



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



CloudMLS: A cloud-based E2EE scheme

Master's Thesis

Lukas Käppeli

lukask@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Ard Kastrati, Karolis Martinkus

Prof. Dr. Roger Wattenhofer

April 11, 2022

Acknowledgements

I thank my supervisors Ard Kastrati and Karolis Martinkus supporting me during the last six months. Their feedback and critical questions guided me toward the astonishing results of this thesis. I further want to thank the Department of Information Technology and Electrical Engineering at ETH Zurich for offering me a virtual machine running my back-end and supporting me in configuring it. Last but not least, I want to thank Prof. Dr. Roger Wattenhofer for giving me the possibility of working on a practical topic. His feedback after the midterm presentation was valuable and had a huge impact on the continuation of the project.

Abstract

While some of the most used instant messengers offer end-to-end encryption and some of them a cloud-based message store, none provides both. In this thesis, we present a Node.js based library providing end-to-end encryption for cloud-based messengers. We use the Message Layer Security protocol as a key establishment protocol and adapt it to a cloud-based setting. By adding this layer of security, users are no longer required to trust the instant messenger platform storing their messages encrypted. We further present an example application securing Telegram chats by utilizing our library.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
1.1 Popular Instant Messengers	1
1.2 Goals	3
2 Background	4
2.1 Group chats	4
2.2 Message Layer Security	4
2.3 Cloud-based E2EE	5
2.3.1 Limitations of cloud support in Signal	5
2.3.2 Storing messages in the cloud	5
2.3.3 Perfect Forward Secrecy	6
2.4 Related work	7
3 CloudMLS - The Implementation	8
3.1 Core Concept	8
3.2 Architecture	8
3.2.1 The core	10
3.2.2 The library	11
3.2.3 The key server	26
3.3 Athena - The Telegram example	35
3.3.1 Application overview	35
3.3.2 Registration and Login	35
3.3.3 Telegram Service	36
3.3.4 Chat overview page	37

CONTENTS	iv
3.3.5 Chat page	37
3.3.6 Contact page	37
3.3.7 Settings page	38
3.4 Possible clients	38
3.5 Email	40
3.5.1 Missing identifiers	40
3.5.2 Group mutations	40
4 Future work	42
4.1 CloudMLS	42
4.1.1 Epoch Base Ratchets	42
4.1.2 Enable native devices	43
4.1.3 Changing the password	43
4.1.4 PBKDF2	43
4.1.5 Public key cryptography	44
4.1.6 Adapt to draft-13	44
4.2 Key server	45
4.2.1 Custom key server	45
4.2.2 Space efficiency	45
4.3 Backlog	45
4.4 EU Regulations	46
5 Conclusion	48
Bibliography	49
A MongoDB Schemata	A-1

Introduction

The most used instant messengers are WhatsApp, Facebook Messenger, WeChat, QQ, and Telegram. While some of those messengers offer end-to-end encryption (E2EE) and some of them are cloud-based, none of them provide E2EE in a cloud-based setting. Taking the example of Telegram offering a cloud-based service, where a user can log in from any device and access his complete message history, Telegram's chats are not end-to-end encrypted by default. There exists the possibility to use E2EE in Telegram's secret chats, these are, however, bound to the device where they were initiated. This limitation showcases the lack of cloud-based E2EE in modern instant messengers. In addition, new messengers suffer from a low adoption rate, due to vendor lock-in, even if they provide excellent security and usability like Signal does. In this Master's Thesis, we present an open-source library providing a fully cloud-based E2EE scheme based on the Message Layer Security (MLS) Protocol [1], currently developed by an IETF working group. Instead of creating a new instant messenger suffering from a low adoption rate, our library can be integrated into any Node.js based client.

1.1 Popular Instant Messengers

We provide an overview of the most used instant messengers. In our comparison, we additionally include Signal as one of the most secure applications as well as Threema, a popular solution from Switzerland. We do, however, omit WeChat and QQ, two Chinese messengers which are not end-to-end encrypted.

WhatsApp With 2 billion monthly active users, WhatsApp is by far the most used instant messenger. It claims to use the Signal Protocol to provide E2EE for every chat, but as WhatsApp is not open source, this claim cannot be verified. The platform is not cloud-based and relies on companion devices authenticating itself to the primary device running the Android or iOS application. This mechanism implies that the primary device has to be online while a companion device wants to use the service.






	WhatsApp	Facebook	Telegram	Signal	Threema
Logo					
Monthly active users in millions	2000 [2]	1300 [3]	400 [4]	40 [5]	8 [6]
Open Source	✗	✗	✓	✓	✓
E2EE	✓	Secret chats only	Secret chats only	✓	✓
cloud-based	✗	✓	✓	✗	✗
Group chat encryption (see 2.1)	Server-side fan-out (Sender keys)	Client-side fan-out	✗	Client-side fan-out	Client-side fan-out

Table 1.1: Instant messenger overview

Facebook Messenger In contrast to WhatsApp, Facebook Messenger is not end-to-end encrypted by default. Users have the option of creating a secret chat, which provides E2EE. These secret chats are not fully cloud-based, as the message history cannot be read on new authenticated devices. There also exists the possibility of creating secret chats for groups.

Telegram Telegram is the most used open-source messenger and has a public API, allowing individuals to create custom telegram clients, as we show later on. Like Facebook Messenger, Telegram does not encrypt messages by default but offers the option of secret chats as well. These secret chats are device-specific which implies that they are accessible exclusively on the device, where they were created. There is further no possibility of creating secret group chats.

Signal In terms of security, Signal is one of the best instant messengers. As an open-source platform, its E2EE is verifiable by everyone. Signal's only drawback is that it is not cloud-based. Messages can be read on any connected device. But when adding a new device, the message history is unavailable.

Threema Developed in Switzerland, Threema is not subject to the CLOUD Act [7] and can protect the users' privacy even more than its competitors by not being forced to disclose user information when requested by the U.S. government.

By deleting delivered messages from its servers, Threema is not a cloud-based platform. At the time of writing, multi-device support is still in development, making Threema currently the least portable messenger in this comparison.

1.2 Goals

When looking at Table 1.1, we note that there does not exist an instant messenger that is both E2EE and cloud-based. We further note that the most used platforms do not provide verifiable E2EE or provide E2EE by limiting the usability. Therefore, our goal is to create a way of providing an open-source and cloud-based plugin for these platforms to enhance their security. On the other side, we want to emphasize that if someone's concern is security and security only, he should consider switching to Signal or Threema. We formulate the main goals of this thesis in the following.

- Create an additional layer of security for existing platforms by providing an end-to-end encryption scheme.
- Implement a cloud-based solution providing a maximum level of usability.
- Keep the required amount of trust in external infrastructures as low as possible.

Background

2.1 Group chats

There are two commonly used approaches to encrypt messages in a group chat setting. WhatsApp uses the server-side fan-out approach [8] with sender keys. These keys get generated upon the first message written to the group and whenever a user leaves the group. After its generation, the key gets distributed over the pairwise encrypted channels between the generator and each other group member. This key then serves for deriving subsequent keys analogous to the symmetric-key ratcheting from the Double Ratchet Algorithm developed by Signal [9]. On the other side, Facebook Messenger, Signal, and Threema use a client-side fan-out approach, where each message is encrypted separately for each group member. In Signal, each user's devices are treated as separate group members, requiring that a message is not only encrypted for each user, but also for each device being part of the group. Both of these approaches are inefficient as they require a linear number of encryption operations per message (client-side fan-out) or for distributing the sender keys (server-side fan-out).

2.2 Message Layer Security

To overcome the issues mentioned in Section 2.1, an IETF working group is currently developing a new asymmetric group key establishment protocol called Message Layer Security (MLS) [1]. The protocol uses a binary tree-based approach to derive encryption keys, where each group member is represented as a leaf of that tree. The required data structure is stored at each group member and kept synchronized by `Commit` messages broadcast upon certain events. `Commit` messages can contain, amongst others, `Add` and `Remove` proposals, indicating that members are added or removed from the group. Similar to the Double Ratchet Algorithm [9], each participant publishes a signed public key, which gets then used when the participant gets added to a group. The MLS Protocol has three main advantages over the client-side and server-side fan-out

approaches. First, it provides perfect forward secrecy by using a symmetric key ratcheting strategy similar to the Double Ratchet Algorithm. Second, the protocol achieves post-compromise security as participants regularly update their signed public keys resulting in **Update** proposals that contribute fresh entropy when committed. Third, using a binary tree structure, all operations on the tree have a time complexity logarithmic to the number of group members.

2.3 Cloud-based E2EE

In this section, we explore how cloud-based E2EE could work. We start with Signal as a basis and then elaborate step by step on how we could achieve a cloud-based E2EE. For each step, we will show its advantages and disadvantages.

2.3.1 Limitations of cloud support in Signal

When a user connects a new device to its Signal account, message histories are not available on that device. Signal only stores messages in transit, meaning that delivered messages get removed from the server. So the first step toward a cloud-based version of Signal would be to keep every message stored in a way that an authenticated user can fetch them. To support multiple devices per user, Signal uses the Sesame algorithm [10]. Sesame maintains a state for every authenticated device of a user. Therefore, if Alice has two authenticated devices and wants to send a message to Bob having two devices as well, Alice has to encrypt the message three times, once for Alice's other device and twice for Bob's devices. A naive approach towards a cloud-based version of Signal would be to simulate that a user has multiple devices already upon registration. In terms of private keys, for each of these virtual devices, the main device would store all private keys up until the point when a new device gets added. When authenticated, the private keys could be transferred to this new device using a secure channel. This approach is very inefficient and does not lead to a fully cloud-based approach as the required number of virtual devices is unknown.

2.3.2 Storing messages in the cloud

In a cloud-based setting, a user can access his complete message history on any device at any time after having authenticated himself. Therefore, all messages must be stored on some server or decentralized message store like matrix [11]. To reduce the trust in external infrastructures, messages should either be stored encrypted or on trusted servers. Establishing trust could be achieved by each user hosting the server itself or by some form of remote attestation like Intel SGX [12]. Both options seem to be impractical in terms of complexity and efficiency.

Let us therefore focus on how to store messages encrypted on untrusted infrastructure. Using asymmetric cryptography, a user needs to store the private key on all his devices that should have access to the message store. This requirement could be met by creating a secure channel for transmitting the private key to new devices associated with a user. From the usability perspective, if a user wants to log in from a new device, it resembles a two-factor authentication, as another registered device is required to be online for the transfer of the private key. Considering symmetric cryptography for encrypting the messages, using a password-based key derivation scheme, a user can authenticate himself using a password, providing a truly cloud-based way of storing encrypted messages. When deriving keys based on user-chosen passwords, we do, however, sacrifice some bits of security compared to using random keys.

2.3.3 Perfect Forward Secrecy

In terms of security, the most interesting question is, if there is a way to provide perfect forward secrecy (PFS) for cloud-based E2EE. There are two approaches to defining PFS.

PFS based on encryption keys

Definition 2.1. Let $c_i = E(m_i, k_i)$, for $i = 0, 1, \dots$, be a series of ciphertexts, each the encryption of some message m_i , encrypted with some key k_i . An encryption scheme provides PFS, if for any leaked key k_i , an adversary cannot decrypt ciphertexts c_j , for all $j < i$.

When considering this first definition of PFS, we see that cloud-based end-to-end encryption schemes are feasible. As the definition only considers leaked encryption keys, it does not matter how the data structures, used to derive these keys, are stored.

