key pairs, and transmitting the information to the concerned members; for adding or removing a member, the delay is around 1 second, while for updating a key, it is close to 0.5 second; and finally, for sending a message, less than 0.2 second is needed. Overall, we obtain good results, showing such a protocol could be developed and used in the real world.

8.2 Future research

There are several topics and subjects which could be studied in the future, to develop the ideas presented in this work, and in this section, we name a few.

The first follows Section 7.3, where it was shown alternative to the SIKE protocol. Between lattice-based and code-based, the most promising in terms of rapidity are the lattice-based algorithm, such as SABER or Crystals-Kyber. Although key encapsulation mechanisms are not easy *plug-and-play* substitute for ECDH key exchange unlike SIKE, the two former are able to replace ECDH, in order to make ART (and more generally ECDH-based protocols) quantum-resistant. Similar tests to the ones in Chapter 7 should obviously be made, in order to compare them with ECDH, throughout the whole ART protocol. And in addition to the time execution and storage space tests, it could be interesting to study energy consumption and network use as well.

If replacing every cryptographic primitive threatened by a post-quantum one is at the end too expensive computationally, another way could be a hybrid form, combining ECDH with PQ algorithm. The reason is that currently, most of the schemes presented until now are quite new and not much analysed, meaning undiscovered vulnerabilities probably exist. In a hybrid encryption scheme, we combine two or more encryption protocols, to resist to both quantum and non-quantum attacks. Hence, if the post-quantum scheme is broken, we can at least rely on the other one(s) to preserve some security.

Eventually, a last idea of topic would about the group creation and update. Throughout the paper, we considered all the members always honest. Nevertheless, what protection are there against malicious ones ? In particular, every member has always trusted the creator of the group. But a malicious creator might be able to secretly add an eavesdropper to a group, leaves this group, and

still be able to decrypt the messages. As much as the users don't trust the intermediate server, they should not trust each other, and should make adequate and precautionary actions.

Appendix A

Algorithms

Algorithm 9: compute_pk

Input : $n, A, P_A, Q_A, P_B, Q_B, k, side$

Output: A, P_B, Q_B

$R \leftarrow$ multiplication(Q, k_A, A) ;
$S_A \leftarrow$ addition(P, R, A)
for $i \in [n-1, ..., 1]$ **do**
    if $side == $ *'initiator'* **then**
        $coef \leftarrow 2^i$
    else
        $coef \leftarrow 3^i$;
    end
    $S'_A \leftarrow$ multiplication$(S_A, coef, A)$;
    $\alpha \leftarrow S'_A[0]$;
    if $side == $ *'initiator'* **then**
        $A' \leftarrow 2(1 - 2\alpha^2)$;
    else
        $A' \leftarrow (A\alpha - 6\alpha^2 + 6)\alpha$;
    end
    $jA \leftarrow 256 \cdot \frac{(A'^2-3)^3}{A'^2-4}$;
    $P'_B \leftarrow$phi$(\alpha, P_B, A', side)$;
    $Q'_B \leftarrow$phi$(\alpha, Q_B, A', side)$;
    if $i \neq 1$ **then**
        $S'_A \leftarrow$phi$(\alpha, S_A, A', side)$;
    end
    $A \leftarrow A'$;
    $P_B \leftarrow P'_B$;
    $Q_B \leftarrow Q'_B$;
    $S_A \leftarrow S'_A$;
end

Return: A, P_B, Q_B

Algorithm 10: shared_secret

Input : $n, A, P_A, Q_A, P_B, Q_B, k, side$

Output: jA

$R \leftarrow$ multiplication(Q, k_A, A) ;

$S_A \leftarrow$ addition(P, R, A)

for $i \in [n-1, ..., 1]$ **do**

  if $side ==$ *'initiator'* **then**

    $coef \leftarrow 2^i$

  else

    $coef \leftarrow 3^i$;

  end

  $S'_A \leftarrow$ multiplication$(S_A, coef, A)$;

  $\alpha \leftarrow S'_A[0]$;

  if $side ==$ *'initiator'* **then**

    $A' \leftarrow 2(1 - 2\alpha^2)$;

  else

    $A' \leftarrow (A\alpha - 6\alpha^2 + 6)\alpha$;

  end

  $jA \leftarrow 256 \cdot \frac{(A'^2-3)^3}{A'^2-4}$;

  if $i \neq 1$ **then**

    $S'_A \leftarrow$phi$(\alpha, S_A, A', side)$;

  end

  $A \leftarrow A'$;

  $S_A \leftarrow S'_A$;

end

Return: jA

Algorithm 11: phi

Input : $\alpha, A, R, side$

Output: $[X, Y]$

$[x, y] \leftarrow R$ **if** $side ==$ *'initiator'* **then**

  $X\_num \leftarrow x(\alpha x - 1)$;

  $X\_deno \leftarrow x - \alpha$;

  $Y\_num \leftarrow (x^2\alpha - 2x\alpha^2 + x)^2$;

