Linköping University | Department of Electrical Engineering
Master's thesis, 30 ECTS | Computer Science and Engineering
2020 | LiTH-ISY-EX–20/5274–SE

Evaluation of the Messaging Layer Security Protocol

-- A Performance and Usability Study

Utvärdering av Messaging Layer Security
-- *En prestanda- och användbarhetsstudie*

Silas Lenz

Examiner : Jan-Åke Larsson
External supervisor : Jonathan Jogenfors

Linköpings universitet
SE--581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Secure messaging protocols have seen big improvements in recent years, with pairwise messaging now being possible to perform efficiently with high security guarantees and without requiring both participants to be online at the same time. For group messaging the solutions have either provided lower security guarantees or used highly inefficient implementations in terms of computation time and data usage, with pairwise channels between all group members, limiting the possible applications. Work is now ongoing to introduce the Messaging Layer Security (MLS) protocol as an efficient standard with high security guarantees for messaging in big groups.

This thesis examines whether current MLS implementations live up to the promised performance properties and compares them to the popular Signal protocol. In general the performance results of MLS are promising and in line with expectations, providing improved performance compared to the Signal protocol as group sizes increase. Two proof of concept applications are created to prove the viability of using MLS in realistic scenarios, one for video calls and one for mobile messaging.

Acknowledgments

I would like to thank Sectra Communications for making this thesis possible, and especially Jonathan Jogenfors for providing me with valuable advice and feedback. I would also like to thank my examiner at Linköping University, Jan-Åke Larsson.

Contents

List of Figures

1 Introduction

1.1 Motivation

Secure group messaging is a challenge where current applications often drop security guarantees provided for groups with two participants when inviting more participants due to performance issues. There has also not been a standard upon which to base new messaging applications. The Messaging Layer Security (MLS) draft with contributions from many industry leaders such as Facebook, Cisco and Google intends to solve both these issues. The aim is to become a Internet Engineering Task Force (IETF) standard with support for scaling up to tens of thousands of group members.

As MLS is promising better performance with retained security guarantees it becomes interesting to evaluate the provided performance and compare it to current solutions. This is the first aim of this thesis. Being in early development there are also no practical applications using MLS in the wild. Therefore, the second aim of this thesis is to test its suitability for real world use by implementing a prototype of video call signalling and text messaging on top of MLS.

1.2 Aim

The aim is to investigate the Messaging Layer Security Protocol (MLS) in terms of functionality and performance. The following research questions should be answered:

RQ 1 Is the MLS specification and the currently available implementations ready for practical use and suitable as a base for encrypted text, voice and video chat using WebRTC?

RQ 2 How competitive are MLS implementations compared to expected theoretical results and to the Signal protocol when comparing CPU time and network data usage for these common operations in a group messaging scenario with different group sizes:

- Creation of a groups
- Addition of a new group member
- Removal of a group member
- Updating key material
- Sending application messages

1.3 Delimitations

This thesis will not perform any security evaluation of the MLS architecture or protocol, the available implementations, or the additional work done during this thesis. Neither will it evaluate whether the used cryptography implementations match the specification. Only calls with two participants will be considered for RQ 1. Unless otherwise mentioned, the report is based on MLS protocol version 07 and architecture version 03.

2 Theory

2.1 Security Properties

The security provided by a protocol can be described by which security properties it provides. Basic features of a secure messaging protocol are encryption and authentication, but many provide some less common features.

2.1.1 End to End Encryption

The purpose of end to end encryption [1] is to allow for secure communication via an unsafe or untrusted channel. This is possible by ensuring that only the sender and receiver know the relevant keys. Contrast this by comparing to email communication where the communication between the sender and the email server and the connection between the server and the receiver may be encrypted, but the intermediary server has the possibility to read or even modify the message.

2.1.2 Backward Secrecy

Backward secrecy provides a self-healing property where even if a session is compromised at one point, the following communication is not compromised if at least one uncompromised

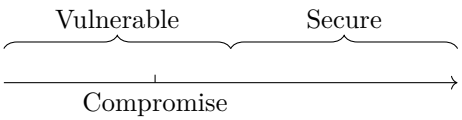


Figure 2.1: Backward secrecy.

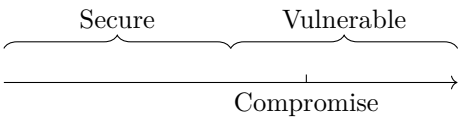


Figure 2.2: Forward secrecy.

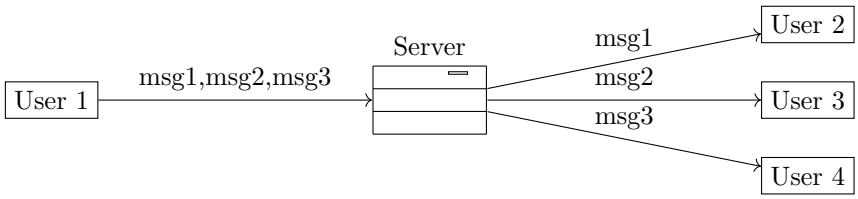


Figure 2.3: Client-side fan-out.

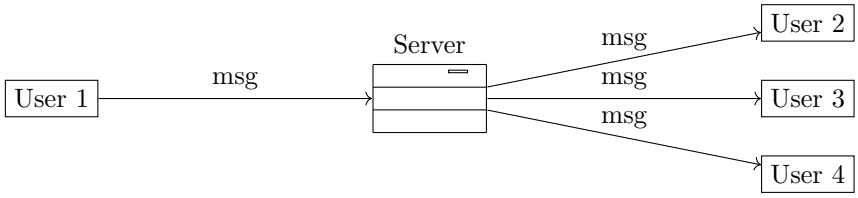


Figure 2.4: Server-side fan-out.

message is sent between the communication participants before the communication is restored, see Figure 2.1 [2].

2.1.3 Forward Secrecy

The opposite of backward secrecy is forward secrecy. It guarantees that if the session is compromised now it can not be used to compromise previous communication, see Figure 2.2. It is safe from compromises happening "forward" in time [2].

2.2 Messaging Related Concepts

Except for the security properties there are some other concepts often mentioned related to a messaging solution.

2.2.1 Fan-Out

When communicating in a group, a message needs to be sent to all participants. This can be done in multiple ways.

- Client-side fan-out means that the client creates a message for each other participant and sends them all separately. This means that the amount of data sent by the client increases linearly with the number of participants in a group. The messages may still go through a server as in Figure 2.3, but separate messages are sent to every other participant.
- Server-side fan-out means that the client creates one message and sends this to a server which will then broadcast it to all other participants, as can be seen in Figure 2.4. This means that the amount of data sent by the client is constant, no matter the number of participants.

2.2.2 Asynchronous

An important property of a messaging protocol that can be used on mobile devices is that it works asynchronously. This means that operations can be done even if participants in a group are not online at the same time.

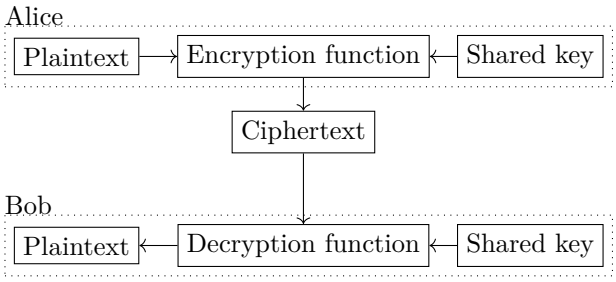


Figure 2.5: Symmetric encryption.

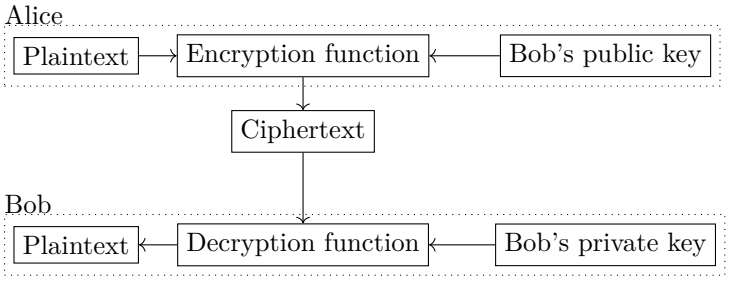


Figure 2.6: Asymmetric encryption.

2.3 Cryptographic Primitives

These are the most important cryptographic primitives relevant to the thesis.

2.3.1 Hash Function

A hash function transforms data of an arbitrary size to a fixed size hash value that can be used for verification of the original data, signatures, and more. A good hash function has properties that makes it hard to find collisions, that is two different input values that generate the same hash value, a pre-image where another input value generates the same hash value as one you already have, or a second pre-image where you have one input value and try to find another that generates the same hash value [1].

Hash functions can also be used together with a Message Authentication Code (MAC), using a key to authenticate the message so the source of the hash can also be verified. This is known as a HMAC (hash-based message authentication code) [1].

2.3.2 Symmetric and Asymmetric Encryption

In symmetric encryption [1] both parties use the same shared secret key to encrypt the plaintext and decrypt the resulting ciphertext, see Figure 2.5. One example of such an encryption function is the Advanced Encryption Standard (AES). In asymmetric encryption [1] the parties share their public key but keep their private key secret. The public key can then be used to encrypt the plaintext, but only the private key can decrypt the ciphertext, see Figure 2.6. To avoid reuse of the encrypted value by a malicious actor, for example to repeat a command, encryption functions may incorporate a nonce, a number used only once, into the encryption.

2.3.3 Authenticated Encryption with Associated Data (AEAD)

When using Authenticated Encryption (AE) the message is both encrypted and authenticated, so it can be verified that the message came from the stated sender and has not been modified.

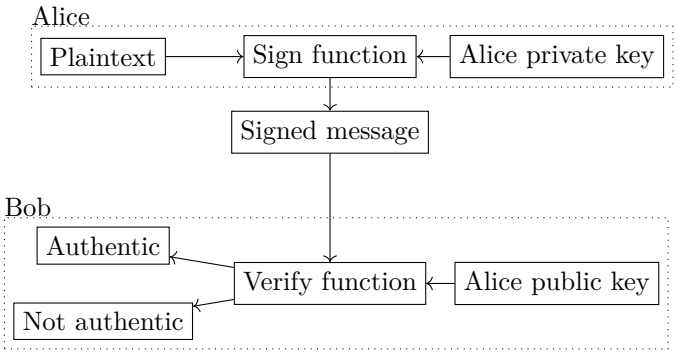


Figure 2.7: Message signatures.

The associated data (AD) is additional data that needs to be authenticated, but not encrypted. An AEAD [3] provides these properties simultaneously, using the only one algorithm and key.