Taking the example of Signals implementation of the Double Ratchet Algorithm, which uses the local storage of its clients to store the data structures used to derive the encryption keys. This approach provides PFS and would still provide PFS if we would use an external infrastructure for storing these data structures. Note that replacing device storage or even secure storage with a third-party server requires the same amount of trust in this third party.

PFS based on long term keys

Definition 2.2. Let $c_i = E(m_i, k_i)$, for $i = 0, 1, \dots$, be a series of ciphertexts. If at any point in time t , the long term private keys of any participating party is leaked, then an adversary cannot decrypt ciphertexts c_j , for all $j < t$.

As this second definition of PFS depends on the secret keys, the question of whether a cloud-based E2EE providing PFS can be created becomes much more complex. As elaborated in Section 2.3.2, most approaches for a cloud-based E2EE rely on a server storing messages in some encrypted form. And as to the cloud-based nature, a user wants to access his messages, therefore requiring some form of authentication based on a password or key. This implies that there cannot be any cloud-based E2EE that provides PFS in terms of a user's long-term secret or password.

2.4 Related work

Despite still being in development, there are already some published papers providing the first security analyses on the Message Layer Security protocol. As MLS is stated to be in a feature freeze by the draft version 11, these security analyses could provide a persistent basis for future versions of MLS. Bhargavan et al. [13] formally defined group messaging in the functional programming language F^* . Using their framework, they were able to find attacks and weaknesses in draft version 7 of MLS. Their proposed improvements were then added to the following versions of MLS.

In a similar work, Almen et al. [14] formalized the core parts of the MLS draft version 10 and analyzed its security guarantees in terms of (adaptive) insider security. Fulfilling the notion of insider security means that it is possible for a group communication protocol to provide private and authenticated message exchange in a network controlled by an adversary also controlling some malicious group members. Their results already led to two improvements in the MLS protocol as well. As the MLS draft version 11 builds the basis of our implementation, we incorporate the improvements of these analyses.

CloudMLS - The Implementation

The CloudMLS project has three core parts, an MLS implementation, a library around it, and a key server. In this section, we are going to show the implementation details of each part and how they interact with each other. Each component is implemented with Typescript as a Node.js project.

3.1 Core Concept

The library creates a layer of security for existing instant messengers. In terms of the OSI model, this library belongs to the sixth layer. The idea is that a user can associate accounts from instant messengers in order to send end-to-end encrypted messages using those instant messengers. For example, assume two users Alice and Bob both have a Telegram account. If then Alice and Bob use a Telegram client with CloudMLS, they can find each other's KeyPackages on the CloudMLS Authentication Server and start writing end-to-end encrypted messages through Telegram. In order to provide this E2EE, CloudMLS uses the MLS protocol and stores the required data structures in an encrypted form on the key server. The current approach for keeping the data on the key server consistent uses a locking mechanism. A client application therefore needs to fetch the specific data structure, modify it in the appropriate manner, and then push it back to the server. To access the different resources on the server, users must authenticate themselves using a username and password. Thus, in order to access Telegram messages in a CloudMLS client, a user has to authenticate itself twice, once for CloudMLS and once for Telegram.

3.2 Architecture

The MLS protocol specifies two services that need to be provided. First, a Delivery Service is expected to ensure in-order delivery of all MLS-specific messages, like `Proposal`, `Commit` and `Welcome` messages. Second, there must be an Authentication Service, where protocol participants can authenticate MLS KeyPackages

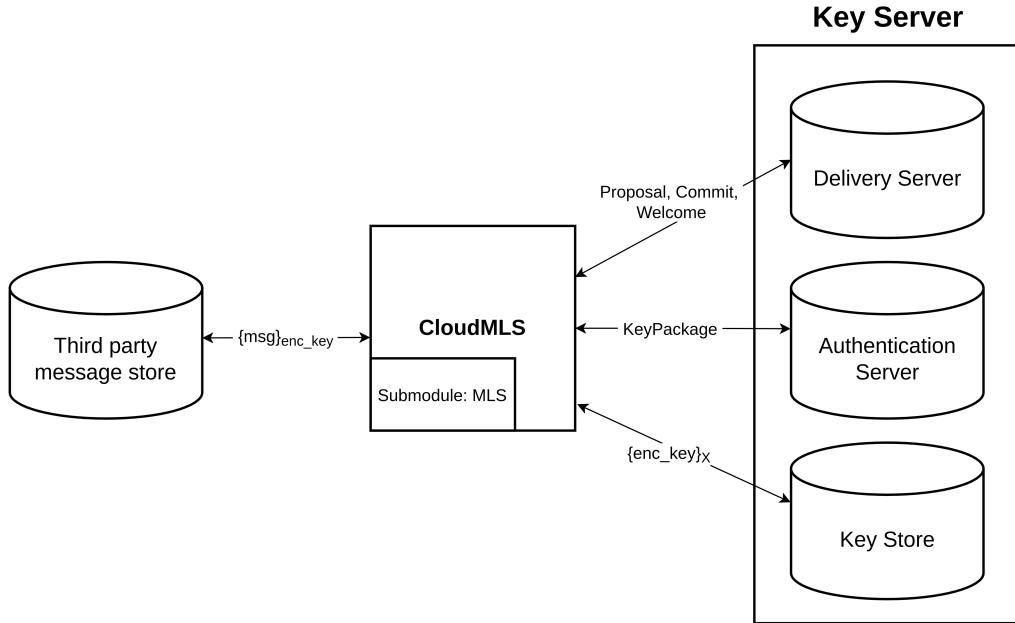


Figure 3.1: High-level library overview.

from other participants. We therefore have integrated these functionalities into our key server as described in Section 3.2.3.

In Figure 3.1, we show the interaction between the various components involved. Starting from left to right, the first instance builds the third party message store, which could be the back-end of any third party messenger. To show how the CloudMLS library gets integrated into a messenger, we provide in Section 3.3 a sample implementation that uses Telegram’s public API. The main idea is to send end-to-end encrypted messages over a third-party platform and store the encryption keys, respectively the data structures used to derive the keys, in a cloud-based key store. How these encryption keys get encrypted in the Key Store is up to the user, marked by X in Figure 3.1. The CloudMLS library, visualized in the middle of the figure acts as a wrapper around the MLS library [15] and functions as an abstraction for the communication with the key server components. The specifications can be found in Section 3.2.2. Finally, on the right side, the three components of the Key Server are shown, where the Delivery Server and the Authentication Server are required services for the MLS protocol. The key store is the core service for the cloud-based E2EE by holding encrypted data structures for the MLS protocol.

3.2.1 The core

As the MLS protocol is still in development, there are few available implementations yet. The basis for our library builds the proof of concept implementation from Hubert Chathi [15]. Besides some minor bug fixes and enabling the serialization of certain objects, the library is used as Hubert Chathi created it. We are going to document all major changes hereafter.

Serializing Group objects The data structure `Group` (`src/group.ts`) is required for all group-specific operations like commits as well as for encrypting and decrypting messages. We were therefore required to serialize the `Group` objects as well as create them from serialized versions. As a data serializable structure, we relied on JSON objects, which can be sent in an HTTP POST request without further transformations. In order to serialize a `Uint8Array`, the Node.js library `byte-base64` is a very efficient option offering fast transformation between `Uint8Array` and base64 strings.

Serializing RatchetTreeView objects Each `Group` object contains a `RatchetTreeView` object representing the current ratchet tree. Because this data structure is required for encrypting and decrypting messages, its state needs to be serialized too. Using the same techniques as for the `Group` objects, efficient serialization and parsing are achieved.

Use of LenientHashRatchets In the implementation from Hubert Chathi are two types of hash ratchets defined. The standard `HashRatchet` object represents the specification of the MLS protocol, whereas the `LenientHashRatchet` objects allow deriving the same encryption keys multiple times. In a cloud-based setting, we require re-deriving the same keys because users might want to read the same message on multiple devices. Another option would be to store the first ratchet of each group member and use this for further key derivations. This approach is documented separately in Section 4.1.1.

Lifetime Extension Missing in the original implementation, we added the required `Lifetime` extension to the `KeyPackage` object. This extension defines the period in which a `KeyPackage` is valid. As users should update their `KeyPackages` before they expire, we implemented a method determining how long a certain `KeyPackage` is valid.

3.2.2 The library

The CloudMLS library provides a simple interface and handles every interaction with the required servers. Between login and logout, a client application is not required to perform any HTTP requests itself but can leave everything to the library. In this section, we first describe how authentication and authorization are implemented before documenting the complete library file per file, including the control flow between each other.

Authentication

A user is authenticated using JSON Web Tokens (JWT) according to RFC 7519 [16]. For detailed documentation of the server-side implementation, see Section 3.2.3. In short, when a user successfully provides correct credentials in the login request, the server signs two JWT's holding the user identifier and attaches them as HTTP-only cookies. Using HTTP-only cookies ensures that these cookies cannot be read or modified by any JavaScript code, mitigating Cross-Site Scripting attacks [17]. The first token is a short-lived access token, whereas the second token is a long-lived refresh token allowing a user to receive new access tokens without providing its credentials again.

Keystore

`src/keystore.ts`

Inside `src/keystore.ts`, we include an abstract `Keystore` class with getters and setters for four variables as well as a `delete()` method. The four variables a `Keystore` implementation has to store are the encryption key for content on the key server, an encryption key for values stored locally, the username of the currently logged-in user as well as a hash of it.

We provide a default implementation of this abstract class, storing all variables in memory. It is also possible to provide a custom key store extending the abstract base class by setting the public field `keystore` of the `CloudMLS` instance. This can be useful for native applications that want to use the file system for storing these values.

Storing these variables in memory has two advantages. First, this solution works in applications running in a browser as well as for native applications. Second, when running in a browser, we do not have to rely on the browser's local storage as this storage can be accessed by any JavaScript application running in the same browser, making it vulnerable to cross-site scripting attacks.

Concerning the values we store, the key for encryption data on the key server is required as we do not want to trust an external infrastructure. We also derive a key for storing values locally. Taking the example of the `MTPProton` library [18] used to make requests to the Telegram API, it requires some persistent storage

for storing authentication keys such that the user does not have to log in again each time. In order to keep the required key in persistent storage, our Telegram example client, documented in Section 3.3, stores these keys encrypted with the local storage encryption key in the browser's local storage. This local storage is a key-value store, for which the key is a hash of the username concatenated with the actual key. Using this approach, an attacker performing a cross-site scripting attack cannot see, who is using the application. For the username, this could be required in several parts of an application so we just store it. And finally, the `delete()` method clears the storage when the user logs out.

Authentication Service

`src/authentication.service.ts`

The Authentication Service authenticates a user to the server and consists of three methods:

- `register(username: string, password: string)`
- `login(username: string, password: string)`
- `logout(navigationCallback?: () => void)`

In the current configuration, registering a user hashes the user-chosen password and uses the second half of the hash output to register the user at the server by sending an HTTP POST request. SHA-512 provided by the Node.js library `crypto-js` is thereby used as a hash function. The reason for hashing the user's chosen password is that the first half of the password is used to derive the keys for encrypting data stored in the browser's local storage as well as encrypting data structures stored on the server. These keys are derived when the user logs into the application using a dedicated salt value retrieved from the server upon successful login. A schematic of this approach is visualized in Figure 3.2. Using this method for deriving these encryption keys, it is not possible to keep a user logged in when using a browser. How this can work for native clients, see Section 4.1.2. After having stored all required values into the Keystore, the third-party platform accounts associated with the current user are added to the Account Service. When calling the logout function, the user is logged out from the server, the states from other services are cleared and an optional navigation callback is called.

