Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Modular Design of the Messaging Layer Security (MLS) Protocol

Master Thesis

Tijana Klimovic

September 29, 2021

Advisors: Prof. Dr. Kenny Paterson, Dr. Igors Stepanovs

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

The Messaging Layer Security (MLS) protocol has been developed by the IETF group in an attempt to standardise an efficient and secure group messaging (SGM) protocol. The MLS protocol can be seen as a composition of three main cryptographic primitives: (1) a continuous group key agreement (CGKA) scheme, (2) a hash function used for maintaining user key-material entropy pools (PRF-PRNG), and (3) a key evolving group symmetric encryption (KEGSE) scheme. Presently, all papers focus on the analysis of CGKA and omit formally defining the other two modular primitives. Furthermore, no prior work has proposed any formal syntax, correctness, or security notion for group messaging schemes. This work aims to fill in the gaps in the formal treatment of the MLS protocol as follows. It defines the syntax, security and construction of the KEGSE building block, as well as a construction of the PRF-PRNG building block based on the MLS specification and architecture documents. In doing so, we define a novel security property we term as forward-secure (sender) anonymity which demands that even upon client compromise, the identity of the sender of past messages remains unknown. Moreover, we show that the MLS-based construction of KEGSE is itself a composition of two primitives: (1) a key-evolving symmetric authenticated encryption (KESE) scheme, representing a single-sender analogue of a KEGSE scheme and (2) a nonce-based authenticated encryption scheme with associated data (NAEAD). We then analyse KEGSE against our proposed security notion and prove that the current version of KEGSE does not provide forward-secure anonymity. We propose a new construction of KEGSE based on puncturable function families (PPRF), to address the insecurity. Moreover, we define a simplified security notion of KEGSE and argue that the current version of KEGSE is secure in this weaker security model. Our work also defines the syntax and correctness of SGM protocols as well as gives a description of the MLS protocol as a whole based on the MLS specification and architecture documents. Finally, we propose a list of potential future research directions in the area.

Contents

Contents iii

1 Introduction 1
1.1 Overview and Motivation 1
1.2 Contributions 2
1.3 Concurrent and Independent work 3
1.4 Outline 3

2 Preliminaries 5
2.1 Notation 5
2.2 Tree Terminology 8
2.3 Cryptographic primitives 10
2.3.1 Key Encapsulation Mechanism (KEM) 10
2.3.2 Key Derivation Function (KDF) 11
2.3.3 Nonce-based Authenticated Encryption with Associated Data (NAEAD) 11
2.3.4 Hash function 13
2.3.5 Digital Signature (DS) 13
2.3.6 Message Authentication Code (MAC) 13
2.3.7 Function families 14
2.3.8 Punctured function families 14

3 Secure Group messaging (SGM) protocol 17
3.1 SGM protocol functional requirements 17
3.2 SGM scheme 18
3.2.1 SGM scheme Syntax 19
3.2.2 Correctness of an SGM scheme 20
3.3 SGM protocol security requirements 22
3.4 Secure Messaging (SM) scheme requirements 23
3.4.1 Secure Messaging (SM) Syntax 23

3.4.2 Secure Messaging (SM) security 23
3.4.3 Modularisation of an SM scheme 24

4 The Messaging Layer Security (MLS) protocol 27
4.1 MLS Terminology 27
4.2 Assumptions and context of MLS 30
4.3 MLS Protocol overview 33
4.4 State of client 42
4.5 Ratchet Tree (RT) 46
4.5.1 Ratchet tree invariants 51
4.6 Group Context 54
4.7 Handshake and Application message plaintext 54
4.7.1 Handshake and Application plaintext 56
4.7.2 Content of a Proposal message 63
4.7.3 Content of a Commit message 68
4.8 Key Schedule 79
4.9 Secret Tree 80
4.10 Handshake and Application message framing 83
4.10.1 Metadata 84
4.10.2 Symmetric hash ratchet 84
4.10.3 Framing creation 85
4.10.4 Framing processing 91
4.11 Welcome message 93
4.11.1 Welcome message creation 95
4.11.2 Welcome message processing 97
4.12 Initialise group 100
4.13 Comparison to MLSv11 101
4.14 MLS protocol security 103

5 Building blocks of MLS 107
5.1 Modular construction of MLS 107
5.1.1 CGKA 108
5.1.2 PRF-PRNG 109
5.1.3 KEGSE 110
5.1.4 Inter-component data flow 110
5.2 PRF-PRNG 111
5.2.1 PRF-PRNG Syntax 111
5.2.2 Instantiating a PRF-PRNG scheme 111
5.3 Key-Evolving Group Symmetric Encryption Scheme (KEGSE) with Associated Data 112
5.3.1 KEGSE Syntax 113
5.3.2 KEGSE Correctness 114
5.3.3 KEGSE Security 115

5.3.4 Key-Evolving (Stateful) Symmetric Encryption Scheme with Associated Data (KESE) 120
5.3.5 KEGSE instantiation 137

6 Conclusions **143**
6.1 Summary 143
6.2 Future work 144

A Appendix **147**
A.1 KESE security definition motivation 147
A.2 FS-GAEAD definition motivation 149

Bibliography **151**

Chapter 1

Introduction

1.1 Overview and Motivation

Instant Messaging (IM) applications such as Whatsapp, Signal and Telegram have gained immense popularity in the recent years [Tan21b, Tan21a, Iqb17]. Naturally, with this increased use of IM technology, security protection of these applications became a pressing concern. Today, many of these applications make use of sophisticated messaging protocols, which can be viewed as asynchronous analogues of the TLS protocol that provide end-to-end guarantees against powerful attackers. Arguably, the most prominent of these solutions has been the Signal messaging protocol [Siga], which has been adopted by WhatsApp [Wha16], Facebook Messenger [Gre16], Wire [Gmb21], Skype [Lun18], Viber [Vib], Google Allo [Mar16] and many others.

The Signal protocol was initially designed, implemented and used in the Signal messenger and it can be roughly divided into two stages. The first stage is a key exchange, or X3DH (extended triple Diffie-Hellman) protocol [Sigb], which combines long-term, medium-term and ephemeral Diffie-Hellman keys to establish a shared secret key. This shared secret key is then further used in the second stage i.e. the Double ratchet protocol [PM16], which ensures that every message sent is encrypted and authenticated using a fresh symmetric key derived by ratcheting already used symmetric keys. These two stages of the Signal protocol were formalized and their security analyzed by researchers in academia, which resulted in a number of papers analysing the security of the X3DH protocol [BFG+19, CGCD+16, HKKP21] and the Double ratchet protocol in [ACD18, CGCD+16] proving that both stages of Signal are secure.

The caveat however is that the Double Ratchet protocol, and hence the security proofs, only focus on the two-party case, despite most messaging applications also supporting group conversations. For groups, most messengers

currently implement ad-hoc solutions that either do not provide end-to-end encryption (meaning the server sees all messages in plain), or are inefficient (e.g. an independent Signal channel is established between each pair of group members). Presently there is an ongoing effort by the IETF working group to develop and standardize both a secure and an efficient group messaging protocol, called Messaging Layer Security (MLS) protocol.

Alwen, Coretti and Dodis [ACD18] analyzed the Double Ratchet protocol in a modular way, by treating it as a composition of 3 cryptographic primitives: continuous key agreement scheme (CKA), hash function used for maintaining user key-material entropy pools (PRF-PRNG), and forward-secure authenticated encryption with associated data (FS-AEAD). This modularisation has been extended to the MLS protocol by Alwen, Coretti, Dodis and Tselekounis [ACDT19], and subsequently used in follow-up work by [ACC+19, ACJM20, AJM20] which focus on the analysis of continuous group key agreement scheme (CGKA) , the group version of CKA, and omit formally defining FS-AEAD and PRF-PRNG in the group setting. Moreover no prior work has yet proposed any formal syntax, correctness or security notion for group messaging schemes.

1.2 Contributions

We provide a brief summary of the modular composition of the bidirectional channel from CKA, PRF-PRNG and FS-AEAD covered in [ACD18]. Based on these ideas and the MLS architecture document [OBR+21], which states the security goals of MLS, we define the syntax and security of a key evolving group symmetric encryption (KEGSE) scheme and a PRF-PRNG scheme representing the missing FS-AEAD and PRF-PRNG primitive in the group context respectively.

In defining security of a KEGSE scheme we provide a formal definition of a novel property called forward-secure (sender) anonymity. Intuitively the property demands that upon client compromise the adversary gains no knowledge about the sender of a received message by the compromised client. We then instantiate these primitives based on the MLS protocol specification [BBM+21]. Subsequently we show an attack on the MLS-based KEGSE scheme instantiation and propose a fixed KEGSE scheme based on puncturable function families (PPRF)-s.

We then define a weaker KEGSE security game (excluding forward secure sender anonymity) and provide a modular construction of the MLS-based KEGSE scheme based on KESE schemes (the latter similar to FS-AEAD primitive used in [ACD18]) and nonce based authenticated encryption (NAE) schemes. Following this we show that the security definition of FS-AEAD schemes in [ACD18] is wrong, fix it and prove the MLS-based instantiation

of a KESE scheme secure in the fixed game. We then give a proof sketch of the MLS-based KEGSE scheme being secure in the weaker security game assuming the security of the underlying KESE scheme. Next to this we provide the syntax and correctness game of secure group messaging (SGM) schemes and give a description of the MLS protocol based on [OBR+21, BBM+21].

1.3 Concurrent and Independent work

We have recently become aware of one concurrent and independent work by Alwen, Coretti, Dodis and Tselekounis [ACDT21]. This work, like ours, defines the syntax and security of two primitives FS-GAEAD and PRF-PRNG, which correspond to the FS-AEAD and PRF-PRNG primitive in the two-user context respectively. Similarly to us they attempt to formally capture the syntax and correctness of an SGM protocol. They additionally define security of an SGM protocol and show how the MLS protocol can be modularly built from their CGKA, FS-GAEAD and PRF-PRNG primitives and analyse its security. However their security games only capture the 'basic' security requirements of an SGM protocol, i.e. confidentiality, authenticity, forward-security and post-comromise security of messages. Hence their work does not analyze MLS against the more advanced features such as forward secure sender anonymity, which we do in our work. Moreover their security notion of an FS-GAEAD scheme has a flaw that extends from the one we found in the security game of the FS-AEAD primitive in [ACD18]. The details of both of these definitional faults is elaborated further in Appendix A.1 and A.2.

1.4 Outline

The remainder of this work is organised as follows. Chapter 2 defines the notation, tree structure terminology and all cryptographic primitives we make use of throughout this work. In chapter 3 we define the syntax and correctness (in terms of a correctness game) of a secure group messaging (SGM) protocol and list the security properties it should provide. We then give a brief summary of the Double ratchet protocol modularization in [ACD18] whose ideas we attempt to extend to the group scenario. Chapter 4 gives a description of the MLS protocol and its assumptions about the context in which it executes based on [OBR+21] and [BBM+21]. In chapter 5 we cast MLS as an SGM protocol composed out of 3 main building blocks: continuous group key agreement (CGKA), key-evolving group symmetric encryption (KEGSE) scheme and a PRF-PRNG scheme; along with a MAC and DS scheme. We then go on to formalise both the KEGSE and PRF-PRNG components and analyse the security of the KEGSE building block. Finally, we conclude with a summary of our results and future work in Chapter 6.

Chapter 2

Preliminaries

2.1 Notation

Throughout this work we make use of the following notation and conventions:

Pseudocode basics. Every value (literal) has a data type. In this work we make use of the basic data types (integers, strings, booleans, bit-strings), collection data types (lists, tuples, maps, sets) and user defined data types (structs). The struct data type is essentially a collection of data elements (also called fields) grouped together under one user specified name. We use a special symbol $\perp$ to indicate an empty position in a list, map or struct; we also return it as an error code indicating an invalid input to an algorithm or an oracle, including invalid decryption. We use the $\leftarrow$ operator to deterministically assign the right operand to the left operand and $\leftarrow_{\$}$ for probabilistic assignment. We will use the shorthand notation $a_1, a_2, \ldots, a_n \leftarrow v$ (for some $n \in \mathbb{N}$ and some value v) to denote $a_1 \leftarrow v, a_2 \leftarrow v \ldots, a_n \leftarrow v$. The $=$ binary operator is used for comparison of equality. We will use $\mathsf{range}(a)$, where $a \in \mathbb{N}_0$ to denote the list $[0, 1, \ldots, a-1]$.

Basic data types. We say a is an integer if and only if $a \in \mathbb{Z}$, a string if and only if $a \in \{"a", \ldots, "z"\}^*$, a boolean if and only if $a \in \{\mathtt{true}, \mathtt{false}\}$ and a bit-string if and only if $a \in \{0,1\}^*$. If a is a string (bit-string) we write $|a|$ to denote the length of the string (bit-string) a, $a[i]$ to denote a's i-th letter (bit) and $a[i, \ldots, j]$ to denote a's substring starting at the i-th letter (bit) and ending with the j-th letter (bit) for $1 \leq i \leq j \leq |a|$. If a and b are strings (bit-strings) then $a \| b$ denotes the concatenation of a and b.

Lists. The list data type is used to store a sequence (ordered collection) of elements of same type. Let T be some data type. We say a is a list of type

T elements if and only if $a \in \{[e_1, e_2, \ldots, e_n] \mid n \in \mathbb{N}_0 \wedge \forall i(i \in \{1, \ldots, n\} : e_i = \bot \vee e_i \textit{ is } T \textit{ type})\}$. Each element in the list has a unique integer (called index) associated to it. A list supports adding, removing, random access and overwriting values of elements. Let A be a list containing elements of some type T. Then the syntax for accessing an element at index i is $A[i]$ and removing of an element at index i is done by assigning $\bot$ to it via $A[i] \leftarrow \bot$. Adding/Rewriting an element at index i with value v is done by $A[i] \leftarrow v$. Lists containing the same type elements can be concatenated via $+$ operator. Let T be some type. If A and B are two lists of type T elements such that $A = [a, b]$ and $B = [c, d]$ for a, b, c, d of T type then $A + B$ returns $[a, b, c, d]$. We denote $|A|$ to be the length of the list.

Tuples. Tuples are used to store a sequence (ordered collection) of elements with potentially different types. Let $T_1, \ldots, T_n$ be some data types and $n \in \mathbb{N}$. We say a is a tuple of type $T_1, \ldots, T_n$ elements if and only if $a \in \{(e_1, e_2, \ldots, e_n) \mid \forall i(i \in \{1, \ldots, n\} : e_i \textit{ is } T_i \textit{ type})\}$. The only operation tuples support is access of its elements. Namely if we have a tuple containing elements $(v_1, \ldots, v_4)$ of type $T_1, \ldots, T_4$, we use v_i to access the value of element v_i for some $i \in \{1, \ldots, 4\}$.

Sets. A set is used to store multiple elements of the same type without an order. A set ensures that each element is unique in value. Let T be some data type. We say a is a set of type T elements if and only if $a \in \{\{e_1, e_2, \ldots, e_n\} \mid n \in \mathbb{N}_0 \wedge \forall i(i \in \{1, \ldots, n\} : e_i \textit{ is } T \textit{ type})\}$. Two sets (containing elements of the same type) A and B can be unified using the binary $\cup$ operator and compared for membership containment using $\subseteq$. If A and B have at least one element with the same value, the $\cup$ operator will ensure that these repetitions do not occur in the set $A \cup B$. Sets also support addition of an element by unifying an existing set S with a set containing the single element we wish to add. Sets also support removing of elements and checking of membership. To remove a subset of elements e_1, e_2 from set $S = \{e_1, e_2, e_3\}$ we use the binary operator $\setminus$ and write $S \setminus R$ where $R = \{e_1, e_2\}$. To check for membership of an element in set S we use the $\in$ operator. If S is a finite set, we let $s \leftarrow\!\!\$\, S$ denote picking an element of S uniformly at random and assigning it to s. Moreover we will sometimes make use of the shorthand notation $S \xleftarrow{+} a$ and $S \xleftarrow{-} a$ to denote the adding and removing of element a from set S respectively.

Maps. Let T_1 and T_2 be some types. A map is used to store elements of type T_2 that have an associated key of type T_1. Let T_1, T_2 be some data type. We say a is a map from type T_1 to type T_2 elements if and only if $a \in \{\{k_1 : e_1, \ldots, k_n : e_n\} \mid n \in \mathbb{N}_0 \wedge \forall i(i \in \{1, \ldots, n\} : k_i \textit{ is } T_1 \textit{ type} \wedge (e_i = \bot \vee e_i \textit{ is } T_2 \textit{ type}))\}$. It supports the same operations as a list data type, i.e. addition, removal, overwriting and access of it's elements. The

only difference is that the operations use keys instead of indices to do these operations. Let A be a map from key types T_1 to value types T_2. Then we use the statement $A[.] \leftarrow y$ to assign to all keys of A the value y (of type T_2 or $\perp$). Moreover, A.keys returns a list of all keys (of type T_1) contained in A, and A.values returns a list of all values (of type T_2) contained in A.

Structs. Let $n \in \mathbb{N}$ and $T_1, \ldots T_n$ be some types. Structures allows us to give a name to a collection of n elements of type $T_1, \ldots T_n$ successively. We use the following syntax to define a user defined type:

```
struct label {
   f1
   f2
   f3
   ⋮
}
```

Where $\mathsf{f_i}$ is the name given to the i-th element of the `label` structure and `label` is some user specified name. The elements of any user defined structure are also called its fields. Let `label` be some structure with $n \in \mathbb{N}$ fields of type $T_1, \ldots, T_n$ respectively. We say a is of type `label` if and only if $a \in \{\texttt{label}(e_1, \ldots, e_n) \mid \forall i (i \in \{1, \ldots, n\} : e_i = \perp \vee e_i \textit{ is } T_i \textit{ type})\}$ where e_i corresponds to the i-th field's value for $i \in [n]$. Any user defined structure supports access and rewriting of its elements. Assume A is a `label` and f a field of `label`. Then we use $A.f$ to access field f of A and $A.f \leftarrow v$ to assign the value v to field f.

Algorithms and adversaries. Algorithms may be probabilistic or deterministic. If A is a probabilistic algorithm, we let $y \leftarrow A(x_1, \ldots; r)$ denote running A with random coins r on inputs $x_1, \ldots$ and assigning the output to y. We let $y \leftarrow\!\!\$\, A(x_1, \ldots)$ be the result of picking r at random and letting $y \leftarrow A(x_1, \ldots; r)$. If A is a deterministic algorithm, we let $y \leftarrow A(x_1, \ldots)$ denote running A on inputs $x_1, \ldots$ and assigning the output to y. Adversaries are algorithms.

Uniquely decodable encoding. We write $\langle a, b, \ldots \rangle$ to denote a bit-string that is a uniquely decodable encoding of $a, b, \ldots$ where each of the encoded elements can have an arbitrary type.

Random number generation. We use rand(n) a probabilistic algorithm that takes in an integer n and outputs a uniformly random and independent bit-string of length n bits.[1]

[1] The standard notation for $x \leftarrow\!\!\$\, \mathsf{rand(n)}$ is $x \leftarrow\!\!\$\, \{0,1\}^n$.

Keywords and wildcards. We use the keyword **req** followed by a proposition at the beginning of procedures in security games. If the proposition is false, then the procedure is exited with a return value $\perp$. We use the * symbol to act as a placeholder (wildcard) for any literal of a certain type.

Security games. We use the code based game playing framework to define security (and sometimes correctness) notions of schemes. We let $\Pr[\mathsf{G}]$ denote the probability that game G returns `true`. All parameters the game takes are specified as subscripts.

2.2 Tree Terminology

Here we give a brief overview of the tree terminology necessary for the Ratchet Tree (RT) data structure covered in Section 4.5 to be understood.

Node. A node (vertex) is a user defined structure.

Undirected graph. A undirected graph G is a pair of sets *(V,E)* where V is a finite set of nodes and E is a finite set of unoredered pairs of nodes in V. The set V is called the **vertex set** of G and E is the **edge set** of G and its elements (unordered pairs of nodes) are called **edges**. More concretely, an edge is a set $\{u, v\}$, where $u, v \in V$ and $u \neq v$. If $\{u, v\}$ is an edge in graph G then we say the edge $\{u, v\}$ is incident on nodes u and v.

Degree of node. The degree of a node (vertex) in an undirected graph is the number of edges incident on it.

Path. A path of length $k \geq 0$ from a vertex u to a vertex v in a undirected graph *G=(V,E)* is a list of nodes $[u_0, u_1, \ldots, u_k]$ such that $u = u_0$ and $v = u_k$, and $\{u_i, u_{i+1}\} \in E$ for all $i \in \{0, \ldots, k-1\}$. The length of the path is the number of edges in the path. We say that the path contains vertices $u_0, \ldots, u_k$ and edges $\{u_0, u_1\}, \{u_1, u_2\} \ldots \{u_{k-1}, u_k\}$. If there exists a path p from node u to v, we say that v is reachable from u via p. In an undirected graph a path exists from node u to v if and only if there exists a path from v to u. A path is simple if all vertices in the path are distinct. It should be noted that each node has a zero length path to itself, and hence any node is reachable from itself.

Cycle. In an undirected graph, a path $[u_0, u_1, \ldots, u_k]$ forms a cycle if $k \geq 3$ and $u_0 = u_k$; the cycle is **simple** if $u_0, u_1, \ldots, u_k$ are distinct. A graph with no cycles is **acyclic**.

Connected component. An undirected graph is connected if every vertex is reachable from all other vertices. The connected components of a graph are the equivalence classes of vertices under the "is reachable from" relation.

Tree. A connected, acyclic, undirected graph is a tree. In a tree any two vertices are connected by a unique simple path.

Rooted tree. A rooted tree is a tree in which one of the nodes is a special node labelled as the 'root' of tree. The root node is special in the sense that all other nodes in the tree are oriented below it in diagrams.

Ancestor and descendant. Let v be a node in a rooted tree G with root r. We call any node u on the unique simple path from r to x an ancestor of x. If u is an ancestor of node x then we say that x is a descendant of node u. Because every node is reachable from itself (0 length path) every node is both an ancestor and a descendant of itself. If u is an ancestor of x and $x \neq u$, then u is a proper ancestor of x and x is a proper descendant of u.

Lowest common ancestor. Let G be a rooted tree. The lowest common ancestor between two nodes v_0 and v_1, denoted as $\mathsf{LCA}(G, v_0, v_1)$ is defined as the lowest node in G that has both v_0 and v_1 as descendants.

Subtree. The subtree rooted at node v is the tree containing only descendants of v, rooted at v.

Parent, child, sibling, leaf and intermediate node. Let G be a rooted tree at node r and let $\{u, v\}$ be the last edge on the simple path from r to node v, then u is the parent of v and v is a child of u. The root node r is the only node in G with no parent. Node u and v are siblings if they have the same parent. A node is called a leaf if it has no children. If a node has both children and a parent it is called an intermediate node.