  $Y\_deno \leftarrow (x - \alpha)^2$;

else

  $X\_num \leftarrow x(\alpha x - 1)^2$;

  $X\_deno \leftarrow (x - \alpha)^2$;

  $Y\_num \leftarrow (x\alpha - 1)(x^2\alpha - 3x\alpha^2 + x + \alpha)$;

  $Y\_deno \leftarrow (x - \alpha)^3$;

end

$X \leftarrow x\_num \cdot$ *inverse*(x_deno) ;

$Y \leftarrow y\_num \cdot$ *inverse*(y_deno) ;

Return: $[X, Y]$

Bibliography

[ACC+17] Reza Azarderakhsh, Matthew Campagna, Craig Costello, LD Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, et al. Supersingular isogeny key encapsulation. *Submission to the NIST Post-Quantum Standardization project*, 2017.

[ATAa14] MA Alia, AA Tamimi, and ONA Al-allaf. Cryptography based authentication methods,‖ vol, 2014.

[Aza] Reza Azarderakhsh. Performance of quantum-safe isogenies on arm processors. ETSI.

[BBC13] Marco Baldi, Marco Bianchi, and Franco Chiaraluce. Security and complexity of the mceliece cryptosystem based on quasi-cyclic low-density parity-check codes. *IET Information Security*, 7(3):212–220, 2013.

[BGJT14] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–16. Springer, 2014.

[BH18] Tanushree Banerjee and M Anwar Hasan. Energy consumption of candidate algorithms for nist pqc standards. Technical report, Technical report, Centre for Applied Cryptographic Research (CACR) at the ..., 2018.

[BMO+19] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The messaging layer security (mls) protocol. *Working Draft, IETF Secretariat, Internet-Draft draft-ietf-mls-protocol-07*, 2019.

[BMS20] Colin Boyd, Anish Mathuria, and Douglas Stebila. Key agreement protocols. In *Protocols for Authentication and Key Establishment*, pages 165–240. Springer, 2020.

[Bou18] Chad Boutin. NIST Issues First Call for Lightweight Cryptography to Protect Small Electronics. https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-smal 2018.

[Bro09] Daniel RL Brown. Sec 1: Elliptic curve cryptography. *Certicom Research*, page v2, 2009.

[BS20] Xavier Bonnetain and André Schrottenloher. Quantum security analysis of csidh. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 493–522. Springer, 2020.

[BW17] William Buchanan and Alan Woodward. Will quantum computers be the end of public key encryption? *Journal of Cyber Security Technology*, 1(1):1–22, 2017.

[Byl90] Czesław Bylinski. The complex numbers. *Def*, 7(1C):1, 1990.

[CGCG+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*, pages 1802–1819, 2018.

[CH17] Craig Costello and Huseyin Hisil. A simple and compact algorithm for sidh with arbitrary degree isogenies. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 303–329. Springer, 2017.

[CJL+17] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. Efficient compression of sidh public keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 679–706. Springer, 2017.

[CJS14] Andrew Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology*, 8(1):1–29, 2014.

[CK01] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 453–474. Springer, 2001.

[CLM+18a] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. Csidh: an efficient post-quantum commutative group action. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 395–427. Springer, 2018.

[CLM+18b] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An Efficient Post-Quantum Commutative Group Action (slides version). https://csidh.isogeny.org/ECCworkshop.pdf, 2018.

[Cos19] Craig Costello. Supersingular isogeny key exchange for beginners. In *International Conference on Selected Areas in Cryptography*, pages 21–50. Springer, 2019.

[Cou06] Jean Marc Couveignes. Hard homogeneous spaces. *IACR Cryptol. ePrint Arch.*, 2006:291, 2006.

[cPy20] cPython. Pickle. https://github.com/python/cpython/blob/master/Doc/library/pickle.rst, 2020.

[CS03] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.

[DF17] Luca De Feo. Mathematics of isogeny based cryptography. *arXiv preprint arXiv:1711.04062*, 2017.

[dF19] Luca de Feo. Isogeny graphs in cryptography. 2019.

[DIK06] Christophe Doche, Thomas Icart, and David R Kohel. Efficient scalar multiplication by isogeny decompositions. In *International Workshop on Public Key Cryptography*, pages 191–206. Springer, 2006.

[DK07] H Delfs and H Knebl. Introduction to cryptography principles and applications, 2007.

[Doe19] John Doe. Measurements of public-key cryptosystems, indexed by machine. http://bench.cr.yp.to/results-stream.html, 2019.

[Dui19] Ines Duits. The post-quantum signal protocol: Secure chat in a quantum world. Master's thesis, University of Twente, 2019.

[EH18] Youssef El Housni. Edwards curves. 2018.

[ES19] Wesam Eid and Marius C Silaghi. Speeding up elliptic curve multiplication with mixed-base representation for applications to sidh ciphers. *arXiv preprint arXiv:1905.06492*, 2019.

