Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

# Analysis and Implementation of the Messaging Layer Security Protocol

Tesi in
Sicurezza delle Reti

Relatore:
Gabriele D'Angelo

Presentata da:
Nicola Giancecchi

Anno Accademico 2018/2019

# Parole chiave

**Network Security**
**Messaging**
**MLS**
**Protocol**
**Ratchet Trees**

*"Oh me, oh vita! Domande come queste mi perseguitano.*
*Infiniti cortei d'infedeli, città gremite di stolti,*
*che v'è di nuovo in tutto questo, oh me, oh vita!*
*Risposta:*
*Che tu sei qui, che la vita esiste e l'identità.*
*Che il potente spettacolo continua,*
*e che tu puoi contribuire con un verso."*
- Walt Whitman

*Alla mia famiglia.*

# Introduzione

L'utilizzo di servizi di messaggistica su smartphone è incrementato in maniera considerevole negli ultimi anni, complice la sempre maggiore disponibilità di dispositivi mobile e l'evoluzione delle tecnologie di comunicazione via Internet, fattori che hanno di fatto soppiantato l'uso dei classici SMS.

Tale incremento ha riguardato anche l'utilizzo in ambito business, un contesto dove è più frequente lo scambio di informazioni confidenziali e quindi la necessità di proteggere la comunicazione tra due o più persone. Ciò non solo per un punto di vista di sicurezza, ma anche di privacy personale. I maggiori player mondiali hanno risposto implementando misure di sicurezza all'interno dei propri servizi, quali ad esempio la crittografia end-to-end e regole sempre più stringenti sul trattamento dei dati personali.

In questa tesi andremo ad illustrare *Messaging Layer Security*, abbreviato in MLS, un nuovo protocollo in fase di sviluppo che garantisce sicurezza ed efficienza in conversazioni di gruppo. Se in una conversazione tra due client la sicurezza può essere garantita tramite crittografia end-to-end e scambio di chiavi, il problema sorge quando più attori partecipano alla conversazione in modo asincrono: in questo caso lo sforzo computazionale è considerevole, a maggior ragione considerando l'uso di dispositivi mobile con capacità di batteria ridotta che lavorano in modo asincrono, non garantendo perciò la presenza continua del dispositivo online.

Verrà trattata sia la parte architetturale, più generale e di indirizzo, che la parte di protocollo, più tecnica e dettagliata. Infine verrà illustrata una implementazione di MLS scritta in Rust e chiamata Melissa, che fornisce tutte le funzionalità base previste dalla versione draft 05 del protocollo.

I lavori sul protocollo, tutt'ora in corso, sono portati avanti da un apposito working group istituito presso l'Internet Engineering Task Force (IETF)

composto da ricercatori universitari ed aziendali. Melissa, invece, è portata avanti da un apposito gruppo all'interno di Wire, servizio di messaggistica sicuro orientato alle aziende.

Il presente elaborato di tesi nasce in seguito ad un'esperienza Erasmus negli uffici di Wire a Berlino, continuata in questi ultimi mesi da remoto.

### Avvertenza

Il presente elaborato di tesi si basa su una versione *Internet-Draft* non definitiva delle specifiche di Messaging Layer Security. I contenuti riportati potranno cambiare in futuro. Tutti gli aggiornamenti relativi a MLS sono disponibili sul sito ufficiale `https://messaginglayersecurity.rocks`.

# Introduction

The use of messaging services on smartphones has increased considerably in recent years, due to the growth in the availability of mobile devices and the evolution of communication technologies via Internet, factors that have effectively replaced the use of text messages.

This increase also concerned the use in the business environment, a context where the exchange of confidential information is more frequent and therefore the need to protect communication between two or more people. This is important not only on a security point of view, but also for personal privacy. The major global players have responded by implementing security measures within their services, such as end-to-end encryption and increasingly strict rules regarding the processing of personal data.

In this thesis we will illustrate *Messaging Layer Security*, shortened as MLS, a new protocol under development that guarantees security and efficiency in group conversations. When in a conversation between two clients, security can be ensured through end-to-end encryption and key exchange. The problem arises when multiple actors participate in the conversation asynchronously: in this case the computational effort is considerable, even more so considering the use of mobile devices with reduced battery capacity that does not guarantee the continuous presence of the online device.

The thesis will deal with both the architectural part, that is more general and traces the outline of the subject, and the protocol part, more technical and detailed. Finally, an implementation of MLS written in Rust and called Melissa will be illustrated, which provides all the basic functionalities indicated in the draft 05 version of the protocol.

Work on the protocol, still in progress, is carried out by a special working group set up at the Internet Engineering Task Force (IETF) composed of

university and business researchers. Melissa, instead, is carried out by a special group within Wire, a secure business-oriented messaging platform.

This thesis document follows an Erasmus experience in the Wire offices in Berlin, which continued remotely in recent months.

### Disclaimer

This thesis work is based on a non-definitive, *Internet-Draft* version of Messaging Layer Security specifications. The contents shown may change in the future. All updates about MLS are available on the official website `https://messaginglayersecurity.rocks`.

# Contents

# Chapter 1

# Messaging and Security

## 1.1 Messaging

*Messaging* is a type of service that allow users to exchange information by using different means of communication. In computer science, communication is intended to take place over an electronic channel. Types of information may include texts, images, videos, voice notes, files, documents, audio and video calls, and much more. Messaging applications include emails, chats, instant messengers, messaging apps and SMS. Communication may take place on computers, phones, tablets and smart devices.

In this document, we will consider the case of *messaging services*, including instant messengers and messaging apps.

### 1.1.1 General use cases of messaging services

In general, messengers provide real-time text transmission service through the Internet between two or more users. To achieve this, we need to specify some fundamental properties of messaging.

Users who wish to exchange messages register to a messaging service using personal credentials, which can be an email and password pair, a username and password, a telephone number, a one-time password, etc. The messaging service is responsible for keeping authentication information safe, or to manage authentication through third-party services (e.g. Single Sign-On) consistently.

Users who join a messaging service are described by different information about them. Typically, users must specify an identifier, which can be a username, an email address, a person's name or other types of identifier (e.g. numeric or UUID). They can also be represented, optionally, by a picture or avatar. Other information is generally defined based on the service and may include, for example, the current status, job position, age, nickname, email or phone number unless previously specified.

The user who enters a messaging service typically sees the list of open conversations, known in instant messengers as *buddy list*. Depending on the service, conversations may be with one person (*one-on-one*) or a group. Group conversations may also be called in other ways, such as *chats*, *channels* or *rooms*, and they are usually distinguished by an identifier (a title or nickname).

Users may typically initiate a one-on-one conversation by knowing the identifier of the other person to contact or, if allowed by the service, by searching inside a global directory. Users may place restrictions on how other users are added, e.g. by sending a request or limiting to already authorized users inside the buddy list. Users can create new conversations, or they can be added to other conversations by other users, or by joining in other ways, such as via an invitation link. Conversations may take place when both or all users are online or not: therefore we talk about *instant messaging* in the first case and *asynchronous messaging* in the latter.

Users of a conversation can view the history of messages and content sent in that conversation since they joined it, in a chronological order, where the last messages sent are usually displayed at the end of the list. Users can participate in the conversation by writing inside the text field of the conversation window. Based on the messaging service, they might also attach emoticons/emojis, images, files, contacts, locations, stickers, surveys, etc.

Some messenger services also provide other advanced features, like audio and video calls with one or more people, bots, ephemeral messages, integration with third-party services, admin control, etc.

## 1.2 Evolution of messaging

Considering the above mentioned information, we can state that messaging is nothing but the exchange of messages between two remote users. We will now retrace the main steps that led to the current state of the industry.

### 1.2.1 Pre-Internet era (1960-1990)

In 1961, the MIT Computation Center developed the *Compatible Time-Sharing System* (CTSS), one of the first time-sharing operating systems that allowed multiple users to share resources over a single mainframe, the IBM 7094. This way, up to 30 simultaneous users were able to communicate by storing files on an online disk. [18] Eight years later, CompuServe was invented: it was the first commercial online service available in the United States, a sort of predecessor of the World Wide Web. CompuServe was known for its online chat and electronic mail service, message forums, software libraries and online games. CompuServe also developed the *Graphics Interchange Format* (GIF) that became popular in the 1990s, returned to the fore in the second half of 2010s.

Shortly after, in 1971, the email service was invented by Ray Tomlinson. His idea was to specify the destination of a message via the @, creating an "address" now known as `username@name_of_computer` (e.g. `john@example.com`). The idea of Tomlinson soon became adopted as the main network email system of ARPANET. In 1985, Quantum Computer Services from Vienna launched *Quantum Link*, an online service for Commodore 64 and 128 which included chat rooms, email and instant messaging services.

Three years later the *Internet Relay Chat* (IRC) protocol was invented. IRC provides a group chat service where users can connect to channels, in order to discuss about different topics. It is also possible to create one-on-one conversations. IRC is still widely used nowadays: for example, it is the principal instant communication channel of the Wikipedia community. In 1989, Quantum Link became *America Online* (AOL). AOL grew exponentially during 1990s, becoming one of the largest internet providers in the United States.

### 1.2.2 Instant messengers (1990-2008)

In 1996, the Israeli company Mirabilis launched *ICQ*, one of the first instant messenger totally dedicated to individual chats. ICQ became very popular in a short amount of time. The following year, AOL launched *AOL Instant Messenger* (AIM), an instant messenger based on the proprietary *OSCAR* and *TOC* protocols. In 1998, AOL acquired ICQ, while Yahoo! launched *Yahoo! Messenger* and, one year later, Microsoft released the first version of *MSN Messenger*.

All these messaging services were united by a common characteristic: the lack of interoperability. A first attempt to let these messengers communicate with each other was brought on by *Jabber* in 1999, an open communication protocol known later as *Extensible Messaging and Presence Protocol* (XMPP). XMPP was used on a large scale by *Google Talk*, an instant messaging and VoIP service platform by Google launched in 2005.

In 2003, Niklas Zennström and Janus Friis founded *Skype*, a revolutionary telecommunication platform that provides encrypted instant messaging and VoIP, originally built with an hybrid peer-to-peer and client-server architecture.

### 1.2.3 Messaging apps (2008-current)

In 2008, Facebook launched *Facebook Chat*, a chat messaging service integrated into the Facebook platform, which later became a standalone app called *Facebook Messenger*. In the same period, Twitter was launched, featuring statuses with a maximum amount of 140 characters and a private messaging service widely known as *direct messages* (DM), initially designed for Twitter users who followed each other.

In November 2009, two former engineers from Yahoo - Brian Acton and Jan Koum - developed the first version of *WhatsApp*, a messaging platform for text and photos, and published it to the iOS App Store. It became really popular and grew exponentially in a short amount of time. WhatsApp Inc. was afterwards acquired by Facebook in 2014. During WWDC 2011, Apple presented *iMessage*, a messaging service dedicated to the Apple platforms and later supported on iOS, macOS and watchOS. The peculiarity is that it

is integrated into the system and users can send free iMessages in the same way they were used to sending SMS.

Google continued to invest in the messenger market with various solutions over time, like *Google Wave* and *Google Voice*, which later became *Google Hangouts*, and the *Allo* and *Duo* apps. In 2013, Nikolai and Pavel Durov, Vkontakte co-founders, launched *Telegram*, a cloud-based messaging platform with all the clients released open source. In the same period, interest in messengers grew in East Asia with the Chinese solution *WeChat* and the South Korean *Line*.

### 1.2.4 Business messengers

The popularity of messaging services has increased lately also on a business level, as a communication tool for teams. Team chats can provide a unique tool for communication within a company and, at the same time, they can reduce the amount of emails exchanged between users. Some of them include Skype for Business, Slack, Wire, Atlassian HipChat, Microsoft Teams and Discord. They often include advanced features like conference video calls, integrations, end-to-end encryption, admin tools, etc.

## 1.3 Security and privacy in communication

When dealing with private conversations, users expect from the messaging services that what they write is kept private, in a way that eavesdropping and interception should not be possible. This leads to the concepts of *security* and *privacy*. According to the Cambridge Dictionary, privacy is *someone's right to keep their personal matters and relationships secret*. This concept has evolved over time, switching from the right concerning the private sphere of a subject, to the right of keeping personal data secret.

### 1.3.1 Cryptography

Cryptography is the study of techniques that can guarantee secure communication in the presence of third parties called adversaries, preventing

them to read private messages. Modern cryptography shares the same basic concepts of computer security, namely:

- **data confidentiality**: protects the information from third parties, guaranteeing access only to authorized parties;
- **data integrity**: attests the originality of data, that is, the data has not been modified by third parties;
- **authentication**: guarantees the identity of the user within a communication;
- **non-repudiation**: impossibility of a party to dispute the authorship of given data.

Cryptography is a horizontal discipline that is based on different fields like computer science, mathematics, electrical engineering and physics, and it is used in various instances of everyday life. Some of them include payment cards, mobile communication, e-commerce and electronic communication, digital signatures and certified electronic delivery services.

Cryptography is made of two fundamental operations: *encryption* and *decryption*. Encryption is the transformation of a plain clear text into a ciphertext using a cipher. Decryption is the inverse operation: given a ciphertext, using a cipher it is possible to decrypt text and transform it back into plaintext. Ciphertexts are usually a sequence of scrambled and apparently unreadable characters. A cipher is a system capable of transforming a plain text into an unintelligible text.

There are two main ways to perform encryption: *symmetric* and *asymmetric* encryption, also known as *Public-Key Encryption*.

### Symmetric encryption

With symmetric encryption, a secret key is shared between sender and receiver, and is used to encrypt plain texts and decrypt ciphertexts. A fundamental requirement for this approach is that the two parties must first have exchanged the secret key in a secure fashion. In addition, the algorithm must be strong enough not to allow a possible opponent to decipher the text.

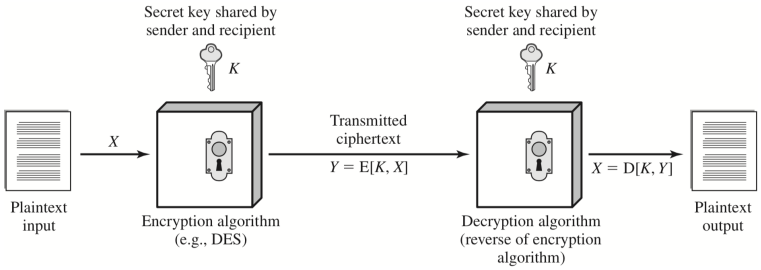



Figure 1.1: Symmetric Encryption. Source: [9]

Two possible ways to attack a symmetric encryption scheme are:

**Cryptanalysis** This approach relies on analysis brought on the content of a ciphertext, encrypted through a key. Cryptoanalysis tries to deduce a specific plaintext or the key used to encrypt it. If the attacker deducts a key successfully, it is compromised.

**Brute-force attacks** This method tries every possible key on a ciphertext, until an intelligible translation is obtained. On average, half of all possible keys must be tried before achieving a successful result.

### Public-key encryption

Asymmetric encryption, mostly known as *Public-Key Encryption* (PKE), is a type of cryptography that makes use of a key pair for each party of a communication. The public key is intended to be shared with third parties, while the private key remains secret and kept by each of the two parties. One key of the pair is used for encryption, while the other one is used for decryption. Another important fact is that PKE is based on mathematical algorithms, rather than operations on bit patterns. Public-Key Encryption was first publicly proposed by Whitfield Diffie and Martin Hellman in 1976.

Public-Key Encryption works as follows. Assuming that the two parties are called Alice and Bob:

1. both Alice and Bob generate a key pair for encrypting and decrypting messages;

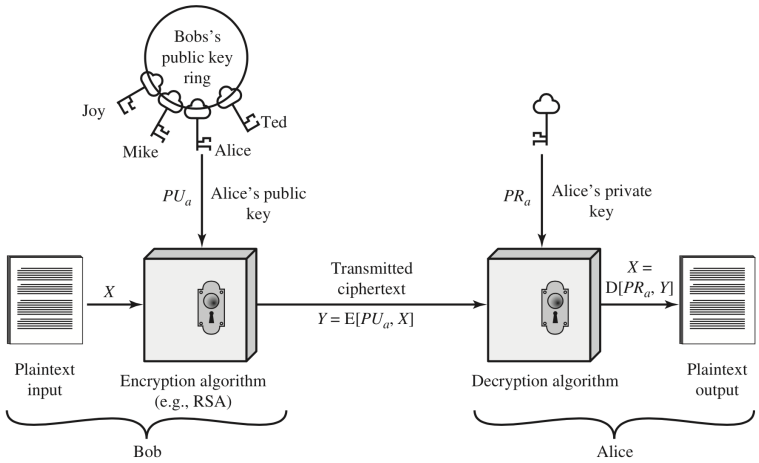



Figure 1.2: Asymmetric Encryption with Public Key. Source: [9]

2. both Alice and Bob place one of the two keys in a publicly accessible register; That is the Public Key, while the companion will be the Private Key, which will be kept private;

3. when Bob wants to send a private message to Alice, Bob encrypts the message using Alice's public key;

4. when Alice receives the message, she decrypts it using her private key. Other recipients cannot decrypt the message, because only Alice knows her private key.

Figure 1.2 shows a message encrypted using Alice's public key. This approach guarantees *confidentiality*, while Figure 1.3 shows a message encrypted using Bob's private key, providing *authentication* and *data integrity*.

### 1.3.2 End-to-End Encryption

*End-to-End Encryption*, shortened as E2EE, is a way to communicate privately where the only parties who can read the messages are the ones who are communicating. This is possible by encrypting and decrypting the messages directly on clients of the authorized users. No one can decrypt data, not even an attacker or the company who runs the service itself, without

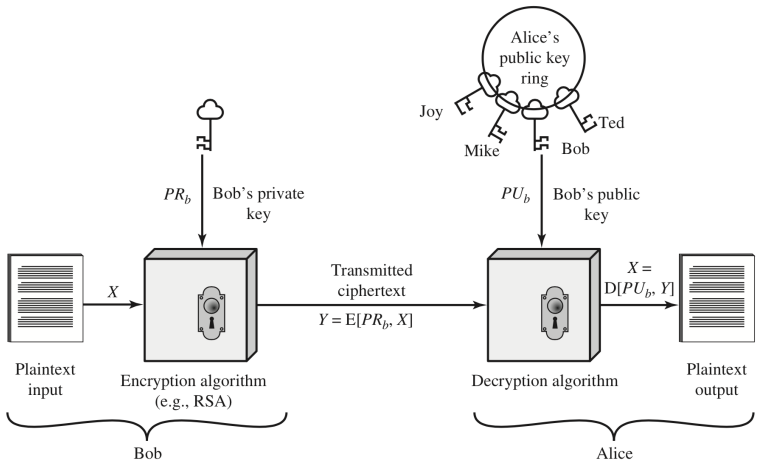



Figure 1.3: Asymmetric Encryption with Private Key. Source: [9]

having the private keys of the parties involved in the communication - that should remain private. Keys can be agreed by using a pre-shared secret, a one-time secret derived from the pre-shared one, or negotiating them by using the Diffie-Hellman Key Exchange method.