Account Service

`src/account.service.ts`

Each user can associate multiple instant messenger accounts with one CloudMLS account. We therefore need some service managing these third party accounts. This service is implemented using the following methods inside the Account Service:

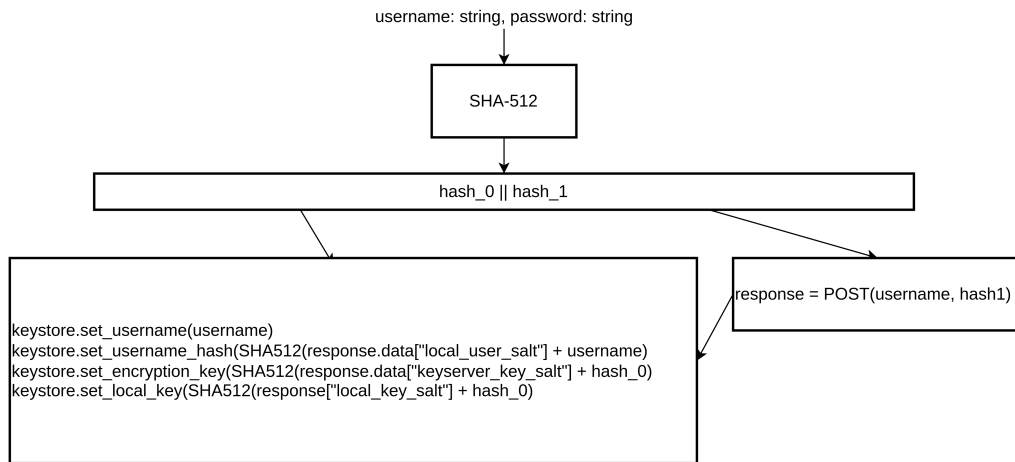


Figure 3.2: Key derivation for keys that encrypt data on the server or in the browser localStorage

- `setAccounts(accounts: Account[])`
- `addAccount(platform: string, account_id: string)`
- `updateKeyPackage(account: Account, forced?: boolean)`
- `createKeyPackage(account: Account)`
- `destroy()`

The data structure `Account` is composed of the platform name (e.g. telegram, facebook, etc.), the platform specific identifier of the user and the serialized version of the corresponding MLS `KeyPackage`. It has the following structure:

```

type Account = {
  platform: string,
  account_id: string,
  keypackage: string
}
  
```

The method `setAccounts()` gets called from the Authentication Service when a user performs a login. The accounts passed as an argument, are included in the response of the server upon a successful login attempt. This response includes all third-party platforms a user has associated with its CloudMLS account. Within `setAccounts()`, for each of these accounts, MLS-specific messages are fetched from the delivery server to process these messages as early as possible. Further, we verify that the `KeyPackages` of all accounts are valid and do not expire in the near future by calling `updateKeyPackage()`. Currently, a `KeyPackage` is replaced seven days before expiring. If a replacement is necessary, `createKeyPackage()` is

called which first fetches from the server all groups of which the specified account is part. Then for each of these groups, a `Commit` message containing the update message is sent to all group members. Note that in the most recent MLS draft, this should be handled differently, which is explained in Section 4.1.6.

```
interface PackageData{
  keypackage: Uint8Array,
  signingPrivateKey: Uint8Array,
  signingPublicKey: Uint8Array,
  hpkePrivateKey: Uint8Array,
  hpkePublicKey: Uint8Array,
  credential: Uint8Array
}
```

Concerning the `KeyPackage`, there are two types of data structures required. As a user must be able to fetch `KeyPackages` from every user, the `KeyPackage` itself is stored in a publicly accessible way. We therefore store the `KeyPackage` itself on the Authentication Server. The corresponding private keys and credentials are stored on the key server only accessible to the creator of the `KeyPackage`. We refer to this data structure as `PackageData`, which includes the signing and HPKE key pair as well as the credential that is signed in the `KeyPackage`. The credential is also required for creating groups as well as committing update messages.

The last method inside the Account Service is the destroy function that clears the state upon a user logs out.

Delivery Service

`src/delivery.service.ts`

The Delivery Service is used for sending and receiving MLS-specific messages like `Welcome` messages for new members and `Commit` messages. These message types are indicated using the following enumeration type:

```
enum MessageType {
  WELCOME = 0,
  COMMIT = 1
}
```

This service is not intended to be used for regular message exchanges between users, we instead rely on third party platforms to deliver these messages. In order to fulfill the MLS protocol requirement of in-order delivery, we use the following methods:

- `fetchMessages(platform: string, account_id: string)`
- `storeMlsMessage(`
 - `from: string,`
 - `to: string,`
 - `platform: string,`
 - `group_id: string,`
 - `message_type: MLS.MessageType,`
 - `mls_message: string,`
 - `creationTime: number``)`
- `sendAll()`
- `destroy()`

The global interface `Message` is used to represent messages in our library and contains the platform-specific identifier of the sender as `src_account`, the group identifier of the group for which the message is intended, the message type according to the enum type from above, the creation time of the message and the serialized MLS message.

```
interface Message {
    src_account: string,
    group_id: string,
    message_type: number,
    creationTime: number,
    mls_message: string
}
```

Within `fetchMessages()`, all MLS messages for the specified account are fetched from the delivery server. For each of these messages, the corresponding group data structure is requested from the key server if existing, then both are passed to the `handleMlsMessage()` method of the MLS Service, documented in Section 3.2.2. The updated `GroupState` is pushed to the key server afterward. Whenever the library needs to send an MLS message, for example when a group is created or members are added to a group, the message is stored using `storeMlsMessage()`. Upon all messages are stored, the complete batch is sent at once ensuring the in-order delivery. Taking the example of a member being added to a group, the member that adds the new member commits an `Add Proposal` resulting in a `Welcome` message for the new member as well as a `commit` message for all other members. It therefore makes sense to store all messages and send them after all operations are finished. Currently, there are two types of MLS messages, `Welcome` and `Commit` Messages. When applying the changes described in Section 4.1.6, there will also be a new type representing `Proposal` messages.

Finally, as in the previous services, the `destroy` method clears the state of the service.

Group Service

`src/group.service.ts`

The Group Service handles creating groups as well as adding and removing members from them. We therefore require a data structure representing the state of a group.

```
interface GroupState {
  group_id: string,
  members: Set<string>,
  creationTime: number,
  mlsEpochState: Map<number, string>,
  latestEpoch: number,
  updateCounter: number
}
```

Each group is identified by a unique identifier, provided by the third-party platform. We do also store the platform-specific identifiers of each group member as a separate set such that we do not have to extract them from the MLS group data structure each time. As some members can create multiple groups with the same id at the same time, we use a timestamp to break ties, documented inside the `checkGroupState()` method. The `mlsEpochState` maps the epoch numbers to the corresponding serialized `GroupState`. As a client may want to decrypt messages from a previous epoch, these group states have to be kept as well. There is, however, the possibility to delete old group states, see Section 4.1.1. When a change is committed, it is always applied to the most recent `GroupState` denoted by the `latestEpoch` variable. The update counter is required to ensure consistency on the key server. Assuming that the same user is logged in on multiple devices at the same time, we ensure that changes of different clients are not overwritten. We therefore increment the update counter each time we update the data structure.

Using the following methods, we provide two approaches to manage the groups itself:

- `checkGroupState(`
 `platform: string,`
 `account_id: string,`
 `group_id: string,`
 `group_members: string[]`
 `)`

- `createGroup()`

```
    platform: string,  
    account_id: string,  
    group_id: string,  
    group_members: string[]  
  )
```
- `addMembers()`

```
    platform: string,  
    account_id: string,  
    group_id: string,  
    newMember_ids: string[]  
  )
```
- `removeMembers()`

```
    platform: string,  
    account_id: string,  
    group_id: string,  
    oldMember_ids: string[]  
  )
```

The first approach is a passive one enabling a client application to just call `checkGroupState()`, where it is determined if a new group can be created or if the existing group requires some members to be added or removed. The logic therefore is shown in Figure 3.3. The first step of the method is to verify if all members have a valid `KeyPackage`. If this is the case and the group is not on the user's key server, it gets created. Else, in the case where the group already exists, the MLS group and the actual group are compared and any change of group members from the client-side (e.g. from the third party platform) are applied to the MLS group. On the other side, if not all `KeyPackages` are valid, there are two possibilities. First, there is no `GroupState` defined, implying that no MLS group was created yet. In this case, nothing has to be done as this group represents a standard group in the third-party messenger. However, if there is a `GroupState` existing, there are two possibilities leading to this state. First, the `KeyPackage` of some members expired. This is per se not a problem as these members are already part of the group. Second, there were added members without a valid `KeyPackage` to the group using the third party platform itself, for example using the Telegram App. In this case, we cannot add these members to the MLS group and they will not be able to decrypt the received messages.

While the first approach managed groups passively, we provide with our second approach the possibility to actively create groups as well as adding and removing members. It contains the methods `createGroup()`, `addMembers()`, and `removeMembers()`. Using these methods to manage groups is more efficient by enabling fine-grained membership control. When creating a group, the `KeyPackages` for all members are fetched from the Authentication Server and passed

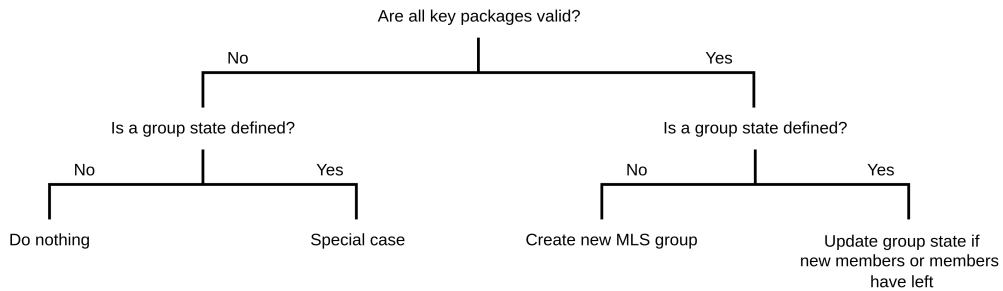


Figure 3.3: Simplified logic of the checkGroupState method

to the MLS Service, where the group is created and the corresponding `Welcome` messages are built. Then, within `createGroup()`, the `Welcome` messages are sent to each member using the delivery service and the `GroupState` is created and posted to the key server. When members are added or removed from the group, we first fetch messages from the Delivery Server to ensure that we do not miss any recent group modifications. We then get the `GroupState` and the `PackageData` from the key server and generate the corresponding update messages using the `generateUpdateMessages()` method within the MLS Service. In the end, all messages are posted to the Delivery Server and the updated `GroupState` is stored on the key server.

Dialog Service

`src/dialog.service.ts`

We refer to a dialog as a chat between two users. The Dialog Service just contains a simplified version of `checkGroupState()`, called `checkDialogState()`. This service is just used for passive dialog management in the same way as groups are managed. For active dialog management, a client application can just use the methods within the Group Service, respectively just the `createGroup()` method, as adding and removing members from a dialog seems to represent a rare use-case. The reason for using MLS in dialogs is mainly simplicity. We are aware of the fact that the Double Ratchet Algorithm would be more efficient, but using MLS enables us to keep the complexity of the project as simple as possible, without sacrificing security.