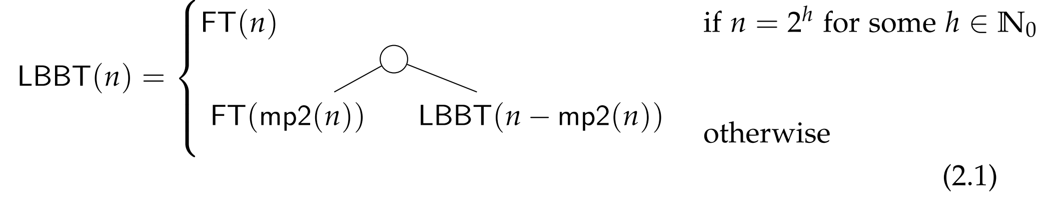
Depth, height and levels. Let G be a rooted tree at node r and u be a node in G. The length of the simple path from r to a node u is the depth of u in G. All nodes with the same depth form a level in G. The height of a node u in G is the number of edges on the longest simple downward path from u to a leaf. The height of a tree is the height of its root. The largest depth of any node in the tree is equal to the height of the tree.

Size of tree. The size of a tree is the number of leaf nodes it contains.

Binary tree. A binary tree is a rooted tree where each node has at most two children, a left child (drawn below and left from node) and a right child (drawn below and right from node). If a node has a left child, its left subtree is the subtree rooted at its left child. Similarly if a node has a right child its right subtree is the subtree rooted at its right child.

Full tree. Let $n \in \mathbb{N}$. Then a full tree with size n denoted as $\mathsf{FT}(n)$ is a binary tree whose height h is such that $n = 2^h$.

Left-balanced binary tree. Let $n \in \mathbb{N}$. A left-balanced binary tree with size n denoted as $\mathsf{LBBT}(n)$ is a binary tree defined over n number of leaves as follows:



(2.1)

where $\mathsf{mp2}(n) = \max\{2^k \mid 2^k \leq n\}$. There is a unique $\mathsf{LBBT}(n)$ for all $n \in \mathbb{N}$.

2.3 Cryptographic primitives

In this section we review all the cryptographic primitives we will make use of in this work.

2.3.1 Key Encapsulation Mechanism (KEM)

Definition 2.1 *A key encapsulation mechanism scheme* KEM *specifies algorithms* $\mathsf{KEM.KGen}$, $\mathsf{KEM.DeriveKeyPair}$, $\mathsf{KEM.Encap}$, $\mathsf{KEM.Decap}$, *where* $\mathsf{KEM.KGen}$, $\mathsf{KEM.Encap}$ *are probabilistic and* $\mathsf{KEM.DeriveKeyPair}$, $\mathsf{KEM.Decap}$ *are deterministic. Associated to* KEM *is a secret key length* $\mathsf{KEM.Nsk} \in \mathbb{N}$, *public key length* $\mathsf{KEM.Npk} \in \mathbb{N}$, *symmetric key length* $\mathsf{KEM.Nsym} \in \mathbb{N}$ *and the KEM ciphertext length* $\mathsf{KEM.Nenc} \in \mathbb{N}$. *The algorithms have the following syntax and semantics:*

- $\mathsf{KEM.KGen}$ algorithm generates a KEM secret key $\mathsf{sk} \in \{0,1\}^{\mathsf{KEM.Nsk}}$ and a KEM public key $\mathsf{pk}\{0,1\}^{\mathsf{KEM.Npk}}$ denoted as $(\mathsf{sk}, \mathsf{pk}) \leftarrow_\$ \mathsf{KEM.KGen}()$.
- $\mathsf{KEM.DeriveKeyPair}$ algorithm takes initial key material $\mathsf{ikm} \in \{0,1\}^{\mathsf{KEM.Nsk}}$ and generates a KEM secret key $\mathsf{sk} \in \{0,1\}^{\mathsf{KEM.Nsk}}$ and a KEM public key $\mathsf{pk}\{0,1\}^{\mathsf{KEM.Npk}}$ denoted as $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KEM.DeriveKeyPair}(\mathsf{ikm})$. Therefore $\mathsf{KEM.KGen}() = \mathsf{KEM.DeriveKeyPair}(\mathsf{rand}(\mathsf{KEM.Nsk}))$.
- Encap algorithm takes in a KEM public key $\mathsf{pk} \in \{0,1\}^{\mathsf{Npk}}$ and produces a KEM symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{KEM.Nsym}}$ and an encapsulation of that key $\mathsf{enc} \in \{0,1\}^{\mathsf{KEM.Nenc}}$ denoted as $(\mathsf{k}, \mathsf{enc}) \leftarrow \mathsf{KEM.Encap}(\mathsf{pk})$.
- Decap algorithm takes as input a secret key $\mathsf{sk} \in \{0,1\}^{\mathsf{KEM.Nsk}}$ and encapsulation $\mathsf{enc} \in \{0,1\}^{\mathsf{KEM.Nenc}}$ and produces some symmetric key $\mathsf{k} \in \{\perp\} \cup \{0,1\}^{\mathsf{KEM.Nsym}}$, denoted as $\mathsf{k} \leftarrow \mathsf{KEM.Decap}(\mathsf{sk}, \mathsf{enc})$.

Correctness Any Key Encapsulation Mechanism K must satisfy the following standard correctness property:

$$\Pr[(\mathsf{sk},\mathsf{pk}) \leftarrow\$ \mathsf{K.KGen}(), (\mathsf{k},\mathsf{enc}) \leftarrow\$ \mathsf{K.Encap}(\mathsf{pk}), \mathsf{k}' \leftarrow \mathsf{K.Decap}(\mathsf{sk},\mathsf{enc}) : \mathsf{k} = \mathsf{k}'] = 1$$

2.3.2 Key Derivation Function (KDF)

Definition 2.2 *A key derivation function* KDF *specifies a pair of deterministic algorithms* KDF.Extract *and* KDF.Expand*. Associated to* KDF *is a pseudorandom key length* $\mathsf{KDF.Nh} \in \mathbb{N}$*, a randomness set* $\mathsf{KDF}.\mathcal{R}$*, a salt set* $\mathsf{KDF}.\mathcal{S}$*.*

- KDF.Extract algorithm takes as input a randomness $\mathsf{r} \in \mathsf{KDF}.\mathcal{R}$ that comes from a distribution that is potentially non-uniform and a salt material $\mathsf{s} \in \mathsf{KDF}.\mathcal{S}$ and produces a pseudorandom key $\mathsf{prk} \in \{0,1\}^{\mathsf{KDF.Nh}}$ denoted as $\mathsf{prk} \leftarrow \mathsf{KDF.Extract}(\mathsf{r},\mathsf{s})$.
- KDF.Expand algorithm takes in a pseudorandom key $\mathsf{prk} \in \{0,1\}^{\mathsf{KDF.Nh}}$, a context $\mathsf{c} \in \{0,1\}^*$ and an integer L and produces a pseudorandom bit-string $\mathsf{k} \in \{0,1\}^{\mathsf{L}}$ denoted as $\mathsf{k} \leftarrow \mathsf{KDF.Expand}(\mathsf{prk},\mathsf{c},\mathsf{L})$.

2.3.3 Nonce-based Authenticated Encryption with Associated Data (NAEAD)

Definition 2.3 *A nonce-based authenticated encryption with associated data* NAE *specifies two deterministic algorithms* NAE.Enc, NAE.Dec*. Associated to* NAE *is a symmetric key length* $\mathsf{NAE.Nk} \in \mathbb{N}$*, a nonce length* $\mathsf{NAE.Nn} \in \mathbb{N}$ *and a ciphertext length function* $\mathsf{NAE.cl} : \mathbb{N} \to \mathbb{N}$*.*

- NAE.Enc takes a symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{NAE.Nk}}$, a nonce $\mathsf{n} \in \{0,1\}^{\mathsf{NAE.Nn}}$, associated data $\mathsf{ad} \in \{0,1\}^*$ and a message $\mathsf{m} \in \{0,1\}^*$ and produces a ciphertext $\mathsf{c} \in \{0,1\}^{\mathsf{NAE.cl}(|\mathsf{m}|)}$ denoted as $\mathsf{c} \leftarrow \mathsf{NAE.Enc}(\mathsf{k},\mathsf{n},\mathsf{ad},\mathsf{m})$.
- NAE.Dec algorithm takes in a symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{NAE.Nk}}$, a nonce $\mathsf{n} \in \{0,1\}^{\mathsf{NAE.Nn}}$, associated data $\mathsf{ad} \in \{0,1\}^*$, a ciphertext c and outputs a message $\mathsf{m} \in \{0,1\}^* \cup \{\perp\}$ denoted as $\mathsf{m} \leftarrow \mathsf{NAE.Dec}(\mathsf{k},\mathsf{n},\mathsf{ad},\mathsf{c})$.

Correctness An nonce-based authenticated encryption with associated data scheme N must satisfy the following standard correctness property for all keys $\mathsf{k} \in \{0,1\}^{\mathsf{N.Nk}}$ nonces $\mathsf{n} \in \{0,1\}^{\mathsf{N.Nn}}$ all associated data $\mathsf{ad} \in \{0,1\}^*$ and all messages $\mathsf{m} \in \{0,1\}^*$:

$$\Pr[\mathsf{c} \leftarrow \mathsf{N.Enc}(\mathsf{k},\mathsf{n},\mathsf{ad},\mathsf{m}), \mathsf{m}' \leftarrow \mathsf{N.Dec}(\mathsf{k},\mathsf{n},\mathsf{ad},\mathsf{c}) : \mathsf{m} = \mathsf{m}'] = 1$$

Consider game $\mathsf{G}^{\mathsf{mae}}$ in Figure 2.1 associated to a nonce-based authenticated encryption scheme with associated data NAE and an adversary $\mathcal{A}$. It extends the definition of authenticated encryption to a multi-user setting (multi-key setting), similarly to [BSJ$^+$16]. The advantage of $\mathcal{A}$ in breaking

the multi-user authenticated encryption (MAE) security of NAE is defined as $\mathsf{Adv}^{\mathsf{mae}}_{\mathsf{NAE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{mae}}_{\mathsf{NAE},\mathcal{A}}\right] - 1$.

The game starts by sampling a challenge bit b and initialises all variables needed to keep track of adversary $\mathcal{A}$ not winning the game trivially. It then asks the adversary $\mathcal{A}$ to guess the value of the challenge bit whilst granting it access to oracles New, Corr, Enc and Dec. The adversary can increase the number of users (and keys) by calling oracle New, which generates a new user key. For any user key, $\mathcal{A}$ is allowed to ask for either message m_0 or m_1 to be encrypted by calling oracle Enc. The Dec oracle allows the adversary to obtain decryptions of ciphertexts. In the real world ($b = 1$) oracle Dec returns the correct decryption and in the random world ($b = 0$) Dec returns the incorrect decryption $\bot$. The adversary $\mathcal{A}$ is also allowed to leak the key of a user if it was not used to create a challenge ciphertext (Enc was called on $m_0 \neq m_1$). To avoid trivial attacks $\mathcal{A}$ is not allowed to call Enc on $m_0 \neq m_1$ if the key of user i was leaked. Likewise, $\mathcal{A}$ is allowed to call Enc only once for every unique user-nonce pair (i, n). The adversary $\mathcal{A}$ is also not allowed to query Dec with ciphertexts that were derived by Enc.

```
G^mae_{NAE,A}:
   b ←$ {0,1}
   ctr ← 0
   safe[.] ← true
   comp[.] ← false
   U ← ∅
   trans ← ∅
   b' ←$ A^{New,Enc,Dec,Corr}
   return b' = b

New():
   keys[ctr] ←$ {0,1}^{NAE.Nk}
   ctr ← ctr + 1

Corr(i):
   req safe[i]
   comp[i] ← true
   return keys[i]

Enc(i, n, m0, m1, ad) :
   req 0 ≤ i < ctr and (i,n) ∉ U
   if comp[i] and m0 ≠ m1:
      return ⊥
   c ← NAE.Enc(keys[i], n, ad, m_b)
   U ← U ∪ {(i,n)}
   if m0 ≠ m1:
      safe[i] ← false
   trans ← trans ∪ {(i, n, c, ad)}
   return c

Dec(i,n,c,ad):
   req 0 ≤ i < ctr and ¬comp[i]
   if (i, n, c, ad) ∈ trans:
      return ⊥
   m ← NAE.Dec(keys[i], n, ad, c)
   if b=1:
      return m
   else:
      return ⊥
```

Figure 2.1: Game $\mathsf{G}^{\mathsf{mae}}$ defining the MAE security of a nonce-based authenticated encryption with associated data NAE.

Definition 2.4 *A nonce-based authenticated encryption with associated data* NAE *is (ϵ, t)-MAE-secure if for all t-attackers $\mathcal{A}$:*

$$\mathsf{Adv}^{\mathsf{mae}}_{\mathsf{NAE}}(\mathcal{A}) \leq \epsilon$$

where an attacker is parametrised over its running time t.

2.3.4 Hash function

Definition 2.5 *A hash function* H *specifies a deterministic algorithm* $\mathsf{H.Ev}$ *and has an associated digest length* $\mathsf{H.Nd} \in \mathbb{N}$.

- $\mathsf{H.Ev}$ algorithm takes as input a message $\mathsf{m} \in \{0,1\}^*$ and outputs the message's digest $\mathsf{d} \in \{0,1\}^{\mathsf{H.Nd}}$ denoted as $\mathsf{d} \leftarrow \mathsf{H.Ev}(\mathsf{m})$.

2.3.5 Digital Signature (DS)

Definition 2.6 *A digital signature scheme* DS *specifies algorithms* $\mathsf{DS.KGen}$, $\mathsf{DS.Sign}$ *and* $\mathsf{DS.Vfy}$ *where* $\mathsf{DS.KGen}$ *is probabilistic and* $\mathsf{DS.Sign}$ *and* $\mathsf{DS.Vfy}$ *are deterministic algorithms. Associated to* DS *is the secret signing key length* $\mathsf{DS.Nsk} \in \mathbb{N}$, *verification key length* $\mathsf{DS.Npk} \in \mathbb{N}$ *and the signature length* $\mathsf{DS.Nsign} \in \mathbb{N}$.

- $\mathsf{DS.Kgen}$ algorithm produces a signing key $\mathsf{tk} \in \{0,1\}^{\mathsf{DS.Nsk}}$ and a verification key $\mathsf{vk} \in \{0,1\}^{\mathsf{DS.Npk}}$, denoted as $(\mathsf{tk}, \mathsf{vk}) \leftarrow_{\$} \mathsf{DS.KGen}()$.
- $\mathsf{DS.Sign}$ algorithm takes as input a secret signing key $\mathsf{sk} \in \{0,1\}^{\mathsf{DS.Nsk}}$ and a message $\mathsf{m} \in \{0,1\}^*$ and produces a signature $\mathsf{s} \in \{0,1\}^{\mathsf{DS.Nsign}}$ over m, denoted as $\mathsf{s} \leftarrow \mathsf{DS.Sign}(\mathsf{tk}, \mathsf{m})$.
- $\mathsf{DS.Vfy}$ algorithm takes as input a verification key $\mathsf{vk} \in \{0,1\}^{\mathsf{DS.Npk}}$ a message $\mathsf{m} \in \{0,1\}^*$ and a signature $\mathsf{s} \in \{0,1\}^{\mathsf{DS.Nsign}}$ and outputs a boolean value b denoted as $\mathsf{b} \leftarrow \mathsf{DS.Vfy}(\mathsf{vk}, \mathsf{m}, \mathsf{s})$.

Correctness A DS scheme DS must satisfy the following standard correctness property for all messages $\mathsf{m} \in \{0,1\}^*$:

$$\Pr[(\mathsf{tk}, \mathsf{vk}) \leftarrow_{\$} \mathsf{DS.KGen}(), \mathsf{s} \leftarrow_{\$} \mathsf{DS.Sign}(\mathsf{tk}, \mathsf{m}), \mathsf{b} \leftarrow \mathsf{DS.Vfy}(\mathsf{vk}, \mathsf{m}, \mathsf{s}) : \mathsf{b} = \mathtt{true}] = 1$$

2.3.6 Message Authentication Code (MAC)

Definition 2.7 *A message authentication code scheme* M *specifies algorithms* $\mathsf{M.KGen}$, $\mathsf{M.Mac}$, $\mathsf{M.Vfy}$, *where* $\mathsf{M.KGen}$, $\mathsf{M.Mac}$ *are probabilistic and* $\mathsf{M.Vfy}$ *is deterministic. Associated to* M *is the symmetric key length* $\mathsf{M.Nk} \in \mathbb{N}$ *and the tag length* $\mathsf{M.Nt} \in \mathbb{N}$.

- $\mathsf{M.KGen}$ produces a symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{M.Nk}}$ denoted as $\mathsf{k} \leftarrow_{\$} \mathsf{M.KGen}()$.
- $\mathsf{M.Mac}$ takes a symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{M.Nk}}$ and a message $\mathsf{m} \in \{0,1\}^*$ and produces a tag $\mathsf{t} \in \{0,1\}^{\mathsf{M.Nt}}$ denoted as $\mathsf{t} \leftarrow_{\$} \mathsf{M.Mac}(\mathsf{k}, \mathsf{m})$.
- $\mathsf{M.Vfy}$ takes a symmetric key $\mathsf{k} \in \{0,1\}^{\mathsf{M.Nk}}$, a message $\mathsf{m} \in \{0,1\}^*$ and a tag $\mathsf{t} \in \{0,1\}^{\mathsf{M.Nt}}$ and outputs a boolean value b denoted as $\mathsf{b} \leftarrow \mathsf{M.Vfy}(\mathsf{k}, \mathsf{m}, \mathsf{t})$.

Correctness A MAC scheme M must satisfy the following standard correctness property for all messages $\mathsf{m} \in \{0,1\}^*$:

$$\Pr[\mathsf{k} \leftarrow\$ \mathsf{M.KGen}(), \mathsf{t} \leftarrow\$ \mathsf{M.Mac}(\mathsf{k}, \mathsf{m}), \mathsf{b} \leftarrow \mathsf{M.Vfy}(\mathsf{k}, \mathsf{m}, \mathsf{t}) : \mathsf{b} = \texttt{true}] = 1$$

2.3.7 Function families

Definition 2.8 *A family of functions* F *specifies a deterministic algorithm* $\mathsf{F.Ev}$*. Associated to* F *is a key bit-length* $\mathsf{F.Kl} \in \mathbb{N}$ *and an output bit-length* $\mathsf{F.Out} \in \mathbb{N}$*.*

- The evaluation algorithm $\mathsf{F.Ev}$ takes in a function key $\mathsf{k} \in \{0,1\}^{\mathsf{F.Kl}}$ and an input $x \in \{0,1\}^*$ and returns an output $y \in \{0,1\}^{\mathsf{F.Out}}$ denoted as $y \leftarrow \mathsf{F.Ev}(\mathsf{k}, x)$.

Consider game $\mathsf{G}^{\mathsf{prf}}$ in Figure 2.2 associated to a function family F and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the PRF security of F is defined as $\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{F}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{prf}}_{\mathsf{F},\mathcal{A}}\right] - 1$.

$\mathsf{G}^{\mathsf{prf}}_{\mathsf{F},\mathcal{A}}$:	Comp(x):

$\mathsf{b} \leftarrow\$ \{0,1\}$
$\mathsf{k} \leftarrow\$ \{0,1\}^{\mathsf{F.Kl}}$
$\mathsf{f}[.] \leftarrow \bot$
$\mathsf{b}' \leftarrow\$ \mathcal{A}^{\mathsf{Comp}}$
return $\mathsf{b}' = \mathsf{b}$
	if b=1:
	  $\mathsf{y} \leftarrow \mathsf{F.Ev}(\mathsf{k},\mathsf{x})$
	else:
	  if $\mathsf{f}[\mathsf{x}] = \bot$:
	    $\mathsf{y} \leftarrow\$ \{0,1\}^{\mathsf{F.Out}}$
	    $\mathsf{f}[\mathsf{x}] \leftarrow \mathsf{y}$
	  else:
	    $\mathsf{y} \leftarrow \mathsf{f}[\mathsf{x}]$
	return y

Figure 2.2: Game $\mathsf{G}^{\mathsf{prf}}$ defining the PRF security of a function family F.

Definition 2.9 *A function family* F *is* (ϵ, t)*-PRF-secure if for all t-attackers* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{F}}(\mathcal{A}) \leq \epsilon$$

where an attacker is parametrised over its running time t.

2.3.8 Punctured function families

Definition 2.10 *A punctured family of functions* G *specifies deterministic algorithms* $\mathsf{G.Ev}$ *and* $\mathsf{G.Punct}$*. Associated to* G *is a key bit-length* $\mathsf{G.Kl} \in \mathbb{N}$*, an input bit-length* $\mathsf{G.In}$ *and an output bit-length* $\mathsf{G.Out} \in \mathbb{N}$*.*

- The evaluation algorithm $\mathsf{G.Ev}$ takes in a function key $\mathsf{k} \in \{0,1\}^{\mathsf{G.Kl}}$ and an input $x \in \{0,1\}^{\mathsf{G.In}}$ and returns an output $\mathsf{y} \in \{0,1\}^{\mathsf{G.Out}} \cup \{\bot\}$ denoted as $y \leftarrow \mathsf{G.Ev}(\mathsf{k}, x)$.
- The puncturing algorithm $\mathsf{G.Punct}$ takes in a function key $\mathsf{k} \in \{0,1\}^{\mathsf{G.Kl}}$ and an input $x \in \{0,1\}^{\mathsf{G.In}}$ and returns a punctured key $\mathsf{k}' \in \{0,1\}^{\mathsf{G.Kl}}$ denoted as $\mathsf{k}' \leftarrow \mathsf{G.Punct}(\mathsf{k}, x)$.

Correctness We make use of the correctness definition provided by [BDdK+21]. A punctured function family G is correct if for all t and $x \in \{0,1\}^{\mathsf{G.In}} \setminus \{x_1, \ldots, x_t\}$ it holds that:

$$\Pr\left[\mathsf{G.Ev}(\mathsf{k_0}, x) = \mathsf{G.Ev}(\mathsf{k_t}, x) : \mathsf{k_0} \leftarrow\$ \{0,1\}^{\mathsf{G.Kl}}, \mathsf{k_i} = \mathsf{G.Punct}(\mathsf{k_{i-1}}, x_\mathsf{i}) \text{ for } \mathsf{i} \in \{1, \ldots, \mathsf{t}\}\right] = 1$$

Consider game $\mathsf{G}^{\mathsf{pprf}}$ in Figure 2.3 associated to a punctured function family G and an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the PPRF security of G is defined as $\mathsf{Adv}^{\mathsf{pprf}}_{\mathsf{G}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{pprf}}_{\mathsf{G},\mathcal{A}}\right] - 1$. We use the security definition provided in [BST13]. The game starts by sampling a challenge bit b and a key k. It then asks the adversary $\mathcal{A}$ to guess the value of the challenge bit whilst granting it access to a Chall oracle. The adversary queries the Chall oracle on x and obtains a punctured key k'. The adversary $\mathcal{A}$ interacts with Chall by picking an input value x and gets back the corresponding punctured key together with a challenge for the value of $\mathsf{G.Ev(x)}$.

```
G^pprf_{G,A}:                 Chall(x):
  b ←$ {0,1}                    k' ← G.Punct(k, x)
  k ←$ {0,1}^{G.Kl}             if b=1:
  b' ←$ A^Chall                   y ← G.Ev(k, x)
  return b' = b                 else:
                                  y ←$ {0,1}^{G.Out}
                                return (k', y)
```

Figure 2.3: Game $\mathsf{G}^{\mathsf{pprf}}$ defining the PPRF security of a punctured function family G.