2.3.4 Signatures

Cryptographic signatures serve the same purpose as handwritten signatures, to allow the recipient to verify that the message was created by a particular sender. Typically, they use a similar structure to asymmetric encryption, where the private key is used to sign a message, and the public key may be used to verify if the signature is authentic, as shown in Figure 2.7.

2.3.5 Key Derivation Function

A Key Derivation Function (KDF) can be used to transform a secret, but unsuitable, value such as a password into a secret key that may be used for encryption, for example by stretching it to the required format. A HKDF is a KDF based on a HMAC [4].

2.3.6 Key Encapsulation

Asymmetric encryption is inefficient for longer messages. It is therefore a common practice to use the asymmetric encryption to transmit a key for symmetric encryption. A Key Encapsulation Mechanism (KEM) [5] is an algorithm that provides the functionality of generating this key and encapsulating it for transmission. The only input is the recipients public key and the output is a key and the encrypted version of that key.

2.3.7 Hybrid Public Key Encryption

Hybrid Public Key Encryption (HPKE) [6] creates a combination of symmetric and asymmetric cryptography together with authentication by using a Key Encapsulation Mechanism, a Key Derivation Function and a method for Authenticated Encryption with Associated Data.

2.3.8 Ratchet Function

A ratchet function can be used to make sure that the key used for communication is ephemeral, i.e. constantly updating and short-lived. It should not be possible to reverse this process. This property is used to achieve forward secrecy, as leaking a key does not compromise previous keys. In its simplest form this may be achieved by repeatedly hashing a value as in Figure 2.8 where part of the output may be used as a key and part as input to the next hash function. It may also include new information in each step, for example by including information from a previous message in the input.

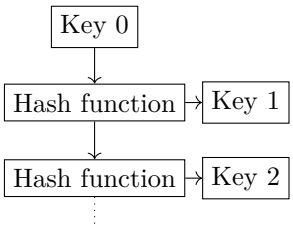


Figure 2.8: A ratchet function.

2.3.9 Diffie-Hellman Key Agreement Protocol

The Diffie-Hellman key agreement protocol [7] is a method for establishing a shared secret over an insecure channel. Using this shared secret then allows for encrypted communication over this insecure channel with symmetric cryptography. It does not include any authentication itself, but can be used together with or as a base for authenticated protocols. The protocol works as follows for user a and user b:

- Users a and b agree to use a base α and a modulus β.
- Both select a secret integer, a "private key", X_a and X_b and use it to calculate and publish their "public keys" $Y_{a/b} = \alpha^{X_{a/b}} \mod \beta$.
- User a calculates $Secret = Y_b^{X_a} \mod \beta$ and user b calculates $Secret = Y_a^{X_b} \mod \beta$.

They now share a common *Secret* that can be used to secure their communication, but by watching the published information an attacker can not recreate *Secret* or $X_{A/B}$ without solving the equation $Y_{a/b} = \alpha^{X_{a/b}} \mod \beta$ for $X_{a/b}$. This is believed to have no efficient solutions for the general case and is called the discrete logarithm problem. Variants of this algorithm using elliptic curves [8] exist and is used in the Signal protocol. Here the security is based on the elliptic curve discrete logarithm problem and the benefit is the ability to use smaller key sizes for the same level of security, reducing the size of storage and transmitted data, but also increasing performance [9]. Diffie-Hellman is not asynchronous in its basic form, but can be used asynchronously, by previously storing one user's information on a server until use.

2.4 The Signal Protocol

The current state-of-the-art protocol for end to end encrypted messaging and calls is the Signal protocol. In addition to the Signal application itself it is in some adaptation used by among others WhatsApp [10], Facebook Messenger [11], Skype [12] and Wire [13]. It provides forward and backward secrecy with its double ratchet algorithm that updates the key with each message.

For group messaging, the Signal application uses multiple pairwise messaging setups. It simply adds a group id to the encrypted message to allow for the receiver to distinguish it from other messages, and keeps a list of members locally. This does mean that it needs to generate $N-1$ different messages and send one to every other participant. Removal of a user is done by sending a message telling the other users to remove the user from their list of group members. The creators of Signal, Open Whisper Systems, have said that they are currently redesigning how Signal handles group messaging [14]. Chase et al. [15] have introduced a new system for maintaining membership lists in an encrypted form on the server and also providing support for roles with different permissions. Initial work on implementation is expected in the coming months [16].

The Signal application improves efficiency when sending attachments by encrypting them using a temporary key and uploading it to a separate server. The key, a hash and a pointer to the encrypted file on the server is then sent over the normal Signal protocol.

While WhatsApp uses the Signal protocol [10], it does not handle group conversations in the same way as the Signal app. Instead, it uses a concept called sender keys where a participant first generates and distributes a sender key over the pairwise protocol. It then uses this sender key to encrypt subsequent messages and deliver messages to participants using server-side fan-out. This means that the message is encrypted once and sent to the server, which will distribute the same message to all participants. This improves efficiency for sending messages, but does have the consequence of losing backward secrecy unless sender keys are regularly replaced [17]. A leaked sender key allows an attacker to eavesdrop on that participant's messages until the sender key is replaced, but replacing sender keys is an expensive procedure. WhatsApp also limits the number of participants in a group to 256, though this can be circumvented client-side [18, 19].

2.4.1 Extended Triple Diffie Hellman (X3DH)

The X3DH protocol[20] is used in the Signal protocol's setup phase for key agreement. It establishes a shared key, `SK`, between two parties. There are three phases in X3DH, illustrated here by an example where Alice establishes a session with Bob

1. Bob publishes a prekey bundle containing an identity key IK_b, a signed prekey SPK_b, a signature of SPK_b created with IK_b and optionally a one-time prekey OPK_b.

2. Alice gets the prekey bundle and performs a set of Diffie-Hellman operations using the values from Bob's prekey bundle, Alice's identity key IK_A and the public part of the ephemeral key EK_A that Alice generates.

   $DH1 = DH(IK_a, SPK_b)$
   $DH2 = DH(EK_a, IK_b)$
   $DH3 = DH(EK_a, SPK_b)$
   $DH4 = DH(EK_a, OPK_b)$
   $SK = KDF(DH1||DH2||DH3||DH4)$

   For the case where the OPK_b does not exist $DH4$ is left out. Alice should now delete all Diffie-Hellman outputs and the public part of EK_a. Using SK (or something derived from SK) Alice can now send an initial message containing, among others, IK_a, EK_a and an identifier of which prekey Alice used. She also includes the first message, encrypted with AEAD, where the associated data is a combination of IK_a and IK_b.

3. Bob receives Alice's initial message and can repeat the same DH and KDF operations as Alice did. If bob can decrypt the first message using SK and the same associated data the protocol has completed successfully.

2.4.2 Double Ratchet

The Double Ratchet algorithm [21] is used by the Signal protocol to exchange messages based on the shared key generated using X3DH. In short this algorithm generates new ephemeral keys for each message. It does this using three chains and two ratchet types.

The sender and receiver chains are updated using the Symmetric-key ratchet. These are equivalent but switched between the participants, so Alice's sender chain matches Bob's receiver chain. The Symmetric-key ratchet is applied when a message is sent or received. It uses what is called a chain key as input and outputs a new chain key and a message key.

Since one compromised chain key would allow an attacker to compromise all the following messages, breaking backward secrecy, this is combined with the root chain which uses the

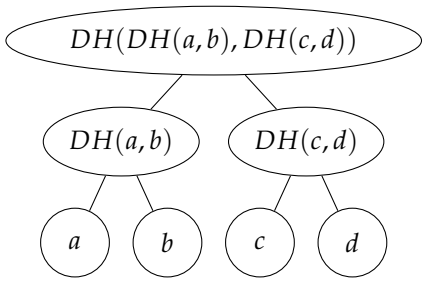


Figure 2.9: Example Asynchronous ratcheting tree.

Diffie-Hellman ratchet. The Diffie-Hellman ratchet uses the opportunity to include a public Diffie Hellman value in the messages. The output of these Diffie-Hellman operations is used to advance the root key twice, which generates two new chain keys, one for the receiver chain and one for the sender chain. Since new material is included in the chain keys, this provides backward secrecy.

2.4.3 Asynchronous Ratcheting Trees (ART)

Traditionally, end to end encrypted group messaging relies on pairwise communication. This does, however, lead to inefficiencies for group messaging as each message has to be encrypted and sent to each participant separately. This scales linearly in both compute and network data usage. The goal of Asynchronous Ratcheting Trees (ARTs) first proposed by Cohn-Gordon et al. [17] is to allow for asynchronous communication where no pairs need to be online at the same time, without the action of sending one message scaling linearly, and keeping security guarantees such as backward secrecy. The idea behind ART is to generate a shared group secret that can be used to encrypt a message only once to the whole group, which can then be sent to everyone with a server-side fan-out.

It does this by creating a binary tree, where each member device is represented by a leaf node. An example is shown in Figure 2.9 where a, b, c and d are group members. Here it can also be seen that the tree has a height of $\mathcal{O}(\log(N))$ where N is the number of members. Their parent nodes are then generated by performing Diffie-Hellman operations between the two children. Intermediate nodes in the tree represent subgroups, so each device is also potentially a member of $\log(N)$ subgroups. The members can then also use a subgroup's public key to send messages to those subgroups, and use the private keys of the subgroups it is a part of to decrypt them. In the example in Figure 2.9, a is also a part of the groups represented by its parent and the root node.

A key update or addition of a new user then results in a fresh leaf key and updates up to the root node along the direct path with Diffie-Hellman operations of two sibling nodes forming the parent node. This means that every group modification changes the root key. This also means all nodes know the private keys of their parents, but not for any other nodes.

A removal will result in the path from the root to the removed member's leaf node being blanked and a new shared group secret derived. Measurements by the authors show that creation of groups is slightly less efficient in computing time, but with the same linear asymptotic trend as pairwise channels, while the number of bytes used for sending a application message including a key update scales logarithmically for ART compared to linear for pairwise, comparing favourably for all but the smallest groups.

TreeKEM has since been developed out of the ideas from ART, and MLS now uses TreeKEM.

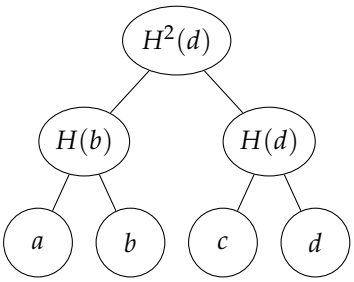


Figure 2.10: Example TreeKEM tree, created in the order a, b, c, d, with nodes represented by private keys.

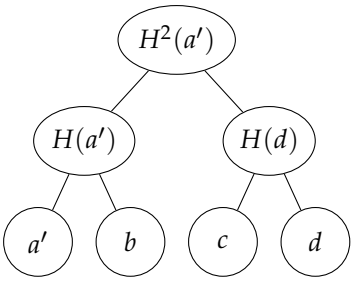


Figure 2.11: Example TreeKEM tree, resulting from a in Figure 2.10 updating its key to a'.