KeyServer Service

`src/keyServer.service.ts`

The purpose of the KeyServer Service is the communication between the library and the key server. This service handles the encryption and decryption for all sensitive data stored externally and uses the keys from the `KeyStore`, see Section 3.2.2. It implements the following methods:

- `getKeyPackage(platform: string, account_id: string)`
- `getKeyPackageData(platform: string, account_id: string)`
- `updateKeyPackage(`
 - `platform: string,`
 - `account_id: string,`
 - `keyPackageData: MLS.PackageData`
 - `old_keypackage?: string``)`
- `getGroups(platform: string, account_id: string)`
- `getGroupState(`
 - `platform: string,`
 - `account_id: string,`
 - `group_id: string,``)`
- `postGroupState(`
 - `account_id: string,`
 - `platform: string,`
 - `groupState: KeyServer.GroupState``)`
- `removeOldStates(`
 - `platform: string,`
 - `account_id: string,`
 - `group_id: string,`
 - `epoch: number``)`

The `getKeyPackage()` method is used to fetch `KeyPackages` from other users if existing. It performs a GET request on the Authentication Server, whereas the `getKeyPackageData()` method fetches `PackageData` from the key server. When requesting `PackageData` from the key server, the response data must first be decrypted using the key stored in the `KeyStore`, see Section 3.2.2. When for an `Account`, a `KeyPackage` is created or refreshed, it is updated on the Authentication Server as well as on the key server. In the current setup, where both servers run on the same machine (and process) using the same database, we do only upload it once, having the `PackageData` encrypted. See Section 4.2.1 for more details on this topic. In order to ensure that a user does not update the `KeyPackage` from two devices concurrently, we include the old `KeyPackage`, if existing, in the request such that the servers can verify that no intermediate update has taken place.

When `KeyPackages` are updated, an update message is issued to all groups, of which the user is part. We provide the method `getGroups()` returning a list of

all group identifiers for which the requested account has a `GroupState` on the key server. This method enables the library to determine which group states have to be updated alongside the `KeyPackage`. The group states itself can be fetched using the `getGroupState()` method and posted using the `postGroupState()` method. Both methods access the encryption key from the `KeyStore` ensuring that the group states on the key server are encrypted and not readable by the external infrastructure.

Despite being tested and functional, the method `removeOldStates()` should not be used at this stage. For further information, see Section 4.1.1.

Message Service

`src/message.service.ts`

The Message Service is used a wrapper for the methods `encryptMessage()` and `decryptMessage()` of the MLS Service. It provides an additional layer of abstraction by handling all interactions with the key server such that the MLS Service can handle encryption and decryption without the need to request external data. The Message Service therefore only provides the following two methods:

- `encrypt(`
 `plaintext: string,`
 `platform: string,`
 `account_id: string,`
 `group_id: string`
 `)`
- `decrypt(`
 `ciphertext: string,`
 `date: number,`
 `platform: string,`
 `account_id: string,`
 `group_id: string,`
 `)`

The encryption of plaintext is straightforward. First, the `GroupState` of the group with the id `group_id` is fetched from the key server. If there is no such group, the plaintext is returned as-is since there is no MLS group for which the message could be encrypted. After calling the MLS Services `encryptMessage()` method, the updated `GroupState` is posted to the key server and the ciphertext is returned to the caller.

For decrypting a message, the corresponding `GroupState` is again fetched from the key server. If the group creation time is after the message creation time passed as the variable `date`, the message is returned as-is since it was created before the group and is therefore most likely just standard plaintext. Using this

first filtering method, the number of calls to the decrypt function is reduced significantly. If the message was created after the group, we pass it to the MLS Services function `decryptMessage()`, upload the updated `GroupState` and return the plaintext in case of successful decryption.

MLS Service

`src/mls-wrapper.ts`

The MLS Service serves as a wrapper for the MLS implementation from Hubert Chathi [15]. It does only import classes and functions from the MLS submodule and types declared in `src/types.ts`. The wrapper is further the only Service importing functionality from the submodule. This approach makes it very simple to adapt the library to changes within the MLS implementation as the wrapper is the only service that has to be adjusted. The service itself provides the following methods:

- `isKeyPackageValid(`
 `keypackage_bytes: Uint8Array,`
 `future_time?: number`
 `)`
- `createKeyPackage(unique_user_id: string)`
- `createGroup(`
 `group_id: string,`
 `keypackage_buffers: Uint8Array[],`
 `my_keypackage: MLS.PackageData`
 `)`
- `commitUpdate(`
 `groupState: KeyServer.GroupState,`
 `newKeyPackageData: MLS.PackageData`
 `)`
- `generateUpdateMessages(`
 `newKeyPackages: Uint8Array[],`
 `leftMembers: string[],`
 `groupState: KeyServer.GroupState,`
 `my_keypackage: MLS.PackageData`
 `)`
- `commit(`
 `groupState: KeyServer.GroupState,`
 `my_keypackage: MLS.PackageData,`
 `proposals: Proposal[]`
 `)`
- `handleMlsMessage(`
 `message: Delivery.Message,`

```

    my_keypackage: MLS.PackageData,
    groupState: KeyServer.GroupState
  )
  • encryptMessage(
    groupState: KeyServer.GroupState,
    message: string,
    keyPackageData: MLS.PackageData
  )
  • decryptMessage(
    groupState: KeyServer.GroupState,
    message: string,
  )

```

The method `isKeyPackageValid()` takes two arguments, the `KeyPackage` itself and possibly a number representing some time. The `KeyPackage` contains a `Lifetime` extension containing two time values `not_before` and `not_after` specifying the time period in the `KeyPackage` is valid. The logic determining what the method returns can be found in Figure 3.4.

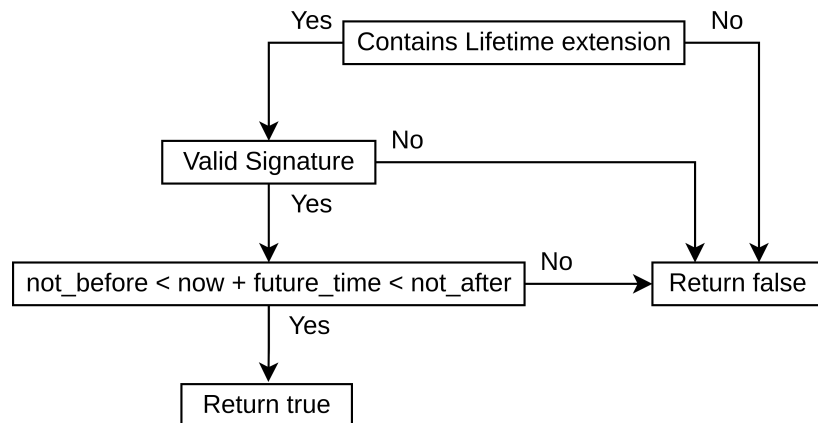


Figure 3.4: Simplified logic that determines if a `KeyPackage` is valid. Note that when `future_time` is not set, it is considered as equal to zero.

Creating `KeyPackages` can be done straightforwardly by generating the required keys and then calling the `KeyPackage.create()` method from the `MLS` submodule. We currently create `KeyPackages` that are valid for 30 days. The `createKeyPackage()` then returns a `PackageData` object that can be uploaded to the key server afterwards, documented in 3.2.2. For group creation, the `MLS` submodule is called directly too. The returned `Group` object is serialized and returned alongside the `Welcome` messages for the other group members.

Whenever a user updates its `KeyPackage`, the method `commitUpdate()` is called for each group where that user is a member. The method creates an

Update Proposal, which gets committed using the `commit()` method. As for updates, the MLS Service provides a method for creating **Add** and **Remove** proposals for members being added or removed from a group. This method, called `generateUpdateMessages()`, commits the updates in the same way as the `commitUpdate()` function by calling `commit()`. Within `commit()`, the credential and signing private key from the user are used to call the `Group.commit()` method from the MLS submodule. The returned `MLSCiphertext` and `Welcome` messages are then returned to the caller of the `commit` function. We do not export the `commit` function as it is only required by the `commitUpdate()` and `generateUpdateMessages()` methods.

For each message received using `fetchMessages()` from the Delivery Service, the `handleMlsMessage()` method from the MLS Service is called. If the incoming message is of type `Welcome`, then we first check, if there already exists a group with the same identifier on the key server. This can happen when using the `checkGroupState()` method is called by two users at the same time. In order to break ties, we compare the creation times of the two groups and going to keep the group that was created first. After having joined the group, the `GroupState` data structure is created and posted to the key server.

When receiving a commit message, the corresponding `GroupState` is fetched from the key server and the commit is applied to it. Then the set members of the `GroupState` is updated by extracting all `KeyPackages` from the Group's `RatchetTreeView`. We further increment the latest epoch by one, serialize the new `Group` and store it inside the `GroupState`.

The functions `encryptMessage()` encrypts the specified message using the group state's latest serialized group, as it does not make any sense to encrypt a message for a past epoch. The `decryptMessage()` method does, however, decrypt messages for any epoch specified in the `MLSCiphertext`. In a cloud-based setting, it must be possible to decrypt messages for all epochs because a user logging in on a new device should be able to read all past messages.

Main

`index.ts`

All the above-mentioned services are created in the main file of the library. Besides constructing the services, we also instantiate an HTTP client with an interceptor. For this purpose, we use the Node.js package `axios` [19], which can be integrated seamlessly into browser-based applications. `Axios` offers the possibility of defining an HTTP interceptor, which we use to request new access tokens when they expire. To see what happens on the server-side, read Section 3.2.3. For the client, it is quite simple. When receiving an HTTP response with the status 401, we send a POST request containing the refresh token to the server, which then issues new access and refresh tokens.

Testing

The library is tested using the Node.js packages karma [20] and jasmine [21]. Jasmine is a JavaScript testing framework that we run in a browser instance by using Karma. Running the tests in a browser is required because we authorize users based on an HTTP-only cookie. In a future version of the library, we might also enable returning these cookies inside the response of a successful login, such that native clients can be built. This is documented in Section 4.1.2. We have created a set of 19 unit tests, testing the following cases:

- **Registration:** Expect that a set of users can register themselves using the Authentication Service.
- **Login / Logout:** For each of the registered users, expect that they can log in and log out using the Authentication Service.
- **Wrong credentials:** Expect that a user cannot log in with the wrong credentials. This does more to verify that the authentication server is correct than testing the library.
- **Key store:** Expect that when a user performs a login, the four variables of the key store are set. This test further checks if the keys are deleted after logout and are the same when the user logs in again.
- **Lifetime:** Expect that the `Lifetime` extension of a newly created `KeyPackage` is indeed 30 days.
- **KeyPackage:** Expect that when a `KeyPackage` is updated, the corresponding `PackageData` is updated as well to ensure consistency between the key server and the authentication server.
- **Accounts:** Calling `addAccount()` from the Account Service should add the account to the user. This test expects that each user can have multiple third-party accounts.
- **Passive group creation:** When calling `checkGroupState()` for a group where each member has a valid `KeyPackage` and the group was not created before, it should be created. This test verifies that each member has received a `Welcome` message and successfully joined the group.
- **Active group creation:** Expect the same behavior as the test for passive group management, by calling the `createGroup()` method instead.
- **E2EE:** This test sends various encrypted messages into a group. The other group members are then decrypting this ciphertext and verifying that its content matches the original message.