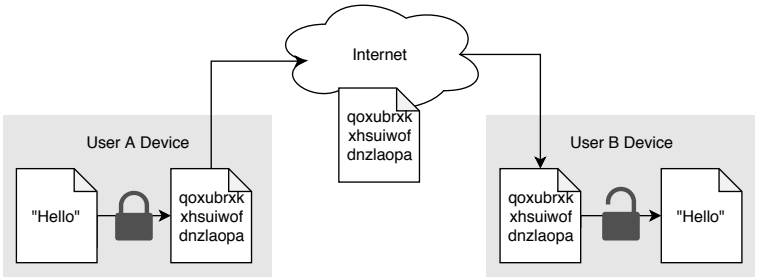



Figure 1.4: Functioning of End-to-End Encryption. Inspired from `https://oreil.ly/2IhYplf`

Even if E2EE guarantees privacy in communication, there are some challenges that should be taken into account. One of them is the possibility of Man-in-the-Middle attacks: rather than breaking the encryption, the sender may try to impersonate a recipient, by sending the message encrypted with their public key. A possible solution is to generate one-time strings based on the public keys of the two parties. In parallel, some security issues linked

to the device of the end user may arise. For example, a private key could be stolen from a device during an attack; or the content of a communication could be shared in other ways on a user's device, like screenshots or chat exports/backups. [20]

## 1.4 Secure messaging

We have seen that messengers exchange data between users, and encryption helps to hide such data from third unauthorized parties. With the increase in the use of messaging services, the need to secure the communications has grown during the last years. This is mainly due to greater user awareness on security issues, including data breaches and identity theft. Increased awareness has brought to the birth of secure messengers, designed from scratch with security in mind like Signal, Wire and Threema, and the implementation of end-to-end encryption to existing apps, like WhatsApp did in 2016.

According to [19], some of the features of a secure messaging service include:

- encryption of both texts and attachments that are exchanged;
- encryption of the content through a private key that stays on the device;
- implementation of *Perfect Forward Secrecy*, a property ensuring that, even if long term keys get compromised, new session keys will remain confidential;
- a recent audit by an independent company;
- a clearly documented service design;
- the type of cryptographic primitives used;
- availability of the code as open source for independent audits and reviews;
- manually verified fingerprints;

- impossibility to log, store and collect any plaintext message, metadata, session or event.

# Chapter 2

# Messaging Layer Security

## 2.1 Background

Each messaging service is characterized by different architectures, features and purposes. The path that led to the definition of Messaging Layer Security passed through several different concepts and protocols.

### 2.1.1 Fundamental properties: Perfect Forward Secrecy and Post-Compromise Security

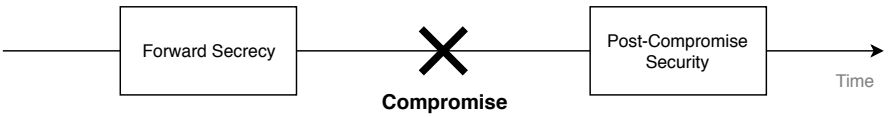



Figure 2.1: The temporal position of the two cryptographic properties of MLS: *Perfect Forward Secrecy* and *Post-Compromise Security*.

As we can see from Figure 2.1, protection about secrecy of past and future messages is guaranteed through two cryptographic properties: *Perfect Forward Secrecy* and *Post-Compromise Security*.

*Perfect Forward Secrecy* (shortened as PFS) means that in a compromised client, the secrecy properties - including access to all encrypted traffic history and current keying material - are guaranteed for messages older than the

oldest key of the client. Clients have an important role of deleting keys as soon as they have been used with the expected message, otherwise secrecy is considered weak.

*Post-Compromise Security* [6] (shortened as PCS) means that if a group member is compromised at a certain time $T$, but it tries to perform an update at a certain time $T'$, with $T' > T$, then all secrecy guarantees are applied for messages sent after $T'$. For instance, if an adversary learns all secrets known by Alice at $T$, and Alice wants to perform a key update at $T'$, the adversary is unable to violate the security properties after $T'$.

### 2.1.2 PGP and OpenPGP

PGP, acronym for *Pretty Good Privacy*, is a family of cryptographic softwares that uses a mix of symmetric-key cryptography, data compression, hashing and public-key cryptography to provide cryptographic privacy and authentication in online communication. It is used to encrypt and decrypt texts, files, emails and whole disk partitions. Despite the fact that this protocol guarantees confidentiality, authentication and integrity check, it does not provide Forward Secrecy and Post-Compromise Security. Furthermore, it does not guarantee deniability.

PGP was initially released by Phil Zimmermann in 1991 as proprietary software, but due to its rapid diffusion, in 1998 the specifications were collected by IETF, which led to the development of PGP as an open protocol, called *OpenPGP*.

### 2.1.3 Off-The-Record

*Off-the-Record* [13] is a cryptographic protocol designed by Ian Goldberg and Nikita Borisov in 2004. It makes use of a ciphersuite composed by *AES-128* as symmetric-key algorithm, *Diffie-Hellman Key Exchange (DHKE)*, and *SHA-1* as hash function. OTR uses a symmetric approach, so the same private key is used to both encrypt and decrypt a message. While this protocol guarantees end-to-end encryption of messages, mutual authentication, Perfect Forward Secrecy and non-repudiation, it does not provide Post-Compromise Security. Furthermore, the symmetrical nature of the algorithm

involves the use of *sessions*, making it suitable for use on instant messengers, but not on asynchronous messengers.

Moreover, one of the most important limitations of OTR is the fact that it can only be used in conversations between two users. The use with multiple clients was treated only in 2009, thanks to mpOTR, *Multi-Party Off-The-Record Messaging* [14]. While it introduces some improvements, there are other major pitfalls, like the lack of in-session Forward Secrecy, that was originally provided by OTR.

### 2.1.4 Double Ratchet Algorithm

The main goal of the MLS protocol is to guarantee an efficient way to manage multiple clients inside a group conversation. The current widespread solution that works efficiently with two participants is the *Double Ratchet Algorithm* [8]. Also referred to as *Axolotl*, Double Ratchet Algorithm is used by two parties to exchange encrypted messages based on a shared secret key. Developed in 2013 by Trevor Perrin and Moxie Marlinspike, it is the base of the *Signal Messaging Protocol* [7] by Open Whisper Systems, whose core is used right now by the most popular messaging platforms.

In a communication between two parties, new keys are derived for every message using the Double Ratchet Algorithm. This way, earlier keys cannot be calculated from the newest ones, thus ensuring Forward Secrecy. Diffie-Hellman public values are also attached to the message by both parties, so later keys cannot be calculated from the earlier ones, providing Post-Compromise Security. These properties protect earlier or late messages in case of a compromised key.

The cryptographic primitives used by the Double Ratchet Algorithm are:

- *Elliptic Curve Diffie-Hellman* (ECDH) with Curve25519 for the Diffie-Hellman ratchet;
- *Keyed-Hash Message Authentication Code* (HMAC) based on SHA-256 for message authentication codes;
- *HMAC* for the hash ratchet;

- *Advanced Encryption Standard* (AES) for symmetric encryption.

Right now, Double Ratchet is implemented with two different approaches:

#### Client-side fan-out

In client-side fan-out, messages in a group conversation are sent directly to the other clients. This way, assuming that $N$ clients take part in a group conversation, each client needs to send a message $N$ times. Clients are responsible of maintaining updated keys for each participant.

This approach has been adopted by Signal, Wire (whose implementation is called *Proteus*), Apple for iMessage and others.

#### Server-side fan-out (sender keys)

Another implementation based on the Signal Messaging Protocol is the *server-side fan-out*, also known as *sender keys*, used for WhatsApp groups and Facebook Messenger, and others. [17]

This is the same way as unencrypted messenger apps are implemented: a client who wants to send a message in a group conversation, transmits a single message to the server, which is then distributed $N$ times by the server to the $N$ different clients. Messages in groups are build on pairwise encrypted sessions, in order to achieve efficient server-side fan-out using sender keys.

## 2.2 The protocol

As we have seen, there are many secure messaging apps, which use similar protocols but with different implementations. The challenges they are called to solve are quite similar.

**Messaging Layer Security**, shortened as MLS, is a new security layer intended to provide end-to-end encryption in group conversations. Messaging services nowadays have to manage both one-on-one and group conversations: it is therefore useful to think of a protocol suitable for group communication, rather than point-to-point communication, in order to reduce the computational effort to encrypt and decrypt messages, and bandwidth required to

send them. Furthermore, another major constraint is asynchronous communication: services should be able to deliver messages, even if clients are offline. MLS is meant to protect against eavesdropping, tampering, and message forgery. The main goals of this protocol are:

- **Groups:** supporting large group conversations efficiently, theoretically up to 50.000 members;
- **Asynchronous:** taking into account that clients may not be online at the same time, including the worst-case scenario where none of the members in a group are available;
- **Security:** managing group membership providing Forward Secrecy and Post-Compromise Security with sub-linear scaling, instead of a linear one.
- **Formal verification:** similar to *Transport Layer Security* (TLS) [5], proving that the implementations satisfy the formal specification of the protocol;
- **Standardized:** providing a standard protocol in order to make MLS implementations interoperable with each other.

MLS is not intended to provide a full secure messaging protocol, but rather to offer security measures for concrete protocols. In this regard, the protocol does not specify a full, concrete implementation, but rather a set of data structures that can be mapped onto concrete encodings like TLS. Implementations that share common encodings could have a certain degree of interoperability, but they might not be compatible because of different authentication infrastructures.

The protocol is being designed by the *MLS Working Group*, a dedicated working group at the *Internet Engineering Task Force* (IETF). The group has already produced an MLS draft specification composed by two documents, that will be analyzed in the following chapters:

- an *architecture* document [1] that sets the domain, the problems and the requirements;

- a *protocol* document [2] that specifies the protocol itself.

A third document is under development and will focus on *federation* [3], describing the changes needed to allow different clients from the same or different entities to communicate with each other, and how the client will interact with the Delivery Service.

## 2.3 History

The history of this protocol is quite recent and still ongoing. It has its roots in 2015, when an increasing number of companies and researchers began to develop an interest in tree-based cryptographic schemes. The intention to converge on a shared protocol for end-to-end encryption applied to messaging arrived in 2016, from an informal meeting during the *IETF 96* conference in Berlin with people from Wire, Mozilla and Cisco.

In 2017, a paper by Facebook and the University of Oxford [4] introduced the concept of *Asynchronous Ratcheting Trees* (ART). During the last months of 2017, several workshops about MLS took place. The first informal meeting of the MLS Working Group, also known as *birds of a feather* (BoF), took place during the IETF 101 conference in February 2018 in London.

In May 2018, the MLS Working Group proposed *TreeKEM* [15], an alternative to ART more cryptographically efficient and better at handling concurrent changes, and they adopted it in the early drafts of the MLS protocol.

Right now, MLS is supported by several organizations like Mozilla, Facebook, Wire, Cisco, MIT, University of Oxford, INRIA, Google and Twitter. The protocol is still a work in progress: the specifications are in a state of "Internet-Draft", so they are informational documents and will be subject to changes in the upcoming months.

## 2.4 Performances of MLS

MLS provides five different possible operations for a group conversation:

- create an empty conversation;

- add a new client to an existing conversation;
- update of the client secrets;
- remove a client from a conversation;
- send a message inside a conversation.

Figure 2.2 shows a comparison in terms of computational complexity for each of the operations previously described, between the two current solutions based on the Double Ratchet Algorithm (client-side fan-out and server-side fan-out) and the latest version of the MLS Protocol Draft.

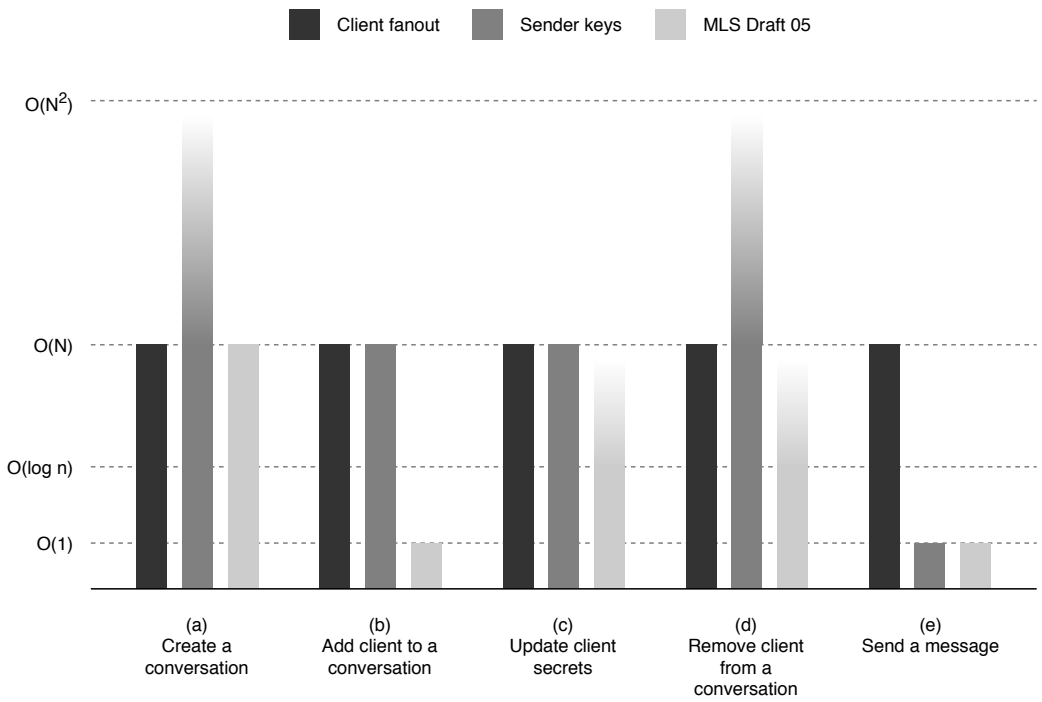



Figure 2.2: Performance comparison of main actions, in terms of complexity, between Client fanout, Sender keys and MLS Draft 05

## 2.5 Current implementations

Companies that are part of the Working Group have started working on some implementations of MLS, written in different programming languages and based on different versions of the draft. Since MLS aims to become an IETF Internet Standard, other companies outside of the Working Group have started showing interest on the matter and are working on their own implementations. A non exhaustive list, that may change over time, is the following:

**Melissa** The proof of concept implementation carried out by the Wire team and object of this thesis. It is written in Rust and publicly available at `https://github.com/wireapp/melissa`. It is based on the draft 05 version of the protocol.

**mlspp** The draft implementation made by Cisco and written in C++. Available open source at `https://github.com/cisco/mlspp`.

**MLS*** This is an implementation written in F* by the *Institut National de Recherche en Informatique et en Automatique* (INRIA).

**Molasses** This is another early implementation written in Rust and based on the draft 04 version of the protocol. It is carried out by Trail of Bits and open sourced at `https://github.com/trailofbits/molasses`.

**RefMLS** An implementation written in JavaScript by the *New York University* in Paris.

**Google** Google is also working to an implementation written in C++.

# Chapter 3

# Messaging Layer Security Architecture

The general architecture of Messaging Layer Security includes the specification of the general setting of the environment, the functional and security requirements, and considerations. These concepts are treated at a higher level than the protocol and they are described in [1]. While the protocol is under development, the architecture is the basis of the entire specification, therefore it is possible to affirm that it will not undergo great changes. We will consider the version 02 of the document in this analysis.

## 3.1 General setting

### 3.1.1 Messaging Service

The base entity is the Messaging Service, shortened as MS. A Messaging Service is a service that provides messaging features to users, that usually have one or more devices, called *clients*. In a Messaging Service, users can exchange messages in a one-on-one conversation or with other users in a group conversation.

MLS calls *members* the set of participants of a Messaging Service and consider one-on-one conversation as group conversations between two people. This is the basic case, since conversations can have up to thousands of

members. The Messaging Service consists of two services that allow clients to send and receive messages correctly, as represented in Figure 3.1:

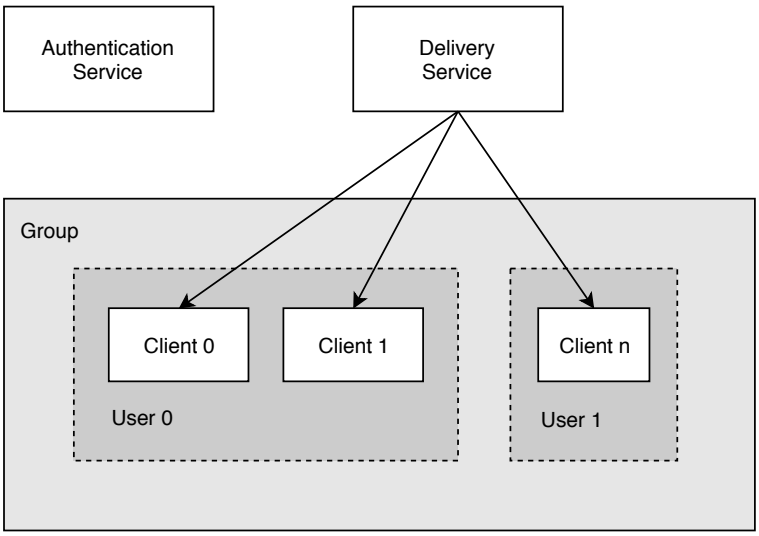



Figure 3.1: The general structure of a Messaging Service.

- an **Authentication Service**, shortened as AS, which is responsible of maintaining user identities and to manage the authentication, approving or rejecting user access to the service. It is also responsible of issuing credentials to new users and it could provide other services like user discovery;
- a **Delivery Service**, shortened as DS, which is responsible to receive and distribute messages to the group members. In group conversations, DS may be responsible of broadcasting messages to all members of the group. It also stores and delivers initial public keys that are required by the clients to proceed with the establishment process of the group secret key.

Both services may be maintained by the same entity or not. They are logically independent: users may use a Delivery Service with an identity issued by a third-party Authentication Service, like a Single Sign-On service.