2.4.4 TreeKEM

Just like ART, TreeKEM by Bhargavan et al. [22] arranges the users or devices as leaf nodes in a left-balanced binary tree. Instead of Diffie-Hellman operations, the parent key is computed by hashing the key of the last modified node. The hash of a key is written as $H(key)$, a secondary hash as $H^2(key)$ and so on. An example would be the tree in Figure 2.10, created by adding nodes in the order a, b, c, d. This tree is used as a source for the following examples.

If a wants to update its private key to a' it results in the tree in Figure 2.11 by sending $H(a')$ to b that can use it to compute $H^2(a')$ and sending $H^2(a')$ to c and d. These keys can be sent by using the fact that we can encrypt messages to a subgroup (recall that every node knows the private keys of its parents). So we send $E(b_{pub}, H(a'))$ and a'_{pub} to b and $E(H^2(d)_{pub}, H^2(a'))$ and $H(a')_{pub}$ to the group with c and d where $E(x, y)$ means that y was encrypted with x. So the updater needs to encrypt and send $\mathcal{O}(\log(N))$ messages. Each other node receives one message containing a secret. Taking the example of b it receives a'_{pub} and $H(a')$, and can compute $H^2(a')$ from $H(a')$, so each device does at most $\log(N)$ hashes. As can be seen in this example, every device needs to know the private keys in the path from its leaf to the root and the public keys for siblings to each node in that path, also known as the copath. This corresponds to a storage requirement of size $\mathcal{O}(\log(N))$ for participating in a group of size N.

Adding a member e to the tree in Figure 2.10 then results in Figure 2.12.

Removing b from Figure 2.10 results in Figure 2.13. Member a may generate a new key b', that b does not know, and then perform a normal update operation using b'. All (sub)groups that b was a part of now has a fresh key unknown to b.

Since add and remove operations behave similarly to update they also need $\mathcal{O}(\log(N))$ encryptions and public key derivations for the issuer and one decryption and $\mathcal{O}(\log(N))$ hashes for the receivers.

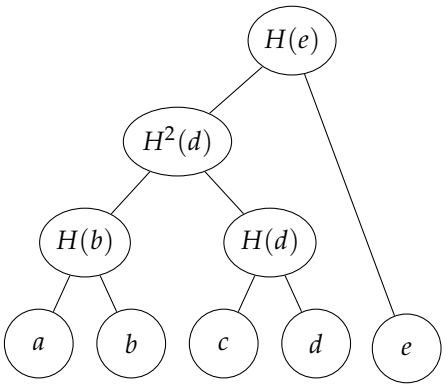


Figure 2.12: Example TreeKEM tree, resulting from adding e to the tree in Figure 2.10.

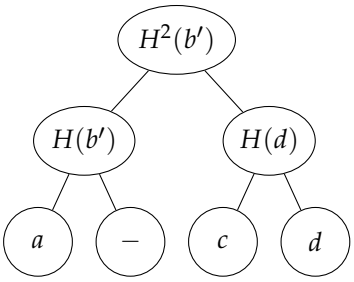


Figure 2.13: Example TreeKEM tree, resulting from removing b from the tree in Figure 2.10.

2.5 Messaging Layer Security (MLS)

MLS is a specification intended as a secure layer for messaging in groups from two to approximately 50 000 users. It is currently a work in progress by a group in the standards organization IETF. The first drafts were based on ART, while current versions are based on TreeKEM. It is not a complete implementation but rather an architecture and protocol specification [23, 24]. There are however some initial implementations, of which MLS++ by Cisco [25], Molasses by Trail of Bits [26] and Melissa by Wire [27] are the ones being focused on in this report. Also, notable is an unpublished implementation by Google [28]. MLS++ and Molasses mostly follow draft version 06 or 07, while Melissa uses version 05. MLS++ is considered the unofficial reference implementation [29]. A client-server implementation without support for application messages based on Melissa, that was last updated in November 2018, exists [30, 31].

MLS is intended to be a quite general specification, of which the most important aspects are described below. This also means that it leaves some important decisions to the application layer, such as the content and format of the message payload and the method of communication. It is presumed that the transport layer is secured, but it is not specified which transport layer should be used, and a compromise of the transport layer will generally be survived. It also lets the application select a cipher suite containing a hash function, a Diffie-Hellman group or curve and an AEAD encryption algorithm. These are then used together with a HPKE cipher suite that additionally specifies a KEM, a HKDF and a `Derive-Key-Pair` function that produces a asymmetric key pair from a symmetric secret.

2.5.1 Trees in MLS

MLS differs in some ways from TreeKEM as originally described by Bhargavan et al. [22] and Section 2.4.4. The hash function for generating parent keys has been replaced with a KDF. The node keys are now generated as described in Listing 1 where `HKDF-Expand-Label(Secret,`

`Label, Context, Length)` is an abstraction on top of `HKDF-Expand(Secret, HkdfLabel, Length)` as described by [32], where the label ("path" or "node") is combined with the context (here empty) and a hash of the current group state to form the `HkdfLabel`. `path_secret[0]` is a random value generated by the leaf doing the update, node keys and path secrets for parents $(n+1)$ are generated from that. An `update` or `remove` then consists of the new `path_secret` values in the direct path to the root encrypted with the public key of the node which is updated. The other members can update their tree by decrypting the path secrets in their direct paths.

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

Listing 1: Pseudocode for updating a ratchet tree in MLS [24].

MLS assumes every participant has a complete view of the public state of the tree with public keys for all nodes, not just those in its copath. For every node in the tree a participant has a public key, for the nodes in its direct path it has a private key, and for leaf nodes it has credentials, so no participant has a complete view of the private state of the tree, only the subgroups it is a member of.

2.5.2 Verifications

There are two main hash values used to verify the group state. One verifies the tree by recursively hashing it. The hash value of each node is based on information about the current node and, for non leaf nodes, the hashes of its children. Additionally, there is a running transcript hash value that is created using the group operations leading to the current state, where in every step a combination of the previous transcript hash and the current operation is hashed.

2.5.3 Key Schedule

Multiple keys and nonces get updated each epoch, that is each time the group state changes, using a number of key schedules. These are used for verification and encryption of different message types.

2.5.4 Server

MLS expects a messaging service to provide two vital services, an Authentication Service (AS) and a Delivery Service (DS). This may be provided by the same service provider, but may also be separated.

Authentication Service (AS)

The Authentication Service is mainly a database and connects an identity (for example a phone number or username) to one or more keys as a long term identifier. A long term identity key can be used by the client to authenticate protocol messages.

Delivery Service (DS)

The Delivery Service is responsible for delivering messages between clients, allowing for asynchronous communication. It can also do a server-side fan-out by broadcasting messages to the

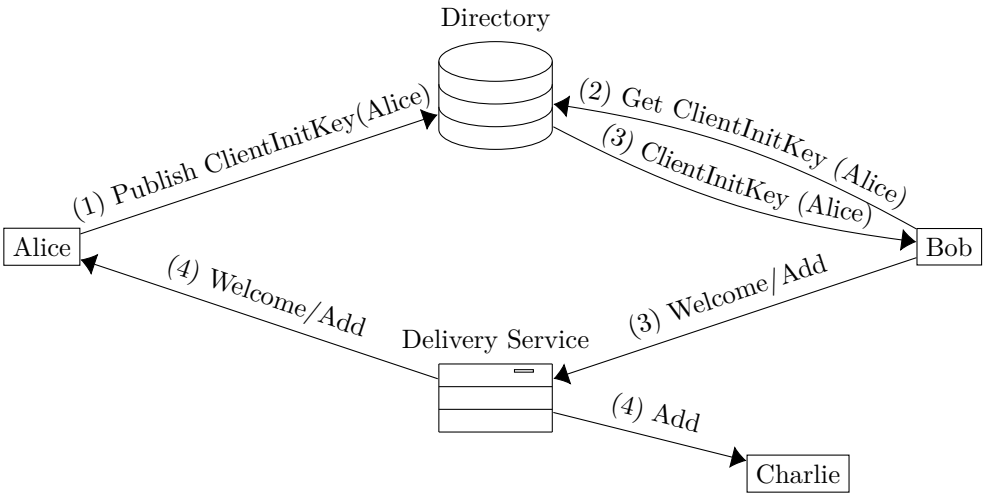


Figure 2.14: Adding a new group member. Bob is already in a group with Charlie and would like to add Alice to the group. Alice has published her `ClientInitKey` to the directory provided by the Messaging service. Bob invites Alice to a group by getting her `ClientInitKey` from the directory and using it to generate a Welcome and Add message. The Welcome gets sent to Alice, the Add to everyone in the group (including Alice).

whole group. It stores messages until the recipient becomes available (or another condition is met, such as a timeout). Clients upload initial keying material and information about the supported cipher suite to a directory provided by the DS, which other clients that would like to communicate with them then can request. These initialization keys are intended to be used just once. Thus, a member may publish multiple initialization keys, as long as they all have a unique identifier. Since it is authenticated with the credentials from the AS the clients can verify its authenticity. MLS does not require the DS to have static knowledge of group constellations, but it is possible for a DS to learn it using traffic analysis. The application may for example include a list of recipients in the message metadata.

There are some requirements on the delivery by the DS, namely that messages will eventually deliver, group operation/cryptographical messages are delivered in order and other messages are delivered approximately in order. It is possible to use a sequence number as an alternative, allowing the clients to reorder after delivery.

2.5.5 Group Operations and Message Types

MLS has four message types corresponding to different group operations, `welcome`, `add`, `remove` and `update`. In addition to this there are application messages containing the actual data.

Group Creation and Addition

Clients publish initialization keys, `ClientInitKeys`, to the DS, containing identifiers and public keys. Any client can request another user's initialization key from the DS. If user a wants to create a group consisting of a, b and c it will first request initialization keys for b and c, then create a group state containing a. After that it will send a `welcome` message representing the current group state to b, then an `add(`b`)` message to a and b upon which a and b update their group state to include b. Then it sends a `welcome` message to c, and an `add(`c`)` message to a, b and c. If any member would then like to add user d it would compute a `welcome` message using the initialization key from d, send it to d and then broadcast the `add(`d`)` message to a, b, c and d. An example adding a user to an existing group can be seen in Figure 2.14. It is recommended that the new member performs an `update` immediately after

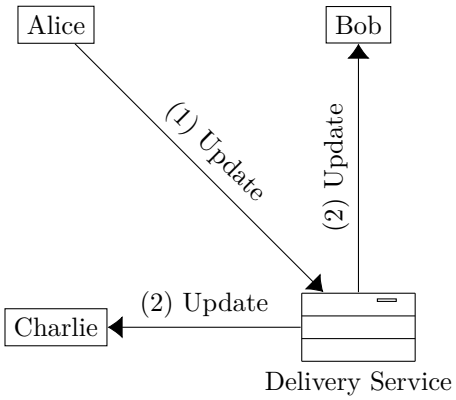


Figure 2.15: Alice updating her key.

being invited, as this will keep the tree balanced. There is also an initial definition of a more efficient initialization procedure in the MLS draft 07 and 08, but this is not yet implemented in most implementations.