- **Passive group management:** Analogous to the passive group creation test, expect that `checkGroupState()` correctly updates the group when the list of members is changed. We further expect that removed members are not able to decrypt messages being sent after they were removed from the group as well as that new member can only read messages being sent after they were added to the group.
- **Active group management:** Performs the same tests as for the passive group management but using the `addMembers()` and `removeMembers()` methods from the Group Service.
- **Decrypt:** Expect that messages from past epochs are decrypted correctly to ensure that clients can access all messages everywhere.
- **Updating KeyPackages:** Expect that whenever a member of the group updates its `KeyPackage` using `updateKeyPackage()` from the Account Service, each group member receives a `Commit` message containing the `Update Proposal`.
- **Old epochs:** Expect that the method `removeOldStates()` from the Key-server Service actually removes the old states.
- **Groups:** Expects that when two groups with the same identifiers are created, all members agree on one of these groups. How it can happen that multiple groups with the same identifiers are created is explained in Section 3.2.2.
- **MLS Messages:** Expects that in `handleMlsMessage()`, malformed messages do not cause any exceptions and are just ignored.

3.2.3 The key server

The key server is currently set up such that it incorporates all three required servers, the key server itself, the delivery server, and the authentication server. It is implemented with Typescript as a Node.js project that uses the Express web framework [22]. In this section, we go through all components of the server and document each part in detail.

Database

The server uses MongoDB to store the records of the users. MongoDB is a NoSQL database using JSON-like tables and can be integrated into Node.js projects by using the mongoose package [23]. The structure of entries in the database is defined using a schema. The schemata used for our key server can be found in Appendix A. In contrast to the definitions of the data structures used in the client, we store most data as Strings on the key server. This comes from the fact that almost all data is encrypted when uploaded to the server. Taking the example of the list of members in a `GroupState`, the `postGroupState()` method of the CloudMLS Keyserver Service documented in Section 3.2.2 encrypts the list and outputs a base64 string which then gets posted to the key server.

Each entry in the schemata is assigned a unique identifier determined by MongoDB. Specifying that a field of the schemata has to be unique does not mean that MongoDB uses this field as a key for the database entries by default. In the following, we present the three schemata used to store the required data on the key server.

User The user schema defines how a user is stored in the database. We require each username to be unique but do not use the username as a key for the entries. Each user authenticates itself using a username and password. This approach requires storing the hash of the password, which is computed using the bcrypt package [24]. The registration and login process are documented in Section 3.2.2. For each user, we generate and store salts for the keys derived for the CloudMLS key store as well as a Boolean indicating if the user runs its own key server for storing the group states. Currently, this variable is not used as there is not yet a key server that can be run by individuals. The last element that is stored is a set of references to accounts.

Account The account schema defines how accounts from third-party platforms like Telegram are stored in the database. Each user has a set of accounts, referenced from the user database entry. We require that the platform-specific identifiers are unique and therefore restrict two users associating the same account.

When adding an account, we concatenate the platform name with the platform-specific identifier such that the account's identifier is guaranteed to be unique. For each account, there are fields for the `KeyPackage`, the corresponding `PackageData`, an inbox, and a set of references to group states. The fields `PackageData` and `GroupState` are only assignable if the user is not using a custom key server, indicated by the Boolean `customKeyServer` within the user schema.

Group State The group state schema defines how group states are stored in the database. As for the account entries, group states have a unique identifier, composed of the account identifier and the actual group identifier. This approach ensures that when fetching a `GroupState`, the actual `GroupState` of that user is returned. We store for each `GroupState` an update counter, ensuring a consistent data structure when a user performs update operations on two devices simultaneously. The group identifier and the update counter are the only entries of the `GroupState` that are not encrypted. The other fields of the `GroupState` schema are encrypted and therefore String values, which represent the fields from the `GroupState` interface defined in Section 3.2.2.

Cross-Origin Resource Sharing (CORS)

When the front-end and the back-end are not hosted on the same domain, we are required to enable CORS. In our setting, having CORS enabled, allows us to host the key server on a separate domain than the client application using the CloudMLS library. In order to allow requests from the client to the server, we need to specify the domain of the client as well as allow credentials. The Node.js package `cors` [25] thereby acts as middleware.

Authentication

When a user logs into the application using his username and password, we issue two tokens using the `jsonwebtoken` package [26]. The two tokens are returned as HTTP-only cookies to the response such that the browser, where the client application is running, can attach them to each request. As already mentioned, HTTP-only cookies prevent the tokens from being exposed to an attacker performing an XSS attack. JSON web tokens (JWT) are a signed payload, where the payload is a JSON formatted object. In our case, we sign the identifier of the user record in the database as well as an expiration date. We use the approach of two tokens, where the first is an access token with a validity of ten minutes and the second is a refresh token that is valid for seven days. Whenever the access token expires, the client sends the refresh token to receive a new access token without the need to provide its credentials again. The cookies, where these tokens are contained have the following configuration:


```
export enum TokenExpiration {
  Access = 10 * 60, // 10 minutes
  Refresh = 7 * 24 * 60 * 60, // 7 days
}

export const refreshCookieOptions: CookieOptions = {
  httpOnly: true,
  secure: process.env.NODE_ENV === 'production',
  sameSite: 'none',
  domain: process.env.BASE_DOMAIN,
  path: '/',
  maxAge: TokenExpiration.Refresh * 1000,
}

export const accessCookieOptions: CookieOptions = {
  httpOnly: true,
  secure: process.env.NODE_ENV === 'production',
  sameSite: 'none',
  domain: process.env.BASE_DOMAIN,
  path: '/',
  maxAge: TokenExpiration.Access * 1000,
}
```

Within the configuration of the cookie itself, we specified the cookie secure option to be true, when running in production mode. This comes from the fact that we have enabled HTTPS in production mode only to facilitate the debugging process when developing. This is documented in Section 3.2.3, where we show how the server is started. Concerning the maxAge option for the cookie, setting this option is not required as we have specified the expiration date for the token itself, but having set the maxAge let the browser drop the cookie once it expires. The sameSite attribute is set to none, enabling the cookies to be sent in cross-origin requests. The optional attributes domain and path are set in order to pass the cookie for subdomains and subdirectories requests as well.

Authorization

When a user is authenticated, the id of the user's record in the database is signed in the JWT. Whenever a user now requests a resource from the key server, this token is sent to the server as well. We have created a middleware function that verifies the validity of the token and extracts the corresponding database key. And as our database is designed using the `has` relation for users, accounts, and group states in the form of "a user has accounts" and "an account has group

states", this database key from the token can be used for authorization. Starting from the database entry of the user, when data for some account is requested by that user, we can iterate through all accounts associated with that user and return the requested resources. The only exception to this approach is the request for KeyPackages of other users, where we query the database containing all accounts instead of starting the search from the user's database entry.

Key Server API

In this section, we document all available API methods and how they interact with the database. The server processes each request sequentially such that we do not need to handle concurrency-related problems. A transformation to a server handling requests in parallel is, however, not very complex. By design, each user only modifies its own records in the database, which implies that a potential locking mechanism would only affect the server on a per-user basis.

POST /auth/register In order to register itself at the key server, a client must provide a username and a password as well as indicate if a custom key server will be used. We then check that the username is not already used by another user and create a new entry for the database as follows.

```
new User({
  username: req.body.username,
  password_hash: hashSync(req.body.password, 8),
  customKeyServer: req.body.customKeyServer,
  keyserver_key_salt: genSaltSync(),
  local_key_salt: genSaltSync(),
  local_user_salt: genSaltSync()
});
```

The user database entry contains a hashed version of the user-chosen password such that the server is not required to store the password itself. We further store three different salt values generated using the bcrypt library. As mentioned in Section 3.2.2, these salt values are used for deriving keys encrypting data on the key server as well as on the client-side.

POST /auth/login When a user sends a POST request to /auth/login, we first find the user with the provided username in the database. If such an entry is found, we compare the hash of the provided password with the stored password hash. Upon a match, the access and refresh tokens are created and set as a cookie in the response. We additionally send the tokens inside the body of the response such that clients not running in the browser can use the tokens as well. In the

body of the response, we also include a set of all accounts that are associated with the user as well. In the CloudMLS library, these accounts are then passed to the Account Service.

POST /auth/logout This method just sets the `maxAge` attribute of the cookies to zero, which triggers browsers to drop the cookie. As the server is stateless and all requests are handled sequentially, logging out a user is this simple.

POST /auth/refresh Whenever the server receives a request containing an invalid access token, it returns a response with the status 401. This triggers the client to request a new access token by providing his refresh token. Upon receiving a valid refresh token, we issue new access and refresh tokens to the client. On the other side, if the provided refresh token is invalid, we return an error 403 indicating the client that he needs to login again.

GET / In order to determine if the key server is reachable, a client can perform a get request to the root directory.

POST /account When a user connects a new third-party platform with his CloudMLS account, the library performs a post request to `/account`. The body of the request must contain the corresponding platform name and an account identifier. The server then creates a new database entry for that account and references it from the user that performed the request.

POST /keypackage After creating a `KeyPackage`, a user uploads it in combination with the `PackageData` object to the key server as well as to the authentication server. As we currently provide a combined version of these servers, we do only provide one API method for this upload. In terms of the authentication server, this function first loads the account for which the `KeyPackage` was created from the database. It then stores the new `KeyPackage` for that account if either no `KeyPackage` was stored before or the previously stored `KeyPackage` matches the `oldKeypackage` field from the POST request. This approach ensures that the client cannot overwrite his own changes creating a possibly inconsistent state when using multiple devices concurrently. Concerning the key server, if the user does not run his own key server, this function does store the `PackageData` as well.

GET /keypackage/:platform/:account_id This function is the only function directly performing the required lookup in the account database instead of starting from the user entry. We do nevertheless require a client to provide a valid access token when requesting the resources. The parameters a client has

to provide are the platform name as well as the account identifier for which the `KeyPackage` should be retrieved. The function then returns the requested `KeyPackage` or responds with the status code 404.

GET /keypackagedata/:platform/:account_id A client can request its `PackageData` using a GET request specifying the platform and account identifier. The server then uses the user identifier from the access token to resolve the corresponding account and return the encrypted `PackageData` to the client.

GET /groups/:platform/:account_id This function returns all group identifiers for which the specified account has stored a group state on the key server. If the user runs his own key server, we return an error message.

GET /groups/:platform/:account_id/:group_id To fetch a group state, a client performs a get request specifying the target platform, account, and group identifier. The server loads the database entry of the user and searches for the corresponding account, then searches the requested group state within the account entry. If the group state is found, it gets returned to the client.

POST /groups Using this method, a client can create a new group state on the server as well as update an existing one. Creating a new group state is simply creating a new database entry and linking it from the user's account entry. For updating a group state, there is an additional step. To ensure consistency, we have defined an update counter, which the client needs to increment each time the group state is modified. The POST request is only successful if the update counter of the modified group state is equal to the update counter stored on the server plus one.

GET /delivery/:platform/:account_id This function is part of the delivery server. It sends all messages from the inbox of the specified account to the client and empties the inbox afterward. A client can, according to this approach, not fetch a message twice and therefore needs to handle the messages after fetching them. A possible improvement could be to assign a counter to each message and then let the CloudMLS library send requests to the delivery server indicating which messages can be deleted.