Definition 2.11 *A punctured function family* G *is* (ϵ, t)*-PPRF-secure if for all t-attackers* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\mathsf{pprf}}_{\mathsf{G}}(\mathcal{A}) \leq \epsilon$$

where an attacker is parametrised over its running time t.

Chapter 3

Secure Group messaging (SGM) protocol

This chapter starts by giving an informal overview of the functional requirements a secure group messaging (SGM) protocol should meet. We then explain how those requirements translate into SGM scheme syntax and informally describe SGM protocol security requirements. Lastly we discuss how the functional and security requirements and syntax simplify when we constrain the protocol to only allow groups containing the same two members at all times. This kind of simplification is called a Secure Messaging (SM) scheme and has been studied by Alwen et al. [ACD18]. We give a brief summary of the main ideas of that work at the end of this chapter.

3.1 SGM protocol functional requirements

An SGM protocol allows any client A to do the following operations:

1. Create a group by inviting a set of other clients.
2. Add a client to a group client A is a member of.
3. Remove a member from a group client A is a member of.
4. Update its own key material.
5. Send an IM application message to everyone in a group client A is a member of.
6. Receive an IM application message from any member of a group client A is a member of.

The first 4 operations are **group operations**, as they are responsible for the changing of the group membership. All operations must be executable in

the asynchronous setting, that is no operation requires two distinct clients to be online simultaneously.

An SGM protocol makes use of an SGM scheme that defines all algorithms a client running an SGM protocol must implement for both functional and security requirements (see section 3.3) to be satisfied.

3.2 SGM scheme

In an SGM scheme, each client is identified by a unique ID from some set SGM.ID and each group by a unique group identifier GID. Prior to defining the syntax of an SGM scheme, we first motivate the choice of algorithms an SGM scheme specifies. An SGM scheme must specify an initialization algorithm that each client can use at the very beginning of the SGM protocol to initialize their state. Each client's state consists of two types of information known to the client: **client data** i.e. information specific to that client and (if the client is a member of some group) **group data** which is information specific to the group the client is a member of.

Given some group G, no member of G will know all information pertaining to it. This separation of knowledge is done in order to facilitate more efficient healing in case of client state compromise. The union of all group members' group data then forms all information there is pertaining to a group, a so called 'group state'. Note that the information contained in one members group data may also be present in another member's group data. Throughout the lifetime of a group the group state will change whenever one of its member's modifies their group data. Of course since this group state is in reality distributed across the group members' states, the members need to ensure that any group data modified locally is propagated to the rest of the group. Namely if members a and b with group data gd_a and gd_b respectively are such that $gd_a \cap gd_b \neq \emptyset$ then any modification a makes to information contained in $gd_a \cap gd_b$ needs to also be propagated to b and vice versa.

With an initialised state (containing only client data at first[1]), each client has all the necessary values to form a group with other clients. Hence an SGM scheme specifies a group creation algorithm that will change the state of the group creator (client running the group creation algorithm) to reflect the formation of a group. In order for other members of this group to be aware of its formation and have all the necessary group data to participate in group conversations, the group creation algorithm also outputs welcome messages (W) for each new member. The SGM scheme also needs to specify

[1] A client that runs the initialising algorithm just joined the system and hence is not a member of any group and therefore its group data is empty.

a processing algorithm that a client can run upon receiving a welcome message in order to extract the group data. Once a group is formed, a group member may wish to send an IM application message (A). To facilitate this, an SGM scheme needs to specify a send algorithm and consequently a receive algorithm, which will enable the other group members to process IM application messages (A). Because a group member may wish to remove some group member a and client a itself may wish to perform an update on its key material, the group members will need to make a choice as to which of these proposed operations will come into effect. To enable this, an SGM scheme specifies algorithms for proposing an add, remove or update operation that produce a proposal message (P) and an algorithm to commit the group to a set of proposals. Similarly to all the other SGM algorithms, the commit algorithm is run by some group member not proposed for removal. The commit algorithm will cause a change to the group data maintained by the caller of the commit to reflect the chosen proposals. Hence just as in the case of the group creation algorithm, the commit algorithm also needs to produce a commit message (T) for members who were already part of the group and possibly welcome messages (W) in case any add operations were chosen to come into effect. The time frame within which the group members use the same group data to perform sends and receives of application data is called an epoch. Each epoch has a so called epoch number associated to it, which is equal to the number of commits processed since the group's creation until this epoch started. Within each epoch, each group member also has their own local message counter, responsible for counting the number of messages sent by this group member within the epoch.

3.2.1 SGM scheme Syntax

Definition 3.1 *A secure group messaging scheme* SGM *specifies ten algorithms,* $\mathsf{SGM.Init}$, $\mathsf{SGM.Create}$, $\mathsf{SGM.Propose\text{-}Add}$, $\mathsf{SGM.Propose\text{-}Remove}$, $\mathsf{SGM.Propose\text{-}Update}$, $\mathsf{SGM.Commit}$, $\mathsf{SGM.Process\text{-}Commit}$, $\mathsf{SGM.Process\text{-}Welcome}$, $\mathsf{SGM.Send}$ *and finally* $\mathsf{SGM.Receive}$*. The* $\mathsf{SGM.Process\text{-}Commit}$, $\mathsf{SGM.Process\text{-}Welcome}$ *and* $\mathsf{SGM.Receive}$ *algorithms are deterministic, the remainder is probabilistic. Associated to SGM is a set of client identifiers* $\mathsf{SGM.ID}$*. The algorithms have the following syntax and semantics:*

- $\mathsf{SGM.Init}$ algorithm takes in an identity $\mathsf{ID} \in \mathsf{SGM.ID}$ and outputs the client's initial state s denoted as $\mathsf{s} \leftarrow_{\$} \mathsf{SGM.Init}(\mathsf{ID})$.
- $\mathsf{SGM.Create}$ algorithm takes in a client's state s and a list of identities of the members to be added to the group $\mathsf{IDs} \subseteq \mathsf{SGM.ID}$ and a group identifier GID, and outputs a new state s' and a list of welcome messages Ws denoted as $(\mathsf{s}', \mathsf{Ws}) \leftarrow_{\$} \mathsf{SGM.Create}(\mathsf{s}, \mathsf{IDs}, \mathsf{GID})$.
- $\mathsf{SGM.Propose\text{-}Add}$ algorithm takes as input a state s and an identifier

$\mathsf{ID} \in \mathsf{SGM.ID}$ and outputs a new state s' and a proposal message P denoted as $(\mathsf{s}', \mathsf{P}) \leftarrow\!\!\$\ \mathsf{SGM.Propose\text{-}Add}(\mathsf{s}, \mathsf{ID})$.

- $\mathsf{SGM.Propose\text{-}Remove}$ algorithm takes as input a state s and an identifier $\mathsf{ID} \in \mathsf{SGM.ID}$ and outputs a new state s' and a proposal message P denoted as $(\mathsf{s}', \mathsf{P}) \leftarrow\!\!\$\ \mathsf{SGM.Propose\text{-}Remove}(\mathsf{s}, \mathsf{ID})$.
- $\mathsf{SGM.Propose\text{-}Update}$ algorithm takes in a state s and outputs a new state s' and a proposal message P denoted as $(\mathsf{s}', \mathsf{P}) \leftarrow\!\!\$\ \mathsf{SGM.Propose\text{-}Update}(\mathsf{s})$.
- $\mathsf{SGM.Commit}$ algorithm takes in a state s and a list of proposal messages Ps and produces a new state s', a commit message T and a list of client identity and welcome message pairs corresponding to each client in a committed add proposal IDWs denoted as $(\mathsf{s}', \mathsf{T}, \mathsf{IDWs}) \leftarrow\!\!\$\ \mathsf{SGM.Commit}(\mathsf{s}, \mathsf{Ps})$.
- $\mathsf{SGM.Process\text{-}Commit}$ algorithm given a state s and a commit message T outputs a new state s' and a list of group members after the commit message is applied G, denoted as $(\mathsf{s}', \mathsf{G}) \leftarrow \mathsf{SGM.Process\text{-}Commit}(\mathsf{s}, \mathsf{T})$.
- $\mathsf{SGM.Process\text{-}Welcome}$ algorithm given a state s and welcome message W outputs a new state s' and a list of group members after the commit message is applied G, denoted as $(\mathsf{s}', \mathsf{G}) \leftarrow \mathsf{SGM.Process\text{-}Welcome}(\mathsf{s}, \mathsf{W})$.
- $\mathsf{SGM.Send}$ algorithm takes in a state s a message m and associated data ad and outputs a new state s' and an application message A denoted as $(\mathsf{s}', \mathsf{A}) \leftarrow\!\!\$\ \mathsf{SGM.Send}(\mathsf{s}, \mathsf{m}, \mathsf{ad})$.
- $\mathsf{SGM.Receive}$ algorithm takes in a state s an application message A and produces a new state s', an epoch number e, an integer i and a message m denoted as $(\mathsf{s}', \mathsf{e}, \mathsf{i}, \mathsf{m}) \leftarrow \mathsf{SGM.Receive}(\mathsf{s}, \mathsf{A})$.

3.2.2 Correctness of an SGM scheme

Informally an SGM scheme is correct if for any group and any application message A sent to the group (by some group member), all members who receive A can immediately decrypt it no matter the order of delivery. The correctness of an SGM scheme is formally captured by the correctness game $\mathsf{G}^{\mathsf{sgm\text{-}corr}}_{\mathsf{SGM}}$ given in Figure 3.1. The game starts by calling $\mathcal{A}_1$ (adversary is seen as a pair of algorithms $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$) to obtain the identifiers of the clients forming the group IDs and checks that no client is repeated twice.

Then for each client in IDs the game initialises an epoch ep, group operation message gop and application message app counter recording the epoch the client is currently in, the number of group messages (P,T,W type) and the number of application messages (type A) sent by client in epoch ep respectively. For each client the game also initialises the set of proposal messages recieved thus far in the current epoch ep. It also maintains maps $\mathsf{trans\text{-}gop}$ and $\mathsf{trans\text{-}app}$ that record all group operation messages and application messages in transmission respectively.

```
G^{sgm-corr}_{SGM,A}:
  (A1, A2) ← A
  (IDs, state) ←$ A1()
  if ∃i, j ∈ range(|IDs|) : i ≠ j ∧ IDs[i] = IDs[j]:
    return false
  for id in IDs:
    st[id] ← SGM.Init(id)
    ep[id], gop[id], app[id] ← 0
    props[id] ← ⊥
  trans-gop[.] ← ⊥
  trans-app[.] ← ⊥
  G[.], lead[.] ← ⊥
  start ← true
  win ← false
  oracles ← {create-group, add-user, remove-user,
  send-update, send-commit, deliver-group-ops,
  send-app-data, deliver-app-data}
  A2^oracles(state)
  return win

create-group(ID0, IDs, GID):
  req ep[ID0] = 0 ∧ start
  start ← false
  (st[ID0], Ws) ←$ SGM.Create(st[ID0], IDs, GID)
  for i in range(|IDs|):
    trans-gop[ID0, IDs[i], ep[ID0], gop[ID0], WEL] ← Ws[i]
  ep[ID0] + +
  G[ep[ID0]] ← [ID0] + IDs
  return Ws

add-user(ID, ID′):
  req ep[ID] > 0 ∧ ID ∈ G[ep[ID]] ∧ ID′ ∉ G[ep[ID]]
  (st[ID],P) ←$ SGM.Propose-Add(st[ID],ID′)
  for id in G[ep[ID]]:
    trans-gop[ID, id, ep[ID], gop[ID], ADD] ← P
  gop[ID] + +
  return P

remove-user(ID, ID′):
  req ep[ID] > 0 ∧ {ID, ID′} ⊆ G[ep[ID]]
  (st[ID],P) ←$ SGM.Propose-Remove(st[ID],ID′)
  for id in G[ep[ID]]:
    trans-gop[ID, id, ep[ID], gop[ID], RMV] ← P
  gop[ID] + +
  return P

send-update(ID):
  req ep[ID] > 0 ∧ ID ∈ G[ep[ID]]
  (st[ID],P) ←$ SGM.Propose-Update(st[ID])
  for id in G[ep[ID]]:
    trans-gop[ID, id, ep[ID], gop[ID], UPD] ← P
  gop[ID] + +
  return P

send-commit(ID):
  req ep[ID] > 0 ∧ ID ∈ G[ep[ID]]
  (st[ID], T, IDWs) ←$ SGM.Commit(st[ID], props[ID])
  for id in G[ep[ID]]:
    trans-gop[ID, id, ep[ID], gop[ID], CMT] ← T
  for i in IDWs:
  (id, W) ← IDWs[i]
    trans-gop[ID, id, ep[ID], gop[ID], WEL] ← W
  gop[ID] + +
  return (T, IDWs)

deliver-group-ops(ID, ID′, e, ctr, flag):
  req trans-gop[ID, ID′, e, ctr, flag] ≠ ⊥∧
  (e = ep[ID′] ∨ ID′ ∉ G[e])∧
  (flag ∈ {CMT, WEL} ⟹ lead[e] ∈ {⊥, (ID, ctr)})
  M ← trans-gop[ID, ID′, e, ctr, flag]
  if flag = CMT:
    (st[ID′], G) ← SGM.Process-Commit(st[ID′], M)
    ep[ID′] + +
  if flag ∈ {ADD, RMV, UPD}:
    props[ID′] ← props[ID′] + M
  if flag = WEL:
    (st[ID′], G) ← SGM.Process-Welcome(st[ID′], M)
    ep[ID′] = e + 1
  if flag = CMT or flag = WEL:
    if lead[e] = ⊥:
      lead[e] ← (ID,ctr)
      G[e+1] ← G
    props[ID′] ← ⊥
    gop[ID′], app[ID′] ← 0
  trans-gop[ID, ID′, e, ctr, flag] ← ⊥

send-app-data(ID, ad, m):
  req ep[ID] > 0
  (st[ID], A) ←$ SGM.Send(st[ID], m, ad)
  for id in G[ep[ID]]:
    trans-app[ID, id, ep[ID], app[ID], m, ad] ← A
  app[ID] + +
  return A

deliver-app-data(ID, ID′, e, ctr, m, ad):
  req trans-app[ID, ID′, e, ctr, m, ad] ≠ ⊥
  A ← trans-app[ID, ID′, e, ctr, m, ad]
  (st[ID′], e′, ctr′, m) ← SGM.Receive(st[ID′], A)
  if (e′, ctr′, m′) ≠ (e, ctr, m):
    win ← true
  trans-app[ID, ID′, e, ctr, m, add] ← ⊥
```

Figure 3.1: The correctness game $G^{\text{sgm-corr}}$ associated to a secure group messaging scheme SGM and an attacker $\mathcal{A}$.

Moreover for each epoch, the G and lead map record the group and leading commit for a given epoch respectively. Finally the start boolean ensures that an adversary is limited to creating at most one group, whilst win denotes whether an attacker won the game. The game then challenges $\mathcal{A}_2$ to set the win flag to `true` whilst being allowed to interact with the **create-group**, **add-user**, **remove-user**, **send-update**, **send-commit**, **deliver-group-ops**, **send-app-data** and **deliver-app-data** oracles which allow it to control the lifetime of the group.

Note that the attacker is forced to call the **create-group** oracle first and is allowed to manipulate only one group throughout a run of the game. This is ensured by the **req** at the begging of the **create-group**. Therefore the correctness game provided here only talks about the correctness of an SGM scheme with a single group. The correctness of an SGM scheme with multiple groups easily extends from this, at the expense of more cumbersome book-keeping.

Correctness An secure group messaging scheme SGM is correct if for all adversaries $\mathcal{A}$:

$$\Pr\left[\,\mathsf{G}^{\mathsf{sgm\text{-}corr}}_{\mathsf{SGM},\mathcal{A}}\,\right] = 0$$

where $\mathcal{A}$ is a pair of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, where $\mathcal{A}_1$ returns a list of identifiers $\mathsf{IDs} \subseteq \mathsf{SGM.ID}$ along with some state state that contains information about its computation; state could include the random coins used by $\mathcal{A}_1$, the list of identifiers IDs returned by $\mathcal{A}_1$, etc. The algorithm $\mathcal{A}_2$ takes in the state information state forwarded by $\mathcal{A}_1$ and uses it to interact with its oracles.

3.3 SGM protocol security requirements

Informally a group messaging protocol is meant to satisfy (as covered in [BBN19] and [OBR+21]) the following security requirements:

- Confidentiality: For any group G, if the current members of G are uncompromised then any message m created by a current member of the group G can not be read by anyone else apart from the current members of group G.
- Anonymity: For any group G, if the current members of G are uncompromised then any message m created by a current member of the group G leaks no information about the sender of the m to anyone else apart from the current members of group G. [2]
- Message and Sender Authenticity: For any group G, if the current members are uncompromised, and a current member of G receives a message, it can detect if it has been tampered with and which member (if any) sent the message.
- Forward-security (FS): For any group G if the state of any current G member is compromised at some point, then all messages sufficiently in the past remain secure.
- Post-compromise security (PCS): For any group G if the state of any current G member is compromised at some point and the adversary upon

[2]This of course assumes a threat model where an adversary compromises for example some central router in the network rather than a switch connected to a single group member.

compromise remains passive, then future messages become secure again after some period of time.

The only non-standard (basic) property is the anonymity property which we add in order to capture the requirement of minimising metadata leakage imposed on MLS by the architecture document [OBR+21]. In particular part of the metadata corresponding to a message is the identity of the sender of the message.

3.4 Secure Messaging (SM) scheme requirements

As explained in the introduction of this chapter, an SM scheme is an SGM scheme where the groups are static w.r.t. adds and removes and contain exactly two members. In other words the two clients who are members of the group in epoch 0 (upon group creation) are the members in all epochs greater than 0. Because of this an SM scheme does not support remove and add operations. Furthermore because the group only has two members the creation and update operations are embedded within the initialisation algorithm and the send and receive algorithms respectively.

3.4.1 Secure Messaging (SM) Syntax

Formally, an SM scheme consists of two initialization algorithms (one for each member), which are given an initial shared key k, as well as a sending and receiving algorithm.

Definition 3.2 *A secure messaging (SM) scheme is a tuple of algorithms* SM=($\mathsf{Init\text{-}A}$,$\mathsf{Init\text{-}B}$, Send, $\mathsf{Receive}$) *defined as follows:*

- $\mathsf{Init\text{-}A}$ ($\mathsf{Init\text{-}B}$) is a deterministic algorithm that takes in a key k and outputs a state $\mathsf{s_A} \leftarrow \mathsf{Init\text{-}A}(\mathsf{k})$ ($\mathsf{s_B} \leftarrow \mathsf{Init\text{-}B}(\mathsf{k})$).
- Send is a probabilistic algorithm that takes in a state s and a message m and outputs a new state and ciphertext $(\mathsf{s}', \mathsf{c}) \leftarrow_{\$} \mathsf{Send}(\mathsf{s}, \mathsf{m})$.
- $\mathsf{Receive}$ is a deterministic algorithm that takes in a state s and a ciphertext c and outputs a new state, an epoch number, an index and a message $(\mathsf{s}', \mathsf{e}, \mathsf{i}, \mathsf{m}) \leftarrow \mathsf{Recieve}(\mathsf{s}, \mathsf{c})$.

3.4.2 Secure Messaging (SM) security

A secure messaging (SM) scheme allows two clients a and b to communicate securely bidirectionally. The exact security requirements it must satisfy are the same as those required by a secure SGM protocol, except we do not require anonymity to hold since it is always the same clients a and b communicating with one another. Additionally confidentiality, authenticity

and forward security are expected to hold even in the case of an adversary controlling the random coins used by a and b. In summary a secure SM protocol must provide confidentiality, authenticity, forward security and post compromise security of its messages.

3.4.3 Modularisation of an SM scheme

In this section we give a summary of the main results of the work by Alwen et al. [ACD18]. This work presents a modular composition of the double ratchet protocol (part of the Signal protocol [PM16]) from three building blocks: continuous key agreement (CKA), forward-secure authenticated encryption scheme with associated data (FS-AEAD) and a two-input hash function (PRF-PRNG).

CKA The CKA primitive is an abstraction of the Diffie-Hellman output derivation part of the public key ratchet. It is a synchronous and passive primitive run between two clients a and b. A primitive is considered synchronous if the clients speak in turns, namely odd rounds consist of a sending and b receiving messages, whereas in even rounds b is the sender and a the receiver. By passive we mean that the primitive assumes that the adversary is passive, i.e. the messages between the clients can only be eavesdropped by the adversary. Note that the adversary is however allowed to control the random coins used by the sender and leak the state of either client. Each round i produces a fresh key k_i which is output by the sender upon sending the message in round m_i and the receiver upon receiving this message m_i. A CKA scheme is considered correct if both the sending and receiving party in each round produce the same fresh key. A CKA scheme is considered secure if given the transcript of messages exchanged $m_1, m_2, \ldots$ the fresh keys underlying these messages $k_1, k_2, \ldots$ look uniformly random and independent. Moreover a secure CKA scheme guarantees that after a state leak, security is restored within two rounds of the scheme (PCS) and that the all messages older than $\Delta_{CKA} \geq \mathbb{N}_0$ rounds remain secure upon state compromise (FS).

FS-AEAD The FS-AEAD primitive represents an SM scheme within a single epoch and hence captures the symmetric key ratchets, i.e. the sending and receiving key derivation (KDF) chains from the double ratchet protocol. The FS-AEAD primitive must then provide confidentiality, authenticity and forward security of messages in the presence of an adversary with capabilities as in an SM scheme. Post compromise security is not required by this primitive since healing of clients upon state compromise is supposed to occur over a certain number of epochs.

PRF-PRNG The PRF-PRNG primitive corresponds to the root KDF chain of the double ratchet protocol which, along with its state, takes the CKA primitive's fresh round keys as input and produces the new state for itself and the FS-AEAD primitive. More specifically, a PRF-PRNG scheme is a two-input function, that takes in a state s and an input $\mathcal{I}$, and produces a new state s' and a bit-string $\mathcal{R}$. A PRF-PRNG is considered secure if, provided that either the input state s is not leaked, or the input value $\mathcal{I}$ is sampled uniformly at random from some (input values set) S, the bit-string produced $\mathcal{R}$ is pseudorandom. Therefore, a PRF-PRNG can be seen to act as a pseudorandom number generator (PRNG) that repeatedly accepts input values $\mathcal{I}$, uses them to refresh its state s to become s' and output pseudorandom bit-strings $\mathcal{R}$ (provided s is not leaked); But it can also be seen to act as a pseudorandom function (PRF), in the sense that the output bit-strings on different $\mathcal{R}$ input values $\mathcal{I}$ (on the same not leaked state s) are indistinguishable from random and independent bit-strings.

Chapter 4

The Messaging Layer Security (MLS) protocol

The MLS protocol is an SGM protocol. In this chapter, we describe how the MLS protocol works based on the specification [BBM+21] and architecture [OBR+21] documents. The description presented here is a slight simplification of the specification given by the IETF group [BBM+21]. The assumptions made by us (in order to accommodate this simplification) and not by the MLS protocol as given in the [OBR+21] and [BBM+21] documents are introduced throughout the description. At the end of the description, we dedicate section 4.13 to summarize all these differences in one place.

4.1 MLS Terminology

This section introduces the basic terminology derived from [BBM+21] and [OBR+21] the MLS protocol makes use of.