Updating

An `update` changes a member's leaf secret and the direct path from that leaf and provides backward secrecy regarding the member's leaf secret. It is up to the application to decide when to do an `update` operation. It may for example be done periodically or after each message. If a member wants to start an `update` it generates a new leaf secret and sends an `update` message which others can use to update their group state. An example of this can be seen in Figure 2.15.

Removal

Removals are done similarly to updates. A member sends a `remove` message with the index of the member to be removed and the direct path from the senders leaf. The direct path from the removed members leaf to the root is blanked and the tree is truncated at the rightmost blank leaf.

Application Messages

These are the actual messages used for application data such as text messages, attachments or in the case of calls for WebRTC signaling. They may contain any data. These are encrypted once for the whole group and broadcasted in the same way as group operation messages.

Message Framing

MLS messages can be framed in two structures, `MLSPlaintext` or `MLSCiphertext`. The plaintext only signs the message while the ciphertext also encrypts it. `MLSCiphertext` should be used for application and group operation messages, but `MLSPlaintext` may be used for group operation messages if there is a need for the delivery service to examine those messages.

2.5.6 Protocol Draft Version 08

A new protocol draft was released in the later stages of this thesis (2019-11-15). The main change is that group operations have been split into proposals and commits, which now also can be sent as a collection of multiple proposals per message. There is also further work on

the efficient creation of groups with multiple initial members. These changes have not been included in this work due to the late release date.

2.6 Theoretical Group Messaging Efficiency Comparison

MLS is theoretically more or equally efficient compared to both the Sender key solution used by WhatsApp and the pairwise based solution used by Signal for all operation types required for a group messaging scenario.

2.6.1 Group Operations

For MLS the creation of `update` and `remove` messages is expected to run in $\mathcal{O}(\log N)$ and result in messages with the same scaling characteristics [33]. Creation of groups is supposed to run in $\mathcal{O}(N)$ for the sender and $\mathcal{O}(1)$ for receivers once efficient group creation is completed [34], but as this is currently done using repeated additions it will show worse scaling.

Signal using pairwise channels should require $\mathcal{O}(N)$ for both sender and receiver during group creation as every participant needs to establish a channel to all other participants. Updates are included in application messages. Removes are not part of the protocol. In the Signal app they are handled by sending an application message asking the participants to remove the relevant channels.

2.6.2 Application Messages

MLS creates and handles application messages in $\mathcal{O}(1)$ regarding the group size, while with Signals pairwise channels a separate copy has to created for every other participant, scaling in $\mathcal{O}(N)$. The amount of data sent scales in the same way [33].

2.7 Web Real-Time Communication (WebRTC)

WebRTC [35] is a project for audio and video based peer-to-peer communication supported by many browsers and used by other application such as the Signal messaging application and Google Hangouts.

2.7.1 Setup/Signaling

WebRTC uses separate channels for signaling and the actual media. The transport mechanism for the signaling is not specified in the standard, but generally communication is done via some server. The signaling channel may be used to negotiate a communication channel for media or a codec with the Session Description Protocol (SDP) and Interactive Connectivity Establishment (ICE).

2.7.2 Two Party Calls

Generally, a WebRTC application tries multiple different routing methods in the setup phase. One is a direct peer-to-peer connection. To facilitate that, the clients need to know each other's public IP addresses which they can receive by contacting a STUN (Session Traversal Utilities for NAT) server and asking for it. In case that does not work, for example because one or both of the devices are behind a NAT (Network address translation) device, a relay server (TURN - Traversal Using Relays around NAT) is used. These connection options are negotiated using ICE by exchanging SDP messages. Some WebRTC applications (FaceTime [36], Slack [37]) use TURN or a similar technology by default or as a preferred option, likely because of the improved setup time.

2.7.3 Conference Calls

A problem with using direct connections is that for conference calls in bigger groups with N participants each device needs to send $N-1$ copies to all other participants with client-side fan-out. Sometimes a Multipoint Conferencing Unit (MCU) or a Selective Forwarding Unit (SFU) is used to help with this issue. A MCU is a centralized server which takes the streams from all inputs and mixes and re-encodes them together into one stream which is sent to the other participants. This is computationally expensive, breaks end to end encryption and introduces latency. A SFU similarly receives all media streams and then decides which streams to forward to which other participants. It is less computationally expensive but also introduces latency and breaks end to end encryption (though there is a proposed standard aiming at solving this issue [38]).

Another WebRTC feature helping with conference calls is simulcast, where multiple streams are sent, for example with different quality. A SFU or MCU may then choose to use one or multiple of these streams, for example sending a lower quality stream to participants using the mobile application.

2.7.4 Security

The draft specifies that the implementations must always encrypt data using SRTP (Secure Real-time Transport Protocol) [39, 35]. This includes end to end encryption, even when a TURN server is used. As the signaling channel is not specified, the security in the general case is unknown. In this thesis however, MLS based signaling will be used to also provide end to end encryption and authentication for the signaling layer. The signaling layer is used for setup and to compare certificate fingerprints for verification of the SRTP connection.

2.7.5 Usage in Messaging Applications

Many major communication providers including WhatsApp [10], Signal [40], Skype [12] and Wire [13] use WebRTC for voice and video calls. For the signaling channel the respective messaging channels are used, providing the same security guarantees as for messages. On top of that they all use SRTP based WebRTC connections with the encryption keys generated over the secured signaling channel.

3 Method

To answer the research questions base implementations for MLS and Signal had to be selected for use in benchmarks, and an MLS implementation for the proof of concept implementations. The call functionality was implemented with Rust based Molasses by Trail of Bits based on a older draft version[1]. A separate messaging implementation was made with MLS++, with Android and Linux console clients. For benchmarks the three main MLS implementations, Melissa, Molasses and MLS++ where selected, together with `libsignal-protocol-java` for the Signal protocol. This was both to allow comparisons between MLS implementations, to get an evaluation of as many aspects as possible since not all features of MLS are supported in all implementations, and to provide some protection from implementation specific issues affecting the results.

All software was developed and tested on Linux, except where otherwise mentioned.

3.1 Messaging and Call Implementations

To answer RQ 1, two proof of concept applications for messaging and audio/video calls have been implemented.

3.1.1 Implementation 1

The first implementation is based on Molasses by Trail of Bits in draft version 04 and Mozilla's browser based WebRTC chat demo[41], with the addition of a custom `MLS-Client` and `MLS-Server` application. It uses the structure in Figure 3.1 where each dashed node represents a client application. Red dashed lines represent communication over unsecured WebSockets, while green solid lines represent communication using MLS on top of WebSockets and the green dotted line represents the peer-to-peer connection used for media in WebRTC calls.

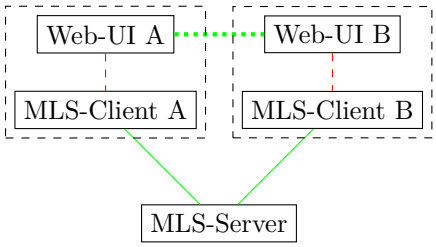


Figure 3.1: Structure of components in MLS setup.

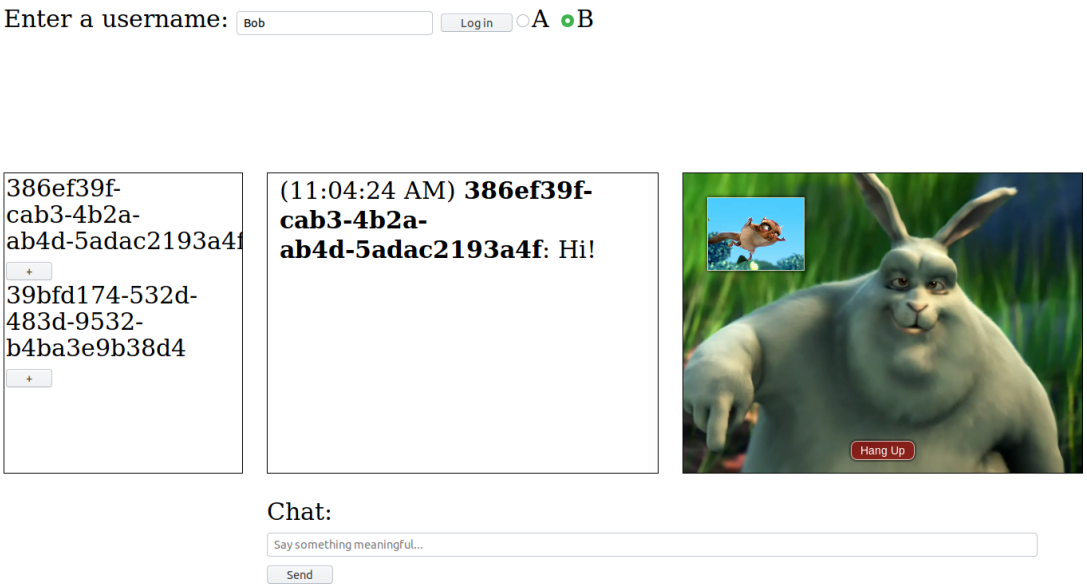


Figure 3.2: Web client with an ongoing video call and a chat message.

Client

The client is based on two components, the `Web-UI` and the `MLS-Client`, as shown in Figure 3.1. The `Web-UI` is Mozilla's WebRTC chat demo with the addition of a selector for whether to act as the interface for Client A or B, and a button for creating a group which sends a message to the `MLS-Client` containing the other person's user ID. See Figure 3.2. On launch a WebSocket connection is opened to the `MLS-Client` application which will act as a proxy creating and handling MLS messages, and handle all communication with the `MLS-Server`. A and B are identical, except for which ports the connection between the `Web-UI` and the `MLS-Client` operate on to allow for separated usage on the same device.

Once connected, the user can choose to open a video call between two users. This results in WebRTC signaling being exchanged over the MLS channel which is used to establish a direct end to end encrypted connection between the two participants.

Server

The `MLS-Server` provides the authentication and delivery service for MLS. It keeps and distributes a list of all available user IDs, and manages a directory of their `UserInitKeys` which can be requested by other clients. The delivery service takes messages and delivers them to

[1]Molasses has a separate Git branch based on draft 07, but master is based on 04, and the new version does not fully support framed messages

`MLS-Clients` associated with a user ID listed in the messages target field. It can also broadcast messages to all available clients.