A typical scenario of a messaging service may be the following:

1. Alice, Bob and Charlie create three different accounts in a Messaging Service, obtaining credentials from the Authentication Service;

2. Alice, Bob and Charlie authenticate to the Delivery Service and store some initial keying material, which can be used to send encrypted messages for the first time. Keying material is authenticated with their long term credentials;

3. when Alice wants to send a message in the conversation, she contacts the Delivery Service and looks up their initial keying material. These keys are then used to create a new set of keys that she can use to send encrypted messages to Bob and Charlie. In the end, she sends encrypted messages to the DS, which forwards them to the recipients;

4. Bob and/or Charlie respond to Alice's message. Their Update messages trigger a new key derivation that allows the shared group key to be updated. This is a required step, in order to guarantee Post-Compromise Security.

### Definitions

As we can see from Figure 3.1, a *group* is basically a set of two or more users called *members* who can interact with the Messaging Service using one or more *clients*.

A *client* is an end-user device designed to access a conversation through an underlying platform, like web, desktop or mobile. Each client owns one or more long-term identity key pairs that uniquely define their identities to other clients in a group. Clients are able to create a group by inviting other users, add or remove users from an existing group, join or leave groups, send and receive messages from/to other members in a group.

A *group* is the set of clients that knows the shared group secret established during the group key establishment. Multiple clients belonging to a user can be grouped together, appearing as one virtual client to the rest of the group. In MLS there is no single "administrator" of a group: any client can add other clients to a conversation. Restrictions can be enforced with access control at the application layer.

### 3.1.2 Authentication Service

The Authentication Service provides a map between user identities and long-term identity keys. As mentioned in Chapter 1.1.1, a user identity can be identified in several ways, like email address, username, phone number, or a unique identifier. The Authentication Service has two main responsibilities: it is the *certification authority* of the Messaging Service, which signs some credentials that link an identity to a key. It is also the *directory server* that provides keys for a given identity. On a security point of view, the connection with the Authentication Service might be secured through a transport security protocol like TLS.

The Authentication Service has a great responsibility in ensuring the authenticity of an identity. In fact, a malicious AS could impersonate any user of the system. To avoid that, it is possible to publish the binding between identities and keys in a public log, such as *Key Transparency*. While this is not required by MLS, it is mandatory to avoid malicious attacks.

#### Key Transparency

*Key Transparency* is an open source library provided by Google [16]. It is an application of a public log to help clients to build trust among them. This happens by providing a public audit record of all changes happened to data, including all the public keys of the actual recipients associated to an account, the times an account was updated and who has updated it. Everything happens preserving privacy.

### 3.1.3 Delivery Service

As with the Authentication Service, the Delivery Service also has multiple responsibilities within a Messaging Service. It is the directory service for the initial keys of the clients, allowing clients to establish a shared key and send encrypted messages to another client, even if it is offline. Delivery Service is also responsible of routing messages between clients and broadcasting them to multiple clients.

### Key handling

Initial cryptographic keys are authenticated using the key of the client and stored within the Delivery Service. These keys will be used in the next steps to establish the shared group secret. Since a user may have multiple clients, each with its own keying material, there might be multiple entries stored by each user. Delivery Service is also responsible for allowing users to manage their initial keys.

When a client wants to create a group and send an initial message, it retrieves the initial keys from the Delivery Service, verifies them using the identity key, and creates the group secret that can be used for message encryption.

### Content delivery

The architecture of MLS assumes that the Delivery Service provides:

- **reliable delivery**: a message sent to the Delivery Service should be delivered to all clients, even if they are not currently available for delivery;
- **in-order delivery**: messages must be delivered in the same order as they are received from a given client, and approximately in the same order in which they are sent by clients - this might happen because multiple clients can send messages at the same time;
- **consistent ordering**: the Delivery Service must ensure that all clients have the same message ordering of operations that are relevant for cryptography, while MLS provides causal consistency of the messages for each sender. Otherwise, it might cause cryptographic errors.

Delivery Service may provide some kind of ordering information to ensure that messages are delivered in the correct order. The protocol itself can verify these properties by detecting inconsistencies in the order of messages, but it does not provide mechanisms to recover from this situation.

Some forms of misbehavior in the Delivery Service could be possible and difficult to detect. Without other side information, clients may not distinguish a Denial of Service attack.

#### Membership

MLS is designed so that neither the Delivery Service nor the Authentication Service know which clients are inside which group. They still might learn this information in different ways under a server-side fan-out model, like traffic analysis or server-stored lists. In the first case, it might be possible to analyze which client has sent the same message to different clients. In the latter, a group membership list could be stored in a server within the Messaging Service.

In addition, since one of the major requirements of MLS is to work asynchronously with online and offline clients, clients may still be holding old keys. This is a problem with two other major constraints, Forward Secrecy and Post-Compromise Security, because they rely on deletion and replacement of keys. Right now, MLS does not provide any specification to solve this problem, but systems that will implement the specification can enforce some mechanism for doing so.

## 3.2 Requirements

As mentioned before, MLS is a group messaging protocol designed to scale easily, from a group involving two up to approximately 50.000 clients, hence maintaining good performance and safety for the users.

### 3.2.1 Functional requirements

Functional requirements include all the features needed in an MLS application in order to exchange messages consistently. They are:

#### Asynchronous usage

All the operations made by clients in a group conversation, including updating keys, adding or removing members, sending messages, should happen asynchronously, assuming that no other client is online simultaneously. Clients do not wait for another reply from a user. The underlying transport layer has to support asynchronous and reliable message delivery.

### Recovery after state loss

Participants in a group conversation whose local state is lost or corrupted can reinitialize their state and continue to participate in the conversation without being removed from the group.

### Support for multiple devices

Users should be able to use multiple devices within the same Messaging Service. New clients are added to a conversation using a previously shared key secrets or a new key. They will not gain access to the previous history of that conversation: history recovery is not allowed by MLS, but may be implemented outside of the scope of this protocol.

Two alternatives regarding the management of devices are currently being analyzed. The first one is treating every device as one participant occupying a leaf in the tree, and then logically grouped as one member by the Messaging Service. The second one is treating every member as one participant occupying a leaf, pairing virtual devices to the member in some other way.

### Extensibility

Messages not affecting the group state can carry an arbitrary payload in any format (e.g. plaintext, JSON, binary, etc.) that can be consumed among group members.

### Privacy

Metadata might be subject to traffic analysis, especially if unprotected. The protocol aims to reduce the metadata footprint on the server side. DS persists just the data needed for message delivery, avoiding to carry any personal information or other sensitive metadata. A Messaging Service that controls both Authentication Service and Delivery Service cannot correlate the delivered messages to the initial public keys.

#### Federation

MLS aims to be compatible with federated environments: multiple implementations can interoperate to form federated systems, if they use compatible message encodings. Federation is an interesting topic currently under analysis by the MLS Working Group and treated in the MLS Federation document [3] of the specifications.

#### Future compatibility

Multiple versions of the protocol should be able to coexist and cooperate in the future. MLS offers a version negotiation mechanism that prevents version downgrade attacks where an attacker would actively rewrite messages with a lower protocol version than the one supported by the endpoints. When multiple versions are available, negotiation guarantees that the version agreed upon will be the highest version supported by the group. Negotiation is usually performed during group creation by fetching the `UserInitKey`s and checking the highest version number.

### 3.2.2 Security requirements

Security requirements include everything needed to ensure secrecy, authentication and security of communication within a conversation. These are:

#### Connections between client and servers

The protocol assumes that all transport connections are secured through a security protocol in the transport layer, such as TLS. However, intrinsic safety of MLS is still guaranteed in case the transport layer gets compromised.

#### Message secrecy and authentication

MLS provides secrecy, integrity and authentication for all messages. *Secrecy* means that a message can only be read by the participants of the group conversation where the message is sent, even in the context of an active adversary. *Message integrity* and *authentication* mean that clients can only

accept messages if sent by a group member from a particular client, refusing to accept messages sent as a different client.

Furthermore, Authentication Service and Delivery Service cannot read messages sent between members of a group, because they are not part of the group. MLS provides optional protections in order to avoid traffic analysis, like messages padding. This way, all ciphertexts are of a standard length, reducing the ability of adversaries to understand the length of messages. Message content can be deniable if the signature keys are exchanged over a deniable channel prior to signing messages.

### Forward Secrecy and Post-Compromise Security

Forward Secrecy and Post-Compromised Security, as already mentioned and explained, are two of the security requirements provided by MLS.

Message encryption keys are derived via a hash ratchet, which provides a form of Forward Secrecy: learning a message key does not reveal previous message or root keys.

Post-compromise security is provided by Update operations, in which a new root key is generated from the latest ratcheting tree. If the adversary cannot derive the updated root key after an Update operation, it cannot compute any derived secrets. Keys are partially generated from the Update message itself, as explained in Chapter 4.8.

### Membership changes

Membership of a group in MLS is managed through agreement. This means that all group members have to agree on the list of members. All members are informed about addition or removal of other members. Once a client is part of a group, the set of devices controlled by the user can only be altered by an authorized member of the group. The authorization is managed by the application, as well as Access Control Lists that allows addition or removal of members to certain members. Members who are removed from a group do not have particular privileges: compromise of a former group participant does not affect the security of the messages sent after the removal, but might affect previous messages if group secrets have not been deleted.

#### Security of attachments

Security properties expected for attachments are similar to the ones expected from messages, except that the download time for attachments might be way longer than that of messages. This means that, in the same way, lifetime of cryptographic keys is higher than for messages, thus slightly weakening the Post-Compromise Security guarantees for attachments.

#### Non-repudiation vs deniability

MLS provides strong authentication within a group. This means that a group member cannot impersonate other members and send messages with another identity. Furthermore, recipients are able to prove that a message was sent by a given client, otherwise the user can report an abuse to the Messaging Service. This verification is usually provided by a third party (*non-repudiation*), but it should also be possible to operate in a *deniable* mode where a proof is not possible. How to supply this is right now an open issue on the protocol side.

## 3.3 Security considerations

MLS assumes that the attacker has a complete control of the network. The protocol provides the security services in front of such attackers. These guarantees have to degrade in presence of compromise of the transport security links and/or clients and elements of the messaging system.

#### Delivery service compromise

MLS provides strong guarantees in case the Delivery Service gets compromised. Even if totally compromised, it should not be able to:

- read messages;
- inject messages that will be acceptable to legitimate clients;
- undetectably remove, reorder or replay messages.

The Delivery Service can mount DoS attacks where it can refuse to forward any messages or specific messages. DoS are partially detectable by clients without an out-of-band channel. The Delivery Service could provide stale keys as initial keys to the client. This does not lead to compromise of the message stream, but it can attack forward security. The solution is to set an expiration to the initial keys.

#### Authentication Service compromise

A compromised Authentication Service could provide incorrect or adversarial identities to clients. This could be mitigated with some kind of transparency/logging mechanism.

#### Client compromise

MLS provides limited protection against compromised clients. In that case, an attacker is able to decrypt messages for groups which the client is a member of and send messages impersonating the compromised client, unless the client updates its keying material. Secrecy is guaranteed in the past and in the future by Forward Secrecy and Post-Compromise Security, already explained in Chapter 2.1.1.

In addition, a client cannot send a message to a group which appears to be from another client with a different identity. Devices from the same user that share keying material will be able to impersonate another device.

# Chapter 4

# Messaging Layer Security Protocol

The protocol document [2] shows how MLS achieves the architectural specification expressed before. We have seen two main cryptographic constraints of the protocol, Forward Secrecy and Post-Compromise Security. In this section, we will see how they are guaranteed in MLS and the way all requirements take shape. This protocol part is based on the draft version 05.

## 4.1 The basics of MLS

MLS is based on the work brought on *Asynchronous Ratcheting Trees* already discussed and explained in [4]. ART has been replaced by TreeKEM [15], a more efficient solution that better handles concurrent changes and tree scaling. Double Ratchet is not efficient for use with bigger groups over networks with low bandwidth.

In fact, for groups with two or more clients, a common strategy is to broadcast symmetric sender keys over shared symmetric channels. Then, each client can send messages in the group with their own sender key. While this can provide Forward Secrecy, it makes it difficult to achieve Post-Compromise Security. An adversary who learns a sender key could potentially eavesdrop messages sent from a member. Generating new sender keys is a solution, but it requires a high computation cost that scales linearly with the group size.

The protocol expects the Messaging Service to provide a *long term identity key provider* - the Authentication Service, a *broadcast channel* for each group that will relay messages to all clients, and a *directory* where clients can publish and get initial keys. The information stored by each client includes both private and public data, and it is called *state*.

The protocol describes four major operations - that are the main operations for group chats:

- initialization;
- adding a member;
- updating member's secret;
- removing a member.

### 4.1.1 Initialization and member addition

An initial state is set up by the group creator with the *Init* message and it is based on information pre-published by clients inside the directory. When exchanging messages, clients produce new shared states that are linked to the predecessors, forming a *Direct Acyclic Graph* (DAG).

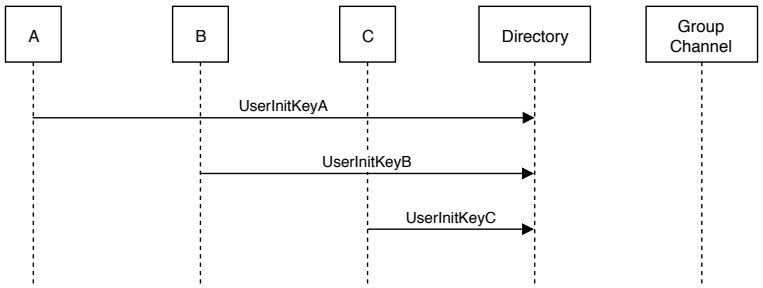



Figure 4.1: Pre-initializing phase: each client publishes a *UserInitKey* inside the directory.

As we can see from Figure 4.1, before the initialization of a group, each client publish its `UserInitKey` inside the directory of the Messaging Service.

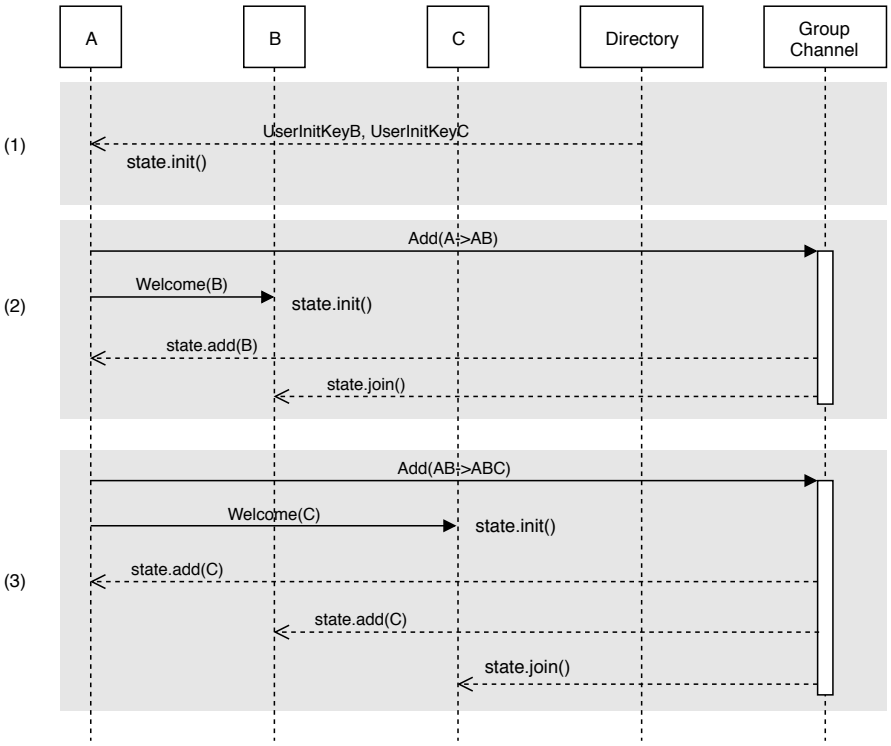



Figure 4.2: Initialization of a group conversation in MLS.

Any client $A$ who wants to create a group with other clients $B$ and $C$, needs to download the `UserInitKey`s of both clients from the directory (1). After that, it creates an empty state within itself and uses the `UserInitKey`s to compute the *Add* and *Welcome* messages for $B$ and $C$. *Add* messages are used to indicate that a new user has been added to the group, while *Welcome* messages are intended to inform the new user that they are added to the group and can start sending messages (2). The procedure is repeated for every new group member (3) and is graphically described in Figure 4.2. Clients cannot get back in time before being added to the group, because new secrets are generated at every new epoch.

### 4.1.2 Periodic updates

In order to provide Forward Secrecy and Post-Compromise Security, each member periodically updates their leaf secret, which represents a contribution

to the group secret.

Any member willing to update their secret can do so by generating a new leaf secret and send an *Update* message, as we can see from Figure 4.3. Once all members have processed the message, the group secrets will be unknown to attackers. The decision about the refresh interval for updates are up to the application.

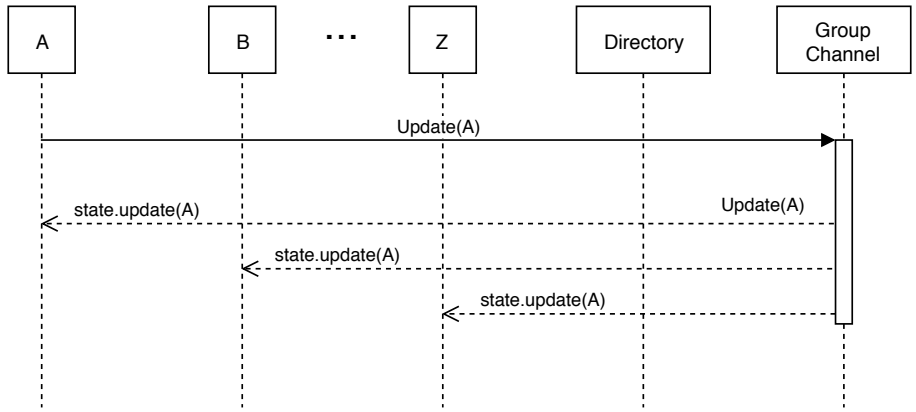



Figure 4.3: Periodic update of secret keys.

### 4.1.3 Removing a member

Members are removed in a similar way, as illustrated in Figure 4.4. Any member can generate a *Remove* message regarding another user (e.g. user A who wants to remove user B) that can be consumed from other participants - except by the removed user - to update their internal state. This will trigger an update of the secrets that will not be delivered to the removed user, so they will not be able to send new messages to the group.