User. A person who has an account in the instant messaging application.

Client. An endpoint device that is used by some user to execute the MLS protocol.[1]

State of a client. All information stored by the client. The state of a client consists of two distinct parts: **client data** which is information specific to that client and (if the client is a member of some group) **group data** which is information specific to the group the client is a member of. The `state` structure encompasses all this information as a collection of fields and is

[1] The **MLS protocol** is a group messaging protocol that **works on the level of clients**. The users and their association with clients is not considered by MLS. This is the task of the messaging application using MLS.

described in more detail in Section 4.4. All of these fields in the `state` structure are only writable by the client itself. However, some of these fields may be readable by all clients in the system (public), whereas others may only be readable by the client itself (private).

Group. A set of clients that have the knowledge of and have contributed to the group's current group secrets' values.

Member of a group. A client that is part of the group.

Group secrets. The group data fields that contain the same value and are private across all group members' states form the group secrets.

MLS ciphersuite. A collection of primitives: key encapsulation scheme KEM, key derivation function KDF, a nonce-based authenticated encryption with associated data NAEAD, a digital signature scheme DS, a hash function H and a MAC scheme MAC. To represent the MLS ciphersuite we define the `Ciphersuite` structure as shown in Figure 4.1:

```
struct Ciphersuite :
  KEM
  KDF
  NAEAD
  DS
  H
  MAC
```

Figure 4.1: Ciphersuite structure

Service Provider (SP). A service provider is a pair of two abstract functionalities, an **Authentication Service (AS)** and a **Delivery Service (DS)**, available to all MLS clients.

Authentication Service (AS). An authentication service is a functionality similar to the functionality provided by a Certificate Authority. Namely, it is responsible for generating and issuing credentials (which are structures similar to certificates see Figure 4.2) of clients. It does so by producing the credential's signature using the AS secret signing key, denoted as AS_sign_sk. This signature is made over the remaining fields in the client's credential, in this way binding a client's unique identity ID to the client's public verification key sign_pk and client's digital signature scheme of choice DS. This functionality must also be able to validate these credentials (by verifying their contained signatures) when requested by MLS clients. The AS ensures that a given ID is only used by one client.

Delivery Service (DS). A functionality composed of two further abstract functionalities: the **DS KeyPackage storage** and a per-group **DS broadcast channel**. In order to facilitate asynchronous addition of clients to a group, each MLS client pre-publishes their own KeyPackages to the **DS KeyPackage storage**. A KeyPackage of client a is a structure containing all information that any other MLS client b may need to add a to its group asynchronously (see Figure 4.3). Therefore the **DS KeyPackage storage** is responsible for storing the set of KeyPackages of each MLS client. Each client a can either add KeyPackages it owns to the DS KeyPackage storage, remove KeyPackages it owns or fetch a KeyPackage belonging to itself or some other client b from the DS KeyPackage storage. Whenever a KeyPackage is used by a client to create a group, the DS removes that KeyPackage from its storage.[2] For each group the DS provides a separate **DS broadcast channel**, that is responsible for relaying any message it receives to all members of the group.

Credential of a client. A `Credential` structure is a structure that groups a (1) a unique identity ID of the client, (2) a digital signature scheme DS, (3) a public key sign_pk corresponding to DS and (4) a signature over (1), (2) and (3) using the secret signing key of the AS AS_sign_sk. Figure 4.2 depicts this structure.

```
struct Credential:
    ID
    DS
    sign_pk
    signature
```

Figure 4.2: Credential structure

KeyPackage of a client. A KeyPackage structure is a structure containing all information needed by a group member to add this client to the group. More specifically it contains: (1) an integer corresponding to the MLS protocol version the client supports version, (2) a `Ciphersuite` suite representing the MLS ciphersuite the client supports, (3) a `Credential` credential being the credential the client obtained from the AS, (4) a KEM public key kem_pk representing the client's KEM public key corresponding to the KEM scheme specified in the suite, (5) a `Capabilities` instance capabilities (see Figure 4.4), (6) a `Lifetime` instance lifetime (see Figure 4.5) and (7) a signature over fields (1) to (6) using the secret signing key corresponding to the public signing key credential.sign_pk. Figure 4.3 depicts this structure:

[2] This prevents using the same KeyPackage to create two different groups, and hence prevents replay attacks.

```
struct KeyPackage:
  version
  suite
  credential
  kem_pk
  capabilities
  lifetime
  signature
```

Figure 4.3: KeyPackage structure

The Capabilities of a client. A Capabilities structure is a structure that groups a list of integers versions corresponding to protocol versions the client supports and a list of Ciphersuite structures suites. It ensures that no downgrade attack can occur undetected once a client wishes to create a group. Figure 4.4 depicts this structure:

```
struct Capabilities:
  versions
  suites
```

Figure 4.4: Capabilities structure

The Lifetime of a KeyPackage. A Lifetime structure is a structure that groups two integers representing the time interval within which the KeyPackage is considered valid. The KeyPackage should be considered invalid outside of this interval. Figure 4.5 depicts this structure:

```
struct Lifetime:
  not_before
  not_after
```

Figure 4.5: Lifetime structure

InitKeys of a client. The set of KeyPackages the client has stored in the DS-s KeyPackage storage.

4.2 Assumptions and context of MLS

In the MLS protocol. each client and each group has a unique identifier ID and GID respectively, which are defined by the instant messaging application. All MLS clients know which client identifier ID belongs to which client. We assume that each client uses exactly one identifier (that never changes)

throughout the entire MLS protocol. Similarly, we assume that once a group is created the GID assigned to it never changes throughout the lifetime of the group. The MLS protocol, as given in [BBM+21], is designed to execute in a context that supplies the service provider (SP) functionalities for its clients to use. The specification makes no assumptions on how the SP's services are implemented (the services could be running on a server, client, or a combination thereof). The MLS architecture document [OBR+21] does lay out some suggestions on how different SP functionalities can be achieved.

For simplicity and in order not to detract from the essential components of MLS, this work models the AS, the DS broadcast channel and the DS KeyPackage storage functionalities as three separate stand-alone servers. The AS and DS KeyPackage storage servers provide an API that each MLS client can use to interact with them via a secure channel. The MLS protocol assumes that the DS broadcast service provides clients with a consistent view of which Commit message (see Section 4.3 and 4.7.3) defines each epoch. That is, while multiple clients may create a Commit message to initiate the next epoch, the DS broadcast service ensures that the first Commit message received by the DS broadcast service, is the one all group members will receive first and hence use to derive the new epoch secrets.[3] We model this assumption by having the channel from the DS broadcast service to the clients be secure against Commit message drops and reorders and having the channel from clients to the DS broadcast service be insecure. Note that we make no assumption about the security of the channel from the DS broadcast service to clients for Proposal and Application messages. The specification makes no assumptions on the communication channels between MLS clients and hence we model them as insecure.

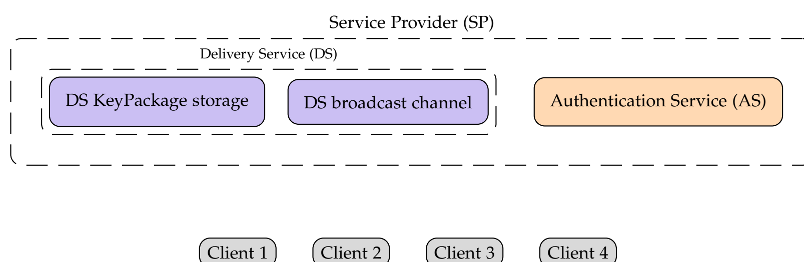


Figure 4.6: Architecture of MLS.

[3]The [OBR+21] document indeed demands the DS Broadcast provide a consistent view to all group members, but it does not require that the first Commit message it receives be the one chosen to be broadcast. Rather it allows for the messages to carry a counter that can be used to break ties when two members send a Commit message at the same time asserting a sort of priority queue. The [OBR+21] suggests that this assumption can be implemented by means of clients checking with one another if such a misbehaviour occurred. We believe that this mechanism is insufficient to realise this assumption.

Given this model of the service provider, the MLS architecture in this work is as shown in Figure 4.6.

The API of the AS is shown in Figure 4.7. The **get_credential** takes in a client identifier ID and a digital signature scheme DS. It verifies that the calling client is indeed represented by this ID, generates a fresh signature key pair (sign_sk, sign_pk) using DS and creates a `Credential` structure using the ID, sign_pk and a signature over these fields using the AS secret signing key AS_sign_sk. It then returns this `Credential` structure and sign_sk to the calling client, which can now use it to authenticate to all MLS clients. The **verify_credential** procedure takes in a `Credential` structure credential and outputs a boolean bool depending on the validity of the signature contained within the credential.

```
get_credential(ID,DS):
  // creates a Credential structure with
  // identifier ID and signature scheme DS
  return (credential, sign_sk)

verify_credential(credential):
  // verifies the signature contained in credential
  return bool
```

Figure 4.7: AS API.

The API of the DS KeyPackage storage is shown in Figure 4.8. The procedure **set_keypackage** takes in a `Credential` structure credential, integer version, `Ciphersuite` structure suite, `Capabilities` structure caps, Lifetime structure time and a digital signature signature. It verifies the credential via the AS, verifies credential.DS=suite.DS and then generates a KEM key pair (kem_sk, kem_pk) using the KEM scheme suite.KEM. It then uses these input values and kem_pk to create the `KeyPackage` structure keypackage for the calling client. It stores keypackage locally to enable other clients to fetch this keypackage (via **get_keypackage** procedure) in order to asynchronously form a group with the owner of keypackage. Finally it returns the KEM key pair (kem_sk, kem_pk) corresponding to the stored keypackage to the calling client.

```
get_keypackage(ID, version, suite):
  // returns KeyPackage structure
  // corresponding to the input values
return keypackage

del_keypackage(keypackage):
  // delete keypackage from local storage

set_keypackage(credential, version, suite, caps, time, sign_sk):
  // generates KEM key pair and corresponding KeyPackage
  // structure for the owner of credential
  // locally stores the KeyPackage structure to
  // facilitate asynchronous addition and group creation
  return (kem_sk, kem_pk)
```

Figure 4.8: DS KeyPackage storage API.

The **get_keypackage** takes in a client identifier ID, integer (representing protocol version) version and `Ciphersuite` structure suite and returns a corresponding `KeyPackage` structure. A single client (uniquely identified by its identity) can support multiple protocol versions and ciphersuites, hence we

need all three values in order to know which `KeyPackage` structure to return. We will use **get_keypackage**(ID,*,*) to fetch all keypackages corresponding to an identifier ID. The **del_keypackage** takes in a `KeyPackage` structure `key-package` and removes it from the DS KeyPackage local storage.

The per group DS broadcast channel is modeled as a standalone server on the network, whose logic is abstracted away. If a group member wishes to send an application, commit or proposal message (see Section 4.3) to the group, it will send that message to the DS broadcast server, which will then fan out the message to all the members of the group (including its creator).

Of course, in reality AS and DS functionalities, as well as the secure channel between the AS and DS KeyPackage storage and the clients, would be implemented in ways described in [OBR+21].

4.3 MLS Protocol overview

The purpose of this section is to give a high level description of how the MLS protocol implements the different operations an SGM protocol must support as described in Section 3.1. For each operation it showcases which messages need to be exchanged for the operation to come into effect. All of these messages go through a 4 step process: message creation, message sending, message receiving, message processing. In our diagrams (message sequence charts) a message is sent by client *m* if and only if there is an arrow coming out from *m*'s lifeline and a message is received by client *m* if and only if there is an arrow in-coming into *m*'s lifeline. As already stated the diagrams only show the exchange of messages (i.e. sending and receiving) but not how the messages are created prior to sending or processed post receival. The details of these two stages (creation and processing) are deferred to Sections 4.7, 4.10 and 4.11.

In this high level description we consider a scenario with four clients A, B, C and D (associated to some users with messaging application accounts) with no group formed yet.[4] In the diagrams we use green arrows in the figures to denote a secure channel, orange arrows to denote a no drop or reorder of messages channel and black arrows denote the network, i.e. insecure channel. We split the protocol description into six steps. The first and second correspond to the setup stage of the protocol. Namely they describe what each MLS client needs to do in order to initialize its state upon IM application installation. The third and fourth step describe the process of group creation. Step five describes how group operations (adding, removing clients and updating keys) occur. The last step describes the sending and receiving of application data.

[4] We skip the part where users would create an account via some client, as that is the job of the messaging application, not MLS.

1. At the very beginning of the MLS protocol, each client first initializes their state. To do this, each client T first forms a `Capabilities` structure T.capabilities that will contain all the MLS protocol versions and `Ciphersuite` structures client T supports ordered by preference. After creating it, the `Capabilities` structure is added to the client's state. Each client T then chooses a digital signature scheme of their liking T.DS and uses it and its identifier T.ID to interact with the AS by calling get_credential(T.ID, T.DS). The AS then returns to client T its credential T.credential and its secret signing key T.sign_sk corresponding to T.credential.sign_pk. Client T then adds the obtained credential to its state (see Figure 4.9).

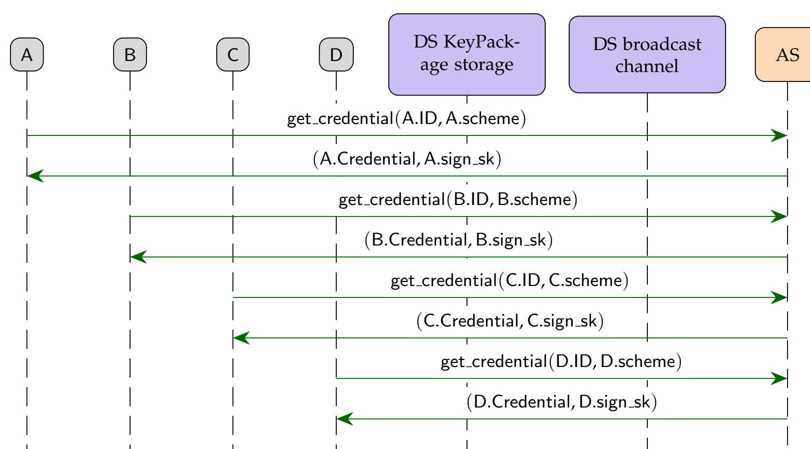


Figure 4.9: Creation of credentials.

2. After this, each client T is ready to form its `KeyPackage` structure, denoted as T.KeyPackage. First T selects one of the `Ciphersuite` structures in its `Capabilities` structure T.capabilities that contains the digital signature scheme chosen in step 1 T.DS. Each client T also selects one of the MLS protocol versions T.v from T.capabilities to be included in its `KeyPackage` structure and a time interval T.t within which T.KeyPackage is considered valid.

   Although a **client may support more than one `Ciphersuite` structure and MLS version**, it **can only advertise one** `Credential` structure and KEM public key per `KeyPackage` structure (and hence one `Ciphersuite` structure and version). Therefore a single client will own one `KeyPackage` structure for each `Ciphersuite` structure and MLS version it supports. In order to ensure that a creator of a group chooses the most preferred `Ciphersuite` structure and MLS version by all group members, the client also includes its `Capabilities` structure T.credential formed in step 1 in

its `KeyPackage` structures.

Finally, each client T calls $\text{set_keypackage}(\text{T.credential}, \text{T.v}, \text{T.DS}, \text{T.capabilities}, \text{T.t}, \text{T.sign_sk})$ (denoted as setT.KeyPackage in Figure 4.10) from the DS KeyPackage Storage API. This call will generate a KEM key pair for T and use them and the client provided information to form T's `KeyPackage` structure T.KeyPackage. The call will also store T.KeyPackage at the DS KeyPackage Storage server in the keypacks map and send back the generated KEM key pair to the calling client $(\text{T.enc_sk}, \text{T.enc_pk})$ (denoted as T.enc_keys in Figure 4.10). Upon receiving its KEM key pair each client T adds T.enc_keys to its state.

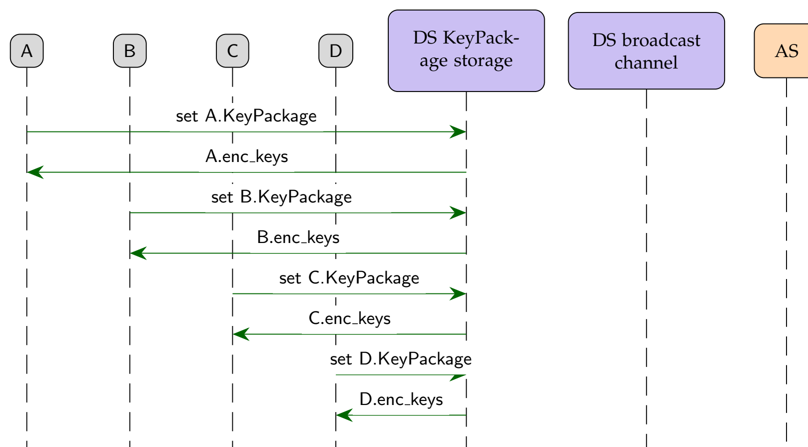


Figure 4.10: Publishing of KeyPackages.

Each client will now repeat steps 1 (excluding the creation of the `Capabilities` structure) and 2 for each combination of `Ciphersuite` and protocol version listed in its `Capabilities` structure. Note that step 1 can be skipped for all version `Ciphersuite` combinations where the `Ciphersuite` contains the same signature scheme. Once this is done the setup stage is complete and the state resulting from steps 1 and 2 is the initial state of an MLS client.

3. Now let us say client A wishes to establish a group with clients B and C. To do this, client A will fetch the `KeyPackage` structures of clients B and C by calling $\text{get_keypackage}(\text{B.id}, *, *)$ and $\text{get_keypackage}(\text{C.id}, *, *)$, from the DS KeyPackage storage API, which returns all `KeyPackage` structures of client B and C stored at the DS KeyPackage storage respectively denoted as B.keypacks and C.keypacks. Client A (group creator) then verifies the validity of the `KeyPackage` structures it fetched.

   A `KeyPackage` structure is considered **valid** if its contained signatures verify (`KeyPackage` structure signature and the contained `Credential`

structure's signature) and if the current time is within the bounds of the KeyPackage structure's lifetime. This validity check ensures that the DS didn't modify the KeyPackage structure and that it didn't serve an expired (stale) one.

If the validity check is passed, the group creator continues to examine the Capabilities structure contained in the KeyPackage structures it fetched and chooses the MLS version and Ciphersuite structure most preferred by all of the clients it wishes to form a group with (B, C and itself). It then updates its state with this choice. Let us call this version and Ciphersuite structure G.v and G.suite. Because the KeyPackage structures are signed and the group creator chooses the best version and Ciphersuite supported by the whole group, the MLS sessions are ensured to be safe from **downgrade attacks**.

Now client A deletes the three KeyPackage structures it uses to form the group with B and C from the DS KeyPackage storage by calling $\mathsf{del_keypacakge}(\mathsf{A.ID}, \mathsf{G.v}, \mathsf{G.suite})$, $\mathsf{del_keypacakge}(\mathsf{B.ID}, \mathsf{G.v}, \mathsf{G.suite})$ and $\mathsf{del_keypacakge}(\mathsf{C.ID}, \mathsf{G.v}, \mathsf{G.suite})$ from the DS KeyPackage storage API. This ensures that no replay attack can occur across different groups containing the same members. Namely, consider what would happen if this deletion did not occur. Clients A, B and C have formed a group and then proceed to send application data, add members, etc which all happen through the DS broadcast channel. Now assume client B wanted to start another group with clients A and C. An attacker who has recorded messages of the first group could impersonate A and C without them taking part in the conversation, just by replaying messages.[5]

Now that we have seen how ciphersuite and version negotiation works in MLS, in order to simplify the clients' state, we assume that all groups are disjoint. Namely we assume that a client can be a member of at most one group. This way a client's state only contains information about the group it is currently a member of. Secondly, we assume that all clients support exactly one same Ciphersuite structure and MLS version and hence that each client only contains one Ciphersuite structure, one integer representing the version, one signature secret key, one Credential structure and one KEM key pair in its state after steps 1 and 2 described above. Note that we do not need to save the Capabilities structure of the client since no downgrade can occur in a scenario where all clients support exactly one and exactly the same Ciphersuite structure. This implies that there is only one KeyPackage structure per client stored at the DS KeyPackage storage.

[5]This is a very weak attack, but none the less one MLS tries to protect against, because it relies on agent B starting with the same randomness as A in the first group and both groups evolving in the exact same way.

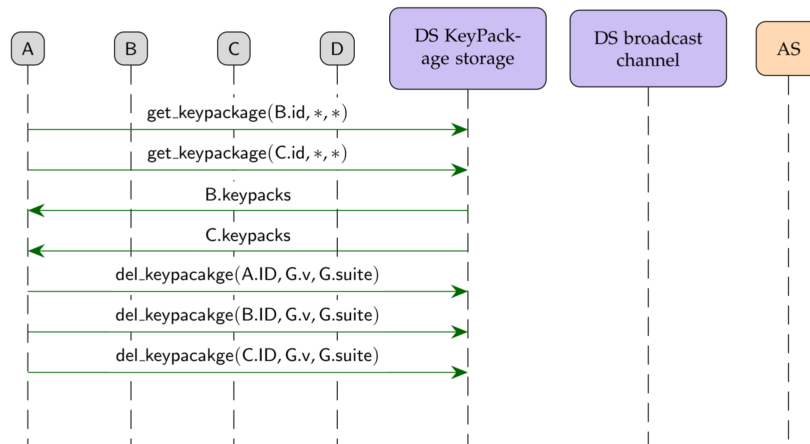


Figure 4.11: Step 1 of group creation algorithm: Fetching group members' KeyPackages.

From now on MLS clients can also send and receive 3 types of messages:

Handshake messages. Messages containing information about group operations. We subdivide them into two further types: **Proposal** messages, used to propose a group operation, and **Commit** messages, used to refresh group secrets and to indicate which of the proposed group operations go into effect.

Application messages. Messages used to send Instant Messaging (IM) application data.

Welcome messages. Messages used to send all data a client would need to join a group it is not yet a member of.

Given any group G and any of its members m, if m wishes to do one of the group operations (add a member, remove a member from a group it belongs to or update its own encryption (KEM) key pair), m first needs to **propose it to other members of group** G. It does so by first creating a **Proposal message** of the corresponding type **Add**, **Update** or **Remove**. Once formed m sends the Proposal to G's DS broadcast channel, which in turn fans out the Proposal to all members of group G (including m).

In order for a sent **Proposal message** p to enter into effect, a **Commit message** containing p must be sent from one of G's members who **received** p. Any member m' who has not received a proposal suggesting to remove m' can act as a committer and create a Commit message. If m' is qualified (did not receive a remove proposal containing itself) and chooses to construct a **Commit message** c one of the things c will contain is a list of all valid Proposal messages that m' received such that each client is targeted by at most one of them; a Commit message can not contain two proposals that apply

to the same client as it would be ambiguous which operation applies to this client. After forming this list, member m' uses it to generate fresh values for the group secrets. Since each group member maintains in their state a local view of the group secrets, the new values generated by m' must be communicated to the rest of the group to ensure consistent group secrets across all group members. Therefore a second thing that m'-s Commit message must contain is information related to these new group secrets values.