3.1.2 Example Interaction

An abstracted example interaction between two `Web-UIs`, two `MLS-Clients` and one `MLS-Server` can be seen in Figure 3.3. Here a group is set up between two users and a single application message is sent.

3.1.3 Implementation 2

To further verify whether MLS is ready for practical use as asked in RQ 1 a separate version based on MLS++ was implemented. This also supports groups of sizes bigger than two. In this case the client is implemented in a more realistic way in a single application.

Client

This client consists of an Android application using the Java Native Interface to call native C++ code that uses MLS++.

Server

This server is a simple SocketIO based server implemented in Python3 with the following methods.

- `publishcik`: Takes a `ClientInitKey` and a username. Is called when connecting to the server and associates the client with the username.
- `getcik`: Takes a username and returns a corresponding `ClientInitKey`. Called before inviting someone to a group.
- `welcome`: Takes a username, welcome message, add message and group id. The server sends this to the client associated with the username.
- `msg`: Takes an application message and a list of usernames. Sends the message to clients associated with the usernames in the list.
- `update`: Takes an update message and a list of usernames. Sends the message to clients associated with the usernames in the list.

These are transmitted as JSON messages with MLS messages in hexadecimal, for example a message sent to a group of two may look like:
`{"msg": "0400[...]29dd", "usernames": ["alice","bob"]}`.

3.2 Benchmarks

To verify the theoretical asymptotically results and to answer RQ 2, the following benchmarks are executed. A comparison of MLS implementations (Molasses [26], MLS++ [25] and Melissa [27]) against a Signal protocol implementation (libsignal-java [42]) are performed. The measurement points are time and amount of network data used for serialized messages. When comparing time between Signal and MLS the most interesting will be to look at asymptotical results. This will show the efficiency of the protocol instead of differences in implementation language and compiler optimizations, with MLS being implemented in Rust and C++ while Signal is running in the Java Virtual Machine (Java and Kotlin). Tests on the sender key method with the Signal protocol are not performed, as it provides different security properties, see Section 2.4.

The following measurements are performed:

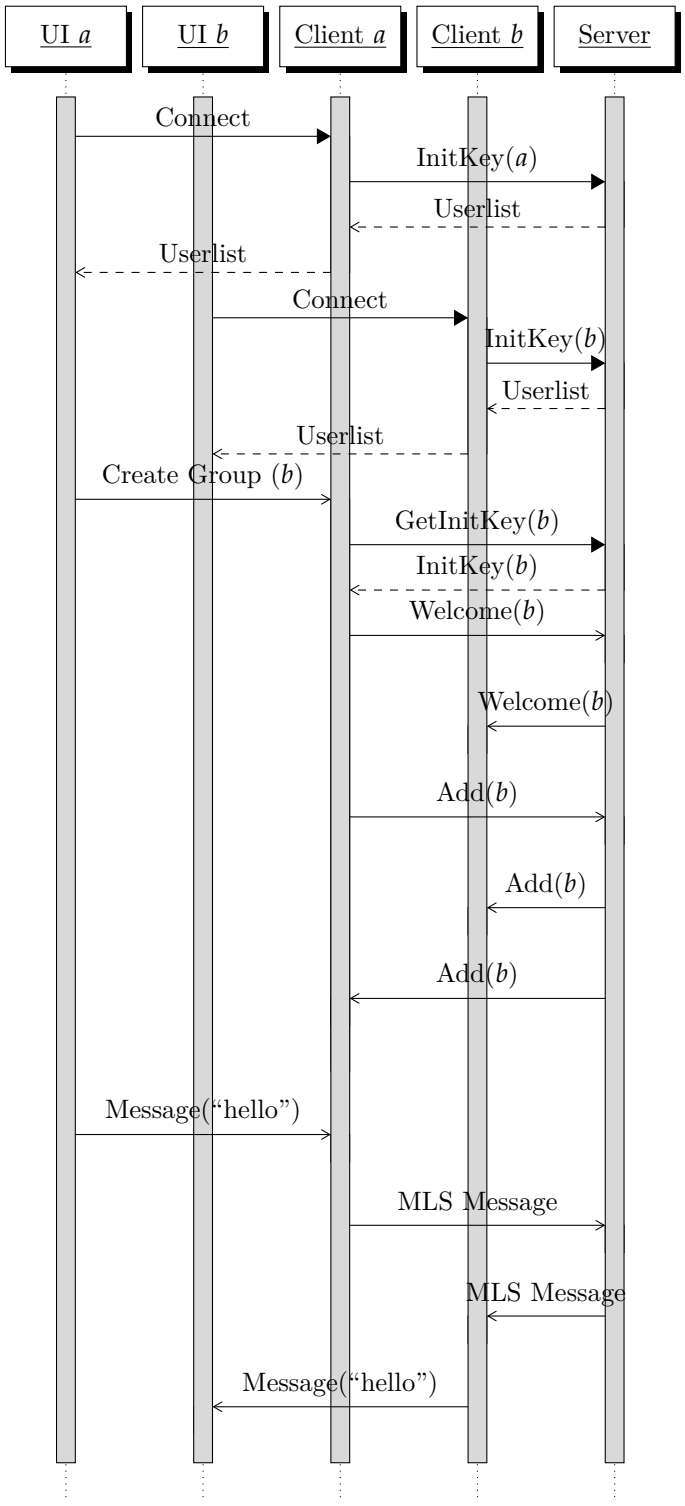


Figure 3.3: Sequence diagram of a sample interaction using Implementation 1. A group consisting of user a and b is set up and a message sent from a to b.

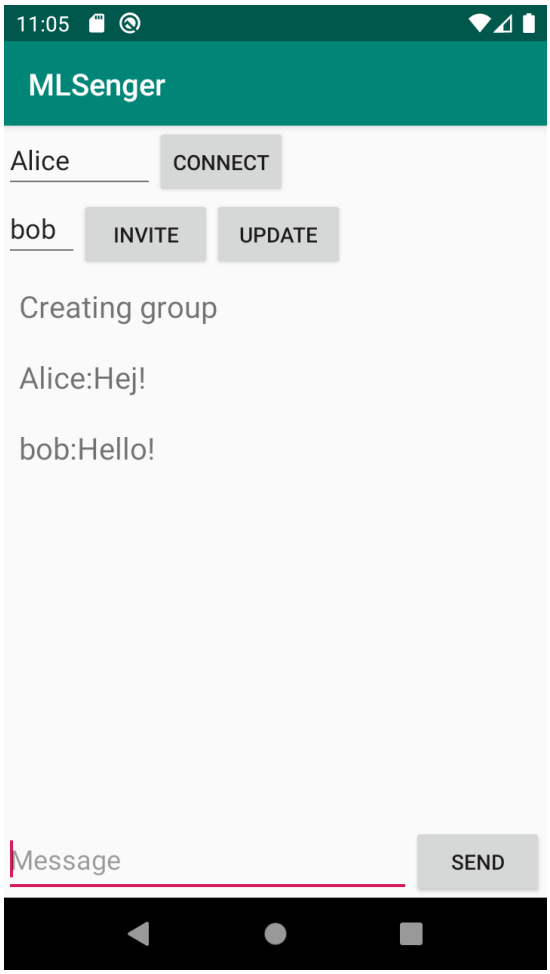


Figure 3.4: Android MLS client. Alice has invited bob to a group and they have both sent a message.

- Total time for creation of a group of size N. As the more efficient group creation procedure in the MLS standard was not completed at the time of measurement, and not supported by implementations, the creation of groups is implemented by doing multiple sequential additions. Pairwise Signal group creation is also be implemented with multiple sequential session establishments.
- The time required for creation of group operation messages and their message size. `Welcome/add`, `update` and `remove`.
- The time required to handle a group operation messages by the recipients. `Add`, `update` and `remove`.
- The time required for creation of an application message sent to the whole group and the handling of it by everyone else. Here in addition to group sizes it is interesting to vary message sizes to simulate common scenarios. Selected sizes of random data are 32 bytes (short text message), 128 bytes (medium size text message), 1024 bytes (long text message) and 5MiB (medium size picture). In addition the amount of data sent compared to the message content and group size is measured.

The most relevant for real use will be application messages and updates, since these happen frequently during a groups lifetime, while setup, add and remove only happens once or infrequently. These are tested for group sizes from 2 (remove from 3) to 100 in steps of 1.

The protocol specification for MLS recommends that after every `add` operation the newly added participant should do a `update` operation as this will keep the tree balanced. The benchmarks will follow this recommendation. Measurements do not include any network latency for sending of the messages. Where applicable, the generation of new material is included, such as new key material for updates and new user identities for `add`s. The first user in a group is the one creating the group operation messages, which is the worst case scenario as a left balanced tree will be at its deepest on the first leaf. The second user will handle the group operation message. The `remove` test case consists of the first user removing the third user. No messages are framed in `MLSCiphertext`, except for application messages where framing into a MLSCiphertext is the main thing being measured. MLS++ frames group operation messages in `MLSPlaintext`, while the others do no framing at all.

All benchmarks were performed on an AMD Ryzen 5 2400G set to performance scheduling aiming to keep a stable frequency and running OpenSuse Tumbleweed. Benchmarks are done without full compiler optimizations enabled (*O*0 for Clang, dev profile for Rust). All tests are done in memory with all users running sequentially in a single thread and without any network traffic.

The benchmarks for Molasses and Melissa where performed using the benchmark library Criterion.rs [43], while MLS++ and Signal have been benchmarked using a combination of the Google Benchmark library [44] and manual benchmark loops. All approaches start by running a warm up period where the routine is executed repeatedly to warm up caches. Then a benchmark loop runs where the routine is again executed a number of times. The total time for the benchmark loop is measured and then divided by the number of operations to get the final result. Where the operation modifies the state a number of copies are made before the measurement and these are then used only once in the benchmark loop.

3.2.1 MLS++

All measurements are run using MLS++, except for creation and handling of `welcome`/`add` messages.

3.2.2 Molasses

All measurements except for those relating to application messages are done using Molasses using the `draft-07` branch at commit `a232445`. Due to the way the Molasses API is structured, creation of group operation messages will also include applying that message to the creators group context.

3.2.3 Melissa

Melissa is based on the relatively old draft version 05 (2019-05-02), and does not support application messages. Group creation and creation of update, remove and add/welcome messages are measured, along with group operation message sizes. Due to a Melissa issue where it uses a 16 bit number to store vector sizes, benchmarks are limited to group sizes of 2 to 75.

3.2.4 Signal

Only creation of groups and application messages are measured as Signal does not have group operation messages in the same way as MLS.

4 Evaluation

This chapter presents and evaluates the results of the thesis.