## 4.2 The logic behind MLS: Ratchet Trees

MLS makes use of *Ratchet Trees* [4] to derive shared secrets between multiple clients. Ratchet Trees are left-balanced binary trees: this means that every node of the tree except leaves has two children - *left* and *right*. In

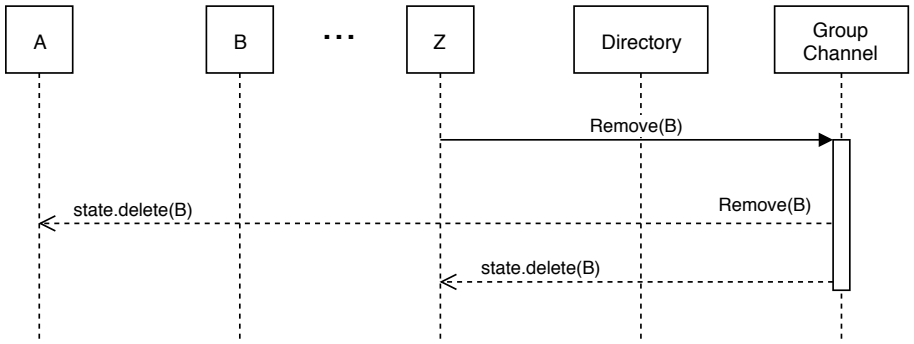



Figure 4.4: Removing a member from a group.

addition to the peculiarities of this type of trees, Ratchet Trees inherit all the relational properties of a normal tree.

A tree can have multiple subtrees, that is, a part of the tree given by the descendants of a node, which becomes the head of the subtree. The size of a tree is the number of leaf nodes it contains.

A tree can be considered *left-balanced* if, for every parent, the subtree is fully balanced or the largest full subtree is the one positioned on the left side. This means that the left subtree is always the one that fills before. A tree is fully balanced when its size is a power of two and both right and left subtrees have the same size.

The *direct path* of a node is the concatenation of the node with the direct path of its parent. The *copath* is the list of siblings of a node in its direct path. The *frontier* is the list of heads of the maximal full subtrees of the trees, ordered from left to right.

Each node has a *node index*, starting at zero and running from left to right, as we can see from Figure 4.5. Since protocol messages only need to refer to the leaves of the tree, they are indexed with a proper index called *leaf index*, always numbered from left to right. Given this numbering, leaf indices are always half of the respective node index, and leaf nodes always have an even node index.

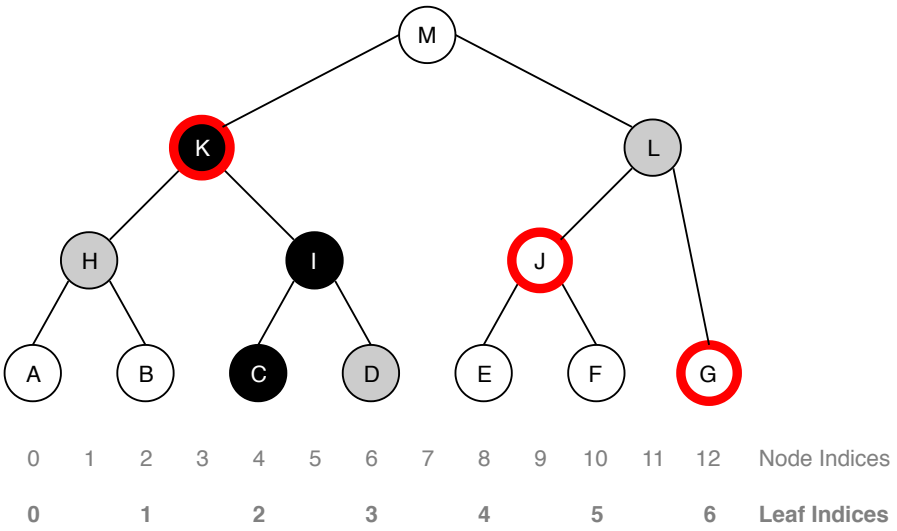



Figure 4.5: A left-balanced tree, with node and leaf indices. Direct path is $C, I, K$ and is marked with black nodes, copath is $H, D, L$ and is marked in gray and frontier is $K, L, G$ and is marked with a red, bolder line on involved nodes.

### 4.2.1 Ratchet Tree nodes

Ratchet Trees are used to generate shared group secrets. A particular instance of a Ratchet Tree is based on these cryptographic primitives:

- a *Hybrid Public Key Encryption* (HPKE) ciphersuite, which specifies a *Key Encapsulation Method* (KEM), an AEAD encryption scheme, and a hash function;
- a *Derive-Key-Pair* function that produces an asymmetric key pair from a symmetric node secret.

Each node of a ratchet tree contains up to three values:

- a private key, within direct paths;
- a public key;
- a credential, in leaf nodes.

### 4.2.2 Blank nodes and resolution

A Ratchet Tree node may be *blank*: this means that no value is present in that node. The *resolution* of that node is an ordered list of non-blank nodes that cover all non-blank descendants. Nodes are ordered according to their indices. The three possible cases of resolution of a node $X$ are:

- non-blank node: $res(X) = \{X\}$;
- blank leaf node: $res(X) = \{\}$;
- blank intermediate node: $res(X) = \{res(left(X)), res(right(X))\}$

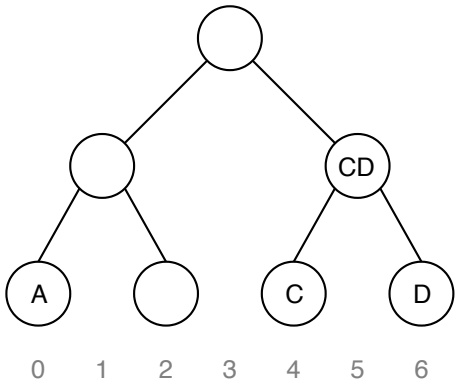



Figure 4.6: A binary tree with some blank nodes (1, 2 and 3).

In the example shown in Figure 4.6, we can see that:

- $res(5) = \{C, D\}$
- $res(2) = \{\}$
- $res(3) = \{A, CD\}$

Every node, including blank nodes, contains a *hash* that summarizes the content of the subtree below that node.

### 4.2.3 View of a Ratchet Tree

MLS assumes that each participant has a complete and up-to-date view of the public state of the group ratchet tree, including public keys and credentials of the leaf nodes. Instead, the secret state is known only to the leaf nodes, which represent the members of the group. This way, no participant in a group has full knowledge of the secret state of the tree - including private keys.

MLS maintains the member views of the tree in a way to maintain the tree invariant. The private key for a node is known to a member of the group only if their leaf is a descendant or equal to the node. This way, each member knows the private keys only for nodes in its direct path.

### 4.2.4 Ratchet Tree updates

The contents of a parent node are based on the latest updated child and are computed from one of its children:

Listing 4.1: Contents of a parent node computed from childrens

```
path_secret[n] = HKDF-Expand-Label(path_secret[n-1], "path", "",
    Hash.Length)
node_secret[n] = HKDF-Expand-Label(path_secret[n], "node", "", Hash
    .Length)
node_priv[n], node_pub[n] = Derive-Key-Pair(node_secret[n])
```

As we can see from Figure 4.7, if participants join a group with leaf secrets $A$, $B$, $C$ and $D$ in this order, the resulting tree will have $KDF(D)$ as parent of $C$ and $D$ and, in turn, its parent will be $KDF(KDF(D))$ (a). If the second participant changes its leaf secret to $X$, the parent secrets will change according to this update (b).

### 4.2.5 Tree view synchronization

Members of a group have to keep their view of the tree synchronized and up to date. When adding or removing clients, an handshake message containing public values for intermediate nodes in the direct path is transmitted.

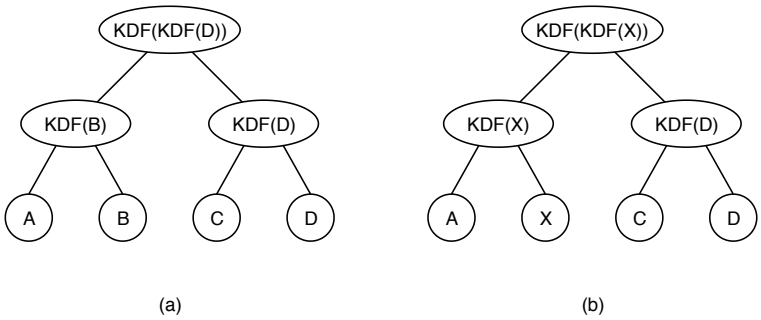



Figure 4.7: An example of leaf secrets update.

Other members can use these public values to update their view of the tree. The member that proposes an update broadcasts a set of values along the direct path of a leaf, as well as the root. These values are the public key of the node and zero or more encrypted copies of the path secret corresponding to the node. Other members can use these nodes of the direct path to update their own view of the tree.

The path secret value is encrypted for the subtree corresponding to the non-updated child of the parent. There is one encrypted path secret for each public key in the resolution of the non-updated child.

The recipient of an update processes it through the following steps:

- Compute the updated path secrets

    1. identify a node in the direct path for which the local member is in the subtree of the non-updated child;
    2. identify a node in the resolution of the copath node for which this node has a private key;
    3. decrypt the path secret for the parent of the copath node using the private key from the resolution node;
    4. derive secret values for ancestors ofthat node using the algorithm described above;
    5. recipient should verify that the received public keys agree with the ones derived from the new `node_secret` values;

- Merge the updated path secrets into the tree
  1. replace the public keys for nodes on the direct path with the received public keys;
  2. for nodes where an updated secret was computed in step 1, replace the secret value for the node with the updated value.

As we can see from Figure 4.7, giving $pk(X)$ the public key of $X$ and $E(K, S)$ the public-key encryption with public key of $K$ of the secret value $S$, when an update is made along the direct path $B - E - G$, the following values will be transmitted by the sender:

- for $pk(G)$, ciphertexts $E(pk(C), G), E(pk(D), G)$
- for $pk(E)$, ciphertext $E(pk(A), E)$
- for $pk(B)$: {}

## 4.3 Ciphersuites

MLS sessions use a single ciphersuite that specifies the following primitives:

- a hash function;
- a Diffie-Hellman finite-field group or elliptic curve;
- an *Authenticated Encryption with Associated Data* (AEAD) encryption algorithm, as described in [10];
- a *Derive-Key-Pair* (DKP) algorithm that maps octets with the same length as the output of the hash function to key pairs for the asymmetric encryption scheme.

Public keys are opaque values in a format defined by the ciphersuite. The two types used are `HPKEPublicKey` and `SignaturePublicKey`.

Implementations may use one of the two ciphersuites supported by MLS and described below. More ciphersuites will be supported in the future. Ciphersuites and signature schemes are defined as follow:

Listing 4.2: Definition of SignatureScheme and CipherSuite

```
1  enum {
2      ecdsa_secp256r1_sha256(0x0403),
3      ed25519(0x0807),
4      (0xFFFF)
5  } SignatureScheme;
6
7  enum {
8      P256_SHA256_AES128GCM(0x0000),
9      X25519_SHA256_AES128GCM(0x0001),
10     (0xFFFF)
11 } CipherSuite;
```

### 4.3.1 Curve25519, SHA-256 and AES-128-GCM

This ciphersuite uses *Curve25519* as Diffie-Hellman group, *SHA-256* as hash function and *AES-128-GCM* as AEAD algorithm. Given an octet $X$, the private key produced by the DKP operation is `SHA-256(X)`. The public key is `X25519(SHA-256(X), 9)`.

*Curve25519* is an elliptic curve offering 128 bits of security. It is designed for use with Diffie-Hellman Elliptic Curve key agreement scheme (ECDH) and it is one of the fastest elliptical curve cryptographies not covered by patents. The DH function name is *X25519*. The curve used by Curve25519 is $y^2 = x^3 + 486662x^2 + x$, a Montgomery curve, over the prime field defined by $2^{255} - 19$ and using $x = 9$ as base point.

*SHA-256* (*Secure Hash Algorithm-256*) is a revision of the SHA-1 algorithm that was originally developed by the National Institute of Standards and Technology (NIST) in 1993. While the algorithm is pretty similar to SHA-1, this revision mainly differs for the message digest size - 256 bits instead of 160 - and the number of steps needed - 64 instead of 80. [9]

*AES-128-GCM* is a cipher block algorithm developed by Vincent Rijmen and Joan Daemen, adopted by NIST and intended to replace the old DES and Triple DES algorithms. It is based on a block length of 128 bits and a key length of 128, 192 or 256 bits. It is combined with GCM, the Galois/Counter

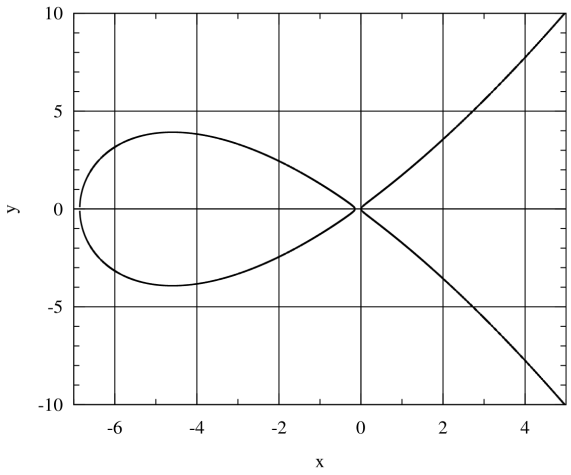



Figure 4.8: Curve25519 elliptic curve

Mode of Operation for cryptographic block ciphers.

### 4.3.2 P-256, SHA-256 and AES-128-GCM

This ciphersuite differs from the previous one for the usage of *P-256* as Diffie-Hellman group. Given an octet $X$, the private key derived by DKP is `SHA-256(X)` interpreted as a big-endian integer. The public key is the result of multiplying the P-256 base point by this integer.

*P-256* is a widely used curve offering 128 bits of security developed by NIST. The curve is described by the function $y^2 = x^3 - 3x + 4105836372515214212932612978004726840911444101599372555483525631403946740129 1$, with modulo $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$. According to [23], P-256 is not considered safe, due to some lacks on the *elliptic-curve discrete logarithm problem* (ECDLP). ECDLP is the problem of finding an ECC user's secret key, given the user's public key.

## 4.4 Credentials

A group member authenticates the identities of other participants with credentials issued by an authentication system. Any credential must express:

- the public key of a signature key pair;
- the identity of the private key holder;
- the signature scheme used by the holder to sign MLS messages;
- some kind of information that allows a party to verify identities (optional).

Listing 4.3: Definition of CredentialType, BasicCredential and Credential types.

```
enum {
    basic(0),
    x509(1),
    (255)
} CredentialType;

struct {
    opaque identity<0..2^16-1>;
    SignatureScheme algorithm;
    SignaturePublicKey public_key;
} BasicCredential;

struct {
    CredentialType credential_type;
    select (credential_type) {
        case basic:
            BasicCredential;
        case x509:
            opaque cert_data<1..2^24-1>;
    };
} Credential;
```

## 4.5 Tree hashes

Group members can agree on the cryptographic state of the group by generating a hash value that represents the contents of the group ratchet tree and the member's credentials. The hash of the tree is the hash of its root node, defined recursively from the leaves. The hash of a leaf is the hash of the `LeafNodeHashInput` object. At the same time, the hash of a parent node including the root, is the hash of a `ParentNodeHashInput` object.

Listing 4.4: Definition of LeafNodeInfo, LeafNodeHashInput and ParentNodeHashInput types.

```
1  struct {
2      HPKEPublicKey public_key;
3      Credential credential;
4  } LeafNodeInfo;
5
6  struct {
7      uint8 hash_type = 0;
8      optional<LeafNodeInfo> info;
9  } LeafNodeHashInput;
10
11 struct {
12     uint8 hash_type = 1;
13     optional<HPKEPublicKey> public_key;
14     opaque left_hash<0..255>;
15     opaque right_hash<0..255>;
16 } ParentNodeHashInput
```

Within `LeafNodeHashInput`, the `public_key` and `credential` fields represent the leaf public key and the credential for the member of that leaf. `info` is null when the leaf is blank.

For `ParentNodeHashInput`, `left_hash` and `right_hash` fields hold the hashes for the left and right children. `public_key` holds the hash of the public key stored in the node, which is null if the node is blank.

## 4.6 Group state

Each member of the group maintains a representation of the state of the group, that is updated after any operation made by other group participants. The state is composed by these fields:

- `group_id`: an unique identifier of the group. Once instantiated, it never changes;
- `epoch`: the current version of the group key. It is incremented by one for each `GroupOperation` processed;
- `tree_hash`: contains a commitment to the contents of the group ratchet tree and the credentials for the group members, as expressed in Chapter 4.5. The hash is updated to represent the current tree and credentials;
- `transcript_hash`: contains the list of `GroupOperation` that led to this state. It is updated as follows:

  `transcript_hash_[n] = Hash(transcript_hash_[n-1] || operation)`

  When a new one-member group is created, this field is set to an all-zero vector of length equal to `Hash.length`.

## 4.7 Direct paths

Each MLS message needs to transmit node values along the direct path of a leaf. The path contains a public key for the leaf node, and a public key and encrypted secret value for intermediate nodes in the path. Path is ordered from the leaf to the root.

Listing 4.5: Definition of HPKECiphertext, RatchetNode and DirectPath types.

```
1 struct {
2     HPKEPublicKey ephemeral_key;
3     opaque ciphertext<0..2^16-1>;
4 } HPKECiphertext;
```

```
5
6  struct {
7      HPKEPublicKey public_key;
8      HPKECiphertext encrypted_path_secrets<0..2^16-1>;
9  } DirectPathNode;
10
11 struct {
12     DirectPathNode nodes<0..2^16-1>;
13 } DirectPath;
```

The length of the secrets is zero for the first node in the path, and equal to the length of the resolution of the corresponding copath node for the remaining elements. `HPKECiphertext` values are computed according to Hybrid Public Key Encryption. [11]

Decryption is performed in the corresponding way, using the private key of the resolution node and the ephemeral public key transmitted in the message.

## 4.8 Key schedule

Group keys are derived using *HMAC-based Extract-and-Expand Key Derivation Function* (HKDF) [12]. The hash function used by HKDF is the ciphersuite hash algorithm. Figure 4.9 explains the functioning of the key schedule.

Each Epoch Secret is combined by the Extract function of HKDF, which takes the Update Secret of the current epoch as argument and it is salted using the Init Secret from the previous epoch.

The Application Secret, the Confirmation Key and the next Init Secret take the secret argument from the incoming arrow of the diagram, along with the `GroupState` of the current epoch, to derive new epoch secrets.