The Commit message is sent to all clients who were members of the group prior to m'-s commitment (termed as existing members). However if the Commit message lists an Add proposal as one of the operations to come into effect, then m' also needs to communicate the new group secrets to the members listed to be added. To do this m' apart from forming a Commit message, forms a Welcome message (if and only if the Commit message lists an Add proposal). The reason why the Commit message can not be sent to the added members as well is because m' used the previous values of the group secrets to form it (see Section 4.7.3). Therefore the Welcome message is necessary in order for the added members to obtain the newest group secrets they are entitled to know.

In case m' does commit, m' is called a **committer**. The committer m' then sends c to G's DS broadcast channel, which delivers the Commit message c to all existing members of G (hence including m'). If the committer m' made a Welcome message as well, it sends it to each added member directly (not via the DS broadcast channel). Each existing member will then upon receiving c process it to obtain new group secrets. Similarly each added member will upon receiving the Welcome message process it to obtain the group secrets as well.

Given all this, changes in the group secrets values occur if and only if some group member processes a Commit message (or a Welcome message). Note that this implies that even the committer does not change its group secrets until it receives and processes its own Commit message from the DS broadcast server. This is because there may be two group members that act as a committer at the same time. Hence a client must wait to see if the next Commit message it receives from the DS broadcast server is its own or someone elses. In this case the DS broadcast channel fans out the Commit message it receives first. This way all members of the group will process the same Commit message and obtain group secrets whose values are consistent across the entire group. These different versions of group secrets split the lifetime of a group into so-called **epochs**. Each of these epochs is uniquely identified by an integer epochID. The epochID is 0 prior to group creation. Each time a Commit message is processed by one of the group members the epochID is incremented by one. An example of a group's lifetime is given in Figure

4.12. [6]

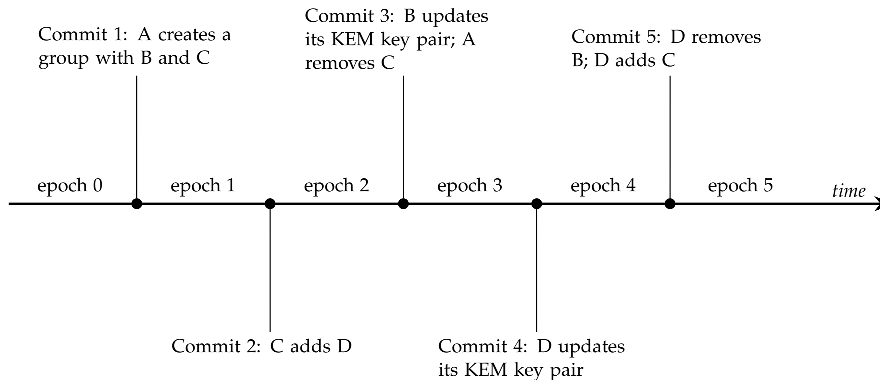


Figure 4.12: Group lifetime through epochs.

Each group member maintains their own epoch counter (representing the member's current epoch) that is in sync with the group's epochID if the member has processed the Commit message corresponding to the current epochID value. This local epoch counter can become out of sync from the epochID if the client is offline and hence does not process the latest Commit messages. For example let us say that since the group's creation there were 6 changes to the group secrets and hence there are 6 epochs. Then a group member who was offline whilst the last 3 changes occurred would still contain group secrets corresponding to epoch 3. Hence this group member's current epoch counter is 3, despite there being 6 epochs overall. If a member joins a group with epochID $= i$ (for some $i \in \mathbb{N}_0$) at the time of joining then that member's local epoch counter is set to be i. The remaining details on how the handshake, application and welcome messages are formed and processed are explained in Sections 4.7, 4.10 and 4.11.

4. At this point client A knows which `Ciphersuite` structure and MLS version is best supported by all the clients it wishes to form a group with. To create a group containing clients of its choice, A starts by creating the smallest non-empty group, a group containing only itself, by running the initGroupData algorithm locally (see Section 4.12). This call essentially generates the initial group data (hence including group secret values) corresponding to epoch 0. The secrets at this point are only known by A, since it is the only member of its one-member group. Then client A creates an Add Proposal message for each client it wishes to add to its one-member group (in our example it wishes to add B and C). Because the group currently only contains A and hence A is

[6] Note that epochs are a property of a group not of group members. The epochID counts the number of changes made to group secrets since the group's creation.

the only client that needs to approve of its own Proposals, there is no need to send the Proposal messages to the DS broadcast. Instead it just immediately creates a Commit message including these Proposals as well as a Welcome message. The Welcome message is then sent to B and C directly (not via the DS broadcast channel). Since, again the existing group member is only A, there is no need to send the Commit message to the DS Broadcast server.[7] The created Commit message is processed by client A only, deriving the group secrets corresponding to epoch 1. Clients B and C upon receiving the Welcome message, process it and derive the group secrets corresponding to epoch 1 as well (See the whole process in Figure 4.13).

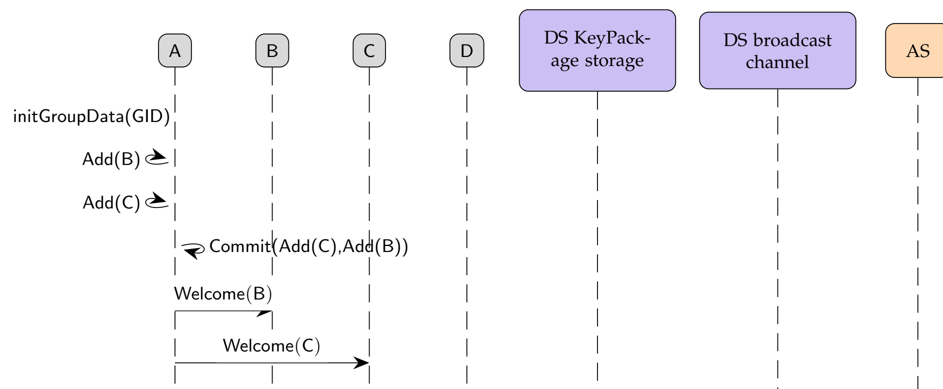


Figure 4.13: Step 2 of group creation: Client A creates a group containing only itself via initGroupData(GID). It then forms an Add Proposal for B and C and commits it locally. It then sends a Welcome message to clients B and C directly.

5. Once a group is created (here the group contains clients A,B,C) any group operation (add,update,remove) occurs by following a two-step process: the group operation is proposed and the group operation is committed. As stated before any group member can propose a group operation by sending a Proposal message to the group. Then for the proposed group operation to come into effect, it needs to be contained (along with possibly other proposals) within a Commit message. This Commit message can be created by any group member who received the Proposal message and who is not proposed for removal by any of the Proposals it received. If a group member wishes to propose an Add operation then the Proposal message is preceded by

[7] The sending of Commit and Proposal messages does not fit what we described before, simply because it is useless to send a message that will only be received by its creator. Whenever a group contains more than one member, the Commit and Proposal messages need to be sent to the DS Broadcast channel in order to achieve agreement on group operations going into effect and consistency of group secrets' values across the group.

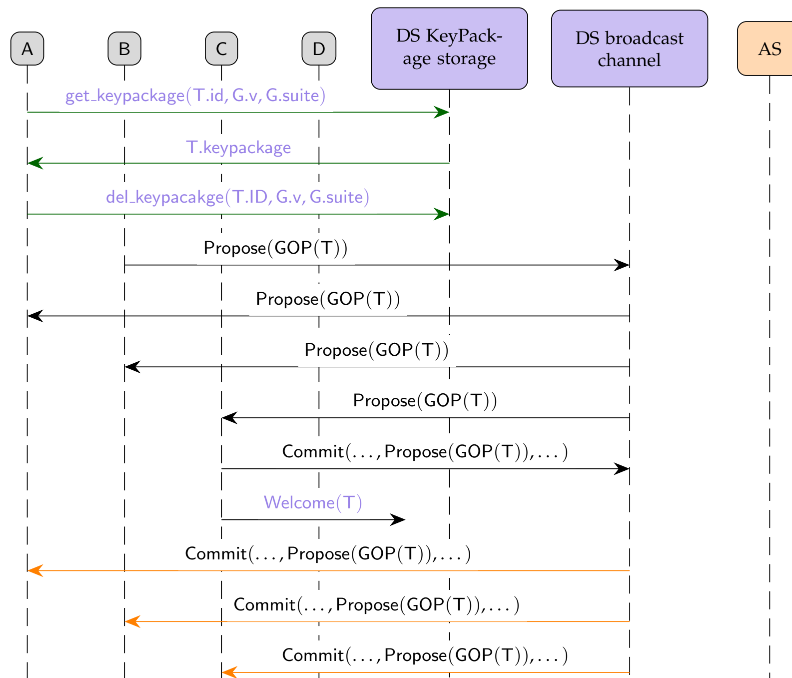


Figure 4.14: Group operations: add, remove and update following the propose and commit paradigm. Client B proposes a group operation GOP applying to some client T and client C sends commits to this group operation by including it in its Commit message. The messages annotated in purple are only present in case GOP=Add.

the proposing client fetching the `KeyPackage` structure of the client it wishes to add by calling get_keypackage(id, G.v, G.suite), where id is the identifier of the suggested client and G.v and G.suite are the MLS version and `Ciphersuite` structure supported by the group respectively. This fetched `KeyPackage` structure, denoted as T.keypackage in Figure 4.14 is then deleted from the DS KeyPackage storage for the same reasons as in step 3. Moreover the Commit message would be sent along with a Welcome message suggested for addition. This process is summarized in Figure 4.14 where B proposes some group operation $\text{GOP} \in \{\text{Add}, \text{Remove}, \text{Update}\}$ for some client T and client C acts as a committer. The additional messages sent in case of $\text{GOP} = \text{Add}$ are colored purple to denote that they are only present in case of a client being added.

6. Finally if a group member wishes to send some IM application data info to the group it can do so by simply forming an application message (see Section 4.7 and 4.10) denoted as Application(info) in Figure 4.15 and sending it to the group via the group's DS broadcast channel. Because the client wishes for the underlying IM application data to be confidential and authentic, it will encrypt it using keys derived from

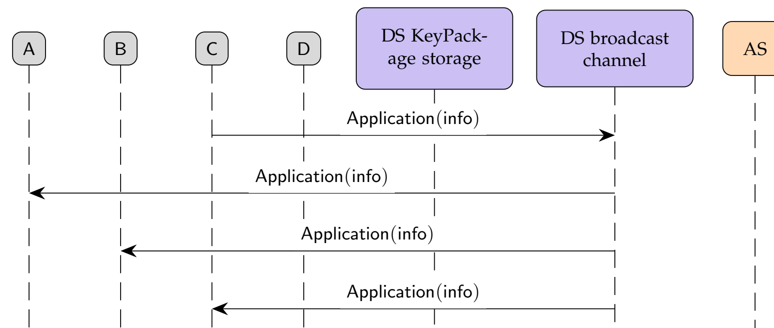


Figure 4.15: Send and receive operation: Client C sends an application message (containing IM application data) to the group. All clients in the group receive it and process it to extract the underlying IM application data.

the group secrets (see Section 4.9 and 4.10). A client upon receiving an application message, decrypts it and processes it to obtain the underlying IM application data. This process is summarised in Figure 4.15.

4.4 State of client

As noted in the MLS protocol overview (see Section 4.3) each MLS client maintains some local information in order to participate in the MLS protocol. This information is grouped together to form a `state` structure shown in Figure 4.16. The fields within this structure can be split into two distinct parts: **client data** i.e. information specific to that client and (if the client is a member of some group) **group data** which is information specific to the group the client is a member of. The **client data** is the set of non-highlighted fields of the `state` structure whereas **group data** is highlighted in green and yellow. As explained in Section 4.1 some fields in a client's `state` structure may be public (readable by everyone in the system) and others may be private (readable by only the client). In Figure 4.16 we add a comment next to the fields MLS requires to be private. The **group data** can be further subdivided into two parts: fields that form group secrets (do not differ in value across group members see Section 4.1)[8] and fields that do not (may differ in value across group members but not necessarily).

A client's group secrets change if and only if the client either processes a Commit message, or it initiates the group, or it processes a Welcome message. When the remaining fields of the `state` structure change will become clear in the following sections. We say that a client knows the value of a if and only if a is present in its state (not equal $\perp$).

[8] This is of course assuming all members processed the latest commit message that lead to the most recent group secret values.

```
struct state:
    version
    ciphersuite
    credential
    sign_sk   // private
    kem_pk
    kem_sk   // private
    ratchet_tree   // private
    group_context
    confirmation_tag
    com_ctr
    prop_ctr
    hand_generation
    app_generation
    prop_pending   // private
    commit_pending   // private
    prop_confirm   // private
    com_confirm   // private
    hand_ratchet_state   // private
    app_ratchet_state   // private
    init_secret   // private
    commit_secret   // private
    welcome_secret   // private
    encryption_secret   // private
    sender_data_secret   // private
    confirmation_key   // private
```

Figure 4.16: State of an MLS client. The non-highlighted fields are set in step 1 of the MLS overview (see Section 4.3), which is responsible for initialising the state of the client. The fields in green are part of group data but not part of the group secrets and the fields highlighted in yellow are part of the group data and group secrets. The fields that are private have a comment whereas the fields that can be made public do not.

As discussed in the protocol overview, we assume that a client can only be a member of one group at a time, so the `state` structure only contains **group data** of a single group.[9] Furthermore another assumption mentioned in the protocol overview is that we assume that all clients support the same and exactly one ciphersuite and version. Therefore instead of **client data** containing a list of ciphersuites, credentials, etc. we only maintain one of each within the `state` structure. This implies that the `state` need not con-

[9]If we were to allow clients to be a member of more groups the bookkeeping would become more complex (`state` would essentially contain a map from group identifiers to group group data) without any added value other than completeness.

tain the Capabilities structure of the client since no downgrade can occur in a scenario where all clients support exactly one and exactly the same Ciphersuite structure and MLS version.

Let s be a state structure of some client a. Then s.ciphersuite is an integer representing the MLS version number the client supports. The s.ciphersuite is the Ciphersuite structure specifying the KEM, KDF, NAEAD, DS, MAC scheme and hash function the client supports. The s.credential is the Credential structure this client obtained from the AS. The s.sign_sk and s.credential.sign_pk form a digital signature key pair and s.kem_pk and s.kem_sk form the KEM key pair the client obtained from the DS KeyPackage storage. The remaining fields' values are all $\perp$ unless the client is part of a group. Thus the state produced by step 1 and 2 in Section 4.3 (the setup stage) would then have the client data fields not equal $\perp$, whilst the group data fields would be $\perp$.

If the client is a member of the group then the group data fields are described as follows. The s.ratchet_tree is a ratchet tree (see Section 4.5), the s.group_context is a GroupContext structure (see Section 4.6) which essentially contains the group identifier, the epoch the client is currently in, a hash of the current s.ratchet_tree value and a hash of Commit messages received and processed thus far. Therefore, the GroupContext structure serves as a summary of some information about the group data. The s.confirmation_tag is a MAC tag that was contained in the most recently processed commit message. It changes if and only if s.group_context changes. Like the group secrets, the s.group_context and s.confirmation_tag contain the same value across all group member's states. However since they are allowed to be public knowledge, we do not classify them as being part of the group secrets.

The fields s.prop_pending, s.commit_pending, s.prop_confirm, s.com_confirm, as well as, s.prop_ctr, s.com_ctr and s.hand_generation are used to keep track of all the proposal and commit messages sent by this client but not yet confirmed by the group in the current epoch. Namely s.prop_pending, s.commit_pending, s.prop_confirm, s.com_confirm are maps that take an integer value and map it to a KEM secret key, a ratchet tree and bit-string (representing the pending commit_secret) pair, Proposal structure (see Section 4.7.2) and Commit structure (see Section 4.7.3) respectively. The fields s.prop_ctr, s.com_ctr and s.hand_generation are integer values representing the number of proposal messages sent, number of commit messages sent and the total number of handshake (proposal and commit) messages sent by a in the current epoch. A client's proposal message is confirmed by the group if the client receives a commit message containing this proposal. Because there may be two clients that send a commit message at the same time, a client must wait to see if the next commit message it receives from the group is its own or someone elses. If it receives its own commit message back that means its been confirmed by the group.

More specifically each time a sends an update proposal message it also saves the new potential KEM secret key associated to this update in the s.prop_pending[s.prop_ctr], saves the content of the sent update proposal message (`Proposal` structure) in s.prop_confirm[s.prop_ctr] and increments s.prop_ctr and s.hand_generation by one. Similarly for each commit message the client sends, it saves the new ratchet tree and commit_secret resulting from processing this commit message in s.commit_pending[s.com_ctr] (see Section 4.5 and 4.7.3), saves the content of the sent commit message (`Commit` structure) in s.com_confirm[s.com_ctr] and increments s.com_ctr and s.hand_generation by one.

Similarly the field s.app_generation is an integer value representing the number of application messages sent by a in the current epoch, and is hence incremented every time a sends an application message.

The hand_ratchet_state and app_ratchet_state are maps that take in an integer ep (representing an epoch) and return a sending and receiving state pair $(\mathsf{st}_S, \mathsf{sts}_R)$. This pair represents all the information a client must maintain in order to encrypt and decrypt handshake and application messages respectively in epoch ep. In particular the sending state st_S is a single bit-string value, whereas sts_R is a map, that takes in a client identifier ID and returns a triple containing a bit-string, an integer and a list of NAEAD key-nonce pairs $(\mathsf{st}_R, \mathsf{i}_R, \mathcal{D})$. A client a will only use its current epoch to send messages to the group. Hence, $\mathsf{st}_S = \perp$ for all epochs other than the current one. In other words, a will only use hand_ratchet_state[s.group_context.epochID] to send messages to its group. However, due to asynchronous communication and possible delays in message delivery, a client can have one or more active receiving epochs and can hence also use information relating to past epochs to successfully decrypt messages. We describe how the hand_ratchet_state and app_ratchet_state entries are initialised in Section 4.8 and 4.9. In Section 4.10 we describe the derivation of NAEAD key nonce pairs used for encrypting handshake and application messages and explain the semantics of the hand_ratchet_state and app_ratchet_state fields.

Finally the init_secret, commit_secret, welcome_secret, encryption_secret and confirmation_key are pseudo-random bit-strings that have been derived according to the key schedule (see Section 4.8). The sender_data_secret is a map from epochs to pseudo-random bitstrings. The reason why this group secret is a map is because it is used to provide metadata protection of application and handshake messages sent and received by this client. Therefore, because the client can receive messages that belong to an epoch older than its current one, the client needs to maintain an sender_data_secret entry for older epochs in order to be able to process the metadata of delayed messages.

4.5 Ratchet Tree (RT)

The MLS protocol uses a ratchet tree in order to derive group secrets efficiently.

Ratchet Tree (RT). A ratchet tree is an LBBT whose nodes have the following structure:

```
struct node:
  kem_pk
  kem_sk
  credential
  unmerged_leaves
```

Figure 4.17: Structure of a node in a ratchet tree.

Let τ be a ratchet tree and n be a `node` structure in τ. The n.kem_pk and n.kem_sk are a KEM key pair and the n.credential is a `Credential` structure. Finally the n.unmerged_leaves is a set containing some of the non-blank leaf nodes in the subtree rooted at n. We say n is **blank** if and only if all its fields' values equal $\perp$. In diagrams we will color the nodes white if they are blank and otherwise black. A client's ratchet tree changes if and only if the client processes a Commit message (or it initiates a group or it processes a Welcome message). Therefore for each new epoch, each client in the group will have a new ratchet tree. In which way the ratchet tree changes from one epoch to the next is described in Section 4.7. Now we turn to explaining the semantics behind the ratchet tree and its nodes' fields.

Let G be a group with ℓ members. Then each member m in its state $m.s$, will have $m.s$.ratchet_tree be a ratchet tree where each non-blank leaf represents one of the ℓ members in G. Hence the ratchet tree maintained by each member of group G will have at least ℓ leaves. Each group member's ratchet tree is almost exactly the same as the ratchet tree maintained by other members in the group. The only difference across the trees of the group members are the values stored in the kem_sk field of nodes. We will explain these differences in Section 4.5.1 after we have defined the relevant ratchet tree terminology.

If n is a non-blank leaf node in $m.s$.ratchet_tree then n.kem_pk and n.kem_sk are the KEM key pair belonging to the group member n represents. Similarly n.credential is the `Credential` structure belonging to the represented group member and the n.unmerged_leaves is always equal to $\perp$.[10]

[10] A subtree rooted at a leaf ℓ contains only ℓ.

If n is a non-blank parent node then n.kem_pk and n.kem_sk values will be derived based on the kem_pk and kem_sk values of one of its children. This derivation is covered in Section 4.7.3. The parent nodes' n.credential is always equal to $\perp$, since only non-blank leaves represent a group member. The n.unmerged_leaves is a set that contains each non-blank leaf n' in the subtree rooted at n such that the member $m' \in G$ represented by n' **does not know** the value of n.kem_sk. More specifically a member $m' \in G$ **knows** n.kem_sk value if and only if the leaf it is represented by is not in n.unmerged_leaves and is a descendant of n. Client m with `state` structure m.s knows the value of n.kem_sk if and only if n.kem_sk $\neq \perp$ in m.s.ratchet_tree.

Ratchet tree representation. Within pseudocode we will represent a ratchet tree as a list of `node` structures. In this list representation, leaf nodes are positioned at even-numbered indices, whereas parent nodes are held at odd-numbered indices; starting at the left-most node in the tree at position zero and running from left to right. For example, a ratchet tree with 5 leaves would have its nodes stored in the list representation as shown in Figure 4.18.

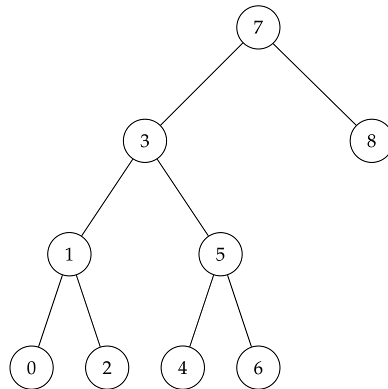


Figure 4.18: Example ratchet tree with 5 leaves. The index of each node in the array representation is written on top of each node.

With this ratchet tree representation we make use of the following functions to obtain the root of a tree, sibling, parent, left child and right child of any given node as well as a function to obtain an index given a node:

- Function Root(τ) returns the root node of the ratchet tree τ.
- Functions siblingNode(τ, n) and siblingIndex(τ, i) return the `node` structure and index of the sibling of node n at index i in ratchet tree τ respectively. If the node has no sibling these two functions return $\perp$.
- Functions parentNode(τ, n) and parentIndex(τ, i) return the `node` structure and index of the parent of node n at index i in ratchet tree τ respectively.

If the node has no parent these two functions return $\perp$.

- Functions $\mathsf{leftNode}(\tau, n)$ and $\mathsf{leftIndex}(\tau, i)$ return the `node` structure and index of the left child of node n at index i in ratchet tree τ respectively. If the node has no left child these two functions return $\perp$.
- Functions $\mathsf{rightNode}(\tau, n)$ and $\mathsf{rightIndex}(\tau, i)$ return the `node` structure and index of the right child of node n at index i in ratchet tree τ respectively. If the node has no right child these two functions return $\perp$.
- Function $\mathsf{node2index}(\tau, n)$ takes in a ratchet tree τ and a `node` structure n and returns the index i at which n is stored in the list of nodes τ. If no such index exists the function returns $\perp$.