4.1 Implemented Software

RQ 1 resulted in the implementation of two demo applications. One where two users could create a group and send messages or make video calls through a relatively separate web interface, and one where text messaging could be done in arbitrarily large groups by using Android and console clients.

4.2 Benchmark Results

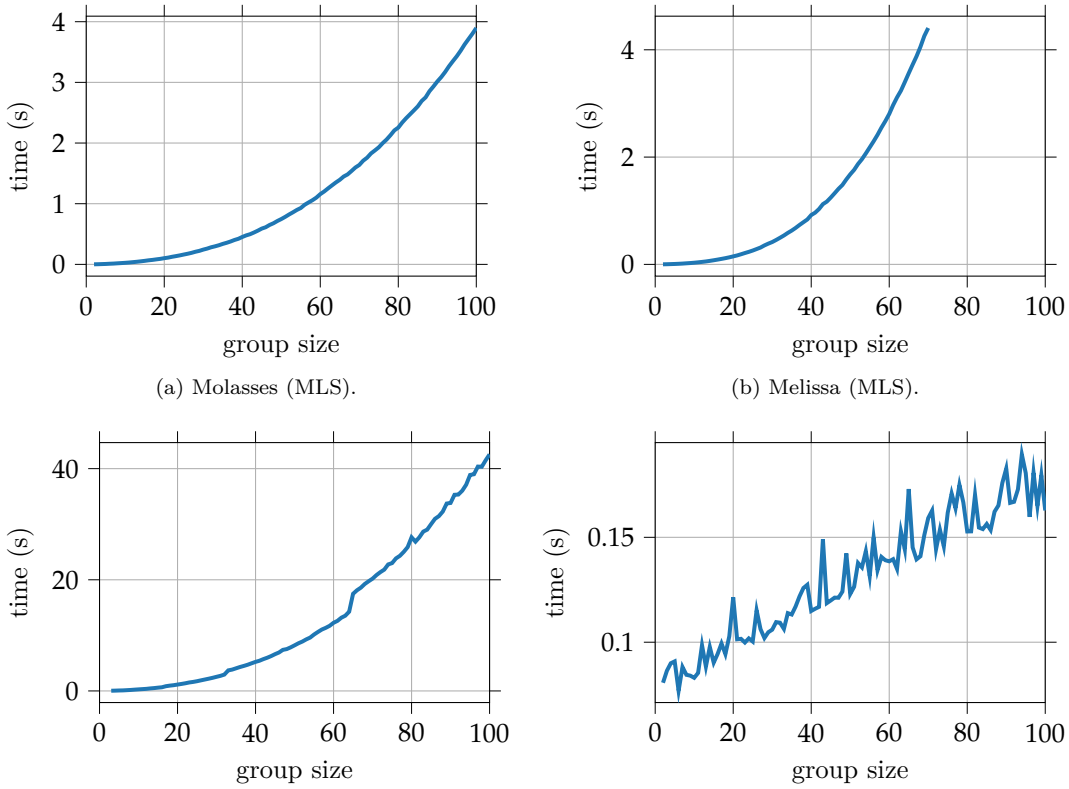
The benchmarks are presented in four different groups, one regarding the creation of a group from scratch, one for creating the group operation messages, one for handling the group operation messages, and one regarding application messages.

4.2.1 Group Creation

Due to the group creation being done sequentially in one thread for all users these measurements correspond to the total work required to create the groups. As can be seen in Figure 4.1 this time shows quadratic growth for all MLS implementations tested, while Signal scales linearly. Most of the time for Signal is taken up by creating users, not sessions, which explains the linear scaling. MLS++ takes an order of magnitude more time compared to the other MLS implementations. This may be due to implementation details or bugs. It does similarly require an order of magnitude more RAM memory relative to the other MLS implementations benchmarked, suggesting memory leaks. Creation of groups in real scenarios may be faster measured in wall clock time due to the work being spread on different devices.

4.2.2 Group Operation Message Creation

As can be seen in Figures 4.2 to 4.4 the creation of update and remove messages scales logarithmically with all MLS implementations, which is the expected behaviour of an efficient implementation, but MLS++ does show a slight linear element to its scaling.



(a) Molasses (MLS).

(b) Melissa (MLS).

(c) MLS++ (MLS).

(d) Signal.

Figure 4.1: Time to create a group using different messaging solutions.

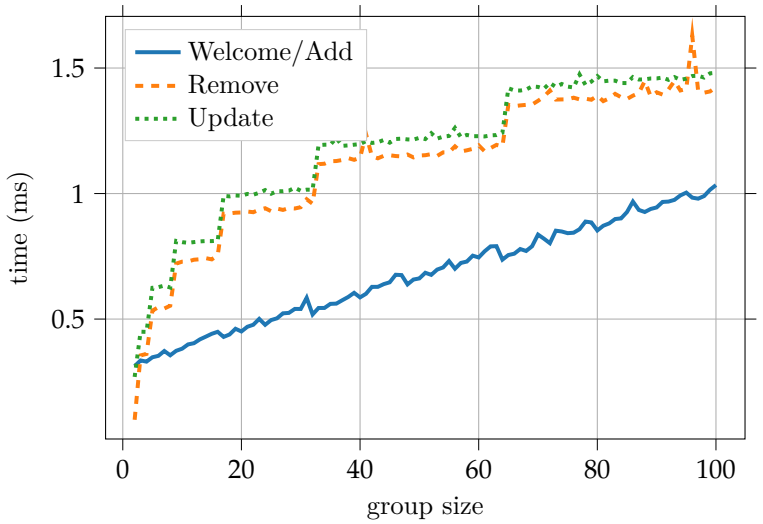


Figure 4.2: Time to make and apply a group operation message in groups of different sizes with Molasses.

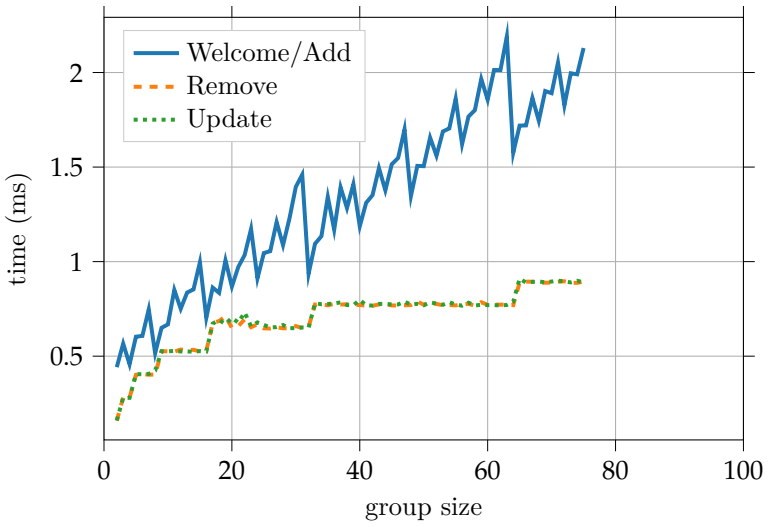


Figure 4.3: Time to make a group operation message in groups of different sizes with Melissa.

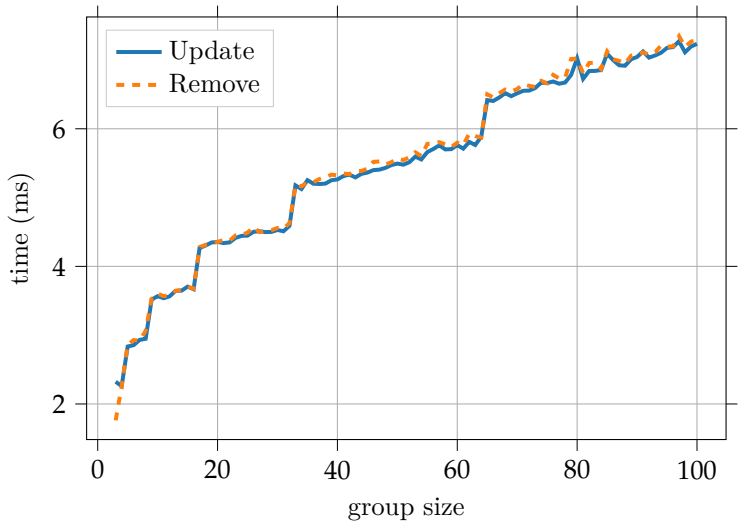


Figure 4.4: Time to make a group operation message in groups of different sizes with MLS++.

Figure 4.5 shows that the size of `welcome` messages with Melissa scales far worse than with MLS++ as it keeps a full transcript of all operations and includes this in `welcome` messages, while others follow the current draft and only include a hash of previous operations. This does increase both message size and creation time as can be seen when comparing Figure 4.3 with Figures 4.2 and 4.4. Figure 4.6 shows that except for Melissa's varying `add` sizes all message types follow the expected scaling, constant for `add` and logarithmic for all others.

4.2.3 Group Operation Message Handling

Handling a group operation message is generally faster than creating it, but here both Molasses and MLS++ show a linear element for all group operations, compare Figure 4.2 with Figure 4.7 and Figure 4.4 with Figure 4.8.

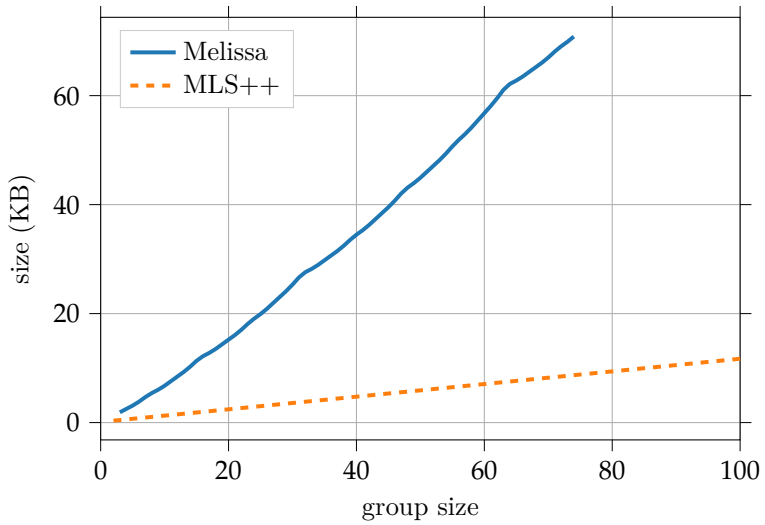


Figure 4.5: Size of `welcome` messages.