### 4.8.1 Encryption keys

MLS encrypts three types of information: metadata about the sender, handshake messages and application messages. Metadata used to lookup the key for encryption is encrypted under AEAD with a random nonce and the `sender_data_key` derived from the `sender_data_secret`:

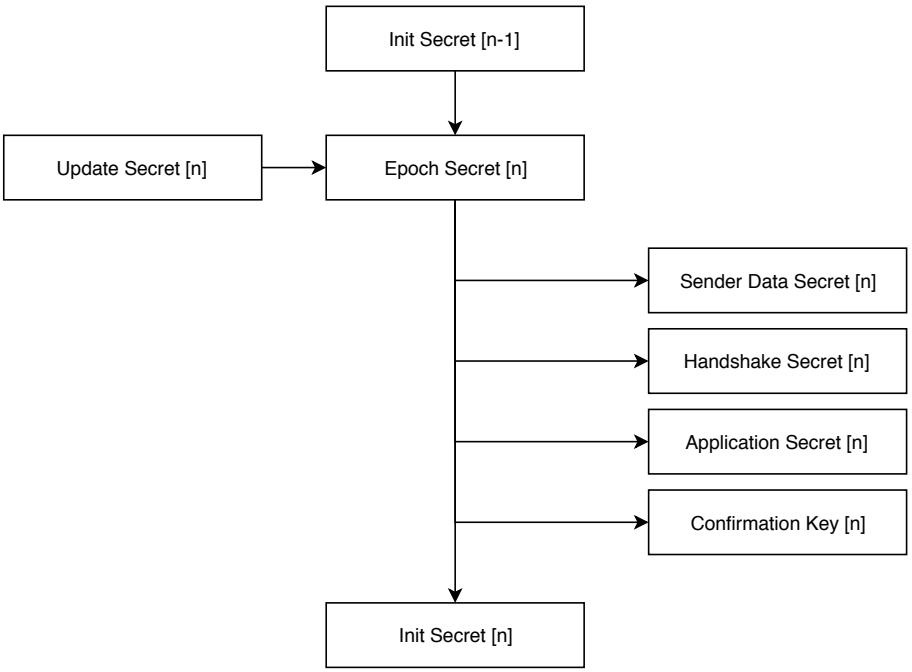



Figure 4.9: Functioning of key schedule

```
1 sender_data_key =
2 HKDF-Expand-Label(sender_data_secret, "sd key", "", key_length)
```

Handshake messages are encrypted using a key and a nonce derived from the `handshake_secret` for a specific sender, in order to prevent multiple senders to perform this way:

```
1 handshake_nonce_[sender] =
2 HKDF-Expand-Label(handshake_secret, "hs nonce", [sender],
    nonce_length)
3
4 handshake_key_[sender] =
5 HKDF-Expand-Label(handshake_secret, "hs key", [sender], key_length)
```

For application messages, a chain of keys is derived for each sender in a similar way. This allows Forward Secrecy on these messages within and out of an epoch. A step of this chain is called *generation*.

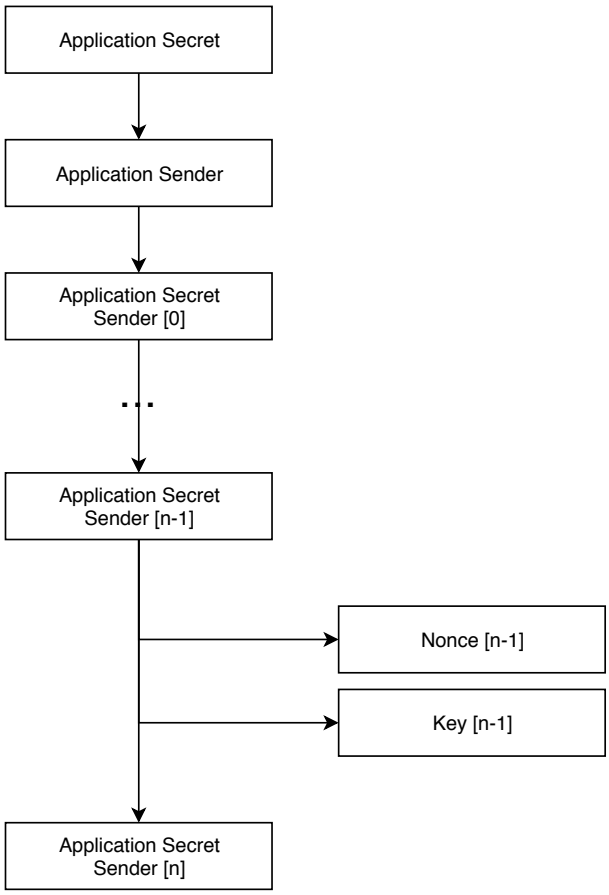



Figure 4.10: Application Key Schedule

The `sender` values are the indices of the member that will use the key to send the message.

### Usage of secrets

In order to provide Forward Secrecy at the level of Application messages, senders must use a given secret once and increment the generation of their secret. This way, an attacker that gets an Application Secret at epoch $n+1$ will not be able to derive the Application Secret at epoch $n$, nor the associated AEAD key and nonce.

Receivers must delete an Application Secret once it has been used to derive the AEAD key and nonce, as well as the next Application Secret.

Receivers may keep the AEAD key and nonce for a reasonable period, but they must delete keys and nonces once they have been used to successfully decrypt a message.

## 4.9 Initialization keys

MLS provides the possibility for users to publish initialization keys that provide some information about the user. This is useful as it allows users to add clients asynchronously to a conversation, even if they are offline. This operation is performed using the `UserInitKey` messages.

`UserInitKeys` specify the ciphersuites supported by the client and provide the public keys that can be used for key derivation and signing. The identity key of the client should be stable throughout the lifetime of the group. Initialization keys should be used only for a limited number of times - potentially once. These keys also contain an identifier chosen by the client, which the client must assure that uniquely identifies a given `UserInitKey` object among the set of keys created by the client.

The `init_keys` array has the same length of the `cipher_suites` array. Each entry inside `init_keys` array must be a public key for the asymmetric encryption scheme defined in the `cipher_suites` array and used in the HPKE construction for TreeKEM. The `UserInitKey` structure is then signed using the identity key of the client. A structure using an invalid signature is considered malformed. The signature process includes all the fields except for the `signature` field.

Listing 4.6: Definition of UserInitKey

```
1 uint8 ProtocolVersion;
2
3 struct {
4     opaque user_init_key_id<0..255>;
5     ProtocolVersion supported_versions<0..255>;
6     CipherSuite cipher_suites<0..255>;
7     HPKEPublicKey init_keys<1..2^16-1>;
8     Credential credential;
9     opaque signature<0..2^16-1>;
```

```
10 } UserInitKey;
```

## 4.10 Message framing

Handshake and application messages share the same framing structure that provides encryption, in order to assure confidentiality, and signing to authenticate the sender within the group.

The structures involved are `MLSPlaintext` and `MLSCiphertext`. The first one is a plaintext message that is only signed, while the latter is both signed and encrypted, with protection on the content of the message and related metadata. `MLSCiphertext` should be used for application and handshake messages; in case the delivery service needs to examine handshake messages, they might be transmitted as `MLSPlaintext`.

Listing 4.7: Definition of ContentType enumerator and MLSPlaintext and MLSCiphertext objects

```
1  enum {
2      invalid(0),
3      handshake(1),
4      application(2),
5      (255)
6  } ContentType;
7
8  struct {
9      opaque group_id<0..255>;
10     uint32 epoch;
11     uint32 sender;
12     ContentType content_type;
13
14     select (MLSPlaintext.content_type) {
15         case handshake:
16             GroupOperation operation;
17
18         case application:
19             opaque application_data<0..2^32-1>;
```

```
20      }
21
22      opaque signature<0..2^16-1>;
23  } MLSPlaintext;
24
25  struct {
26      opaque group_id<0..255>;
27      uint32 epoch;
28      ContentType content_type;
29      opaque sender_data_nonce<0..255>;
30      opaque encrypted_sender_data<0..255>;
31      opaque ciphertext<0..2^32-1>;
32  } MLSCiphertext;
```

The signature can be computed this way:

- gathering the required metadata: group identifier, epoch, content type (copied from the `MLSPlaintext` object), nonce, sender index, and key generation;
- signing the protected content and metadata;
- encrypting the sender information, using the random nonce and the key derived by the `sender_data_secret`;
- encrypting the content through a content encryption key.

The `ciphertext` field of `MLSCiphertext` is populated during the content encryption process, while the `encrypted_sender_data` is populated during the content metadata step. Decryption is made in a reverse manner, so first it decrypts metadata, the message, and then verifies the content signature.

### 4.10.1 Metadata encryption

The sender data used for content encryption is encrypted through AEAD using the `sender_data_nonce` and `sender_data_key` fields of `MLSCiphertext`, and encoded into the `MLSSenderData` object. Additional authenticated data (AAD) are handled through `MLSCiphertextSenderDataAAD`. Both structures

are described in Listing 4.8. The Delivery Service cannot detect the sender of the message, since `sender` is encrypted inside `MLSSenderData`.

When parsing sender data during decryption, the recipients have to verify that the `sender` field is an occupied leaf in the ratchet tree. To do so, the index value must be lesser than the number of leaves of the tree.

Listing 4.8: Definition of MLSSenderData and MLSCiphertextSenderDataAAD objects

```
1  struct {
2      uint32 sender;
3      uint32 generation;
4  } MLSSenderData;
5
6  struct {
7      opaque group_id<0..255>;
8      uint32 epoch;
9      ContentType content_type;
10     opaque sender_data_nonce<0..255>;
11 } MLSCiphertextSenderDataAAD;
```

### 4.10.2 Content signing and encryption

`MLSPlaintext` objects are signed using the signing private key corresponding to the credentials of the sender at the leaf in the tree. The signature includes metadata and message content without the `signature` field.

The `ciphertext` field of the `MLSCiphertext` object is produced by supplying the inputs specified in Listing 4.9 to the AEAD function provided by the ciphersuite in use. The plaintext input contains content and signature of the `MLSPlaintext` object, with an optional padding.

Keys and nonces vary depending on the content type of the message. The handshake key can be chosen from the application key chain for the current epoch, according to the message type.

Additional Authenticated Data (AAD) used during encryption include the value specified inside `MLSCiphertextContentAAD`. They are used to identify the key and nonce.

The `ciphertext` of `MLSCiphertext` is produced by supplying AAD to the AEAD function.

Listing 4.9: Definition of MLSCiphertextContent and MLSCiphertextContentAAD objects

```
1
2  struct {
3      opaque content[length_of_content];
4      uint8 signature[MLSInnerPlaintext.sig_len];
5      uint16 sig_len;
6      uint8 marker = 1;
7      uint8 zero_padding[length_of_padding];
8  } MLSCiphertextContent;
9
10 struct {
11     opaque group_id<0..255>;
12     uint32 epoch;
13     ContentType content_type;
14     opaque sender_data_nonce<0..255>;
15     opaque encrypted_sender_data<0..255>;
16 } MLSCiphertextContentAAD;
```

## 4.11 Handshake messages

As mentioned before, the group state will change after one of the following four basic operations:

- group initialization;
- client addition;
- client removal;
- client update of the leaf key.

These operations are performed by broadcasting *handshake* messages to the group. There is not a consolidated handshake phase to the protocol, be-

cause these broadcast messages are exchanged throughout the entire lifetime of the group, when a client needs to inform the whole group about changes.

Handshake messages encapsulate a `GroupOperation` message that performs the change to the group state. The handshake is carried in a `MLSPlaintext` that provides the signature of the sender, or in a `MLSCiphertext` if the application wants to send it encrypted.

Listing 4.10: Definition of GroupOperationType, GroupOperation and Handshake types.

```
1  enum {
2      init(0),
3      add(1),
4      update(2),
5      remove(3),
6      (255)
7  } GroupOperationType;
8
9  struct {
10     GroupOperationType msg_type;
11     select (GroupOperation.msg_type) {
12         case init: Init;
13         case add: Add;
14         case update: Update;
15         case remove: Remove;
16     };
17     opaque confirmation<0..255>;
18 } GroupOperation;
```

The general flow for processing a handshake message is as follows:

1. if the handshake is encrypted in a `MLSCiphertext`, decrypt it;
2. verify that the `epoch` field of the `MLSPlaintext` is equal to the epoch of the current `GroupState` object;
3. verify that the signature of the `MLSPlaintext` is verified using the public key from the credential stored at the leaf indicated by the `sender` field;

4. use the `operation` message to produce an updated and provisional `GroupState` object incorporating the changes;

5. use the `confirmation_key` for the new epoch to compute the confirmation MAC for this message and verify that it is the same as the `confirmation` field;

6. if the above checks are successful, consider the updated `GroupState` as the current state of the group.

The `signature` and `confirmation` values are computed over the transcript of group operations, using the transcript hash from the provisional `GroupState` object:

```
1 GroupOperation.confirmation = HMAC(confirmation_key, GroupState.
    transcript_hash)
```

HMAC uses the Hash algorithm for the ciphersuite in use. Sign uses the signature algorithm indicated by the signer's credential in the roster.

### 4.11.1 Init

Direct initialization messages are currently undefined in draft 04 and 05. The actual workaround is to create a group state including only the client that creates the group, and then add the initial members to the group. This has a communication complexity of $O(N\ log\ N)$, rather than $O(N)$ of the direct initialization.

### 4.11.2 Add

When adding a new member to a group, an existing member should send a Welcome message to the new member and an Add message to the group.

#### Welcome message

The Welcome message specifies that the new member needs to initialize a `GroupState` object that can be updated to the current state using the Add message, encrypted to the new member using HPKE. The recipient key pair

for the HPKE encryption is the one included in the indicated `UserInitKey`, corresponding to the related ciphersuite.

Listing 4.11: Definition of RatchetNode and WelcomeInfo structures and Welcome message.

```
1  struct {
2      HPKEPublicKey public_key;
3      optional<Credential> credential;
4  } RatchetNode;
5
6  struct {
7      ProtocolVersion version;
8      opaque group_id<0..255>;
9      uint32 epoch;
10     optional<HPKEPublicKey> tree<1..2^32-1>;
11     opaque transcript_hash<0..255>;
12     opaque init_secret<0..255>;
13 } WelcomeInfo;
14
15 struct {
16     opaque user_init_key_id<0..255>;
17     CipherSuite cipher_suite;
18     HPKECiphertext encrypted_welcome_info;
19 } Welcome;
```

When describing a tree through a list of nodes, `credential` for a node must be populated only if that node is a leaf in the tree.

The `init_secret` of the Welcome message is the output of Figure 4.9. The new member can combine the Init secret with the Update secret transmitted in the corresponding Add message to get the epoch secret in which it is added. Prior epoch secrets are never revealed to new members.

The new member process the Add message for itself, so the Welcome message should reflect the state of the group before the new user is added. The Welcome message will contain a copy of the `GroupState` object owned by the sender.

In a conversation with Alice and Bob, Bob can decrypt a Welcome mes-

sage, but he does not have the cryptographic assurance that Alice is the real sender. For each Update message, participating clients contribute to the signing, this way authenticating the path in the tree. If the chain is included in the Welcome message, Bob can ensure that Alice is not lying by verifying that other members have signed the inner nodes as well.

### Add message

The Add message provides the information needed by the existing group members in order to update their `GroupState` with the new member:

Listing 4.12: Definition of Add message.

```
1 struct {
2     uint32 index;
3     UserInitKey init_key;
4     opaque welcome_info_hash<0..255>;
5 } Add;
```

The `index` field specifies where in the tree the new member should be added: into an existing blank node, or at the right edge of the tree.

In both cases, the index $i$ should be strictly comprised between 0 and the size of the group ($0 \leq i \leq n$). When $i = n$, the node is added at the right edge. If the index already exists ($i < n$) and the node is not blank, then it means that the Add message is malformed and should be rejected by the recipient.

The `welcome_info_hash` field contains a hash of the WelcomeInfo object. The message is generated by requesting the `UserInitKey` for the user to be added from the directory and encoding it into an Add message.

### Joining a group

A client joining the group processes Welcome and Add messages by preparing a new `GroupState` object based on the Welcome message, and processing the Add message as an existing member would.

An existing member receiving an Add message verifies the signature of the message, then updates its state as follows:

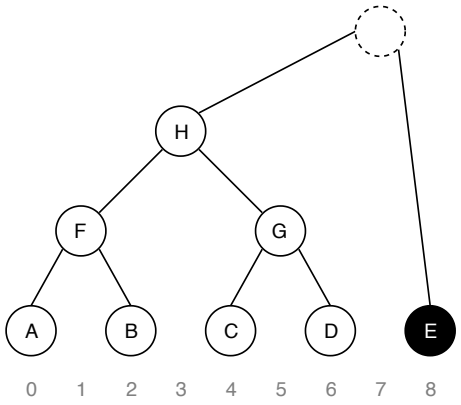



Figure 4.11: The ratchet tree after adding the new client $E$ to a group.

1. if `index` is equal to the size of the group, increment the group size and extend tree accordingly;

2. verify the signature of the included `UserInitKey`. If the verification fails, abort;

3. generate a WelcomeInfo object that describes the state prior to the addition, and verify that the hash is the same as `welcome_info_hash`;

4. update the ratchet tree by setting all the nodes in the direct path of the new node to blank;

5. set the leaf node at position `index` to a new node containing the public key from the `UserInitKey` in the Add corresponding to the ciphersuite in use, as well as the credential under which the `UserInitKey` was signed.

The update secret resulting from this change is an all-zero octet string of length `Hash.length`. Right after processing the Add message, the new member should send an Update message in order to update its key. This will help to limit the tree structure degrading into subtrees, and thus maintain efficiency.

### 4.11.3 Update

A member can send an Update message in order to update its leaf secret and key pair. This provides Post-Compromise Security on the prior leaf private key of the member.

Listing 4.13: Definition of Update message.

```
1 struct {
2     DirectPath path;
3 } Update;
```

The sender creates a message by generating a fresh leaf key pair and computing the direct path in the ratchet tree. A member that receives an Update message verifies the signature of the message, then updates its state by updating the cached ratchet tree. This is made possible by replacing nodes in the direct path from the updated leaf, using the information contained in the Update message. If the tree contains blank nodes, the resolution of the direct path will be used instead.

The update secret resulting from this change is the path secret for the root node of the ratchet tree.