POST /delivery

This function is used to send MLS-specific messages to other users. The client posts a batch of messages to the delivery server, which ensures that each recipient receives all messages in order. We achieve this using the simple approach of

handling all these requests sequentially and all messages for a certain group are received in one batch.

Error handling

There are many possibilities why errors can be thrown in the server implementation. Having a central error handling middleware controller enables a simple way to handle these errors and to set the appropriate status codes to the corresponding responses. We extend the Error class multiple times to distinguish various error categories. The following code snippet shows how the errors are treated.

```
switch (error.constructor) {
  case MongoDBError:
    res.status(error.code).send({ message: error.message })
    next(error)
    break

  case TokenRefreshError:
    res.cookie(
      "accessToken", '', {...accessCookieOptions, maxAge: 0}
    )
    res.cookie(
      "refreshToken", '', {...refreshCookieOptions, maxAge: 0}
    )
    res.status(error.code).send({ message: error.message })
    break

  case BadRequestError:
  case NotFoundError:
  case AccountError:
  case ConflictError:
  case CustomKeyServerError:
  case TokenVerificationError:
    res.status(error.code).send({ message: error.message })
    break
  default:
    next(error)
}
```

- MongoDB errors can be caused by caused when trying to insert values declared to be unique multiple times. We therefore return the error message, giving the client a hint about what went wrong.

- **TokenRefresh** errors are a special error case, where we do not only send back the error message of the underlying problem but also set the `maxAge` attribute of the cookie to zero. This will then let the browser drop the cookie and prevents causing the same error multiple times.
- In case a client sends a malformed request where required parameters are missing, we return a `BadRequestError`.
- **NotFound** errors are returned each time a non-existent resource is requested. An example of such an error is when a client wants to discover which of his contacts has a valid `KeyPackage` on the authentication server. For each contact not having registered itself a `NotFoundError` will be returned.
- **Account** errors are returned in a single case, where a client wants to register an account that is already associated with the same user.
- **Conflict** errors are caused by providing a wrong update counter when posting a group state or by providing the wrong `oldKeypackage` when trying to update a `KeyPackage`. These errors may only occur in cases where a user is online with two devices simultaneously.
- **CustomKeyServer** errors are returned each time a client requests or tries to store data that is supposed to be on a custom key server run by the user. This error is returned if and only if the database entry of the user has the field `customKeyserver` set to true.
- **TokenVerification** errors occur when the user provides an expired or invalid token. We return a status code 401, indicating the CloudMLS library to send a request to `/auth/refresh` to receive a new access token.

Environment File

Some variables of the key server are dependent on the environment where the service is being hosted. A simple solution to simplify the setup of a server in a new environment is to specify a file in which all these variables are set. These variables are the following.

- **NODE_ENV** This variable has to be set either to development or production. In production mode, HTTPS is used and the cookie attribute `secure` will be true.
- **BASE_DOMAIN** The base domain can be localhost or the actual domain of the key server. This variable is used for the cookie attribute `domain`.
- **PORT** This variable denotes the port number on which the server should listen. The server will be accessible on `http://BASE_DOMAIN:PORT`, respectively `https://BASE_DOMAIN:PORT`

- **ACCESS_TOKEN_SECRET** Contains the secret for the JWT signing operation of the access token. The `sign()` function uses this secret as input for the HMAC.
- **REFRESH_TOKEN_SECRET** Analogous to the access token secret but this variable is used for the signing operation of the refresh token.
- **MONGODB_URL** This variable contains the URL of the MongoDB. The database runs on `mongodb://localhost:27017` by default
- **MONGODB_DATABASE** The name of the database to be used for storing the entries.

Starting the server

Creating the HTTP server is achieved using the express method `listen()`. In production mode, we want to secure the communication between the client and the key server and use the `https` module from Node.js. We therefore need to provide a certificate and a private key.

```
if (process.env.NODE_ENV === 'production') {
  createServer({
    key: readFileSync(process.env.SSL_PRIV_KEY!),
    cert: readFileSync(process.env.SSL_CERT!)
  }, app).listen(process.env.PORT, () => {
    console.log(
      'Production server at https://localhost:\${PORT}'
    );
  });
} else {
  app.listen(process.env.PORT, () => {
    console.log(
      'Development server at http://localhost:\${PORT}'
    );
  });
}
```

Starting the server itself can be done in many ways. During development, we ran the server using the `pm2` package [27].

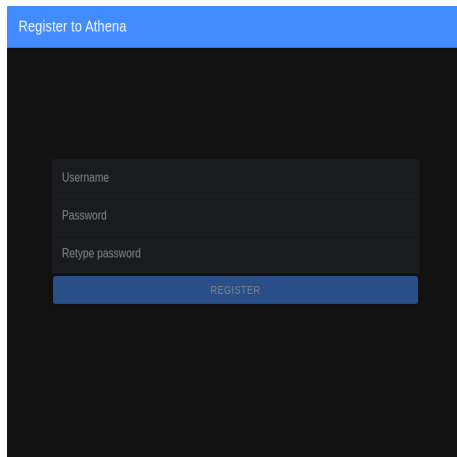
3.3 Athena - The Telegram example

To demonstrate the capabilities of the CloudMLS library in combination with our key server, we have created a client that is able to send and receive end-to-end encrypted text messages through Telegram. This client does only serve as a simple proof of concept and shows how CloudMLS could be integrated into an existing Node.js based instant messenger. The basis for this implementation was the Athena project from Noah Zarro [28]. In this section, we are going to show how the Telegram client is implemented and the integration of CloudMLS.

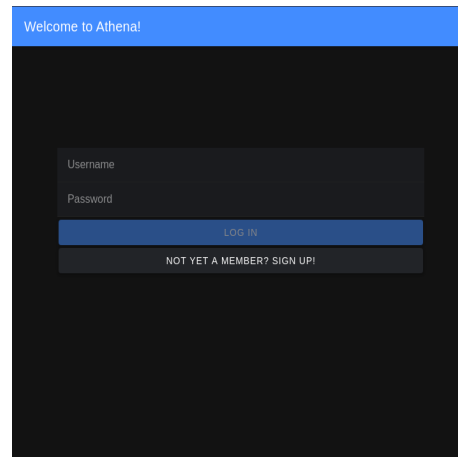
3.3.1 Application overview

The Telegram example client is an Angular [29] based web application, which can be compiled to Android and iOS using the Ionic framework [30]. The main advantage of Ionic is that it allows cross-platform development using a single code base. The project structure of our client is kept very simple. As an entry point for the application, we use a login screen that forwards the user to a tab overview once his login was successful. We provide three tabs, one for an overview of the Telegram chats, one for an overview of the contacts, and a settings page. To call the API methods from the Telegram back-end, we use a service combining the required logic.

3.3.2 Registration and Login



(a) Registration page



(b) Login page

The registration and login pages let a user authenticate himself to the CloudMLS library. They are simple user interfaces with the purpose of exposing the Authentication Service's `register()` and `login()` methods. Besides calling the

functions from the libraries, they display error messages resulting from these function calls.

3.3.3 Telegram Service

Within the Telegram service's page, we use two helper classes. First, we provide a `CustomLocalStorage` class which provides persistent storage for the Telegram API. This class enables saving the required keys for authenticating a Telegram account at the Telegram API. To ensure that only the corresponding user has access to those keys, we use the encryption key for local content and the username hash from the CloudMLS `KeyStore` described in Section 3.2.2. Second, we use a class called `API` for performing all calls to the Telegram API and handling results and errors. The Telegram service itself provides the following functionalities.

- **Authentication** The authentication process with Telegram uses two methods, one for requesting an authentication code for the user-provided phone number and one for sending the authentication code to the Telegram API. Note that when requesting an authentication code, Telegram sends this code as a Telegram message to the user.
- **Dialogs** To fetch and parse all dialogs, we use the method `getDialogs()`, which returns a list of all dialogs including a decrypted version of the last message sent. The method uses various helper functions to parse the response of the Telegram API call `'messages.getDialogs'`.
- **Contacts** Contacts are retrieved using a call to `'contacts.getContacts'`, parsing the result and loading the corresponding profile images.
- **Getting Messages** To load all messages from a specific dialog or group, the API method `'messages.getHistory'` is called. For each received message object, we extract the message itself, the corresponding sender, and the date. We then try to decrypt the message using the CloudMLS function `decrypt` of the Message service and set a flag, if the message was encrypted or not.
- **Sending Messages** When sending a message, we first try to encrypt it using the CloudMLS function `encrypt` of the Message service. Note that the encryption can only be successful if the group or dialog identifier has a stored group state on the key server. We then send the resulting message using the API method `'messages.sendMessage'`.

3.3.4 Chat overview page

The chat overview page fetches the dialogs and group chats from the Telegram service and displays them as shown in Figure 3.6. For each of these chats, the last message sent is decrypted such that it can be read without opening the chat itself. We further display a blue lock on the right-hand side, if the chat is end-to-end encrypted. In other words, there is a blue lock if the corresponding chat has a group state on the key server. In our example, each time when entering the chat overview page, we call for each group the `checkGroupState()` and for each dialog `checkDialogState()`. As we are not using any form of a database for the client, this method is called to determine if the chat is E2EE or not.

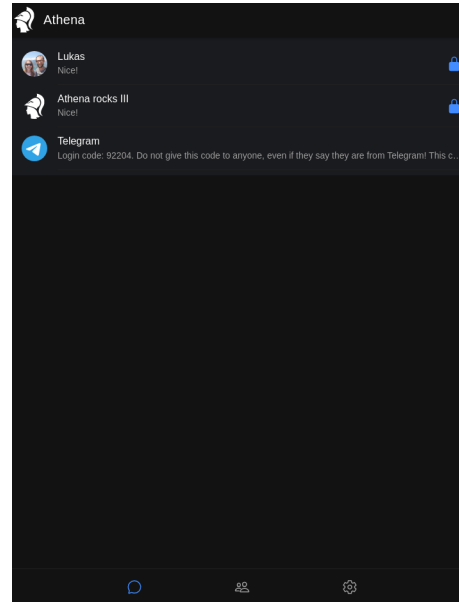


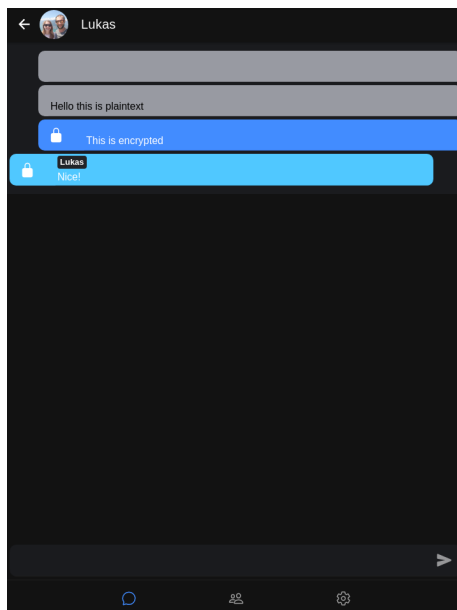
Figure 3.6

3.3.5 Chat page

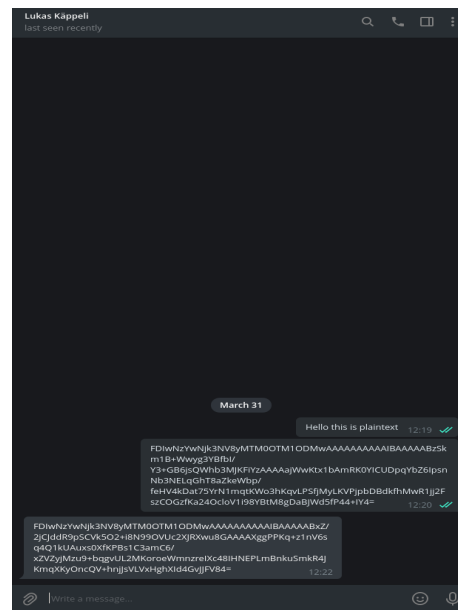
When entering a chat, we first load all messages using the Telegram service. Then, we try to decrypt each message and display the content. As visible in Figure 3.7a, we use three different colors to display messages. Gray indicates that a message is sent without being encrypted with MLS. It can therefore be read by any client of the corresponding third-party platform. The dark blue color represents messages that are sent encrypted from the current user. On the other side, the light blue color indicates that an encrypted message is sent from a chat partner.