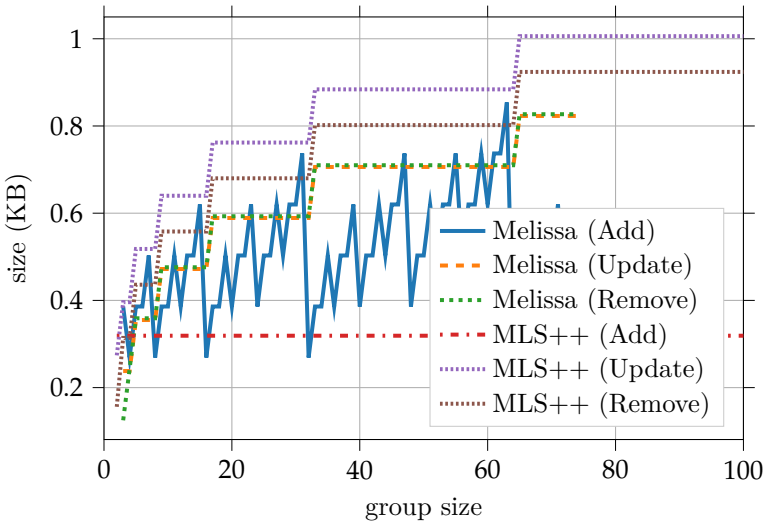


Figure 4.6: Size of other group operation messages.

4.2.4 Application Messages

Application messages have an additional parameter where the size of the plaintext being sent can also be varied. Application messages do not change in size with different group sizes using MLS, see Figure 4.9. Sizes increase linearly as the message content size increases, where the framed message has a constant overhead compared to the plaintext. Only one message needs to be generated and distributed, running in constant time and size for the sender.

For Signal a separate message needs to be generated and sent to each member, so as can be seen in Figure 4.9 the total amount of data that the sender of the message needs to send increases when increasing the group and message size for Signal, while the data amount for MLS does not change when increasing the group size.

The time required to create the messages to be sent follows the same pattern as the message size, see Figure 4.10. Notable is the fact that there is little change between 32 bytes, 128 bytes and 1024 bytes for both Signal and MLS++, suggesting a high constant overhead. Note that large messages can be optimized, see Section 2.4, but this is true for both Signal and MLS.

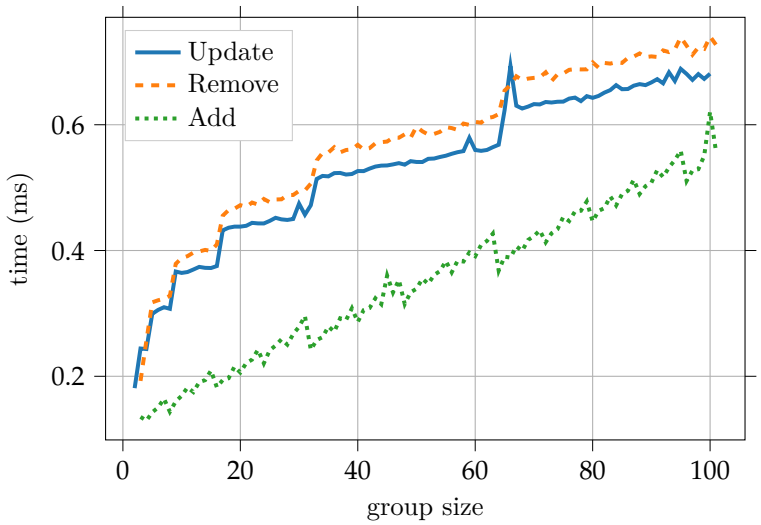


Figure 4.7: Time to handle a group operation message in groups of different sizes with Molasses.

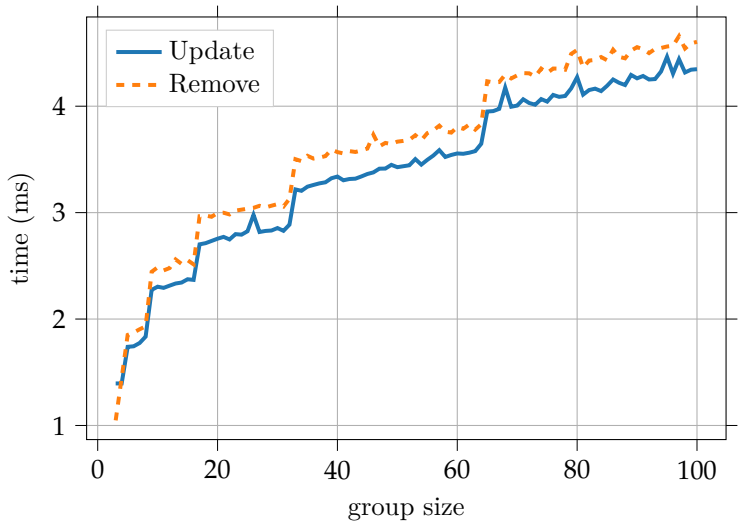


Figure 4.8: Time to handle a group operation message in groups of different sizes with MLS++.

As a Signal double ratchet message also introduces new key material providing backward secrecy it might be more fair to include a MLS update operation, increasing the time for creation and message size to $\mathcal{O}(plaintext\_size + \log(group\_size))$. As it is up to the application to choose the frequency of MLS updates it is possible to balance less frequent updates against better performance, which would tip the performance benefit towards MLS.

4.2.5 Performance of MLS Implementations

All MLS implementations have shown roughly the expected scaling characteristics, with MLS++ generally being a bit slower than Molasses and Melissa. Melissa shows its age in some tests where performance is impacted by changes between draft versions.

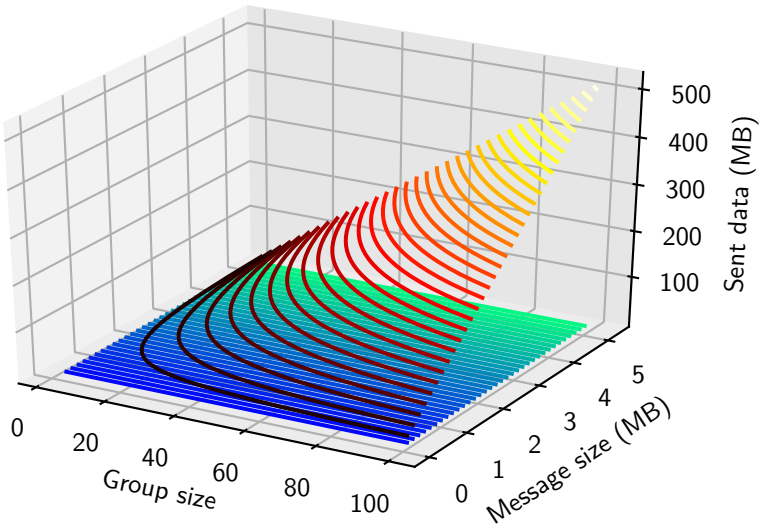


Figure 4.9: Total size of the application messages sent by MLS++ (blue-green) and Signal (red-yellow) for different (plaintext) message and group sizes.

4.3 Method

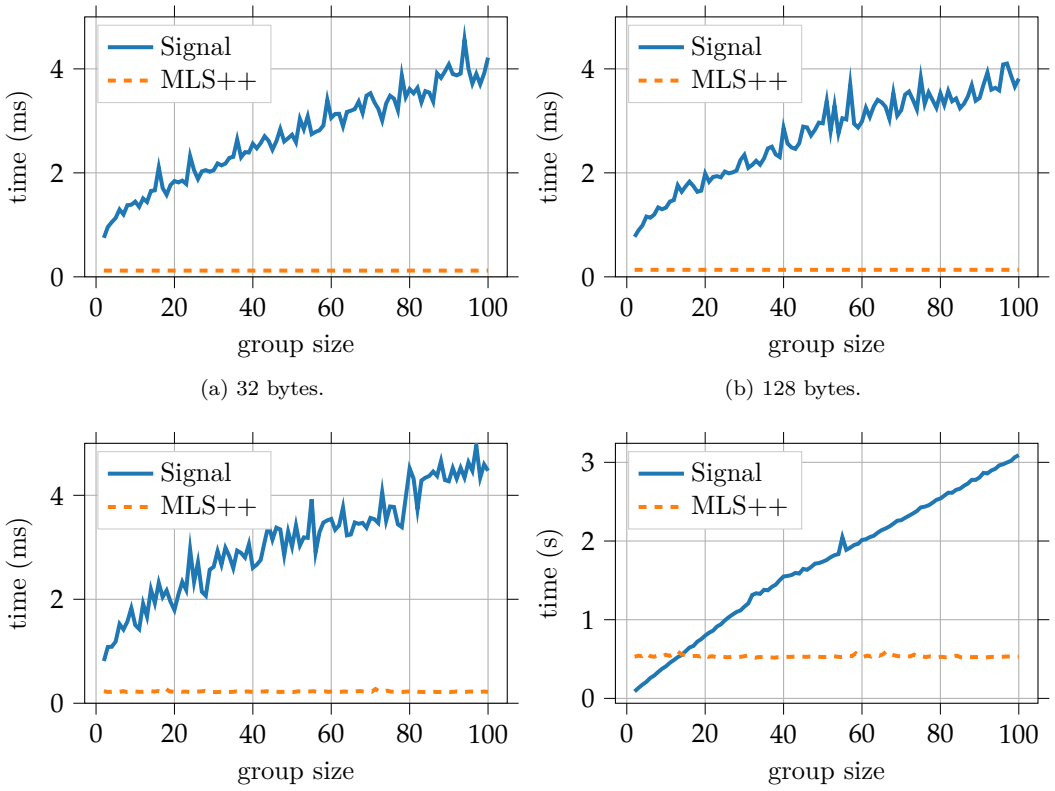
The chosen method was generally successful at producing results that answer the research questions, but some aspects could have been improved.

4.3.1 Messaging

The selection of Molasses as the base for the video call implementation was based on the completeness of it's example code, which included examples with transmission of messages, even though the used version was based on MLS draft version 4. This resulted in having to create a custom implementation of message framing which does not provide the same security benefits as the specification introduced in draft 5. The video call implementation also uses two client components which make it easier to implement but makes for a rather unrealistic architecture. In the Android application, Cisco's MLS++ was used, which supports message framing. This also provides a much more realistic implementation as it keeps the client in one application, rather than using a completely separate user interface.

4.3.2 Benchmarks

There is a degree of implementation specific influence in the benchmark results. Signal uses 256-bit AES vs 128-bit for MLS. Molasses and Melissa are also implemented in Rust, MLS++ in C/C++ and Signal in Java/Kotlin. While testing multiple MLS implementations does create the opportunity of comparing different MLS implementations and increases the reliability of the results, the difference in implementation language also makes it harder to fairly compare Signal to MLS. It does however not change the asymptotical results, which are the main takeaways from these benchmarks. The method should also describe the conditions of the



(a) 32 bytes.

(b) 128 bytes.

(c) 1KB.

(d) 5MB.