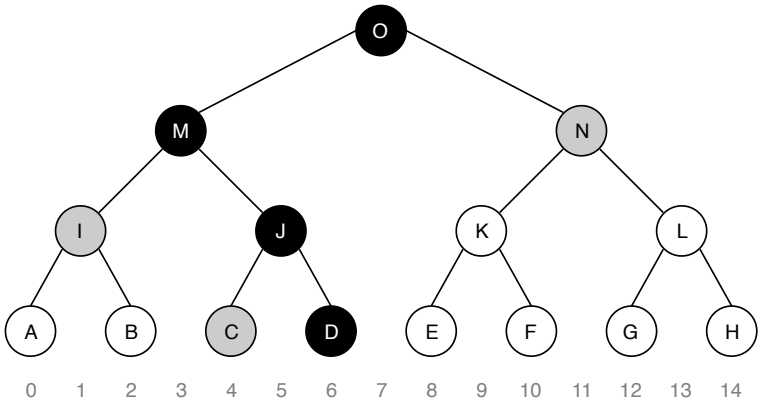



Figure 4.12: The ratchet tree after updating the client $D$. Direct path is represented in black, copath is represented in gray.

An open issue here is that, when a user is added to a group conversation, they have to wait until each member performs an Update in order to get the

complete knowledge of the group participants.

### 4.11.4 Remove

A Remove message is sent from a group member who wants to remove one or more members from the group conversation. Remove messages are not intended to be used when a member wants to remove themselves from the group. In this case, when a member receives a Remove message where the `removed` index is equal to the signer index, the recipient must discard the message because it is malformed.

Listing 4.14: Definition of Remove message.

```
1 struct {
2     uint32 removed;
3     DirectPath path;
4 } Remove;
```

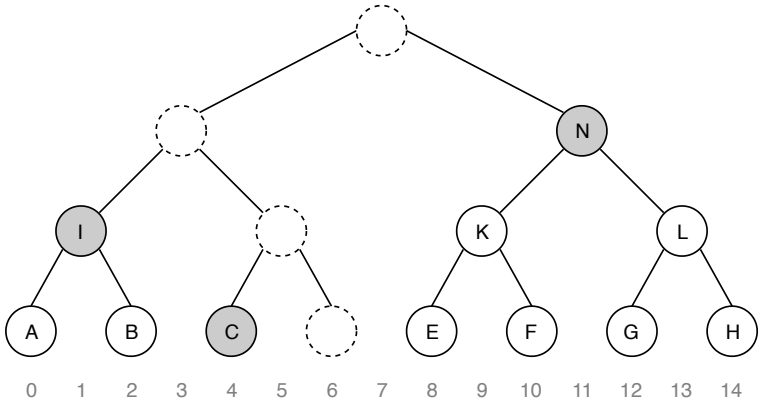



Figure 4.13: The ratchet tree after removing the client $D$. Blank nodes are represented with dashed outline. Copath is represented in gray.

Remove message is generated by the sender creating a fresh leaf key pair and computing its direct path in the current ratchet tree, starting from the removed leaf. A member that receives a Remove message verifies the signature of the message, then it updates its state as follows:

1. update the roster by setting the credential in the removed slot to `null`;

2. update the ratchet tree, replacing nodes in the direct path from the removed leaf using the information contained in the Remove message;

3. reduce the size of the roster and the tree until the rightmost element and leaf node are non-null;

4. update the ratchet tree by setting to blank all nodes in the direct path of the removed leaf, together with the root node. We assume here that there must be at least one non-null element in the tree, since any `GroupState` must have the current member in the tree and self-removal is prohibited;

5. truncate the tree, so the rightmost non-blank leaf is the last node of the tree.

The update secret will be the path secret for the root node of the ratchet tree in the first step.

## 4.12 Sequencing of state changes

Each handshake message is based on a given starting state called *epoch*, represented in Figure 4.14 and indicated with the `prior_epoch` field. Any changes to the state made from a different state will generate incorrect results.

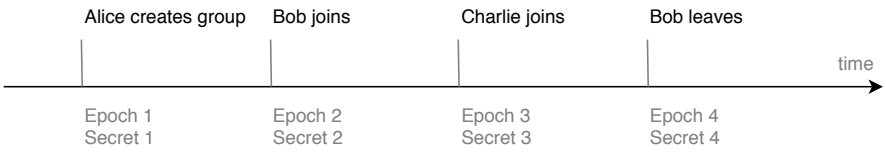



Figure 4.14: Graphical representation of epochs

Sequencing changes are not a problem as long as any handshake message is based on the latest state of the group. However, there is the risk that two members will generate handshake messages based on the same state. In this case, members of a group should deconflict the simultaneous handshakes.

MLS specifies two approaches: the Delivery Service that enforces a total order, or a signal in the message that clients can use to break ties.

As long as handshakes cannot be merged, there is a risk of starvation: in a busy group, a member may never be able to send handshakes because they always lose to other members. Handling this problem depends on the dynamics of the application.

With both approaches, implementations must update the cryptographic state only when a valid handshake message is received. Generation of handshakes must be stateless, because the endpoint does not know at that time if the changes to the state will succeed or not.

### 4.12.1 Server-enforced ordering

With server-enforced ordering, the delivery service keeps a queue for every incoming message. This way, outgoing messages are processed in the same order. The server is entitled to resolve conflicts during race conditions and it is trusted, since it does not know the content of the messages.

Messages should have a clear-text counter that can be checked by the server for tie-breaking. Counter starts from zero and is incremented for every new message. If two members send a message with the same counter, the first one to arrive will be accepted by the server, and the second one will be rejected. The rejected message needs to be sent again with the correct counter.

To prevent manipulation, the integrity of the counter can be guaranteed by including the counter in a signed message envelope.

### 4.12.2 Client-enforced ordering

Order enforcement can be implemented client-side by using a two step update protocol. The first client sends a proposal to update, which is accepted only when it gets more than 50% of approval from the group. Then, it sends the approved update. Clients that do not get their proposal accepted will wait for the winner to send their update before retrying new proposals.

This approach seems to be safer, as it does not rely on the server. Nevertheless, it is more complex and harder to implement, and it can also cause

starvation for clients that keep failing to get their proposal accepted.

### 4.12.3 Merging updates

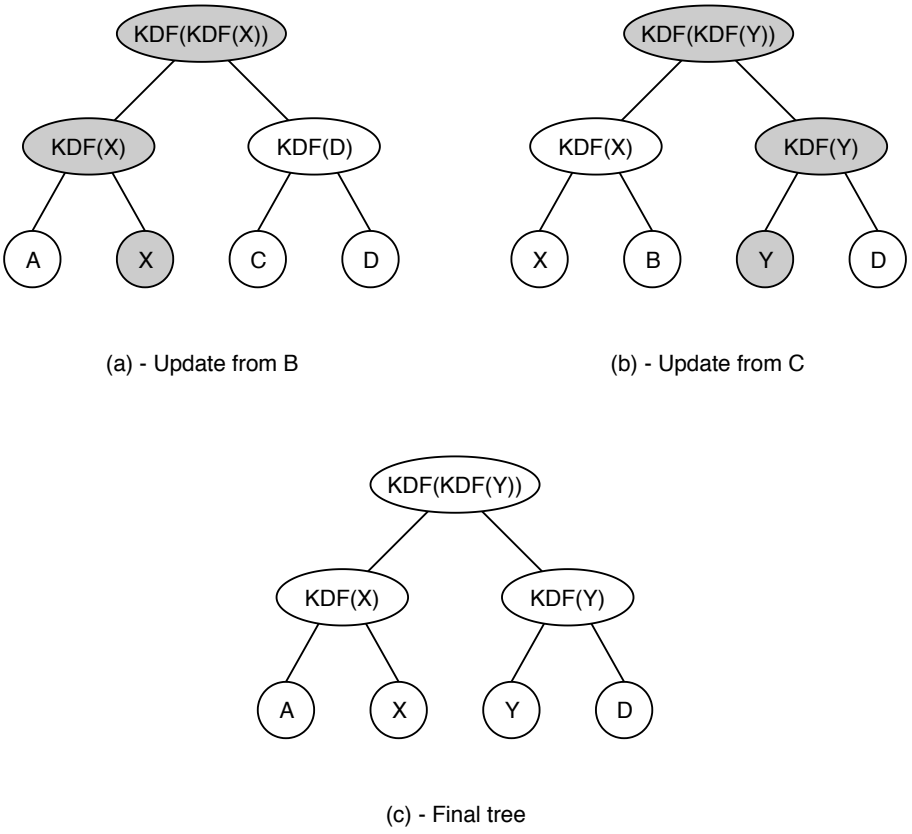



Figure 4.15: Merging two simultaneous updates from B and C

It is possible to address the problem of concurrent changes by having the recipients of the changes merge them. Since the value of intermediate nodes is determined by its last updated child, updates can be merged by recipients as long as the recipients agree on an order.

As previously mentioned, processing an update is made in two steps: compute updated secrets by hashing up the tree, and update the tree with new secret and public values. To merge an ordered list of updates, a recipient simply performs these updates in the specified order.

If we consider the tree represented in Figure 4.7 (a) and we suppose that both B and C simultaneously decide to update to X and Y, then they will send out updates as represented in Figure 4.15 (a) (b).

Assuming that the ordering agreed by the group says that the update from B should be processed before the update from C, the other members in the group will overwrite the root value for B with the root value from C, all resulting in the state represented in Figure 4.15 (c).

While Handshake messages should be ordered, Update messages do not have to be ordered.

## 4.13 Application messages

The handshake protocol provides an authenticated group key exchange to clients. The Application secret provided by the Handshake key schedule is used to derive encryption keys for the *Message Protection Layer*. Application messages must be protected with the AEAD encryption scheme associated with the ciphersuite used.

Each member maintains their own chain of Application secrets, where the first one is derived based on a secret chained to the Epoch secret. The initial Application secret is bound to the identity of each client, in order to avoid collisions and allow support for decryption of reordered messages.

Subsequent secrets must be rotated for each message sent in order to provide stronger cryptographic security guarantees. Application Key Schedule uses this rotation to generate fresh AEAD keys and nonces used to encrypt and decrypt future Application messages.

Each change to the Group through handshakes will cause a change of the group secret: this means that changes must be applied before encrypting new Application messages. This is needed for confidentiality, in order to avoid receiving messages from the former group members after leaving, being added to or excluded from the group.

### 4.13.1 Message encryption and decryption

Group members must use the AEAD algorithm associated with the negotiated MLS ciphersuite, in order to encrypt and decrypt Application messages according to the Message Framing section.

The `group` identifier and `epoch` allow a device to know which group secrets should be used and from which Epoch secret to start computing other secrets and keys. The `sender` identifier is used to derive the Application secret chain of the member from the initial group Application secret. The application `generation` field is used to determine which Application secret should be used from the chain to compute the correct AEAD keys before performing decryption.

Application messages should be padded to provide resistance against traffic analysis techniques. This avoids additional information to be provided to an attacker in order to guess the length of the encrypted message.

Padding should be used on messages with zero-valued bytes before AEAD encryption. Upon decryption, the length field of plaintext is used to compute the number of bytes to be removed from the plaintext to get the correct data.

#### Delayed and reordered Application messages

Each Application message contains the group identifier, the epoch and a message counter. This way, a client can receive messages out of order. However, if they can retrieve or recompute the correct AEAD decryption key, they can decrypt messages.

Clients might be required to keep the AEAD key and nonce for a certain amount of time to retain the ability to decrypt delayed or out of order messages.

## 4.14 Security considerations

This section describes how security goals of MLS, previously described in the architectural part, are achieved at protocol level.

### 4.14.1 Confidentiality of the group secrets

Group secrets are derived from the previous ones and the root key of a ratcheting tree. The root key of the group ratcheting tree, and all the values derived from it, are secret because only group members know their leaf private key in the group.

Initial leaf keys are known only by their owner and the group creator, because they are derived from an authenticated key exchange protocol. Subsequent leaf keys are known only by their owner. Long term identity keys must be distributed by the Authentication Service to clients in order to authenticate their legitimate peers.

### 4.14.2 Authentication

MLS considers two forms of authentication:

**Authentication with respect to the group.** Group members can verify a message coming from one member of the group. This is guaranteed by the secrecy of the shared key derived from the ratchet trees. If all members are honest, then the shared group key is only known to the group members. Further guarantees about the sender of a message take place by using AEAD or appropriate MAC with the shared key.

**Authentication with respect to the sender.** Group members can verify that a message was sent from a particular member of the group. This is guaranteed by digital signatures on the messages.

### 4.14.3 Init key reuse

Initialization keys are intended to be used only once and then deleted. Reuse of initial keys is not unsafe at all, but it may complicate protocol analyses, because it is difficult to know how many times the initial key was used.

# Chapter 5

# An Implementation of MLS: Melissa

## 5.1 Melissa

**Melissa** is a proof-of-concept implementation of the Messaging Layer Security protocol created by the Wire team. It is written in Rust and it is based on the draft 05 version of the protocol. The development started in September 2018 and the code is publicly available open source at `https://github.com/wireapp/melissa`. In order to stick to the changes expected from the specification of MLS, new updates are implemented accordingly. Melissa is released under the terms of the GNU General Public License.

### 5.1.1 Work brought on Melissa

My contribution to Melissa included the update of the project to adapt it to the draft 04 and 05 versions of the protocol. These changes include:

- updating the terminology used for variables and functions;
- replacing the hash inside TreeKEM with KDF;
- removing the `secret` field from the tree nodes;
- adding credentials to the tree and removing the `roster` at the same time;

- converting from states to commitment using tree hashes;
- adding framing for handshake and application messages, together with encryption of handshake messages.

### 5.1.2 Rust

Rust is a compiled, multi-purpose programming language focused on memory safety and concurrency. It aims to be an efficient, safe and suitable language for developing concurrent software.

Rust syntax is influenced by *Cyclone*, a safe dialect of C, with aspects of object-oriented features from C++ and functional features from Haskell and OCaml. This makes Rust a procedural, functional and object-oriented programming language.

Rust is a language whose popularity has grown in recent years, so much so that it is considered the "most loved programming language" by developers since 2016 according to the Stack Overflow Developer Survey [21]. Rust is also used as the programming language of Servo, an experimental browser engine that inspired some of the major improvements brought by *Firefox Quantum*.

## 5.2 The project

The project can be cloned or downloaded from the GitHub repository. The only requirements are that Rust must be installed on the system - this can be done via `rustup`, the official Rust installer available at `https://rustup.rs` - and that the package manager should update the dependencies needed by the project. This can be done by calling `cargo update` in the terminal from the root folder of the project.

## 5.3 Project structure

The project is structured in *modules*, declared inside the `lib.rs` file together with the dependencies. Modules are public and they are implemented

inside their respective files: for example, `treemath` module is declared inside the `treemath.rs` file.

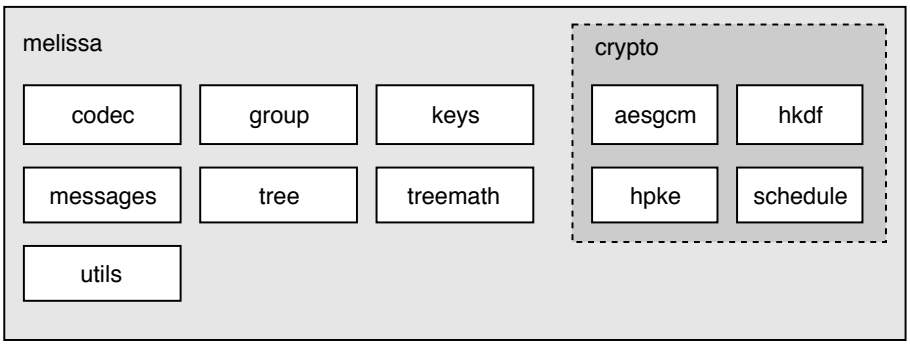



Figure 5.1: Modules available in Melissa

### 5.3.1 The `crypto` submodule

The `crypto` folder is a submodule of the project that contains all the cryptographic modules used in Melissa.

`aesgcm` contains the functions for using the AES-128 and AES-256 block cipher algorithms. In fact, the three structures available here are `Nonce`, `Aes128Key` and `Aes256Key`. AES keys implement the `From` and `Drop` traits - the main abstraction construct of Rust, equivalent in some way to Java interfaces - respectively the converter from `u8` vectors and the destructor. They also specify functions to seal (encrypt) and open (decrypt) data. AES-128 seals and opens data by using the `seal_in_place()` and `open_in_place()` primitives of `ring`. AES-256 methods are currently left unused, since the two ciphersuites specify AES-128 as the encryption algorithm.

`hpke` handles the Hybrid Public Key Encryption (HPKE). A payload can be encrypted through HPKE by calling the `encrypt` method on the `HpkeCiphertext` object. The payload will be sealed with a fresh key pair by using the ciphersuite composed by X25519, SHA-256 and AES-128-GCM. Ciphersuites are specified by the `HpkeCiphersuite` enumerator and four of them are declared, mostly for future implementations (e.g. with P256):

- P256, SHA-256 and AES-128-GCM;

- P512, SHA-512 and AES-256-GCM;
- X25519, SHA-256 and AES-128-GCM;
- X448, SHA-512 and AES-256-GCM.

Messages can also be encrypted with a specified ephemeral key pair by calling `encrypt_with_ephemeral()`. Both encryption and decryption procedures call `setup_base_x25519_aes_128()` and `setup_core_x25519_aes_128()`, that first creates a `HpkeContext` containing information like the ciphersuite in use, then it expands both key and nonce through HKDF by using the `hkdf` module. Since HPKE makes use of AES algorithms, ciphertexts are sealed and opened by using the `aesgcm` module seen before.

`hkdf` handles the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [12], already seen in Chapter 4.8. HKDF is useful to hide the input keying material from malicious attackers that may have some partial knowledge about it.

When calling HKDF, it takes four arguments in input: `salt`, `input`, `info` and `length`:

```
1 pub fn hkdf(salt: Salt, input: Input, info: Info, Len(len): Len) ->
      Key {
2     Key(expand(extract(salt, input), info, len as usize))
3 }
```

The `extract()` function takes the input keying material to concentrate dispersed entropy into a short but cryptographically strong pseudorandom key (PRK) that is produced as the output of the function:

```
1 pub fn extract(Salt(s): Salt, Input(i): Input) -> Prk {
2     Prk(hmacsha256::authenticate(i, &hmacsha256::Key(mk_salt(s))).0)
3 }
```

Then, the generated output key is expanded to create a larger cryptographically independent output. The number and length of the output keys may depend on the algorithm used.

```
1 pub fn expand(prk: Prk, Info(info): Info, len: usize) -> Vec<u8> {
2     let n = (len as f32 / HASH_LEN as f32).ceil() as usize;
```

```
3     let mut t = Vec::new();
4     let mut okm = Vec::new();
5
6     for i in 1..=n {
7         let mut buf = Vec::with_capacity(t.len() + info.len() + 1);
8         buf.extend(&t);
9         buf.extend(info);
10        buf.push(i as u8);
11
12        let key = hmacsha256::Key::from_slice(&prk.0).unwrap();
13        let t_i = hmacsha256::authenticate(&buf, &key);
14        okm.extend(&t_i.0);
15
16        t.clear();
17        t.extend(&t_i.0);
18    }
19
20    okm.into_iter().take(len).collect()
21 }
```

`schedule` contains functions and structures for the key schedule. Particularly, it includes the declaration of the Init Secret, the Epoch Secrets and HKDF Label objects. It also comprehends the function to derive secrets: given a pseudorandom key secret, a HKDF Label and a context, the secret is derived by calling `expand(secret, label, context)` of the HKDF module.