The [BBM+21] has python code for each of these operations in its Appendix, and hence we just use them without specifying their implementation.

Resolution. Intuitively the resolution of a node n is the smallest set of nodes that covers all non-blank leaves in n-s subtree. Let τ be a ratchet tree and n a node in τ. The resolution $\mathsf{Res}(n)$ of node n is a set of nodes defined recursively as follows:

$$\mathsf{Res}(n) = \begin{cases} \{n\} \cup n.\mathsf{unmerged_leaves} & \text{if } n \text{ is not blank} \\ \emptyset & \text{if } n \text{ is a blank leaf} \\ \bigcup_{n' \in C(n)} \mathsf{Res}(n') & \text{if } n \text{ is a blank parent node.} \end{cases}$$

where $C(n)$ are the children of node n.

Direct path. Intuitively a direct path of a node n is the simple path from n to the root node excluding n. Let τ be a ratchet tree and n a node in τ. The direct path of node n, denoted as $\mathsf{dPath}(\tau, n)$ is a list of nodes defined recursively as follows:

$$\mathsf{dPath}(\tau, n) = \begin{cases} [] & \text{if } n \text{ is a root node} \\ [\mathsf{parentNode}(\tau, n)] + \mathsf{dPath}(\tau, \mathsf{parentNode}(\tau, n)) & \text{otherwise.} \end{cases}$$

Co-path. Intuitively a co-path of a node n is a list of sibling nodes of each node in n-s simple path excluding the root node. Let τ be a ratchet tree and n a node in τ. The co-path of node n, denoted as $\mathsf{coPath}(\tau, n)$ is a list of nodes defined recursively as follows:

$$\mathsf{coPath}(\tau, n) = \begin{cases} [] & \text{if } n \text{ is a root node} \\ [\mathsf{siblingNode}(\tau, n)] + \mathsf{coPath}(\tau, \mathsf{parentNode}(\tau, n)) & \text{otherwise.} \end{cases}$$

An example of a direct path and co-path of the leaf node representing member C is given in Figure 4.19.

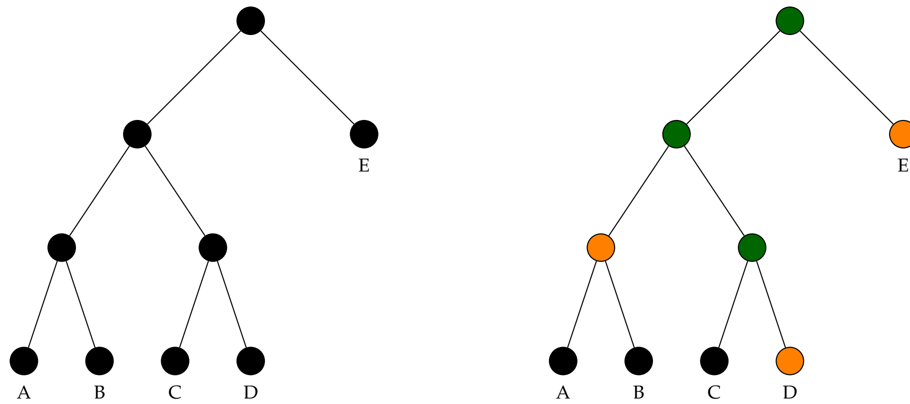


Figure 4.19: Example ratchet tree of a group containing A,B,C,D,E (left) and the coPath and dPath of node representing member C in this tree colored orange and green respectively (right).

Common Path. A common path of two nodes n and n' in a tree τ, denoted as $\mathsf{commonPath}(\tau, n, n')$, is the simple path from $\mathsf{LCA}(, \tau, n, n')$ to the root node.

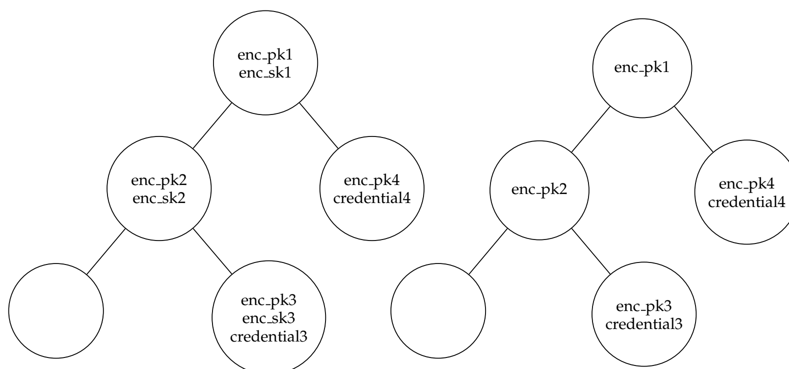


Figure 4.20: Example RT τ (left) and its public state (right). Note that the sibling, children and parent fields of the nodes are made clear from the picture and hence are for readability reasons not included explicitly within the nodes. The remaining 5 fields are all made explicit if present. Note that the node furthest to the left is blank, and hence already has its enc_sk value equal $\perp$.

Public state. Let τ be a ratchet tree. The public state of τ, is a ratchet tree τ' that is identical to τ w.r.t. all fields in all nodes except for the kem_sk field which is set to $\perp$ in τ'. More specifically the function in Figure 4.21 returns the public state given a ratchet tree τ. An example of a public state is given in Figure 4.5.

```
get-PS(τ):
  τ' ← τ
  for node in τ':
    node.kem_sk ← ⊥
  return τ'
```

Figure 4.21: Function that computes the public state of the input ratchet tree.

Ratchet tree operations. In this paragraph we describe some more advanced ratchet tree operations we will make use of in Section 4.7.

The procedure AddLeaf takes in a ratchet tree τ and returns a ratchet tree with an additional leaf τ' and the added leaf ℓ node denoted as $(\tau', \ell) \leftarrow$ AddLeaf(τ). Given a ratchet tree τ, AddLeaf is specified as follows:

- Let n be the number of leaves contained in τ. If $2^k = n$ for some $k \in \mathbb{N}_0$ then create two new node structures ℓ and r' with all their fields set to $\perp$. Now attach ℓ to r' as its right child and the root of τ as r'-s left child. The new ratchet tree τ' then has r' as a root, the input ratchet tree τ as the left subtree of r' and a blank leaf ℓ as the right child of r'.
- Otherwise, let r be the root node of τ and τ_L and τ_R be the left subtree and right subtree of r's respectively. Now call AddLeaf(τ_R) that outputs a new tree τ'_R. Return the ratchet tree τ' with a r as a root, τ_L as its left subtree and τ'_R as its right subtree.

The procedure Trunc takes in a ratchet tree τ and returns a truncated ratchet tree τ' such that its rightmost leaf is non-blank denoted as $\tau' \leftarrow$ Trunc(τ) and specified as follows:

- Let v be the rightmost leaf in τ and v' its sibling. If v is blank and is not the root then remove v as well as its parent from τ and place v' where the parent was. Use this new ratchet tree τ' to call Trunc(τ') and return this call's output.
- Otherwise return the input ratchet tree τ.

The procedure dPathBlank takes in a ratchet tree τ and a client identity ID and returns a ratchet tree τ' that is a copy of τ with all nodes in dPath(τ,n) blanked where n is the leaf node representing client identified by ID. This is denoted as $\tau' \leftarrow$dPathBlank(τ, ID).

Let n be the leaf node representing a client with identity ID. The procedure Unmerge takes in a ratchet tree τ and a client identity ID and returns a ratchet tree τ' that is a copy of τ such that each non-blank and not root node v in dPath(τ,n) has n added to v.unmerged_leaves. This is denoted as $\tau' \leftarrow$Unmerge(τ, ID).

The procedure BlankNode takes in a ratchet tree τ and a client identity ID and returns a ratchet tree τ' that is a copy of τ with the leaf node representing client ID blanked, denoted as $\tau' \leftarrow$BlankNode(τ, ID).

The procedure SearchNode takes in a ratchet tree τ and a client identity ID and returns the node n representing the client identified by ID. This is denoted as $n \leftarrow$SeachNode(τ, ID). If no such node exists SearchNode returns $\perp$.

4.5.1 Ratchet tree invariants

In this section we introduce the invariants the ratchet trees across the group should satisfy (assuming all group members are in the same epoch).[11] These invariants govern the conditions under which a `node` structure has its fields set (or not) to $\perp$ and which fields contain the same value across group members. We end this section by providing some motivation behind these invariants whose purpose will become even more clear in the sections succeeding Section 4.5.

Let G be a group and $m \in G$ be a member of group G with $m.s$ denoting the `state` structure of client m. Let n denote the number of nodes in $m.s$.ratchet_tree. We use m_{leaf} to denote the leaf node representing member m in $m.s$.ratchet_tree and m_{root} to denote the root node in $m.s$.ratchet_tree.

Invariant 1. Given any group G, all its members' ratchet trees have the same public state.

$$\forall G : \forall m, m' \in G : \text{get-PS}(m.\text{s.ratchet_tree}) = \text{get-PS}(m'.\text{s.ratchet_tree}). \tag{4.1}$$

Invariant 2. Given any two members m and m' of a group and any node v, if v-s KEM secret key is present (not equal $\perp$) in both m-s and m'-s ratchet tree then v in m-s ratchet tree and v is m'-s ratchet tree has the same KEM secret key value.

$$\begin{gathered}\forall G : \forall m, m' \in G : \forall i \in \text{range}(n) : m.s.\text{ratchet_tree}[i].\text{kem_sk} = \perp \vee \\ m'.s.\text{ratchet_tree}[i].\text{kem_sk} = \perp \vee m.s.\text{ratchet_tree}[i].\text{kem_sk} = m'.s.\text{ratchet_tree}[i].\text{kem_sk}.\end{gathered} \tag{4.2}$$

Invariant 3. In a ratchet tree, only non-blank leaf nodes have their `Credential`

[11] If one of the members is offline for two epochs, clearly that member's ratchet tree would be behind. However after the offline member processes the two commit messages that initiated the two epochs it missed while it was offline, its ratchet tree will also be in sync with the ratchet trees of other members.

structures not equal $\bot$.

$$\forall G : \forall m \in G : \forall i \in \mathsf{range}(n) : m.s.\mathsf{ratchet_tree}[i].\mathsf{credential} \neq \bot \iff m.s.\mathsf{ratchet_tree}[i] \text{ is a non-blank leaf.} \tag{4.3}$$

Invariant 4. In a ratchet tree, only the non-blank parent nodes have their $\mathsf{unmerged_leaves}$ not equal $\bot$.

$$\forall G : \forall m \in G : \forall i \in \mathsf{range}(n) : m.s.\mathsf{ratchet_tree}[i].\mathsf{unmerged_leaves} \neq \bot \iff m.s.\mathsf{ratchet_tree}[i] \text{ is a non-blank parent node.} \tag{4.4}$$

Invariant 5. A non-blank parent node in a ratchet tree must have its KEM public key and unmerged leaves present (i.e. not equal $\bot$). Note that the unmerged leaves may be an empty set ($\emptyset$) however.

$$\forall G : \forall m \in G : \forall i \in \mathsf{range}(n) : m.s.\mathsf{ratchet_tree}[i].\mathsf{kem_pk} \neq \bot \land m.s.\mathsf{ratchet_tree}[i].\mathsf{unmerged_leaves} \neq \bot \iff m.s.\mathsf{ratchet_tree}[i] \text{ is a non-blank parent node.} \tag{4.5}$$

Invariant 6. A non-blank leaf node in a ratchet tree must have its KEM public key and credential present (i.e. not equal $\bot$).

$$\forall G : \forall m \in G : \forall i \in \mathsf{range}(n) : m.s.\mathsf{ratchet_tree}[i].\mathsf{kem_pk} \neq \bot \land m.s.\mathsf{ratchet_tree}[i].\mathsf{credential} \neq \bot \iff m.s.\mathsf{ratchet_tree}[i] \text{ is a non-blank leaf node.} \tag{4.6}$$

Invariant 7. The KEM secret key of a non-blank node v in the ratchet tree is known to a member of the group if and only if that member's leaf is a descendant of v and is not in $v.\mathsf{unmerged_leaves}$.

$$\forall G : \forall m \in G : \forall i \in \mathsf{range}(n) : m.\mathsf{ratchet_tree}[i].\mathsf{kem_sk} \neq \bot \iff m_{leaf} \notin m.\mathsf{ratchet_tree}[i].\mathsf{unmerged_leaves} \land m_{leaf} \text{ is a descendant of } m.\mathsf{ratchet_tree}[i]. \tag{4.7}$$

Invariant 8. All members of a group know the root node's KEM key pair.

$$\forall G : \forall m \in G : m_{root}.\mathsf{kem_pk} \neq \bot \land m_{root}.\mathsf{kem_sk} \neq \bot. \tag{4.8}$$

In summary the above ratchet tree invariants ensure that the ratchet trees across the group members are identical, except for the KEM secret key values of nodes, which if present have the same value and otherwise $\bot$. Therefore, if any group member modifies its own ratchet tree, then that member needs to ensure the ratchet tree invariants hold, by essentially sending this new local information to the group. This way other members can update their ratchet tree appropriately and the ratchet tree invariants hold once again on this new set of group ratchet trees. We say that the group's ratchet

trees are in sync if the ratchet tree invariants hold on the set of group ratchet trees.

Motivation behind invariants

Let G be a group and m a member of that group G. The first invariant promises that all members' ratchet trees have the same public state, which implies that all members know the same KEM public keys. Let n be a node in m's ratchet tree.

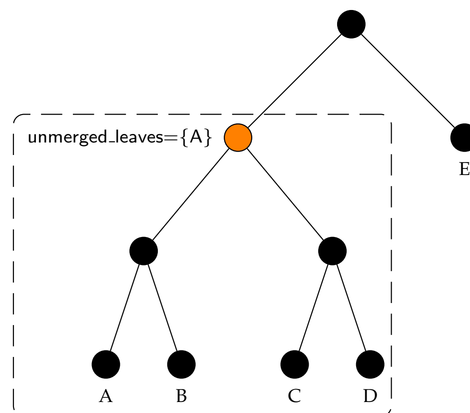


Figure 4.22: If a client uses the orange node's KEM public key to create a ciphertext, then members B,C,D know the corresponding KEM secret key and can decrypt that ciphertext. Because client A is in the unmerged_leaves of the orange node, A does not know KEM secret key of the orange node, and hence is unable to decrypt.

If n is not blank then by invariant 5 and 6 we have $n.\mathsf{kem_pk} \neq \perp$ which m can use to encrypt a message of its choosing. All the members who know $n.\mathsf{kem_sk}$ will be able to decrypt this message. According to invariant 7 the exact members who have $n.\mathsf{kem_sk} \neq \perp$ for a non-blank parent node n are the members represented by the leaves (in the subtree rooted at n) not in $n.\mathsf{unmerged_leaves}$. In particular because of invariant 8, if a member uses the KEM public key of its ratchet tree's root node to encrypt a message, all members will be able to decrypt and obtain that message successfully.

Now let n be a parent node (blank or non-blank) in the tree. To encrypt the same secret to all members represented by leaves in the subtree rooted at n we can again ask ourselves who knows $n.\mathsf{kem_sk}$. If n is non-blank and $n.\mathsf{unmerged_leaves} = \emptyset$ then we can encrypt the secret under $n.\mathsf{kem_pk}$ and know that all members in n's subtree will be able to decrypt. Therefore in this case we would only need to form a single ciphertext. However if $n.\mathsf{unmerged_leaves} \neq \emptyset$ then there are some members in n's subtree that can not decrypt a ciphertext created under $n.\mathsf{kem_pk}$. Moreover if n is blank

then by definition no one in the group knows neither n.kem_sk nor n.kem_pk. Clearly encrypting under one KEM public key will not suffice. To ensure that we use the minimal number of KEM public keys in these cases we use the resolution of node n. Namely $\mathsf{Res}(n)$ returns the smallest set ℓ of non-blank nodes that cover all leaves in the subtree rooted at n. To encrypt to all leaves in this subtree, for each node $n' \in \ell$ client m will encrypt the secret using n'.kem_pk, in turn forming a total of $|\ell|$ ciphertexts.

4.6 Group Context

A `GroupContext` structure defined in Figure 4.23 represents a summary of all the information common across all group members' states. The GID is a group identifier, epochID is an integer representing an epoch, the tree_hash is a digest of the public state of a ratchet tree and confirmed_transcript_hash is a digest of commit messages. As mentioned in Section 4.4 each client m who is part of some group G maintains a `GroupContext` structure as part of its state. Let $m.s$ be the state of m. Then $m.s$.group_context.GID represents the group identifier of G, the $m.s$.group_context.epochID represents m's current epoch, $m.s$.group_context.tree_hash is essentially a hash over the public state of $m.s$.ratchet_tree; Finally $m.s$.group_context.confirmed_transcript_hash contains a running hash over the commit messages that led to the current m-s state $m.s$.

```
struct GroupContext:
    GID
    epochID
    tree_hash
    confirmed_transcript_hash
```

Figure 4.23: `GroupContext` structure.

Like the ratchet tree structure and group secrets, a client's `GroupContext` structure also changes if and only if the client processes a Commit message (or it initiates a group or it processes a Welcome message). The exact way in which these fields are computed will be explained in Section 4.7 when we explain how Commit message processing occurs.

4.7 Handshake and Application message plaintext

As we know (from Section 4.3) MLS has 3 different types of messages: Handshake messages (Proposal and Commit), Application messages and Welcome messages. The Handshake and Application messages are represented by an `MLSmessage` structure shown in Figure 4.24.

The first 4 fields represent the associated data of the message: GID is a group identifier, epochID is an integer representing an epoch, content_type is a string that can take on 3 possible forms: "proposal", "commit" and "app" and user_ad is a string representing any data the creator of the message specified and wanted only integrity protected. The enc_metadata and enc_data are NAEAD ciphertexts over some metadata and the underlying plaintext respectively.

The associated data used in both NAEAD encryptions are all the first four fields in the MLSmessage structure combined. The key and nonce pairs used to produce enc_metadata are (in part) derived from the sender_data_secret stored in a client's state. The key and nonce pairs used to produce enc_data of a handshake or application message, are derived from the hand_ratchet_state and app_ratchet_state respectively (also stored in a client's state, see Section 4.4). The details of how the metadata and plaintext key nonce pairs are derived are deferred to Section 4.10, since we will need the understanding of the MLS key schedule (Section 4.8) and secret tree (Section 4.9).

```
struct MLSmessage:
  GID
  epochID
  content_type
  user_ad
  enc_metadata
  enc_data
```

Figure 4.24: MLSmessage structure. Both (confidential) handshake and application messages have this structure.

Given all this, creating a new or processing a received handshake or application message (represented by MLSmessage) can be viewed as the following two-step procedure. Namely, a handshake or application message *msg* is created by first creating the plaintext underlying *msg*.enc_data. Then, using the results of this step, the *msg*.enc_data and all the other fields in *msg* are created. Similarly, a handshake or application message *msg* is processed by first processing (decrypting) the *msg*.enc_metadata and *msg*.enc_data to obtain the underlying metadata and plaintext after which the plaintext itself is processed (using the results of the first step). We refer to the creating and processing of the MLSmessage structure excluding the plaintext as the creating and processing of the framing.

In this section, we focus on the plaintext of the Handshake and Application messages (MLSmessage structures). First, we define the MLSCiphertextContent structure used to represent the plaintext underlying MLSmessage structure's enc_data. We then proceed to describe how this MLSCiphertextContent

structure is created (prior to sending) and how it is processed (post receival). Finally we explain how the Proposal and Commit structures (used to represent the 'content' of a Handshake or Application message) are formed and processed. The creation and processing of the framing will be explained in Section 4.10. We postpone it until then, because we need the understanding of the MLS key schedule (Section 4.8) and secret tree (Section 4.9).

4.7.1 Handshake and Application plaintext

The plaintext underlying the MLSmessage structure's enc_data (Figure 4.24) is an (encoded) MLSCiphertextContent structure defined in Figure 4.58. An MLSCiphertextContent structure contains commit, proposal and application fields which represent the so called content of a Commit, Proposal and Application message being constructed (represented by an MLSmessage structure) respectively.

More specifically the commit is a Commit structure (see Section 4.7.3) representing the content of a Commit message, proposal is a Proposal structure (see Section 4.7.2) representing the content of a Proposal message and application is a string representing the IM application content the creator of the message wishes to send to the group. At any point in time an MLSCiphertextContent structure can only have one of these three fields be present (not equal $\perp$). The field that is present is the message type (Proposal, Commit or Application) the MLSCiphertextContent structure will be used for.

struct MLSCiphertextContent:	struct MLSPlaintextTBS:
commit	group_context
proposal	GID
application	epochID
signature	sender
confirmation_tag	user_ad
	content_type
	commit
	proposal
	application

Figure 4.25: MLSCiphertextContent structure and MLSPlaintextTBS structure.

In addition to these fields, an MLSCiphertextContent structure contains a signature signature produced over the (encoded) MLSPlaintextTBS structure (defined in Figure 4.25) using the creator's signing secret key. The confirmation_tag field is only present (not equal $\perp$) if and only if the commit field is itself not $\perp$. If present, the confirmation_tag is a MAC tag computed using

the creator's confirmation_key over the creator's current confirmed_transcript (stored in the group_context field of the creator's `state` structure).

The `MLSPlaintextTBS` structure contains the following. The group_context is a `GroupContext` structure, GID is a group identifier, epochID is an integer representing an epoch and sender is a client identifier. The user_ad is a string and content_type is a string that can take on 3 possible forms: "proposal", "commit" and "app" (similarly to the fields present in `MLSmessage` structure). The commit, proposal and application are exactly the same as the fields in `MLSCiphertextContent` structure; exactly one of these 3 fields can be not $\perp$ and now the field that is present needs to correspond to the value of the content_type string.

Plaintext creation In this section, we explain how the `MLSCiphertextContent` structure is created. Let a be a client, with state $a.s$, who wishes to create a Handshake or Application message, and hence starts by creating the `MLSCiphertextContent` structure. To form the `MLSCiphertextContent` structure, a first creates an empty `MLSCiphertextContent` structure ptx by doing:

$$ptx \leftarrow \texttt{MLSCiphertextContent}(\perp, \perp, \perp, \perp, \perp) \quad (4.9)$$

that it then proceeds to populate field-by-field as follows. Depending on whether a wishes to create a Proposal, Commit or Application message, a will either create a `Proposal` structure (according to Section 4.7.2) and assign it to ptx.proposal, create a `Commit` structure (according to Section 4.7.3) and assign it to ptx.commit or create a string respectively (representing the IM application data a wants to send) and assign it to ptx.application respectively. Recall that a can only populate one of these three ptx fields in order to ensure the 'single present' rule holds. Note that, as described in Sections 4.7.2 and 4.7.3, creating of a `Proposal` or `Commit` structure modifies a-s state in order for some pending secrets to be saved.

After having populated ptx.proposal, ptx.commit or ptx.application (and possibly modifying its state $a.s$), client a proceeds by choosing some string user_ad, which it wants to only integrity protect by the `MLSmessage` structure (containing ptx's encryption). It calls form_Plaintext_Signature(a.s, user_ad, ptx) defined in Figure 4.26 and sets ptx.signature to this call's output.