Figure 4.10: Time to create application messages with different plaintext sizes on MLS++ and Signal. Note the different y-axis scale on the 5MB plot.

benchmarks clearly enough to make replication possible. It may have been interesting to also run benchmarks using Signal's C library, or using the sender key method even though the security properties differ, but this was deemed out of scope.

4.4 Societal Impact

The existence of an efficient messaging protocol with high security properties and an open specification may increase the likelihood of secure messaging being used in more scenarios, and increase the privacy and security of the individual user. This may also increase difficulty for law enforcement operations. The work done in this thesis confirms the claims made about the performance properties of MLS and shows the protocol used in realistic end user applications. This may increase the legitimacy of the MLS protocol and help with future adoption.

5 Conclusion

The aim of this thesis was to evaluate the Messaging Layer Security protocol from a performance and usability standpoint. This was done by comparing the time and data requirements of typical group messaging operations in different MLS implementations to the Signal protocol, and by implementing two proof of concept applications using MLS.

5.1 Messaging and Call Implementations

RQ 1 asks whether the MLS specification and the available implementations are ready for practical use. This was evaluated by implementing two proof of concept applications. One implemented video call functionality based on WebRTC in a desktop application with a web based user interface, using the Molasses MLS implementation. The other was an Android application using the MLS++ implementation supporting text messaging. Both had a server component handling communication between the clients.

These proof of concept messaging applications show that the evaluated version of MLS and the draft implementations are already in a state where it is possible to make functional implementations for realistic scenarios. The video call application also shows that it is general enough to be used for more than a pure text messaging application. The evaluated draft and the implementations do however lack a complete feature set, where efficient group initialization and message framing are not yet completely supported. Also, as of this date there is only a quite old MLS server implementation published, though two basic examples where implemented as part of this thesis.

5.2 Benchmarks

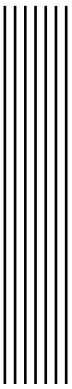
RQ 2 is about evaluating the performance of current MLS implementations compared to both the Signal protocol and the theoretical results that can be expected from the MLS specification. This was performed by measuring the computation time and message sizes when performing a variety of group operations. These where about creating groups, adding and removing group members, updating key material and sending application messages.

The performance results show that the current MLS implementations mainly follow the expected asymptotic behaviour, with constant or logarithmic growth relative to group size in

computation time and message sizes for most operations. This does confirm that the performance benefits compared to the Signal protocol are real. The measurements have also shown that the data usage benefits are significant. When sending application messages the amount of data sent does not increase with bigger group sizes while for the Signal protocol this increases linearly. While the Signal protocol had much better performance when creating groups, this is likely to change once the efficient group creation from MLS version 08 is implemented.

5.3 Impact and Future Work

The work done in this thesis confirms the performance benefits of MLS compared to the Signal protocol, and the implemented proof of concept applications increase its legitimacy as an upcoming standard for secure messaging. But before this happens some future work remains. One is a thorough security analysis. An extension to the work done in this thesis may be a large scale performance evaluation based on the proof of concept applications running on different devices, instead of using separate benchmark implementations. An updated performance evaluation once implementations catch up with MLS protocol draft version 08 would also be interesting, especially regarding group creation.



Bibliography

[1] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Ed. by Phil Sutherland. 2nd. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0471128457.

[2] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. "SoK: Secure Messaging". In: *IEEE Symposium on Security and Privacy*. San Jose, CA, 2015-05, pp. 232–249. DOI: 10.1109/SP.2015.22.

[3] David McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116. 2008-01. URL: https://rfc-editor.org/rfc/rfc5116.txt.

[4] Hugo Krawczyk. "Cryptographic Extraction and Key Derivation: The HKDF Scheme". In: *Advances in Cryptology – CRYPTO 2010*. Ed. by Tal Rabin. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 6223. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648. DOI: 10.1007/978-3-642-14623-7_34.

[5] Victor Shoup. "A proposal for an ISO standard for public key encryption (version 2.1)". In: *IACR e-Print Archive* 112 (2001).

[6] Richard Barnes and Karthikeyan Bhargavan. *Hybrid Public Key Encryption*. Internet-Draft draft-irtf-cfrg-hpke-02. Work in Progress. Internet Engineering Task Force, 2019-11. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-02.

[7] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976-11), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[8] Victor S Miller. "Use of elliptic curves in cryptography". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 417–426.

[9] Julio López and Ricardo Dahab. *An Overview of Elliptic Curve Cryptography*. 2000-05-22.

[10] WhatsApp Inc. *WhatsApp Encryption Overview*. Tech. rep. 2017-12-19.

[11] Facebook Inc. *Messenger Secret Conversations Technical Whitepaper*. Tech. rep. 2017-05-18.

[12] Microsoft Corporation. *Skype Private Conversation*. Tech. rep. Technical white paper. 2018-06-20. URL: https://az705183.vo.msecnd.net/onlinesupportmedia/onlinesupport/media/skype/documents/skype-private-conversation-white-paper.pdf.

[13] Wire Swiss GmbH. *Wire Security Whitepaper*. Tech. rep. Technical white paper. 2018-08-17. URL: https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf.

[14] "WhatsApp Flaws Could Allow Snoops to Slide Into Group Chats". In: *Wired* (2019-01). ISSN: 1059-1028. URL: https://www.wired.com/story/whatsapp-security-flaws-encryption-group-chats/ (visited on 2019-10-21).

[15] Melissa Chase, Trevor Perrin, and Greg Zaverucha. *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*. Cryptology ePrint Archive, Report 2019/1416. 2019.

[16] *Technology Preview: Signal Private Group System*. Signal Messenger LLC. 2019-12-09. URL: https://signal.org/blog/signal-private-group-system/.

[17] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS*. Toronto, Canada: ACM Press, 2018, pp. 1802–1819. DOI: 10.1145/3243734.3243747.

[18] Paul Rosler, Christian Mainka, and Jorg Schwenk. "More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. London, 2018, pp. 415–429. DOI: 10.1109/EuroSP.2018.00036.

[19] *What Are Good Ways to Add 3000 Users in a Whatsapp Group?* URL: https://www.quora.com/What-are-good-ways-to-add-3000-users-in-a-whatsapp-group/answer/Prakash-Sharma-338 (visited on 2019-10-15).

[20] Moxie Marlinspike. *The X3DH Key Agreement Protocol*. Ed. by Trevor Perrin. 2016-11-04. URL: https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[21] Moxie Marlinspike. *The Double Ratchet Algorithm*. Ed. by Trevor Perrin. 2016-11-20. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[22] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups*. Published in MLS mailing list https://mailarchive.ietf.org/arch/msg/mls/v1CY0jFAOVOHokB4DtNqS__tX1o. 2018-05-03.

[23] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. *The Messaging Layer Security (MLS) Architecture*. Internet-Draft draft-ietf-mls-architecture-03. Work in Progress. Internet Engineering Task Force, 2019-09-13. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-03.

[24] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol*. Internet-Draft draft-ietf-mls-protocol-07. Internet Engineering Task Force, 2019-07-08. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-07.

[25] *Cisco/Mlspp*. Cisco Systems. URL: https://github.com/cisco/mlspp (visited on 2019-10-25).

[26] *trailofbits/molasses*. URL: https://github.com/trailofbits/molasses (visited on 2019-09-30).

[27] *Wireapp/Melissa*. Wire Swiss GmbH. URL: https://github.com/wireapp/melissa (visited on 2019-11-12).

[28] Richard Barnes, Benjamin Beurdouche, Karthik Bhargavan, Katriel Cohn-Gordon, Cas Cremers, Jon Millican, Emad Omara, Eric Rescorla, and Raphael Robert. *Messaging Layer Security - The Beginning*. Real World Crypto. 2019. URL: https://rwc.iacr.org/2019/slides/MLS%20@%20RWC%202019.pdf.

[29] *Better Encrypted Group Chat*. 2019-08-06. URL: https://blog.trailofbits.com/2019/08/06/better-encrypted-group-chat/.

[30] *Wireapp/Mls-Server*. Wire Swiss GmbH. URL: https://github.com/wireapp/mls-server (visited on 2019-11-22).

[31] *Wireapp/Mls-Client*. Wire Swiss GmbH. URL: https://github.com/wireapp/mls-client (visited on 2019-11-22).

[32] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. 2010-05. URL: https://rfc-editor.org/rfc/rfc5869.txt.

[33] Richard Barnes. *Messaging Layer Security - Past, Present, and Future*. EuroCrypt 2019. 2019-05-18. URL: https://people.cispa.io/cas.cremers/events/WSM2019/MLS_WSM%202019.pdf.

[34] *Specify an Init Message by Bifurcation · Pull Request #171 · Mlswg/Mls-Protocol*. URL: https://github.com/mlswg/mls-protocol/pull/171 (visited on 2019-12-04).

[35] Cullen Jennings, Daniel Burnett, Taylor Brandstetter, Adam Bergkvist, Anant Narayanan, Jan-Ivar Bruaroey, and Bernard Aboba. *WebRTC 1.0: Real-time Communication Between Browsers*. Candidate Recommendation. https://www.w3.org/TR/2018/CR-webrtc-20180927/. W3C, 2018-09.

[36] *Facetime Doesn't Face WebRTC*. 2015-06-09. URL: https://webrtchacks.com/facetime-decode/ (visited on 2019-12-10).

[37] *Is Slack's WebRTC Really Slacking? (Yoshimasa Iwase)*. 2016-03. URL: https://webrtchacks.com/slack-webrtc-slacking/ (visited on 2019-09-13).

[38] Cullen Jennings, Paul Jones, Richard Barnes, and Adam Roach. *SRTP Double Encryption Procedures*. Internet-Draft draft-ietf-perc-double-12. Work in Progress. Internet Engineering Task Force, 2019-08-29. URL: https://datatracker.ietf.org/doc/html/draft-ietf-perc-double-12.

[39] Colin Perkins, Magnus Westerlund, and Joerg Ott. *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. Internet-Draft draft-ietf-rtcweb-rtp-usage-26. Work in Progress. Internet Engineering Task Force, 2016-03. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtcweb-rtp-usage-26.

[40] *Video Calls for Signal Now in Public Beta*. Signal Messenger LLC. 2017-02-13. URL: https://signal.org/blog/signal-video-calls-beta/.

[41] *Mdn/Samples-Server*. MDN Web Docs. URL: https://github.com/mdn/samples-server/tree/master/s/webrtc-from-chat (visited on 2019-10-25).

[42] *Signalapp/Libsignal-Protocol-c*. Signal Messenger LLC. URL: https://github.com/signalapp/libsignal-protocol-c (visited on 2019-10-25).

[43] Brook Heisler. *Bheisler/Criterion.Rs*. URL: https://github.com/bheisler/criterion.rs (visited on 2019-11-25).

[44] *Google/Benchmark*. Google. URL: https://github.com/google/benchmark (visited on 2019-11-25).