3.3.6 Contact page

After getting all contacts using the Telegram service, we check for each contact if we can start an end-to-end encrypted conversation with it. We achieve this by requesting and verifying the `KeyPackage` of every contact. If the `KeyPackage` exists and is valid, we show the blue in the list. Note that this is just a contact list and does not support the navigation to the corresponding chat upon selection.



(a) Chat page inside the Telegram example client



(b) The same chat as in the left Figure, but using the official Telegram client

Figure 3.7

3.3.7 Settings page

On the settings page are three functionalities incorporated. First, when selecting the "show credentials" button, the current username and associated accounts are displayed. Second, a user can perform a logout which then logs the user out from the CloudMLS library and clears the state of the Telegram service. Third, a user can connect to a Telegram account by entering his phone number and selecting 'Request auth code'. Telegram will then send a message containing an authentication code, which the user then should enter in the second field in the login form. If the authentication code is correct, the account will be connected to the current device even when the user logs out. As we are storing the session keys in encrypted form using the CloudMLS KeyStore keys, only the authorized user has access to the session.

3.4 Possible clients

As already mentioned, our Telegram example client serves as a simple proof of concept on how the CloudMLS library could be used. There is no support for sending and receiving media messages, seeing who is online, nor are the messages stored in a database and therefore have to be decrypted each time when entering

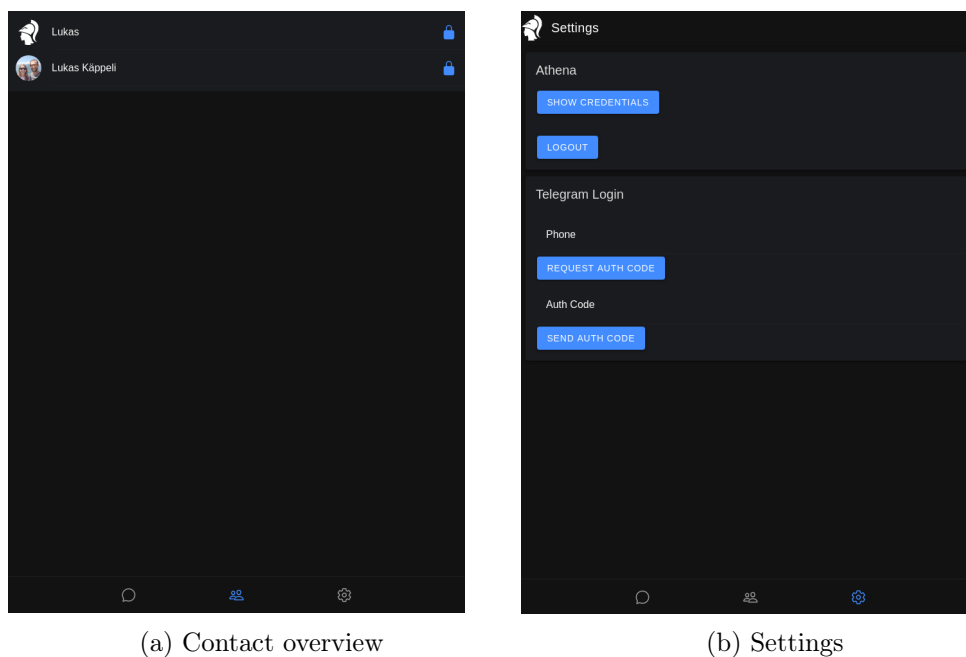


Figure 3.8

a chat. But instead of creating another instant messenger, we want to list some possible clients into which CloudMLS could be integrated.

- **Telegram:** Evgeny Nadymov created a Telegram client [31] based on React Native that could serve well for integrating the CloudMLS library. Considering the structure of the project, it seems possible to integrate CloudMLS using the existing functions. Taking the example of the `sendMessage()` method in "telegram-react/src/Components/ColumnMiddle/InputBox.js", which is the only place where messages of any type are sent, calling `CloudMLS.messageService.encrypt()` before sending is a lightweight adjustment.
- **WhatsApp (unofficial):** Baileys [32] is an unofficial WhatsApp client without an User Interface. It is a really well-written and documented open-source project that makes a perfect fit for CloudMLS. The two disadvantages are the lack of a graphical user interface and that using it may lead to a ban from WhatsApp. Baileys further relies on web sockets that spoof the origin to be `web.whatsapp.com`. This can only be achieved when running as a native project and will not work when used in a browser-based application.
- **Facebook (unofficial):** Alex Rose created a Facebook Messenger command-line interface [33] for the Facebook Messenger. CloudMLS could be inte-

grated in a simple way by adding a layer between the command line and the rest of the application. But again, this client will not work in a browser, even if we do not use the command line, because the library is built using native Node.js packages.

- **SMS:** There does not seem to exist a Node.js application that supports sending SMS. But as there is an Ionic Cordova plugin that enables sending SMS from Android and iOS devices, one could easily adapt the telegram example client to support sending and receiving SMS.

3.5 Email

While developing the CloudMLS library, we have tested how it could be used in terms of email messages. As it is not possible for browsers to use the SMTP protocol directly, we can not create an email client like the telegram example client. It is still possible to create a native email client that uses the CloudMLS library, but we leave that for future work. We have, however, integrated the google mail API [34] to send and receive emails from Gmail accounts. Our approach still exists in the branch `feature/email` of the `main_app` repository. In the following, we are going to elaborate on how the email setting differs from instant messaging and how to overcome the resulting issues.

3.5.1 Missing identifiers

Email messages themselves usually have an identifier, but the concept of continuous communication between two or more users does not exist. The only way that messages are grouped into a conversation is based on the subject field of the email. To use CloudMLS in the email setting does require a notion of identifiable conversations. The approach to assigning a random group identifier for each message would work but does not scale as for each message a new group is created. For dialogs, we have used the approach of using the combination of the two email addresses to generate a group identifier. By sorting them in alphabetical order their concatenation builds a unique group identifier. This uniqueness is implied as each email address is required to be unique as well. When we then try to adapt this approach for communications between more than two members, it quickly gets complicated. The concept of blind copies does not work in this setting as well, because adding a new member to a group is a visible process.

3.5.2 Group mutations

Consider an email conversation between three users that was started by one user, writing an email message to two others. These three users then build a group that

can be well represented by a group state. When then one of these users wants to add a new member to that group, there does not seem to be a trivial solution on how the group identifier should behave. Using the approach of concatenating all group member's email addresses would generate a new group upon adding a member. We therefore opted for generating random group identifiers upon sending an email to multiple recipients. This does also enable group evolution in terms of group members. When a group member sends a message to a group and adjusts the set of recipients, the group gets updated accordingly. On our feature/email branch, we provide a proof of concept that works, but we strongly recommend using other mechanisms to encrypt email messages.

Future work

There is a large potential for improvements to the CloudMLS library as well as for the underlying MLS implementation from Hubert Chathi. Both libraries are still not implemented completely nor were any performance optimizations made. In terms of security, both the implementations themselves and the imported Node.js packages need to be analyzed to rule out security vulnerabilities. This section will be a bit longer than the usual future work sections, which shows the potential of our library.

4.1 CloudMLS

In terms of the library, the basic functionality is in place and works. There are, however, important features missing. These features will improve usability and performance.

4.1.1 Epoch Base Ratchets

In a previous version of our library, we have only stored the most recent group state on the key server. The problem thereby was that users that were offline during two or more group updates could not decrypt messages sent in between these updates. More formally, if a user has seen the group at epoch x , and is then offline until epoch $x + 2$, the user will never be able to decrypt messages from epoch $x + 1$.

In this previous version, instead of storing each group state, we simply stored the first hash ratchet of each member of the group. From this ratchet, all subsequent keys were derived. Thus, to reduce the number of group states on the key server, we could re-add these epoch base ratchets and find some way to determine when we can delete past group states.

4.1.2 Enable native devices

As we have aimed for a browser-compatible implementation of our library, there currently exists a lack of support for native applications. The problem is that the access tokens that are sent as cookies, which do only work in a browser. To support native devices, the following requirements must be met:

- **Key server:** Upon successful login, the tokens must be sent inside an HTTP-only cookie as well as in the response body. This is already implemented.
- **Key server:** When receiving a request, do not only search the token in the cookies but also in the request itself.
- **Key server:** When refreshing the tokens, send the resulting tokens back as a cookie and inside the response body.
- **CloudMLS:** When receiving the tokens from the key server, either due to a login or refresh request, store them in memory. If the client application provided some form of persistent `KeyStore`, we may store the tokens in it.
- **CloudMLS:** Add an additional axios interceptor, adding the token to each request. It may be beneficial to filter the URLs for which the tokens are useless, like refresh, login, and registration requests.

4.1.3 Changing the password

As we are using a password-based encryption scheme to encrypt data stored on the key server and possibly in local storage, changing the password is a non-trivial task. Changing the password does therefore also require to re-encrypt local and remote data structures, which is an approach that does not scale. An alternative would be to generate a random key that is used to encrypt these data structures and then encrypt this random key using the password-based encryption scheme. Concerning performance, this approach would be very efficient as changing the password only requires the re-encryption of one element. On the other side, changing the password does not change the encryption keys. We could, however, provide a method that allows a user to re-encrypt the complete content on the key server.

4.1.4 PBKDF2

In the current version of the library, we derive the key for encrypting data on the key server using SHA-512 and a salt value from the server. In order to provide more security in the password generation process, we should switch to

the standardized approach of using PBKDF2 from the `crypto-js` [35] package to provide a more secure way of deriving encryption keys.

4.1.5 Public key cryptography

We do only support symmetric key cryptography for the encryption of data on the key server. It is absolutely possible to allow a user to encrypt the data structures using asymmetric cryptography. The resulting E2EE would, however, not be a cloud-based approach. A user accessing his account on a new device or in a browser would have to provide his private key in some way. This could only be achieved if the user is online with a primary device at the same time, resembling a two-factor authentication or WhatsApp-like approach. Despite the disadvantages, a user should have the choice of how he wants to secure his data.