[FHLOJRH17] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, 2017.

[FMB+16] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is textsecure? In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 457–472. IEEE, 2016.

[Gal01] Steven D Galbraith. Supersingular curves in cryptography. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 495–513. Springer, 2001.

[gmp21] GNU Multiiple Precision. https://gmplib.org/, 2021.

[GPST16] Steven D Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 63–91. Springer, 2016.

[JDF11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[Kra05] Hugo Krawczyk. *Perfect Forward Secrecy*, pages 457–458. Springer US, Boston, MA, 2005.

[LL93] Arjen K Lenstra and Hendrik W Lenstra. *The development of the number field sieve*, volume 1554. Springer Science & Business Media, 1993.

[LP07] Hyun-Yong Lee and In-Cheol Park. Balanced binary-tree decomposition for area-efficient pipelined fft processing. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(4):889–900, 2007.

[LXY19] Zhe Li, Chaoping Xing, and Sze Ling Yeo. Reducing the key size of mceliece cryptosystem from automorphism-induced goppa codes via permutations. In *IACR International Workshop on Public Key Cryptography*, pages 599–617. Springer, 2019.

[Mar03] John Levi Martin. What is field theory? *American journal of sociology*, 109(1):1–49, 2003.

[Mat19] Dark Matter. Introduction to Post-Quantum Cryptography. https://crypto.polito.it/content/download/200/1205/file/slide_bellini.pdf, 2019.

[MLS19] Michiel Marcus, T Lange, and P Schwabe. White paper on mceliece with binary goppa codes, 2019.

[Nie10] Jakob Nielsen. Website Response Times. https://www.nngroup.com/articles/website-response-times/, 2010.

[Pau17] Eduard Paul. What is Digital Signature- How it works, Benefits, Objectives, Concept. https://www.emptrust.com/blog/benefits-of-using-digital-signatures, 2017.

[Pie17] Margarita Pierrakea. Supersingular isogeny key-exchange. In *Encyclopedia of Cryptography and Security*, 2017.

[PM16a] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *GitHub wiki*, 2016.

[PM16b] Trevor Perrin and Moxie Marlinspike. The x3dh key agreement protocol. *Specification. Nov*, 2016.

[QQL10] Bing Qi, Li Qian, and Hoi-Kwong Lo. A brief introduction of quantum cryptography for engineers. *arXiv preprint arXiv:1002.1237*, 2010.

[Ren18] Joost Renes. Computing isogenies between montgomery curves using the action of (0, 0). In *International Conference on Post-Quantum Cryptography*, pages 229–247. Springer, 2018.

[Riv21] Quentin Rivollat. Sike and art. https://github.com/QuentinRiv/SIKE_ART, 2021.

[RSS+18] Avi Rosenfeld, Sigal Sina, David Sarne, Or Avidov, and Sarit Kraus. A study of whatsapp usage patterns and prediction models without message content. *arXiv preprint arXiv:1802.03393*, 2018.

[SBBB12] William Stallings, Lawrie Brown, Michael D Bauer, and Arup Kumar Bhattacharjee. *Computer security: principles and practice*. Pearson Education Upper Saddle River, NJ, USA, 2012.

[Sil09] Joseph H Silverman. *The arithmetic of elliptic curves*, volume 106. Springer Science & Business Media, 2009.

[Spa16] John Spacey. What is a Key Derivation Function. https://simplicable.com/new/key-derivation-function, 2016.

[Str11] Emma Strubell. An introduction to quantum algorithms. *COS498 Chawathe Spring*, 13:19, 2011.

[Sug18] YK Sugi. What is a quantum computer? explained with a simple example. https://www.freecodecamp.org/news/what-is-a-quantum-computer-explained-with-a-simple-example-b 2018.

[Sut15a] Andrew Sutherland. Elliptic Curves - Lecture 14. https://math.mit.edu/classes/18.783/2015/LectureNotes14.pdf, 2015.

[Sut15b] Andrew Sutherland. Elliptic Curves - Lecture 7. https://math.mit.edu/classes/18.783/2015/LectureNotes7.pdf, 2015.

[Vél71] Jacques Vélu. Isogénies entre courbes elliptiques. *CR Acad. Sci. Paris, Séries A*, 273:305–347, 1971.

[Vin03] Ėrnest Borisovich Vinberg. *A course in algebra*. Number 56. American Mathematical Soc., 2003.

[Whi16] WhitePaper. Whatsapp encryption overview. 2016.

[Yel16] Baris Yeldiren. What is End-to-end encryption (E2EE) ? https://www.printfriendly.com/p/g/Sddiq3, 2016. [Online; accessed 19-July-2008].

[ZSP+18] Gustavo HM Zanon, Marcos A Simplicio, Geovandro CCF Pereira, Javad Doliskani, and Paulo SLM Barreto. Faster key compression for isogeny-based cryptosystems. *IEEE Transactions on Computers*, 68(5):688–701, 2018.