The algorithm form_Plaintext_Signature takes in a client's state state, user specified associated data and its current `MLSCiphertextContent` structure and outputs a signature over: (1) a's current group_context, (2) the group identifier corresponding to the group a is a member of, (3) a's current epoch, (4) a's identity, (5) the associated data a specified, (6) the type of message a wishes to form and (7) the content of the message a wishes to create. Some of the values of the signed fields are also included in the associated data part of the entire `MLSmessage` structure that would contain the encryption of

ptx; others are part of ptx; and some (sender) underlay the enc_metadata of the MLSmessage structure encrypting ptx (see Section 4.10).

```
form_Plaintext_Signature(state, user_ad , ptx):
  id ← state.credential.ID
  ep ← state.group_context.epochID
  gid ← state.group_context.GID
  gctx ← state.group_context
  if ptx.commit ≠ ⊥:
    type ← "commit"
  if ptx.proposal ≠ ⊥:
    type ← "proposal"
  if ptx.application ≠ ⊥:
    type ← "app"
  ptx' ← MLSPlaintextTBS(gctx, gid, ep, id, user_ad, type, ptx.commit, ptx.proposal, ptx.application)
  signature ← state.DS.Sign(state.sign_sk, ⟨ptx'⟩)
  return signature
```

Figure 4.26: form_Plaintext_Signature algorithm.

Finally a forms the ptx.confirmation_tag as follows. If a did not choose to create a plaintext for a Commit message then ptx.confirmation_tag is set to $\perp$ as said before. If however a did choose to create a Commit message, then a calls form_Plaintext_MAC($a.s$) defined in Figure 4.27 and sets ptx.confirmation_tag to this call's output. The form_Plaintext_MAC algorithm takes in a state structure and returns a mac tag over the confirmed_transcript of the GroupContext structure stored in the input state.

```
form_Plaintext_MAC(state):
  MAC ← state.ciphersuite.MAC
  gctx ← state.group_context.confirmed_transcript
  k ← state.confirm_key
  return MAC.Mac(k,gctx.confirmed_transcript)
```

Figure 4.27: form_Plaintext_MAC algorithm.

Therefore the first step of creating an MLSmessage structure can be summarised as given by the create_Step1 algorithm in Figure 4.28. It takes in a string user_ad, a Commit structure (or $\perp$) commit, a Proposal structure (or $\perp$) proposal and a bit-string (or $\perp$) application and produces all the information that needs to be passed to the second stage of creation (covered in Section 4.10).

Client a with state $a.s$ would therefore (depending on which kind of message it wants to send) form either the Commit structure commit, Proposal structure proposal (according to Sections 4.7.3, 4.7.2 respectively) or a bit-string application (setting the rest to $\perp$). As explained before, the creation of these may modify the state of $a.s$. Then a would choose some data it only wants integrity protection for user_ad and call create_Step1($a.s$, user_ad, com-

mit, proposal, application) to execute the first step of creating an `MLSmessage` structure. The output of this call is then used by a to construct the framing of the `MLSmessage` structure as covered in Section 4.10.3.

```
create_Step1(user_ad, commit, proposal, application):
  ptx ← MLSCiphertextContent(commit, proposal, application, ⊥, ⊥)
  signature ← form_Plaintext_Signature(state,user_ad,ptx)
  if commit ≠ ⊥ :
    tag ← form_Plaintext_MAC(state)
  else:
    tag ← ⊥
  ptx.signature ← signature
  ptx.confirmation_tag ← tag
  return (ptx, user_ad)
```

Figure 4.28: create_Step1 algorithm.

Plaintext processing In this section we explain how the processing of the plaintext (underlying the `MLSmessage` structure's enc_data) occurs. Let a be a client who received a Handshake or Application message and has managed to process the frame successfully.[12] Moreover, we assume that the frame processing step (much like we did in the create_Step1 algorithm) passes on all information we need to process the plaintext. Namely, the frame processing will supply us with the sender's identity ID, sender specified authenticated data user_ad, the content_type claimed by the sender and of course the extracted `MLSCiphertextContent` structure ptx representing the actual plaintext. How this information is extracted by the frame processing is explained in Section 4.10.

```
verifyContentAndSignature(state, user_ad, id, ptx, content_type):
  ep ← state.group_context.epochID
  gid ← state.group_context.GID
  gctx ← state.group_context
  if ptx.commit ≠ ⊥:
    type ← "commit"
  if ptx.proposal ≠ ⊥:
    type ← "proposal"
  if ptx.application ≠ ⊥:
    type ← "app"
  if type ≠ content_type
    return false
  ptx' ← MLSPlaintextTBS(gctx, gid, ep, id, user_ad, type, ptx.commit, ptx.proposal, ptx.application)
  n ← SearchNode(state.ratchet_tree, id)
  b ← state.DS.Vfy(n.sign_pk, ⟨ptx'⟩, ptx.signature)
  return b
```

Figure 4.29: verifyContentAndSignature algorithm.

Given this information client a with state $a.s$ starts by verifying that in-

[12]We assume that the frame has been processed successfully as otherwise processing would terminate.

deed only one of the three possible contents ptx.commit, ptx.proposal and ptx.application is present. It then goes on to verify ptx's signature ptx.signature and if the type of content present in ptx matches the declared type content_type by calling the method verifyContentAndSignature($a.s$, user_ad, ID, ptx, content_type) defined in Figure 4.29.

The verifyContentAndSignature procedure takes in a `state` structure, string user_ad, client identifier id, `MLSCiphertextContent` structure ptx and a string content_type that can take on 3 possible forms: "proposal", "commit" and "app" and returns a boolean value.

If the verification of either the content_type or signature fails, then the client aborts the plaintext processing. If the signature and type of content in ptx are valid, a continues to verify the mac tag ptx.confirmation_tag (for commit messages) by simply calling the verify_Plaintext_MAC(state, ptx.tag) algorithm defined in Figure 4.30. The verify_Plaintext_MAC algorithm takes in a `state` structure and mac tag tag and returns boolean.

```
verify_Plaintext_MAC(state, tag):
  MAC ← state.ciphersuite.MAC
  gctx ← state.group_context.confirmed_transcript
  k ← state.confirm_key
  return MAC.Vfy(k,gctx.confirmed_transcript,tag)
```

Figure 4.30: verify_Plaintext_MAC algorithm.

Again if the mac tag verification fails then the plaintext processing abort. Otherwise, it continues to process the content part of plaintext ptx, i.e. the field that is present out of the ptx.commit, ptx.proposal and ptx.application fields.

Let us assume ptx.application is present. Then the data contained within ptx.application would be forwarded to the IM application and the processing of an Application message is finished successfully.

Let us assume ptx.proposal is present. Then client a simply buffers the `Proposal` structure contained in ptx.proposal locally. Client a delays processing the `Proposal` structure until it wishes to create a Commit message.

Let us assume ptx.commit is present. The client processes the `Commit` structure contained in ptx.commit, according to Section 4.7.3. The result of successfully processing[13] a `Commit` structure is a state s that now contains a new ratchet tree s.ratchet_tree and a new commit secret s.commit_secret. Client a now overwrites its state $a.s$ with s.

Recall (from Section 4.4) that the group secrets of a client are modified if and only if the client processes a Commit message (or it initiates a group

[13] If the processing of the ptx.commit fails, then the processing of the entire Commit message naturally fails too.

or it processes a Welcome message). Moreover, as mentioned in Section 4.6, the group_context of a client also changes if and only if the client processes a Commit message (or if it initiates a group or if it processes a Welcome message). Therefore client a now uses its updated state $a.s$ to modify its group_context and the remaining group secrets (init_secret, welcome_secret, encryption_secret, sender_data_secret, confirmation_key).

Client a first updates its `GroupContext` structure $a.s$.group_context by forming an `MLSPlaintextCommitContent` structure, defined in Figure 4.31 based on the received Commit message. The `MLSPlaintextCommitContent` structure groups fields that are present (not $\bot$) in either: (1) the associated data part of the received Commit message (first 4 fields of the `MLSmessage` structure), (2) data underlying the enc_metadata field of the Commit message (see Section 4.10) or (3) data underlying the enc_data field of the Commit message.

```
struct MLSPlaintextCommitContent:
  GID
  epochID
  sender
  user_ad
  content_type
  commit
  signature
```

Figure 4.31: `MLSPlaintextCommitContent` structure.

More concretely a forms the `MLSPlaintextCommitContent` structure by calling form_Body($a.s$, ptx, ID, content_type, user_ad), defined in Figure 4.32, and sets body to this call's output. The form_Body algorithm takes a `state` structure state, an `MLSCiphertextContent` structure ptx, a client identifier sender and two strings content_type and user_ad (first of which can take on only three values "proposal", "commit" or "app") and produces the corresponding `MLSPlaintextCommitContent` structure.[14]

```
form_Body(state,ptx,sender,content_type, user_ad):
  ep ← state.group_context.epochID
  gid ← state.group_context.GID
  body ← MLSPlaintextCommitContent(gid, ep, sender, user_ad, content_type, ptx.commit, ptx.signature)
  return body
```

Figure 4.32: form_Body algorithm.

[14]Note that the group identifier and epoch are used from client a's state, instead of being passed as an argument by a successful frame processing. As we shall see in Section 4.10, a successfully processed frame must have the same group identifier and epoch as the client receiving the Handshake or application message.

Finally, client a calls update_GroupContext($a.s$,body,ptx.confirmation_tag), defined in Figure 4.33, and sets $a.s$ to the output of this call. The algorithm update_GroupContext takes a `state` structure state, an `MLSPlaintextCommitContent` structure and a mac tag and produces a new `state` structure (which contains an updated `GroupContext` structure and confirmation_tag and is otherwise identical to the input state). The new `GroupContext` structure essentially has its epochID incremented by 1 (since it corresponds to the new epoch initiated by the received Commit message), its tree_hash now contains a digest of the new ratchet tree (produced by the processing of the `Commit` structure) and its confirmed_transcript hash now incorporates the body corresponding to the received Commit message. The confirmation_tag is overwritten to contain the ptx.confirmation_tag contained in the received Commit message.

```
update_GroupContext(state,body,tag):
  gctx ← state.group_context
  gctx.epochID + +
  H ← state.ciphersuite.H
  τ ← state.ratchet_tree
  cth ← gctx.confirmed_transcript_hash
  gctx.tree_hash ← TH(H, τ, Root(τ))
  ith ← H(⟨cth, state.confirmation_tag⟩)
  gctx.confirmed_transcript_hash ← H(⟨ith, body⟩))
  state.group_context ← gctx
  state.confirmation_tag ← tag
  return state
```

Figure 4.33: update_GroupContext algorithm.

The update_GroupContext algorithm makes use of TH in order to update the tree_hash (which will now correspond to the new ratchet tree produced by the processed `Commit` structure). The TH algorithm is deterministic and takes in a hash function H, a ratchet tree τ and a `node` structure n and outputs a digest (using H) of the Public state of the subtree rooted at n in τ. It is defined recursively, starting with the leaves as follows:

$$\mathsf{TH}(\mathsf{H},\tau,n) = \begin{cases} \langle \mathsf{node2index}(\tau,n), n.\mathsf{kem_pk}, n.\mathsf{credential}\rangle & \text{if } n \text{ is a leaf} \\ \langle \mathsf{node2index}(\tau,n), \mathsf{pn}(n), \mathsf{hash}(\mathsf{H},\tau,n,\text{`}r\text{'}))\rangle & \text{if } \mathsf{leftNode}(\tau,n) = \bot \\ \langle \mathsf{node2index}(\tau,n), \mathsf{pn}(n), \mathsf{hash}(\mathsf{H},\tau,n,\text{`}l\text{'}))\rangle & \text{if } \mathsf{rightNode}(\tau,n) = \bot \\ \langle \mathsf{node2index}(\tau,n), \mathsf{pn}(n), \mathsf{hash}(\mathsf{H},\tau,n,\text{`}l\text{'})), \mathsf{hash}(\mathsf{H},\tau,n,\text{`}r\text{'}))\rangle & \text{otherwise.} \end{cases}$$

where $\mathsf{pn}(n)$, takes in a `node` structure n and outputs the bit-string:

$$\mathsf{pn}(n) = \langle n.\mathsf{kem_pk}, n.\mathsf{unmerged_leaves}\rangle$$

and $\mathsf{hash}(\mathsf{H},\tau,n,side)$ takes in a hash function H, a ratchet tree τ, a `node` structure n and $side \in \{\text{`}l\text{'}, \text{`}r\text{'}\}$ and returns the following:

$$\mathsf{hash}(\mathsf{H},\tau,n,side) = \begin{cases} \mathsf{H.Ev}(\mathsf{TH}(\mathsf{H}, \mathsf{rightNode}(\tau,n),\tau)) & \text{if } side = \text{`}r\text{'} \\ \mathsf{H.Ev}(\mathsf{TH}(\mathsf{H}, \mathsf{leftNode}(\tau,n),\tau)) & \text{if } side = \text{`}l\text{'}. \end{cases}$$

Note that $\mathsf{TH}(\mathsf{H}, \tau, n)$ incorporates all fields apart from the KEM secret key kem_sk of each node in the subtree rooted at n in τ, which is exactly what the Public state of the subtree rooted at n in τ would contain.

Lastly, in order to update the remaining group secrets, client a runs the MLS key schedule, described in Section 4.8, on $a.s.$commit_secret, $a.s.$group_context and $a.s.$init_secret. Note, the first two values are new, i.e. they belong to the new epoch being initiated by the processed Commit message; whilst $a.s.$init_secret is an old value (belonging to the previous epoch). The $a.s.$init_secret will be overwritten by the MLS key schedule, just like all the other group secrets, to store values corresponding to the new epoch.

Once the MLS key schedule derives the remaining group secrets of the new epoch, the (new) $a.s.$encryption_secret will be used by the secret tree (see Section 4.9) to initialise the new epoch's sending and receiving state in $m.s.$hand_ratchet_state and $m.s.$app_ratchet_state. Of course, since a processed Commit message initiates a new epoch, all counters $a.s.$com_ctr, $a.s.$prop_ctr, $a.s.$hand_generation, $a.s.$app_generation will be reset to 0. Moreover all Commit messages pending for approval $a.s.$com_confirm, buffered Proposal messages $a.s.$prop_confirm and their corresponding (pending) secrets $a.s.$commit_pending and $a.s.$prop_pending will be erased, since they belong to an old epoch (and the only active sending epoch is the newest one). With this, the processing of a Commit message is completed successfully.

4.7.2 Content of a Proposal message

There are 3 types of proposal messages: add, update and remove. In our MLS simplification, an operation (add,remove,update) can be proposed (via a proposal message) to a group G by some member of group G, and only the existing members of group G can process it. MLS uses a `Proposal` structure defined in Figure 4.34 to represent the content of any Proposal message type.

```
struct Proposal:
  proposal_type
  add
  update
  remove
```

Figure 4.34: Proposal content structure.

Let p be a `Proposal` structure. Then $p.$proposal_type is a string that can take on 3 possible forms "add", "update" or "remove" and is used to represent the type of Proposal message p corresponds to; $p.$add is an `Add` structure (Figure 4.35), $p.$update is an `Update` structure (Figure 4.38), $p.$remove is a `Remove` structure (Figure 4.42). At any point in time p can have only two

fields present (not equal $\perp$): p.proposal_type and one of p.add, p.update or p.remove fields depending on the string p.proposal_type contains.

We next describe the `Add`, `Update` and `Remove` structures contained within the `Proposal` structure. For each proposal type we also explain the creation and processing of the content part of the proposal message.

Add

Add content creation An Add proposal message is used to suggest an MLS client, who is not yet a member of the group, be added to the group. Let p be a `Proposal` structure of an Add proposal message. Then p-s fields are set as follows: p.proposal_type = "add", p.update = $\perp$, p.remove = $\perp$ and p.add is an `Add` structure defined in Figure 4.35. The p.add.key_package is a `KeyPackage` structure owned by the client suggested to be added. Recall (from step 5 in Section 4.3) the client wishing to propose some client A to be added to the group would fetch A's `KeyPackage` structure stored at the DS KeyStorage service prior to sending the Proposal message. Therefore the p.add.key_package would be set to A's fetched `KeyPackage` structure.

```
struct Add:
    key_package
```

Figure 4.35: `Add` structure.

More specifically let m be a client and let m.s denote its `state` structure and ID be the identity of the client m wishes to add to the group. Then client m will call the Propose-Add-Content(m.s, ID) to create a `Proposal` structure of an Add proposal message:

```
Propose-Add-Content(st, ID):
    keypackage ← get_keypackage(ID, st.version, st.ciphersuite)
    del_keypacakge(ID, st.version, st.ciphersuite)
    add_p ← Add (keypackage)
    p ← Proposal("add", add_p, ⊥, ⊥)
    return (st, p)
```

Figure 4.36: Algorithm responsible for creating an Add proposal message content.

Add content processing Let p be a `Proposal` structure representing the content of an Add proposal message and let m be a client of some group G with `state` structure m.s. Client m processes p by first checking that p.proposal_type = "add". It then picks the leftmost blank leaf n in m.s.ratchet_tree and sets the n.kem_pk field to p.add.key_package.kem_pk and n.credential to p.add.key_package.credential. If no blank leaf exists then m extends its ratchet tree m.s.ratchet_tree by one leaf and assigns p.add.key_package.kem_pk and

p.add.key_package.credential to the added leaf's kem_pk and credential fields respectively.

Let n be the node that is chosen to represent the added member. Then for each non-blank node (excluding the root) n' in dPath(m.s.ratchet_tree, n) client m adds n to n'.unmerged_leaves. There is no need to overwrite the unmerged_leaves field of the root node, since (by invariant 8 in Section 4.5.1) all clients know its KEM secret key. This entire procedure is summarized by the Process-As dd-Content algorithm shown in Figure 4.37.

Let p be an Add proposal message and p.content be its content. We say p **applies** to an MLS client m if and only if p.content.add.key_package is set to be m's `KeyPackage` structure.

```
Process-Add-Content(st, p):
  req p.proposal_type = "add"
  ℓ ← ⊥
  for n in st.ratchet_tree:
    if n is blank and n is leaf:
      ℓ ←n
  if ℓ = ⊥:
    (st.ratchet_tree, ℓ) ← AddLeaf(st.ratchet_tree)
  ℓ.kem_pk ← p.add.keypackage.kem_pk
  ℓ.credential ← p.add.keypackage.credential
  index ← node2index(st.ratchet_tree, ℓ)
  st.ratchet_tree[index] ← ℓ
  ID ← p.add.keypackage.credential.ID
  st.ratchet_tree ← Unmerge(st.ratchet_tree, ID)
  return st
```

Figure 4.37: Processing algorithm for Add proposal content.

Update

Update content creation An Update proposal message is used to suggest that its sender should refresh (update) its KEM key pair. Let p be a `Proposal` structure of an Update proposal message. Then p.proposal_type = "update", p.add $= \bot$, p.remove $= \bot$ and p.update is an `Update` structure defined in Figure 4.38. The p.update.key_package is a newly created `KeyPackage` structure by the sender of the Update proposal message. The goal of an Update proposal is to have each group member m replace the existing KEM public key (in m.ratchet_tree) associated to the Update proposal's sender to the KEM public key contained in the new `KeyPackage` structure. The sender also saves the KEM secret key corresponding to the KEM public key contained in p.update.key_package in its state. This way if p is confirmed by the group (via a Commit message) the sender of p is able to add its new KEM secret key in its ratchet tree at its representative node.

```
struct Update:
  key_package
```

Figure 4.38: `Update` structure.

More specifically let m be a client and let m.s denote its `state` structure. Then client m will call the Propose-Update-Content(m.s) to create a `Proposal` structure of an Update proposal message:

```
Propose-Update-Content(st):
  (sk, pk) ←$ st.ciphersuite.KEM.KGen()
  keypackage ← Form-KeyPackage(st, pk)
  update_p ← Update (keypackage)
  p ← Proposal("update", ⊥, update_p, ⊥)
  st.prop_pending[st.prop_ctr] ← sk
  st.prop_confirm[st.prop_ctr] ← p
  return (st, p)
```

Figure 4.39: Algorithm responsible for creating an Update proposal message content. It makes use of the Form-KeyPackage function algorithm defined in 4.40.

```
Form-KeyPackage(st, kem_pk):
  t ← Lifetime(time.now, time.tomorrow)
  caps ← Capabilities([st.version], [st.ciphersuite])
  m ← ⟨st.version, st.ciphersuite, st.credential, kem_pk, caps, t⟩
  signature ← st.ciphersuite.DS.Sign(st.sign_sk, m)
  keypackage ← KeyPackage(st.version, st.ciphersuite, st.credential, kem_pk, caps, t, signature)
  return keypackage
```

Figure 4.40: Form-KeyPackage algorithm. The time.now and time.tomorrow calls return an integer corresponding to the current time and an integer corresponding to the time in 24 hours respectively. This interval is arbitrary.

Update content processing Let p be a `Proposal` structure representing the content of an Update proposal message and let m be a client of some group G with `state` structure m.s. Client m processes p by first checking that p.proposal_type = "update". It then checks which member's key material should be updated with p, by reading the identity set in the field p.update.key_package.credential.ID. It finds the leaf node n in m.ratchet_tree representing the client with this identity. Subsequently it replaces n.credential with p.update.key_package.credential and n.kem_pk with p.update.key_package.kem_pk. If client m is the one represented by leaf node n then m also sets n.kem_sk to m.s.prop_pending[ctr] where ctr is the group operation number corresponding to p. Finally m blanks all the nodes on dPath(m.s.ratchet_tree, n). This entire procedure is summarized by the Process-Update-Content algorithm shown in Figure 4.41.

```
Process-Update-Content(st, p):
  req p.proposal_type = "update"
  ID ← p.update.keypackage.credential.ID
  n ← SearchNode(st.ratchet_tree, ID)
  n.kem_pk ← p.update.keypackage.kem_pk
  n.credential ← p.update.keypackage.credential
  if ∃ctr : prop_confirm[ctr] = p:
     n.kem_sk ← prop_pending[ctr]
     st.kem_sk ← n.kem_sk
     st.kem_pk ← n.kem_pk
     st.credential ← n.credential
  index ← node2index(st.ratchet_tree, n)
  st.ratchet_tree[index] ← n
  st.ratchet_tree ← dPathBlank(st.ratchet_tree, ID)
  return st
```

Figure 4.41: Processing algorithm for Update proposal content.

Let p be an Update proposal message and p.content be its content. We say p **applies** to an MLS client m if and only if p.content.update.key_package.credential.ID is set to be m's identity.

Remove

Remove content creation A Remove proposal message is used to suggest that a member of the group be removed from the group. Let p be a `Proposal` structure of a Remove proposal message. Then p.proposal_type = "remove", p.add $= \perp$, p.update $= \perp$ and p.remove is a `Remove` structure defined in Figure 4.42. The p.remove.ID is the identity of the client suggested to be removed.

```
struct Remove:
  ID
```

Figure 4.42: `Remove` structure.

More specifically let m be a client and let m.s denote its `state` structure. Then client m will call the Propose-Remove-Content(m.s, ID) to create a `Proposal` structure of a Remove proposal message:

```
Propose-Remove-Content(st, ID):
  p ← Remove (ID)
  return (st, p)
```

Figure 4.43: Algorithm responsible for creating a Remove proposal message content.