Listing 5.1: Deriving secrets in the `schedule` module

```
1 pub fn derive_secret(secret: hkdf::Prk, label: &str, context: &[u8
    ]) -> Vec<u8> {
2     let context_hash = sha256::hash(context).0;
3     let hkdf_label = HkdfLabel::new(&context_hash, label,
          HASH_LENGTH);
4     let state = &hkdf_label.serialize();
5     println!("HKDFLabel: {}", bytes_to_hex(&state));
6     let info = hkdf::Info(state);
7     hkdf::expand(secret, info, HASH_LENGTH)
8 }
```

### 5.3.2 Group handling (`group.rs`)

The `group` module contains the core logic for managing a group in MLS. The module handles the initialization of the group, the creation and processing of the *Welcome*, *Add*, *Update*, *Remove*, *Handshake* messages, and deals with init and epoch secrets. After processing every type of message, the epoch secret gets updated by calling the `rotate_epoch_secret()` method. This ensures Perfect Forward Secrecy.

Groups can be created starting from a Welcome message or initialized through a `Credential` object and a `GroupId`. In the first case, the tree is generated from the public keys, while in the latter it is generated starting from the client's own leaf.

The *Add* message is created at the same time of the *Welcome* message, as illustrated in Figure 4.2. The *Add* message is created by encrypting the tree with the new index of the member and the new size, together with a fresh random leaf secret. As seen before, the new size is the number of leaves plus one, while the expected index will be the number of leaves multiplied by 2. Alongside, the *Welcome* message is created by processing the *Add* message on a mutable copy of the current group.

The *Update* message is created by encrypting the own leaf index, the current size of the tree and a fresh random leaf secret. Then, the message is hashed and the secret gets updated. The *Update* is processed by checking that hashes match: in this case, it merges the direct path with the hashed nodes. Otherwise, KEM is applied to the Update path.

The *Remove* message is created by passing the index of the participant that should be removed, together with the nodes and ciphertext encrypted from the tree. The *Remove* is processed by applying the KEM path to the path specified by the Remove message.

*Handshake* messages are created by passing the prior epoch, the index of the signer and the algorithm used. After that, the message gets signed. Handshakes are processed by verifying the signature and continuing the processing by calling the proper `process_` function described before and associated to

the respective `group_operation_value` (add, update or remove).

New epoch secrets are generated by encoding the group identifier, the group epoch, the tree and the transcript, and by updating the Init secret. The group epoch is then incremented by 1.

### 5.3.3 Keys handling (`keys.rs`)

`keys` provide data structures and utilities for key handling. Particularly, the module implements the structures for key agreement over the two curves supported by the protocol, that is, *Curve25519* and *P-256*. The latter is implemented through its public key. For *Curve25519*, the public and private keys are stored into a `X25519KeyPair` object, respectively encoded as `X25519PublicKey` and `X25519PrivateKey` types. Key pairs can be generated from scratch, starting from an existing secret or from a private key. Private keys can generate shared secrets and derive the public key by calling the two methods `shared_secret()` and `derive_public_key()`:

```
1  pub fn shared_secret(&self, p: &X25519PublicKey) -> Result<[u8;
       32], Zero> {
2      let group_element = scalarmult::curve25519::GroupElement::
           from_slice(&p.0).unwrap();
3      let scalar = scalarmult::curve25519::Scalar::from_slice(&self.0)
           .unwrap();
4      scalarmult::curve25519::scalarmult(&scalar, &group_element)
5          .map(|ge| ge.0)
6          .map_err(|()| Zero {})
7  }
8
9  pub fn derive_public_key(&self) -> X25519PublicKey {
10     let scalar = scalarmult::curve25519::Scalar::from_slice(&self.0)
           .unwrap();
11     X25519PublicKey(scalarmult::curve25519::scalarmult_base(&scalar)
           .0)
12 }
```

Furthermore, the `keys` module provides the `Identity` object that can be used for identity keys. It exposes the identifier and the public key, thus

maintaining the private key. Objects can be signed with a given `Identity` by implementing the `Signable` trait, that provides the `sign()` method to perform the signing action and `verify()` to ensure that the object is signed correctly.

`BasicCredential` is the base structure for credentials that includes a public key, together with an identity. Credentials can be verified through the `verify()` method.

In the end, `keys` provides the mechanism for group initialization and adding members to a group, through the `UserInitKey` object. The personal `UserInitKey` is usually pushed to the Directory through the `UserInitKeyBundle`, that includes the initialization key and a set of private keys that can be used in the future by other clients, as described in Figure 4.1.

### 5.3.4 Encoding and decoding (`codec.rs`)

The `codec` module provides traits for encoding and decoding primitive data types and custom data structures, in a similar fashion as TLS does. Encoding and decoding is provided to submodules by implementing the `Codec` trait. `Codec` makes available two methods for encoding and decoding, plus two `detached` method versions that perform the respective operations on a specified instance of the object:

Listing 5.2: The `Codec` trait

```
1  pub trait Codec: Sized {
2      fn encode(&self, buffer: &mut Vec<u8>);
3      fn decode(&mut Cursor) -> Result<Self, CodecError>;
4      fn encode_detached(&self) -> Vec<u8> {
5          let mut buffer = vec![];
6          self.encode(&mut buffer);
7          buffer
8      }
9      fn decode_detached(buffer: &[u8]) -> Result<Self, CodecError> {
10         let mut cursor = Cursor::new(buffer);
11         Self::decode(&mut cursor)
12     }
13 }
```

Implementations of the `Codec` trait are available throughout the project. An example implemented for the `HpkeContext` object is available in Appendix A.1.

The `codec` module also provides implementations of the `encode()` and `decode()` methods for the primitive unsigned integer types `u8`, `u16`, `u32`, `u64`, besides respective encoding and decoding methods for unsigned integer vectors, named respectively `encode_vec_uXY` and `decode_vec_uXY` (where XY is the size of the unsigned type).

### 5.3.5 Messages specification and protection (`messages.rs` and `mp.rs`)

The `messages` module declares the four main types of operations: *Welcome*, *Add*, *Update*, *Remove*, together with the enclosing *Handshake* message. The core structure here is `GroupOperation`: it contains the type of message, declared as a `GroupOperationType` enumerator, the content of the group operation to be performed, declared inside `GroupOperationValue`, and the delivery confirmation. The `GroupOperation` is then enclosed into a `Handshake` message. The structure of these messages follows the specifications brought by the protocol described inside Chapter 4.11, with some minor adaptations. The module also contains the structures for the message framing, such as `MLSPlaintext` and `MLSCiphertext`. Message framing was introduced with draft 05 and is described in Chapter 4.10.

The `mp` module is about message protection and provides structures like `ApplicationMessage` that wrap the encrypted content of a message, and `SignatureContent` that contains the signed content. It also includes structures for stage secrets (nonce and key) like `StageSecrets` and `SenderApplicationSecret`.

### 5.3.6 Tree structure and math (`tree.rs` and `treemath.rs`)

The `tree` module contains declarations of the main components of a tree: nodes, node secrets and the tree itself.

The `Node` object represents an instance of a node in the tree. As reported

in Chapter 4.2.1, the structure contains three optional values: a `NodeSecret` object, a Diffie-Hellman public key and a private key. Fields are valued based on their position in the tree, whether they are leaves, a direct path or standard nodes. If no fields are valued, the node is considered *blank*. The `NodeSecret` can be created randomly, starting from bytes, or hashed.

The `Tree` object describes the structure of a ratchet tree. It is basically a set of `Node` objects, together with the index of its own leaf. Trees can be created starting with the own leaf node or from a set of public keys, together with the own leaf index and private key. Besides implementing utility methods such as getting own leaf node, leaf count or the root node, the object also provides operations that can be performed on the tree for encryption and decryption purposes. They include:

- `resolve()`, that recursively resolves the tree as explained in Chapter 4.2.2 and illustrated below:

Listing 5.3: Resolution of a tree for a given node

```
1  pub fn resolve(&self, x: usize) -> Vec<usize> {
2      let n = self.get_leaf_count();
3      if !self.nodes[x].is_blank() {
4          return vec![x];
5      }
6
7      if treemath::level(x) == 0 {
8          return vec![];
9      }
10
11     let mut left = self.resolve(treemath::left(x));
12     let right = self.resolve(treemath::right(x, n));
13     left.extend(right);
14     left
15 }
```

- `blank_up()` blanks the nodes of the tree recursively from the leaves up to the root, excluding it;

- `merge()` merges a set of nodes on a given path;
- `hash_up()` returns the hashes for nodes on the direct path on a `index` leaf node;
- `kem_to()` applies the key encapsulation to the tree, given the set of nodes composing direct path and the copath. The encapsulation is made by encrypting the public key of a copath node with the secret of the direct path node for every direct-copath node pair, this way:

```
1 let (dirpath_node, copath_node) = node_pair;
2 let public_key = copath_node.dh_public_key.unwrap();
3 let ciphertext =
4 HpkeCiphertext::encrypt(&public_key, &dirpath_node.secret.
      unwrap().0[..]).unwrap();
5 path.push(ciphertext);
```

- `encrypt()` and `decrypt()` respectively for encrypting and decrypting the tree;
- `apply_kem_path()` applies the key encapsulation to a given path.

The module also provides data structures for tree hashes, introduced by draft 05 and previously explained in Chapter 4.5.

The `treemath` module contains utility methods for handling trees, like finding the left or right children, parent, siblings, etc. More precisely:

- `log2()` for calculating base 2 logarithm ($log_2(n)$) and `pow2()` for a power of 2 ($2^n$);
- node level and width;
- methods for relationships in a tree: root of a tree, left and right children, parents and siblings;
- methods for calculating the direct path and copath;
- `leaves()` for the list of leaves in a tree.

The module also provides utility methods to generate and read test vectors, that will be covered in the next chapter.

### 5.3.7 Utilities (`utils.rs`)

The `utils` module provides some utility methods to convert hexadecimal strings to bytes and vice versa, in addition to the call to the `memzero()` method provided by `sodiumoxide` to clear vectors.

## 5.4 Dependencies

Dependencies in Rust are called *crates* and they are handled through **Cargo**, the main package manager for Rust. Information about the current package and dependencies are specified through the `Cargo.toml` file, positioned inside the root directory. Another file, `Cargo.lock`, is generated automatically by Cargo starting from the `Cargo.toml` file while installing or updating crates and specifies exact information about dependencies. `Cargo.toml` specifies:

- name, version and authors of the current package;
- general dependencies: `sodiumoxide`, `libsodium` and `ring`;
- dependencies for testing and benchmarking: `criterion`;
- the position of the benchmark folder.

### 5.4.1 NaCl, `libsodium` and `sodiumoxide`

`sodiumoxide` is a type-safe and efficient Rust binding around Sodium (`libsodium`), which in turn is a portable, cross-compilable and packageable fork of NaCl. NaCl, abbreviation for *Networking and Cryptographic Library* and pronounced as "salt", is a public domain library for network communication, encryption, decryption and signatures. NaCl provides all the core operations needed to build higher level cryptographic tools. [22] It was created by Daniel J. Bernstein, mathematician and cryptologist who also created *Curve25519*.

### 5.4.2 `ring`

`ring` is another hybrid Rust, C and assembly library that provides a set of general-purpose cryptographic operations. `ring` makes it easy to build and integrate it into high level frameworks and applications; it also works optimally on small devices and microcontrollers, in order to support Internet-of-Things applications. `ring` derives from BoringSSL, which in turn is derived from OpenSSL, it is developed by Brian Smith and available open source at `https://github.com/briansmith/ring`. `ring` is used within the project to provide AES-128-GCM, which is not provided by default by `libsodium`.

### 5.4.3 Usage within Melissa

NaCl is used in Melissa for several operations, including:

- **Cryptographic random bytes generation** for nonces, keys and key pair generation within the whole project;
- **Public-key signatures** inside `keys.rs` using *Ed25519*, a EdDSA signature scheme that makes use of SHA-512 and Curve25519;
- **Secret-key authentication** inside the HKDF-Extract and HKDF-Expand functions of `hkdf.rs`;
- **Hashing** to generate SHA-256 hashes for the HKDF-Extract and HKDF-Expand functions, the derive secret function of `schedule.rs` and the specification of the ciphersuite that includes SHA-256 inside of `keys.rs`;
- **Scalar multiplication** used to generate shared secrets, Derive-Public-Key function, and new X25519 key pairs inside of `keys.rs`.

`ring` is mostly used in `aesgcm.rs` to provide Authentication Encryption with Associated Data (AEAD) for sealing and opening keys, and for sealing and opening operations.

# Chapter 6

# Melissa: Tests and Benchmarks

## 6.1 Tests

As a strategic component of any software, tests help to verify the correctness of instructions, procedures and functions of the entire software or a particular module of it. Verification is typically specified through dedicated *test cases*, functions that enclose various assertions that have to succeed in order for the test to be verified.

There are four main levels of testing:

- *unit tests* that verify the functionality of a portion of code, usually classes or single functions;
- *integration tests* that verify the interaction between different components, classes or modules
- *system tests* that verify if a system meets its requirements;
- *acceptance tests* used to certify the readiness of a product before the release, usually as part of a quality assurance system.

Tests can be natively supported by the system or may require the use of external libraries for the whole task or some parts (e.g. usage of *mock objects*).

### 6.1.1 Tests in Rust

Within a Rust project, tests are declared inside each module. They are recognizable by the `#[test]` attribute specified before each test method. On larger projects, tests are usually grouped into a module called `tests` and marked with the `#[cfg(test)]` attribute. [24]

Tests are verified through two main tools: *panics* and *assertions*. Panics are usually placed inside the code where the functions should fail. In the following example, a panic is thrown when dividing a number by zero:

```
1 pub fn divide(a: u32, b: u32) -> u32 {
2     if b == 0 {
3         panic!("Divide-by-zero error");
4     }
5     a / b
6 }
```

Assertions are placed inside the test cases to verify a particular condition. If the assertion fails, a panic is thrown. The assertions available in Rust are `assert!(expression)`, that verifies the `expression` to be true, and both `assert_eq!(left, right)` and `assert_ne!(left, right)`, verifying that the `left` item is equal or not to the `right` item.

Tests are handled through `cargo` and can be launched by calling `cargo test` in the terminal.

## 6.2 Test cases

Test cases are available inside each module. They cover the main operations to be carried out within that module.

#### Encoding and decoding

We have seen that the `codec` module includes operations for encoding and decoding data implementable through the `Codec` trait. Tests here verify that:

- primitive types, including unsigned integers `u8`, `u16`, `u32` and `u64` are correctly encoded through the `encode()` function;
- vectors based on primitive types are correctly encoded through the respective `encode_vec_uXY` function (where XY is the size of the unsigned type).

An example of test with 16 bit unsigned integer primitive type and vector can be found in Appendix A.2.1.

### Sealing and opening in AES

The `aesgcm` module contains a test for sealing and opening a payload encrypted through AES by using both AES-128 and AES-256 algorithms. It basically seals a payload and opens it, to see that the two copies of the message are the same:

```
1 let payload = vec![1, 2, 3];
2 let key: Aes128Key = Aes128Key::from(randombytes::randombytes(
      AES128KEYBYTES));
3 let nonce = Nonce::new_random();
4 let encrypted = aes_128_seal(&payload, &key, &nonce).unwrap();
5 let decrypted = aes_128_open(&encrypted, &key, &nonce).unwrap();
6 assert_eq!(decrypted, payload);
```

### HPKE

The `hpke_encrypt_decrypt_x25519_aes()` test from the `hpke` module is used for testing encryption and decryption through Hybrid Public Key Encryption. It simply encrypts a cleartext composed by a vector of integer with the public key, and decrypts it with the private key of a `X25519KeyPair`, using a `HpkeCiphertext` object. Then, the cleartext and decrypted payloads are compared.

```
1 #[test]
2 fn hpke_encrypt_decrypt_x25519_aes() {
3     let kp = X25519KeyPair::new_random();
```

```
4     let cleartext = vec![0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
5
6     let encrypted = HpkeCiphertext::encrypt(&kp.public_key, &
          cleartext).unwrap();
7     let decrypted = HpkeCiphertext::decrypt(&kp.private_key, &
          encrypted).unwrap();
8
9     assert_eq!(cleartext, decrypted);
10 }
```

### HKDF

Three tests are described into the `hkdf` module to verify the Extract-and-Expand functions provided by HKDF. The tests declare the initial keying material, the salt and the info that will be provided to the two functions, and they verify the correctness by comparing the pseudorandom key for the extract function, and the output keying material for the expand phase to the expected values. `test_case_1()` and `text_case_2()` are basically the same using keys of different lengths, where `test_case_3()` verifies the functions against empty salt and info values.

```
1  #[test]
2  fn test_case_1() {
3      use utils::*;
4
5      let ikm = hex_to_bytes("0
           b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b");
6      let salt = hex_to_bytes("000102030405060708090a0b0c");
7      let info = hex_to_bytes("f0f1f2f3f4f5f6f7f8f9");
8      let len = 42;
9
10     let expected_prk =
11         hex_to_bytes("077709362
               c2e32df0ddc3f0dc47bba6390b6c73bb50f9c3122ec844ad7c2b3e5")
               ;
12     let expected_okm = hex_to_bytes(
```

```
13          "3cb25f25faacd57a90434f64d0362f2a2d2d0a90cf1a5a4c5db02d5
                {...}85865",
14      );
15
16      let prk = extract(Salt(&salt), Input(&ikm));
17      let okm = expand(prk, Info(&info), len);
18
19      assert_eq!(&expected_prk, &prk.0);
20      assert_eq!(&expected_okm, &okm);
21  }
```

### Group conversations

The `group` module contains a test named `alice_bob_charlie_walk_into_a_group()`, which verifies the process of creating a group with three participants: Alice, Bob and Charlie.