4.1.6 Adapt to draft-13

At the beginning of March, the IETF working group developing the MLS protocol published draft version 13 of the protocol. While version twelve of the draft did not force us to adapt our implementation, version 13 has an important change in it. In our implementation of the library, when a user updates his `KeyPackage`, he creates an `Update` proposal that he commits directly. This self-update is forbidden as of version 13 and we have to change our approach. It therefore requires a third MLS message type for proposal messages, which should be enabled in the delivery service and the `handleMlsMessage()` function. The user updating his `KeyPackage` then sends an `Update` proposal to all other members of a group, from which anyone then issues a commit incorporating this proposal. The most complex change will be to determine if a received proposal was already included in a commit. An idea for solving this issue is, to iterate through all received messages and filter out proposals incorporated in a recent `Commit`, if such a `Commit` exists.

The draft version 13 does also specify that a `KeyPackage` should only be used once to add a client to a group. This implies some changes to our implementations. We first have to differentiate calls requesting a `KeyPackage` itself from calls just verifying that there is a valid `KeyPackage`. Then the authentication server must be able to verify the validity of `KeyPackages`. We may further provide the possibility to upload multiple `KeyPackages` per `Account` such that reusing the same `KeyPackage` could be minimized.

4.2 Key server

The implementation of the key server is complete and only lacks small adjustments. The most important missing feature is, however, letting users run their own key server.

4.2.1 Custom key server

The current implementation of the key server incorporates all three functionalities, the key store, the delivery server, and the authentication server. The next step in the development of this project should be to implement a version of the key store that can be run by individuals. This step involves creating a mechanism ensuring consistency between the key server and the authentication and delivery server. Besides consistency, the question of how authentication and authorization on the custom key server can be handled arises. As the main motivation for a self-hosted key server is reducing the amount of trust in external infrastructures, it does not make any sense to let the authentication server handle access to the user's key server.

A simple way of achieving this separation could be to perform registration and login requests twice, one to each server. Despite a small overhead, this approach would perfectly split up the two services. The next question arising is then how a user could switch from one to another key server. We leave this question up for future work.

4.2.2 Space efficiency

We currently store the group states as an encrypted version of a serialized JSON object. The approach of using JSON objects has the disadvantage that the field identifiers are included in the serialization introducing an overhead. The `Group` and `RatchetTreeView` data structures store redundant data, which is not beneficial in terms of storage efficiency as well. Using our approach, we do, however, only store data in the order linear to the number of group mutations and `KeyPackage` updates. This is already a huge improvement compared to storing data linear to the number of messages sent. Nevertheless, there is room for improvements in terms of the amount of storage required.

4.3 Backlog

While writing this thesis, the backlog for the three repositories constantly grew. For the sake of completeness, we list each item of it.

- **(Key server)** Remove the field `CLIENT_URLS` from `.env.template`: This field is not used anymore, so we can simply remove it.
- **(CloudMLS)** Handle server URLs differently: Currently, the URLs are hardcoded in `/src/server_urls.ts`. We should move them to an environment file and just use two URLs, the `AUTH_SERVER_URL` for the authentication and delivery server, and the `KEYSERVER_URL` for the key server.
- **(CloudMLS)** `DeliveryService.destroy()` calls `sendAll()`: It should rather delete the message store instead of sending the messages from it.
- **(General)** Licences: Before publishing the source code of the project, we have to determine the license under which we make it open-source.
- **(Key server)** Move the cors origin URL. This URL should be specified within the `.env` file, not hard-coded in the cors configuration in `/src/index.ts`
- **(CloudMLS)** Only call `DeliveryService.sendAll()`, if and only if `postGroupState()` was successful. We therefore need to turn around the order of calls to first call `postGroupState()`, then upon success send all stored messages.
- **(General)** Check comments: Verify that function documentation comments still match the behavior of the method itself.
- **(General)** Delete delivered messages manually: Currently, we delete MLS-specific messages from the inbox once the client has fetched them. We should, however, keep the messages until the client has confirmed to successfully have applied the resulting changes. We could implement this by adding some form of counter to each message and then let the library confirm the latest message that could be deleted on the delivery server.
- **(General)** Find out how to prevent a user from adding an Account that does not belong to him.

4.4 EU Regulations

On 25th March 2022, the council of the EU agreed provisionally on the Digital Markets Act (DMA) forcing large platforms like WhatsApp and Facebook Messenger to provide some kind of interoperability to other platforms [36]. There are some concerns about providing interoperability could weaken the security and privacy of these platforms. We want to pose the question, why not use CloudMLS for solving this problem? Each instant messenger provider could host an authentication, delivery, and key server. Then, for the sake of being interoperable, the providers just need to create three services. First, an authentication server where their own clients can post their `KeyPackages` and an API, where other provider's clients can fetch them. Second, a delivery server to which users

from other platforms can post MLS-specific messages. Third and last, an API, where other providers can send and receive the encrypted messages. We leave a detailed analysis in terms of security, privacy, etc. for future work.

Conclusion

We have presented a library enabling secure communications over existing instant messengers. CloudMLS provides excellent security guarantees in a cloud-based setting, by using a password-based encryption scheme. Despite some missing features and a large backlog, we consider the library a success. It builds a solid basis with potential in various areas. From regular (and legal) Telegram Clients, to unofficial WhatsApp clients to even a way to provide interoperability between large platforms, we consider all to be possible. We could even think of securing email communications using our library, having in mind that this does impose a significant overhead compared to other solutions.

In terms of trust, we have reached our goal of minimizing the amount of trust required in external infrastructure. The only infrastructure that needs to be trusted is the authentication server and the delivery server. And this approach drastically reduces the trust in companies generating money using the data of users of their platform.

Bibliography

- [1] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, Mar. 2022. [Online]. Available: <https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html>
- [2] Meta, “Two billion users — connecting the world privately,” Feb. 2020. [Online]. Available: <https://about.fb.com/news/2020/02/two-billion-users/>
- [3] Messenger, Sep. 2017. [Online]. Available: <https://www.facebook.com/messenger/posts/1530169047102770/>
- [4] Telegram, Apr. 2020. [Online]. Available: <https://telegram.org/blog/400-million/de>
- [5] D. Curry, Jan. 2022. [Online]. Available: <https://www.businessofapps.com/data/signal-statistics/>
- [6] Threema Press, Jan. 2020. [Online]. Available: https://threema.ch/press-files/1_press_info/Press-Info_Threema_EN.pdf
- [7] “Clarifying lawful overseas use of data (cloud) act’,” Mar. 2018. [Online]. Available: <https://www.justice.gov/dag/cloudact>
- [8] WhatsApp, Nov. 2021. [Online]. Available: <https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf>
- [9] M. Marlinspike, Nov. 2016. [Online]. Available: <https://signal.org/docs/specifications/doubleratchet/>
- [10] M. Marlinspike, Apr. 2017. [Online]. Available: <https://signal.org/docs/specifications/sesame>
- [11] [Online]. Available: <https://matrix.org>
- [12] Intel. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html>
- [13] K. Bhargavan, B. Beurdouche, and P. Naldurg, “Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS,” Inria Paris, Research Report, Dec. 2019. [Online]. Available: <https://hal.inria.fr/hal-02425229>

- [14] J. Alwen, D. Jost, and M. Mularczyk, “On the insider security of mls,” Cryptology ePrint Archive, Report 2020/1327, 2020, <https://ia.cr/2020/1327>.
- [15] H. Chathi, Gitlab. [Online]. Available: <https://gitlab.matrix.org/matrix-org/mls-ts>
- [16] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, May 2015. [Online]. Available: <https://www.rfc-editor.org/info/rfc7519>
- [17] KirstenS, “Cross site scripting (xss),” OWASP. [Online]. Available: <https://owasp.org/www-community/attacks/xss/>
- [18] godliked, “mtproton,” npm. [Online]. Available: <https://npmjs.com/package/mtproton>
- [19] mzabriskie, nickuraltsev, emilyemorehouse, and jasonaayman, “axios,” npm. [Online]. Available: <https://npmjs.com/package/axios>
- [20] dignifiedquire, johnjbarton, zzo, and k. vojtaajina, “karma,” npm. [Online]. Available: <https://npmjs.com/package/karma>
- [21] slackersoft, sgravrock, dwfrank, and amavisca, “mongoose,” npm. [Online]. Available: <https://npmjs.com/package/jasmine>
- [22] mikeal, dougwilson, and jasnell, “express,” npm. [Online]. Available: <https://npmjs.com/package/express>
- [23] aaron, rauchg, tjholowaychuk, and vkarпов15, “mongoose,” npm. [Online]. Available: <https://npmjs.com/package/mongoose>
- [24] jfirebaugh, tootallnate, ncb000gt, defunctzombie, and amitosh, “bcrypt,” npm. [Online]. Available: <https://npmjs.com/package/bcrypt>
- [25] dougwilson and troygood, “cors,” npm. [Online]. Available: <https://npmjs.com/package/cors>
- [26] dschenkelman, iaco, jaredhanson, jfromaniello, jstrutz, lbalmaceda, woloski, and ziluvatar, “jsonwebtoken,” npm. [Online]. Available: <https://npmjs.com/package/jsonwebtoken>
- [27] tknew, “pm2,” npm. [Online]. Available: <https://npmjs.com/package/pm2>
- [28] N. Zarro, “End to end encryption in a cloud-based messenger,” Jun. 2021.
- [29] [Online]. Available: <https://angular.io>
- [30] [Online]. Available: <https://ionicframework.com>
- [31] [Online]. Available: <https://github.com/evgeny-nadymov/telegram-react/>

- [32] [Online]. Available: <https://github.com/adiwajshing/Baileys/>
- [33] [Online]. Available: <https://github.com/Alex-Rose/fb-messenger-cli>
- [34] [Online]. Available: <https://developers.google.com/gmail/api>
- [35] [Online]. Available: <https://www.npmjs.com/package/crypto-js>
- [36] [Online]. Available: <https://www.consilium.europa.eu/en/press/press-releases/2022/03/25/council-and-european-parliament-reach-agreement-on-the-digital-markets-act/>

MongoDB Schemata

```
const groupStateSchema = new Schema<GroupState>({
  group_id: {
    type: String,
    unique: true,
    lowercase: true,
    trim: true,
    required: [true, "group_id not provided"],
  },

  updateCounter: {
    type: Number,
    default: 0
  },

  members: {
    type: String
  },

  creationTime: {
    type: String
  },

  mlsGroup: {
    type: String
  },

  latestEpoch: {
    type: String
  }
})
```

```
const accountSchema = new Schema<Account>({
  account_id: {
    type: String,
    unique: true,
    lowercase: true,
    trim: true,
    required: [true, "account_id not provided"],
  },

  keypackage: {
    type: String,
    default: ""
  },

  inbox: [{
    type: String,
    default: []
  }],

  keypackageData: {
    type: String
  },

  groupStates: [{
    type: Schema.Types.ObjectId, ref: 'GroupState',
    default: []
  }]
})
```

```
const userSchema = new Schema<User, Account>({
  username: {
    type: String,
    unique: true,
    lowercase: true,
    trim: true,
    required: [true, "username not provided"],
    minLength: 8
    maxLength: 20
  },

  password_hash: {
    type: String,
    required: true
  },

  customKeyServer: {
    type: Boolean,
    default: false
  },

  keyserver_key_salt: {
    type: String,
    default: ""
  },

  local_key_salt: {
    type: String,
    default: ""
  },

  local_user_salt: {
    type: String,
    default: ""
  },

  accounts: [{
    type: Schema.Types.ObjectId, ref: 'Account',
  }]
});
```