More in depth, it creates a conversation with three identities, each of them distinguished by a credential that includes the identity. Alice creates the group with her identity, while Bob and Charlie create their `UserInitKey`s. Then, Alice adds Bob to the group by creating a *Welcome* and *Add* message, the latter being processed by Alice. In the end, Bob creates their representation of the group starting from the *Welcome* message from Alice.

After that, both Bob and Alice update their secret by creating an *Update* message and processing it. Then, Bob adds Charlie to the group and processes the *Add* message together with Alice. Charlie and Alice try to update their secrets, which is updated by all the three participants.

Finally, Bob removes Charlie from the group, causing the remove message being processed by all participants.

### Keys

The `keys` module contains verification for `crypto.bin` test vector, together with a test that signs a payload and verifies it through the `verify_detached()` method. The module also tests the generation of `UserInitKey`s.

#### Tree and tree math

The tests for `tree` and `treemath` modules are composed by the verification of the test binaries dedicated respectively to tree resolution (`resolution.bin`) and tree math (`treemath.bin`).

### 6.2.1 Test vectors

*Test vectors* are binary files released by the MLS Working Group that are used to verify the compliance to the protocol. They are based on the draft 04 version of the protocol and they are available in the MLS Implementations repository at `https://github.com/mlswg/mls-implementations/tree/master/test_vectors`. Due to open issues on the protocol, there are some minor differences between the specifications and the results of the vectors.

Right now, there are six test vector files available covering the tree math, tree resolution, cryptographic functions, key schedule, application key schedule, message parsing and serialization and sessions. The three test vectors currently available in the project and tested are `treemath.bin`, `resolution.bin` and `crypto.bin`.

#### Tree math

The test vector for tree math aims to verify the relationships between nodes, as defined in the protocol. This test vector is verified by the `verify_binary_test_vector_treemath()` test inside the `treemath` module.

To do so, the vector is represented through a `TreeMathTestVectors` object that contains the following values:

- `tree_size`: the size of the tree to be tested;
- `root[i]`: index of the root of a tree with $i + 1$ leaves;
- `left[i]`, `right[i]`, `parent[i]`, `sibling[i]` : respectively, indices of the left child, the right child, the parent and the sibling of the node at the index $i$;

The result of the respective function that should be called on the `treemath` module has to be the same as the one declared in the vector.

### Resolution

Resolution vectors verify the output of the tree resolution algorithm. This test vector is verified by the `verify_binary_test_vector_resolution()` test inside the `tree` module.

The vector is represented this way:

```
1 uint8_t Resolution<0..255>;
2 Resolution ResolutionCase<0..2^16-1>;
3
4 struct {
5     uint32_t n_leaves;
6     ResolutionCase cases<0..2^32-1>;
7 } ResolutionTestVectors;
```

The tree is distinguished by a number of leaves equal to `n_leaves`. The `cases` vector has $2^{(2 \times n\_leaves - 1)}$ entries; the entry at index $t$ represents the set of resolutions with a blank/filled pattern matching the bit pattern of the integer $t$. When `((t >> n) & 1) == 1`, the node $n$ is filled, otherwise it is blank.

The `ResolutionCase` vector contains the resolutions of every node in the tree, so `case[t][i]` contains the resolution of the node $i$ in the tree $t$.

### Crypto

The test vectors about cryptographic features are declared through `CryptoTestVectors`, the inputs of the functions, and `CryptoCase` that holds the outputs using the specified ciphersuite. This test vector is verified by the `verify_binary_test_vector_crypto())` test inside the `keys` module.

The test vector is used to verify the correctness of the following methods:

- *HKDF-Extract*, the first step of Extract-and-Expand, over a given salt and input keying material;
- *Derive-Secret*;

Table 6.1: Code coverage results for Melissa

| Module name | Lines covered | Percentage |
|---|---|---|
| `crypto/aesgcm` | 95/101 | 94.06% |
| `crypto/eckem` | 101/107 | 94.39% |
| `crypto/hkdf` | 62/71 | 87.32% |
| `crypto/schedule` | 55/69 | 79.71% |
| `codec` | 140/178 | 78.65% |
| `group` | 190/255 | 74.51% |
| `keys` | 172/241 | 71.37% |
| `messages` | 22/123 | 17.89% |
| `mp` | 86/98 | 87.76% |
| `roster` | 0/5 | 0% |
| `tree` | 183/254 | 72.05% |
| `treemath` | 150/180 | 83.33% |
| `utils` | 13/13 | 100.00% |
| Total | 1269/1695 | 74.87% |

- *Derive-Key-Pair*;
- *ECIES*, using the key pair generated during Derive-Key-Pair.

### 6.2.2 Code coverage

Code coverage was implemented during this thesis work for testing purposes. The component used here is *Tarpaulin*, a library for code coverage purposes that provides line coverage, reporting tools and upload to online coverage services like Coveralls and Codecov. Due to its requirements, Tarpaulin is compatible with Linux only, so it was run under a Ubuntu distribution.

Tarpaulin revealed that tests executed on the draft 04 version of the project cover 74.87% of the entire codebase. More in depth, the results for the single modules are listed in Table 6.1.

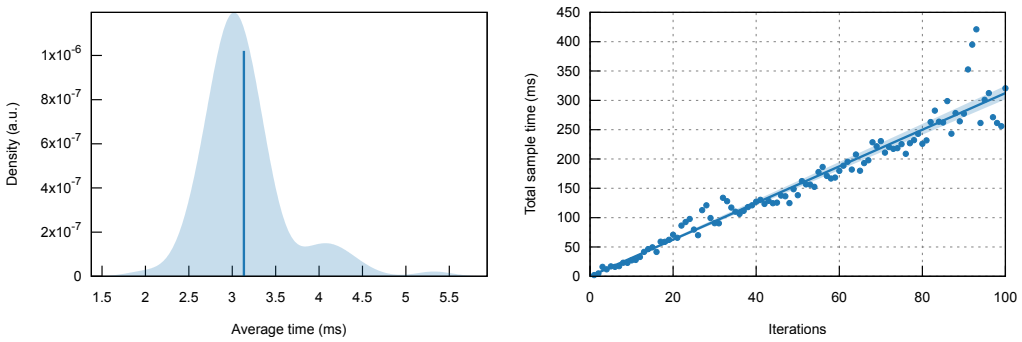
## 6.3 Benchmarks

Benchmarks are tests that measure the performances of a software by running the test code several times. The results of a benchmark can be used to compare the performances with other softwares or solutions.

As previously seen, benchmarks are implemented inside Melissa through the use of *Criterion.rs*, a benchmarking library created by Jorge Aparicio and maintained by Brook Heisler, written in Rust and available open source at `https://github.com/bheisler/criterion.rs`

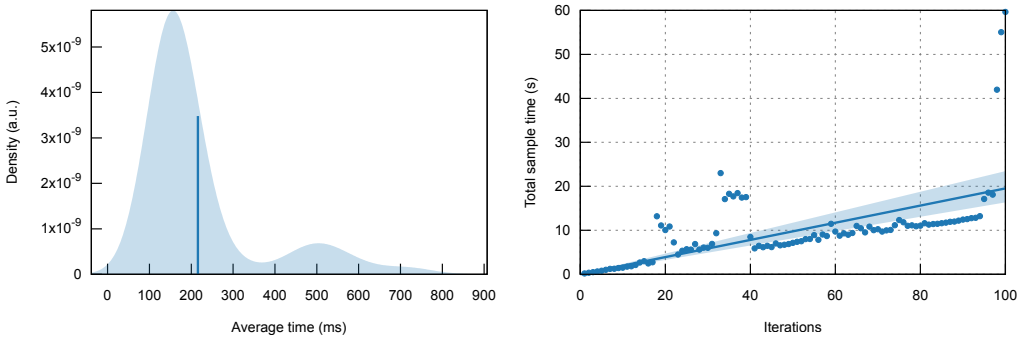
Benchmark tests are declared inside the `benches/benchmark.rs` file. They are written inside the `criterion_benchmark()` function and they can be launched by calling `cargo bench` inside the `benches` folder. These tests aim to measure the performances of:

- **HKDF Extract-and-Expand functions.** The two functions are performed in an average time of $20.19\mu s$.
- **encryption and decryption with ECKEM.** The two operations are performed in a similar time: encryption took an average of $239.53\mu s$, while decryption took $222.61\mu s$;
- **encryption and decryption with AES-128-GCM.** Encryption took an average of $9.24\mu s$, while decryption took $2.85\mu s$;
- **creation of a `UserInitKeys` bundle.** Bundle is created within an average of $83.42\mu s$;
- **creation of a group** with two members (Alice and Bob) and a larger group of 10 members. Benchmarks revealed that group creation took an average time of $3.13ms$ for a group with two participants (Figure 6.1) and $215.82ms$ for a group of 10 participants (Figure 6.2).




(a) Average time per iteration (b) Linear regression

Figure 6.1: Benchmark results for creating a group with two participants




(a) Average time per iteration (b) Linear regression

Figure 6.2: Benchmark results for creating a group with 10 participants

# Chapter 7

# Conclusions

Messaging is continuously evolving and will play an increasingly central role in a constantly connected world. Security in communications will be of primary importance in order to protect private conversations from third parties, and the industry is increasingly moving in this direction.

Although end-to-end encryption has recently been implemented worldwide in major messaging services, the focus on performance remains paramount.

In this context, Messaging Layer Security can play a fundamental role, both for the unity of intent of the participants of the working group and their companies, and for the intrinsic efficiency of the protocol that allows its use on mobile devices and possibly embedded.

It will be interesting to follow the future of this protocol in several aspects, first of all the transposition of the Internet Draft as IETF Standard, the evolutions, the various implementations and finally the adoption by the major players in the industry.

On a personal level, it was extremely interesting and compelling to follow the various phases of the project, observing the first intentions of work on the protocol, the decision of the name, and then go in depth during the recent months through the analysis of the protocol and the work done on Melissa.

# Appendix A

# Code examples

## A.1 Implementation of the `Codec` trait

```
1  pub struct HpkeContext {
2      ciphersuite: u16,
3      mode: u8,
4      kem_context: Vec<u8>,
5      info: Vec<u8>,
6  }
7
8  impl Codec for HpkeContext {
9      fn encode(&self, buffer: &mut Vec<u8>) {
10         self.ciphersuite.encode(buffer);
11         self.mode.encode(buffer);
12         encode_vec_u8(buffer, &self.kem_context);
13         encode_vec_u8(buffer, &self.info);
14     }
15
16     fn decode(cursor: &mut Cursor) -> Result<Self, CodecError> {
17         let ciphersuite = u16::decode(cursor)?;
18         let mode = u8::decode(cursor)?;
19         let kem_context = decode_vec_u8(cursor)?;
20         let info = decode_vec_u8(cursor)?;
21         Ok(HpkeContext {
```

```
22              ciphersuite,
23              mode,
24              kem_context,
25              info,
26          })
27      }
28  }
```

## A.2 Tests

### A.2.1 Encoding

```
1  #[test]
2  fn test_encode_primitives() {
3      // ...
4      let uint16: u16 = 1;
5      let mut buffer = Vec::new();
6      uint16.encode(&mut buffer);
7      assert_eq!(buffer, vec![0u8, 1u8]);
8      // ...
9  }
10
11 #[test]
12 fn test_encode_vec_u16() {
13     let v: Vec<u16> = vec![1, 2, 3];
14     let mut buffer = Vec::new();
15     encode_vec_u16(&mut buffer, &v);
16     assert_eq!(buffer, vec![0u8, 6u8, 0u8, 1u8, 0u8, 2u8, 0u8, 3u8])
           ;
17 }
```

# Bibliography


[1] E. Omara, B. Beurdouche, E. Rescorla, S. Inguva, A. Kwon, A. Duric, *The Messaging Layer Security (MLS) Architecture*. IETF MLS Working Group Internet-Draft Version 02, March 11, 2019. `https://tools.ietf.org/html/draft-ietf-mls-architecture-02`

[2] R. Barnes, J. Millican, E. Omara, K. Cohn-Gordon, R. Robert, *The Messaging Layer Security (MLS) Protocol*. IETF MLS Working Group Internet-Draft Version 05, May 02, 2019. `https://tools.ietf.org/html/draft-ietf-mls-protocol-05`

[3] E. Omara, R. Robert, *The Messaging Layer Security (MLS) Federation*. IETF MLS Working Group Internet-Draft, June 25, 2019. `https://datatracker.ietf.org/doc/draft-omara-mls-federation/`

[4] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, K. 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 (CCS '18). ACM, New York, NY, USA, 1802-1819. DOI: `https://doi.org/10.1145/3243734.3243747`

[5] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*. IETF RFC 8446, August 2018. `https://tools.ietf.org/html/rfc8446`

[6] K. Cohn-Gordon, C. Cremers, L. Garratt, *On Post-compromise Security*. 2016 IEEE 29th Computer Security Foundations Symposium (CSF), Lisbon, 2016, pp. 164-178. DOI: `https://doi.org/10.1109/CSF.2016.19`

[7] K. Cohn-Gordon, C. Cremers, B. Downling, L. Garratt, D. Stebila, *A Formal Security Analysis of the Signal Messaging Protocol*. 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, 2017, pp. 451-466. DOI: https://doi.org/10.1109/EuroSP.2017.27

[8] T. Perrin, M. Marlinspike, *The Double Ratchet Algorithm*. Signal Specifications, November 11, 2016. https://www.signal.org/docs/specifications/doubleratchet/doubleratchet.pdf

[9] W. Stallings, L. Brown, *Computer Security: Principles and Practice, Global Edition*. Pearson Education Limited, 2014.

[10] D. McGrew, *An Interface and Algorithms for Authenticated Encryption*. IETF Network Working Group RFC 5116, January 2008. https://tools.ietf.org/html/rfc5116

[11] R. Barnes, K. Bhargavan, *Hybrid Public Key Encryption*. IETF Network Working Group Internet-Draft Version 01, March 11, 2019. https://tools.ietf.org/html/draft-barnes-cfrg-hpke-01

[12] H. Krawczyk, P. Eronen, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. IETF RFC 5869, May 2010. https://tools.ietf.org/html/rfc5869

[13] N. Borisov, I. Goldberg, E. Brewer, *Off-the-Record Communication, or, Why Not To Use PGP*. In Proceedings of the 2004 ACM workshop on Privacy in the electronic society (WPES '04). ACM, New York, NY, USA, 77-84. DOI: http://dx.doi.org/10.1145/1029179.1029200

[14] I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, H. Chen, *Multi-party Off-the-Record Messaging*. In Proceedings of the 16th ACM conference on Computer and communications security (CCS '09). ACM, New York, NY, USA, 358-368. DOI: https://doi.org/10.1145/1653662.1653705

[15] K. Bhargavan, R. Barnes, E. Rescorla, *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. A protocol*

*proposal for Messaging Layer Security (MLS)*. May 3, 2018. https://mailarchive.ietf.org/arch/attach/mls/pdf1XUH6o.pdf

[16] R. Hust, G. Belvin, *Security Through Transparency*. Google Security Blog, January 12, 2017. https://security.googleblog.com/2017/01/security-through-transparency.html

[17] WhatsApp Inc., *WhatsApp Encryption Overview: Technical White Paper*. December 19, 2017. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

[18] T. Van Vleck, *Electronic Mail and Text Messaging in CTSS, 1965-1973* in IEEE Annals of the History of Computing, vol. 34, no. 1, pp. 4-6, Jan. 2012. DOI: https://doi.org/10.1109/MAHC.2012.6

[19] M. Williams, *Secure Messaging Apps*. https://www.securemessagingapps.com

[20] A. Greenberg, *Hacker Lexicon: What Is End-to-End Encryption?*, Wired.com, November 25, 2014. https://www.wired.com/2014/11/hacker-lexicon-end-to-end-encryption/

[21] StackOverflow, *StackOverflow Developer Survey Results 2016*. https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted

[22] D. J. Bernstein, *NaCl: Networking and Cryptographic library*. http://nacl.cr.yp.to

[23] D. J. Bernstein, *SafeCurves: choosing safe curves for elliptic-curve cryptography*. http://safecurves.cr.yp.to

[24] Rust By Example, *Unit testing*. Official Rust Documentation. https://doc.rust-lang.org/rust-by-example/testing/unit_testing.html

# List of Figures

# Listings

# Acknowledgements

First of all I would like to thank prof. Gabriele D'Angelo for following me in recent months as a supervisor on this thesis. Thanks also to Alan Duric and Raphael Robert for their important contribution in MLS and the invaluable help provided during the writing of the thesis, and of course my colleagues at Wire and *Berliner* friends with whom I spent and am going through some wonderful moments.

Thanks to my friends who supported me - and endured - during these three years, very often in front of *ein gutes deutsches Bier*: Daniele, Thomas, Alessandro, Fabrizio, Carla, Vanessa, Lorenzo, Giovanni and many others!

Thanks to my fellow students, Giacomo, Michele, Giacomo and Aldo, for having been with me in this crazy undertaking and for having shared long days of study, pre-exam anxieties and post-exam joys, but above all laughter and many beautiful moments spent together.

A special thanks also to the TEDxCittàdiSanMarino team for helping me carry out the organization of the event. Thanks to Carlotta for the linguistic revision of this thesis.

Last but not least, a very special thanks to my family, to whom this thesis is dedicated. You are my strength. Always.

*Nicola*
*July 18th, 2019*

# Ringraziamenti

Vorrei innanzitutto ringraziare il prof. Gabriele D'Angelo per avermi seguito negli ultimi mesi come relatore di questa tesi. Grazie anche ad Alan Duric e Raphael Robert per il loro importante contributo in MLS e il prezioso aiuto fornito durante la stesura della tesi, e naturalmente i miei colleghi di Wire e i miei amici berlinesi con i quali ho passato e sto passando dei bellissimi momenti.

Un grazie ai miei amici che mi hanno supportato - e sopportato - durante questi tre anni, molto spesso davanti a un boccale di buona birra tedesca: Daniele, Thomas, Alessandro, Fabrizio, Carla, Vanessa, Lorenzo, Giovanni e tanti altri!

Grazie ai miei compagni di studio, Giacomo, Michele, Giacomo e Aldo, per essermi stato accanto in quest'impresa folle e per aver condiviso lunghe giornate di studio, ansie pre-esame e gioie post-esame, ma soprattutto risate e tanti bei momenti passati assieme.

Un ringraziamento d'obbligo anche al team di TEDxCittàdiSanMarino per avermi dato una mano a portare avanti l'organizzazione dell'evento. Grazie a Carlotta per la revisione linguistica di questa tesi.

*Last but not least*, un grandissimo ringraziamento speciale alla mia famiglia, alla quale questa tesi è dedicata. Siete la mia forza. Da sempre.

*Nicola*
*18 luglio 2019*