Remove content processing Let p be a `Proposal` structure representing the content of a Remove proposal message and let m be a client of some group G with `state` structure m.s. Client m processes p by first checking

that p.proposal_type = "remove". Then it searches its ratchet tree to find a leaf node n such that n.credential.ID = p.remove.ID. Remember since the client identities are unique across all MLS clients, there can be at most one leaf with an identity matching the one contained in p. Client m then sets all the fields of n to $\perp$, turning the node blank. Then for each node n' in the dPath(m.s.ratchet_tree, n) client m blanks n'. Now if n is the rightmost blank node then m truncates its ratchet tree until the rightmost node becomes non-blank.

```
Process-Remove-Content(st, p):
  req p.proposal_type = "remove"
  ID ← p.remove.ID
  st.ratchet_tree ← BlankNode(st.ratchet_tree, ID)
  st.ratchet_tree ← dPathBlank(st.ratchet_tree, ID)
  st.ratchet_tree ← Trunc(st.ratchet_tree)
  return st
```

Figure 4.44: Processing algorithm for Remove proposal content.

Let p be a Remove proposal message and p.content be its content. We say p **applies** to an MLS client m if and only if p.content.remove.ID is set to be m's identity.

4.7.3 Content of a Commit message

Commit content creation

Consider a client m who is a member of some group G. Assume m has buffered all `Proposal` structures within the current epoch (that have been received and processed up until the content) in a list ℓ'. Assume further that m has not been suggested for removal by any of them.[15] In MLS if $|\ell'| > 0$ then m is not allowed to send application messages. This way MLS ensures that group members immediately act on any group operations. Therefore, at this point m can either (case 1) create and send a proposal message itself or (case 2) create and send a commit message to each existing member of G and (if the commit message contains any Add proposals) create and send a welcome message to each member contained in a committed Add proposal. Assume m wishes to do the latter i.e. create a commit message and possibly (if it commits an Add proposal) welcome message. In this section we describe the construction of the commit message content. The creation of welcome messages will be covered in Section 4.11.

A commit message content is represented by the `Commit` structure displayed in Figure 4.45. Let c be a `Commit` structure. Then c.proposals is a list of `Proposal` structures and c.path is an `UpdatePath` structure defined in Fig-

[15] If m is proposed to be removed then it should not create a commit message.

ure 4.45 which consists of a `KeyPackage` structure keypackage and a list of `UpdatePathNode` structures nodes defined in Figure 4.48. The c.path must be present (not equal $\perp$) if the c.proposals contains at least one Remove or Update proposal or if $|c.\text{proposals}| = 0$. Otherwise it can be omitted (equal to $\perp$). Recall, from Section 4.3, the purpose of a Commit message is two-fold. Namely it is used to (1) indicate which proposed group operations go into effect and (2) refresh the group secrets, more specifically the commit_secret, which is then used to derive the remainder of the new group secrets (see Section 4.8). The c.proposals is used to serve the first purpose and c.path is used for the second.

struct Commit:	struct UpdatePath:
proposals	keypackage
path	nodes

Figure 4.45: `Commit` structure which represents the content part of a Commit message and `UpdatePath` structure.

A client m with state m.s creates a `Commit` structure c by following a sequence of steps. It starts by forming the c.proposals. As stated before each client upon receiving a proposal message, processes everything apart from the contained `Proposal` structure, which is buffered. It defers the processing of buffered proposal content (`Proposal` structures) to the point when it wishes to create a Commit message content. The c.proposals list is formed by including each buffered `Proposal` structure p that meets the following conditions:

- p is a valid `Proposal` structure. A `Proposal` structure p is valid if all of the following conditions hold. All the appropriate fields are present given a certain value of the p.proposal_type field. If p.proposal_type = "add" or p.proposal_type = "update" then p.add.keypackage or p.update.keypackage must be a valid `KeyPackage` structure respectively. A `KeyPackage` structure k is valid if and only if k.signature verifies against the k.credential.sign_pk, k.version is equal to the group's version, k.ciphersuite is equal to the group's `Ciphersuite` structure, the k.k.credential.signature verifies against the AS signature key AS_sign_sk and the time at which p is being processed is within the bounds specified in k.lifetime. If p.proposal_type = "remove" then m's ratchet tree must contain a leaf representing client identified by p.remove.ID.
- If there are multiple valid `Proposal` structures that apply to the same client a, the committer m chooses one `Proposal` structure (out of the multiple) and includes only that one in the Commit, considering the rest invalid. For all clients a the committer m must prefer any Remove received, or the most recent Update applying to a if there are no Remove propos-

als. If there are multiple Add proposals for the same client, the committer again chooses one to include at random and considers the rest invalid.

Once having formed the list of `Proposal` structures c.proposals, client m then proceeds to process each element of c.proposals on a **copy of its local ratchet tree** m.s.ratchet_tree[16]. It first processes all Update `Proposal` structures in c.proposals and then all Remove `Proposal` structures. Finally it processes all Add `Proposal` structures in the order listed in c.proposals. So in the example list of `Proposal` structures given in Figure 4.1 client m would process the `Proposal` structures in the following order: 2,3,0,4,1,5,6.

0	1	2	3	4	5	6
Rem(A)	Add(C)	Upd(B)	Upd(D)	Rem(F)	Add(G)	Add(E)

Table 4.1: Example list of `Proposal` structures contained in c.proposals where c is a Commit message. The Remove, Add, Update `Proposal` structures are denoted as Rem(.), Add(.) and Upd(.) respectively where the . is a placeholder for any client identifier.

Now if $|c.\mathsf{proposals}| \neq 0$ and c.proposals contains only Add `Proposal` structures then m is allowed to not populate c.path, i.e. set c.path to $\perp$, and finish forming the `Commit` structure. If m in such a scenario chooses not to populate the c.path field, then m would store the ratchet tree to which the c.proposals were applied, along with the new commit_secret (set to an all-zero bit-string of KDF.Nh bits length) in $m.s.\mathsf{commit_pending}[m.s.\mathsf{com_ctr}]$. Moreover it would save the produced `Commit` structure in $m.s.\mathsf{com_confirm}[m.s.\mathsf{com_ctr}]$. This way, if it receives its own Commit message back from the DS Broadcast service, m can obtain the corresponding ratchet tree and commit_secret.

Otherwise client m proceeds to populate the c.path field by creating an `UpdatePath` structure. Let v_0 be the leaf node representing client m in m.s.ratchet_tree. On a high level in order to create an `UpdatePath` structure client m first samples seed values and uses them to generate fresh KEM key pairs for itself and every node in $\mathsf{dPath}(m.\mathsf{s.ratchet_tree}, v_0)$. Then for every node v on $\mathsf{coPath}(m.\mathsf{s.ratchet_tree}, v_0)$ it encrypts specific information under the KEM public keys of nodes in $\mathsf{Res}(v)$. This encrypted information allows each party in the subtree of v to learn all new KEM secret keys and seed values from v's parent up to the root. The seed value corresponding to the root node is then used to generate the commit_secret of the new epoch this commit message would define. The new KEM public keys and resulting ciphertexts are grouped together to form an `UpdatePath` structure.

More specifically, let client m (acting as a committer) be represented by leaf node v_0 and let $[v_1, v_2, \ldots, v_n]$ (for some $n \in \mathbb{N}_0$) be the nodes on the

[16] It must do it on a copy and not on the original tree since there may be another client m' who is also creating a Commit message at the same time as m. Therefore m does not yet know if its Commit message will be the one defining the next epoch.

direct path of v_0, i.e. $[v_1, v_2, \ldots, v_n] = \mathsf{dPath}(m.\mathsf{s.ratchet_tree}, v_0)$. Client m (acting as a committer) starts by sampling a fresh random 'seed' value $s_0 \leftarrow_{\$} \{0,1\}^{\mathsf{KDF.Nh}}$. Then for all $i \in \{0, 1, \ldots, n\}$ client m using the generated seed s_0 computes a new KEM key pair $(\mathsf{kem_sk}_i, \mathsf{kem_pk}_i)$ for node v_i in the following way:

$$(s_{i+1}, \mathsf{kem_sk}_i, \mathsf{kem_pk}_i) \leftarrow \mathsf{Prg}(m.\mathsf{s}, s_i). \quad (4.10)$$

where Prg is a deterministic algorithm that takes in a `state` structure state and a pseudorandom value s and computes a new pseudorandom value and KEM key pair. It is defined as follows:

```
Prg(state, s):
  KDF ← state.ciphersuite.KDF
  ns ← KDF.Expand(s, ⟨KDF.Nh, "mls10node"⟩ , KDF.Nh)
  s' ← KDF.Expand(s, ⟨KDF.Nh, "mls10path"⟩ , KDF.Nh)
  (kem_sk, kem_pk) ← DeriveKeyPair(ns)
  return (s', kem_sk, kem_pk)
```

Figure 4.46: Prg algorithm specification.

The seed value s_{n+1} is then set to be the new commit_secret value. Finally for all $i \in \{1, \ldots, n\}$, client m sets v_i.unmerged_leaves to $\emptyset$. Now that m has computed a new KEM key pair for its representative node and all nodes in its direct path, m needs to send some information to the other group members in order to keep all members' local ratchet trees in sync.[17]

Concretely, client m needs to send its new KEM public key $\mathsf{kem_pk}_0$ and the KEM public key of all nodes in v_0-s direct path, i.e. $\mathsf{kem_pk}_i$ for all $i \in \{1, \ldots, n\}$ to all members. This is necessary to preserve the first ratchet tree invariant (see Section 4.5.1), which demands that the public state of each group members' ratchet tree is the same.

Moreover, client m needs to ensure that the KEM secret key of any given node in the direct path of v_0 is propagated to the clients eligible to know it. This is again necessary in order to maintain invariant 7 across the group. Since for all $i \in \{1, \ldots, n\}$ $v_i.\mathsf{unmerged_leaves} = \emptyset$ then node v_i-s KEM secret key must be sent to all members represented by a leaf in the subtree rooted at v_i. Equivalently, a given member m' represented by some leaf u is only allowed to learn the new KEM secret keys belonging to nodes in $\mathsf{commonPath}(m.\mathsf{s.ratchet_tree}, v_0, u)$.

Let $[v'_0, v'_1, \ldots, v'_{n-1}]$ be the nodes on the co-path of v_0, i.e. v'_t is the sibling node of v_t for all $t \in \{0, \ldots, n-1\}$. Let $i \in \{1, \ldots, n\}$. Because seed s_i is used to derive the KEM key pair belonging to node v_i in v_0-s direct path

[17] Note that the group we want the ratchet tree invaraints to hold is the group resulting from the commit message we are creating.

(equation 4.10), it is enough to send s_i to each client represented by some leaf u in v'_{i-1}-s subtree and ensure that the member represented by u obtains the KEM secret keys of nodes in $\mathsf{commonPath}(m.\mathsf{s.ratchet_tree}, v_0, u)$.

Of course, client m can not send s_i plainly, since the non-eligible group members could simply eavesdrop on the network and obtain the value of s_i. Recall from Section 4.5.1 that $\mathsf{Res}(v'_{i-1})$ returns the smallest set of non-blank nodes that cover all leaves in the subtree rooted at v'_{i-1}. Therefore it is enough to use the KEM public key of each node in $\mathsf{Res}(v'_{i-1})$ to encrypt s_i to all members represented by some leaf in v'_{i-1}-s subtree. Then for all $i \in \{1, \ldots, n\}$ and for every node $v_j \in \mathsf{Res}(v'_{i-1})$, client m computes:

$$c_{ij} \leftarrow \mathsf{SetSeal}(m.s, v_j.\mathsf{kem_pk}, s_i). \tag{4.11}$$

where SetSeal is defined in Figure 4.47. The SetSeal algorithm takes in a `state` structure state, KEM public key kem_sk and a message m and outputs a ciphertext (a KEM encapsulation enc and NAEAD ciphertext c′ pair). [18]

```
SetSeal(state, kem_pk, m):
  KEM ← state.ciphersuite.KEM
  NAEAD ← state.ciphersuite.NAEAD
  ctx ← ⟨state.group_context⟩
  (k, enc) ← KEM.Encap(kem_pk)
  (key,nonce) ← KeySchedule(state, k)
  c′ ← NAEAD.Enc(key, nonce, ctx, m)
  return (enc, c′)

KeySchedule(state, k):
  KEM ← state.ciphersuite.KEM
  KDF ← state.ciphersuite.KDF
  NAEAD ← state.ciphersuite.NAEAD
  suite ← ⟨KEM, KDF, NAEAD⟩
  psk_id_hash ← KDF.Extract(⟨"HPKE-v1", suite, "psk_id_hash"⟩, " ")
  info_hash ← KDF.Extract(⟨"HPKE-v1", suite, "info_hash"⟩, " ")
  ks_ctx ← ⟨"mode_base", psk_id_hash, info_hash⟩
  secret ← KDF.Extract(⟨"HPKE-v1", suite, "secret"⟩,k)
  key ← KDF.Expand(secret, ⟨NAEAD.Nk, "HPKE-v1", suite, "key", ks_ctx⟩, NAEAD.Nk)
  nonce ← KDF.Expand(secret, ⟨NAEAD.Nn, "HPKE-v1", suite, "base_nonce", ks_ctx⟩, NAEAD.Nn)
  return (key,nonce)
```

Figure 4.47: SetSeal algorithm specification.

Finally client m has all the material it needs to form the `UpdatePath` structure c.path, which encompasses all information m needs to broadcast to the group (for the trees to remain in sync). Namely it creates a `KeyPackage` structure c.keypackage by calling the $\mathsf{Form\text{-}KeyPackage}(m.\mathsf{s}, \mathsf{kem_pk}_0)$ function defined

[18] The SetSeal and OpenSeal (see Figure 4.52) algorithms given here are simplifications of the HPKE algorithms in the base mode given by [BBLW21]. Moreover MLS uses a `GroupContext` structure that is slightly modified from the one that is stored in the client's state.

in Figure 4.40, where m.s is the state of client m. The keypackage is the part of an `UpdatePath` structure responsible for sending m's new KEM public key $\mathsf{kem_pk}_0$ to all other group members.[19]

```
struct UpdatePathNode:
  kem_pk
  enc_path_secret
```

Figure 4.48: `UpdatePathNode` structure.

To form c.path.nodes client m needs to create an `UpdatePathNode` structure for each node in the direct path of v_0 (leaf representing client m). An `UpdatePathNode` structure defined in Figure 4.48 contains a KEM public key kem_pk and a list of ciphertexts enc_path_secret. For all $i \in \{1, \ldots, n\}$ client m creates an `UpdatePathNode` structure *node* where *node*.kem_pk is set to $\mathsf{kem_pk}_i$ and *node*.enc_path_secret is set to contain all ciphertexts encrypting the seed value s_i, i.e. all elements of the set $\{c_{ij} \mid j \in \{0, \ldots, |\mathsf{Res}(v'_{i-1})|\}\}$.

Client m also saves the ratchet tree (produced by applying the c.proposals, generated KEM keys and empty unmerged_leaves) and the produced commit_secret in the $m.s$.commit_pending map (at the current index $m.s$.com_ctr). Moreover it saves the created `Commit` structure in the $m.s$.com_confirm map at the same index at which it saved the ratchet tree and commit_secret pair in $m.s$.commit_pending. This way if the commit message is confirmed by the group, the client (who created it) can obtain the corresponding ratchet tree and commit_secret.[20]

The entire process of creating the `Commit` structure, provided the list of `Proposal` structures has been formed, is summarized in the form_Commit algorithm defined in Figure 4.49. It takes in a `state` structure st, and a list of `Proposal` structures proposals and produces a `state` structure and an `Commit` structure. Note that the computePath algorithm omits setting the path if it can (i.e. if $|\mathsf{proposals}| \neq 0$ and proposals only contains Add proposals). We do this here to illustrate all the possibilities of commit content creation. In reality the client creating a commit message can populate the c.path no matter the case.

[19] Note that this KeyPackage is not published to the DS KeyPackage storage since its not intended to be used to initialize a group. Its purpose is to prove to the members of its existing group the contained KEM public key really belongs to m.

[20] Note that, allowing the creator to encrypt the seed value s_0 under its own KEM public key would hinder the Post-Compromise security of MLS. Namely an attacker who has leaked the state of the committer would then have the KEM public key the committer used to encrypt the seed value to itself.

```
form_Commit(st, proposals):
  (τ,setPath) ← applyProposals(st, proposals)
  if ¬setPath:
    commit_secret ← {0}^KDF.Nh
    commit_pending[com_ctr] ← (τ, commit_secret)
    com_confirm[com_ctr] ← c
    return (st, path)
  KDF ← st.ciphersuite.KDF
  s[.], kem_pk[.], kem_sk[.] ← ⊥
  enc_path_secret[.][.] ← ⊥
  nodes[.] ← ⊥
  v_0 ← SearchNode(τ, st.credential.ID)
  dp ← dPath(τ, v_0)
  s[0] ←$ {0,1}^KDF.Nh
  for i in range(|dp| + 1) :
    (s[i + 1], kem_sk[i], kem_pk[i]) ← Prg(st, s[i])
  commit_secret ← s[|dp| + 2]
  cp ← coPath(τ, v_0)
  for i in range(|cp|) :
    res ← Res(cp[i])
    j ← 0
    for v in res :
      enc_path_secret[i][j] ← SetSeal(st, v.kem_pk, s[i + 1])
      j + +
  keypackage ← Form-KeyPackage(st, kem_pk[0])
  for i in range(|dp|) :
    nodes[i] ← UpdatePathNode(kem_pk[i + 1], enc_path_secret[i])
  path ← UpdatePath(keypackage, nodes)
  τ ← computeTree(τ, v_0, kem_sk, kem_pk, keypackage)
  c ← Commit(proposals, path)
  commit_pending[com_ctr] ← (τ, commit_secret)
  com_confirm[com_ctr] ← c
  return (st, c)
```

Figure 4.49: Algorithm form_Commit. It uses the applyProposals and computeTree functions defined in Figure 4.50.

The form_Commit algorithm makes use of two helper algorithms applyProposals and computeTree defined in Figure 4.50. The applyProposals takes in a `state` structure state, and list of `Proposal` structure proposals. It produces a copy of the ratchet tree contained in state to which proposals have been applied to and a boolean setPath which is set to `true` if and only if the `UpdatePath` structure can be excluded from a `Commit` message containing proposals.

The computeTree algorithm takes in a ratchet tree τ, a `node` structure n (an element of τ), a list of KEM secret keys kem_sk and corresponding KEM public keys kem_pk and a `KeyPackage` structure keypackage. It produces a copy of the ratchet tree τ in which: (1) the KEM keys along the direct path of node n have been replaced by kem_sk and kem_pk, (2) the unmerged_leaves set of each node in n-s direct path has been overwritten with $\emptyset$ and (3) the `Credential` structure and KEM key pair of n has been set to the values contained in keypackage.

```
applyProposals(st, proposals):
  types ← ["update", "remove", "add"]
  setPath ← false
  for type in types:
    for p in proposals:
      if p.proposal_type=type and type="update":
        st ← Process-Update-Content(st,p)
        setPath ← true
      if p.proposal_type=type and type="remove":
        st ← Process-Remove-Content(st,p)
        setPath ← true
      if p.proposal_type=type and type="add":
        st ← Process-Add-Content(st,p)
  if |proposals| = 0:
    setPath ← true
  return (st.ratchet_tree, setPath)
```

```
computeTree(τ, n, kem_sk, kem_pk, keypackage):
  i ← node2index(τ, n)
  τ[i].kem_sk ← kem_sk[0]
  τ[i].kem_pk ← kem_pk[0]
  τ[i].credential ← keypackage.credential
  for j in range(|dp|) :
    i ← node2index(τ, dp[j])
    τ[i].kem_sk ← kem_sk[j + 1]
    τ[i].kem_pk ← kem_pk[j + 1]
    τ[i].unmerged_leaves ← ∅
  return τ
```

Figure 4.50: applyProposals and computeTree algorithms.

Therefore a client m, with state $m.s$, will call computePath($m.s$, proposals) to obtain a new state $m.s$ and a `Commit` structure c containing proposals.

Example We illustrate this process with an example. Namely let client m identified as A with state m.s be in a group containing clients identified as A,B,C,D and E. Assume m started creating a Commit message c and has formed and applied a list of all valid received `Proposal` structures c.proposals to a copy of its m.s.ratchet_tree and is now ready to form the `UpdatePath` structure c.path. Assume that the ratchet tree as a result of applying c.proposals is the one shown in Figure 4.51. Since m identifies as A the leaf node it is represented by is the left-most leaf in this tree annotated as v_0. Hence its direct path is $[v_1, v_2, v_3]$ and its co-path is $[v_0', v_1', v_2']$.

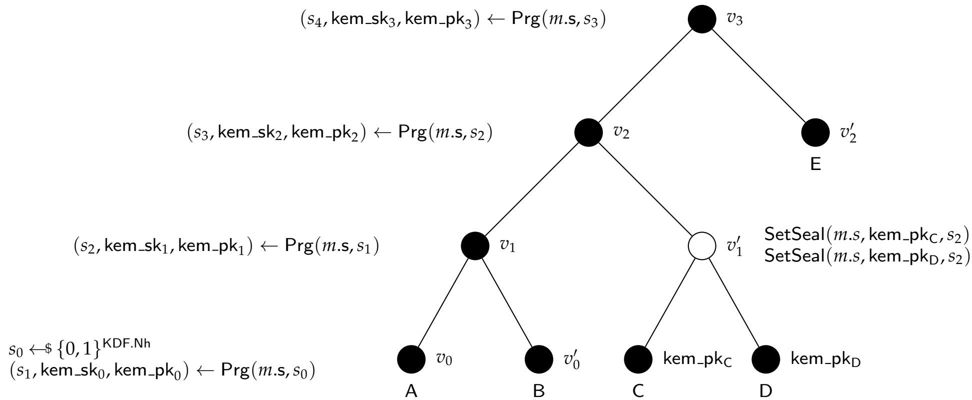


Figure 4.51: Example process of assembling the `UpdatePath` of a commit message. The diagram only illustrates the ciphertexts corresponding to the seed of v_2 and excludes the ciphertexts corresponding to v_1 and v_3 due to their simplicity.

Client m first chooses a random value s_0. Next, m calls the function Prg

for each level $i \in \{0,1,2,3\}$ on the simple path from v_0 to the root v_3 and derives a seed value s_{i+1} for the next level and a new KEM key pair for that level's node v_i. Every seed value s_i in the direct path of v_0 (s_1, s_2, s_3) is then encrypted with setSeal using the KEM public key of each node in the resolution of v_{i-1}-s sibling node v'_{i-1}. This ensures that all leaves in the subtree rooted at v'_{i-1} can learn the value s_i. Because all group members use the same KDF algorithm and equation 4.10 holds, all leaves in the subtree rooted at v'_{i-1} can also compute all new KEM key pairs and seeds from v'_{i-1}-s parent (v_i) up to the root.

So in our example the seed value s_2 corresponding to node v_2 will be encrypted using the KEM public keys of nodes in the resolution of v'_1. Since v'_1 is blank (indicated white in diagrams) then the resolution includes the leaves representing clients C and D. Therefore we will have two ciphertexts encrypting s_2, one using $\mathsf{kem_pk}_C$ and another using $\mathsf{kem_pk}_D$ in order to ensure all clients represented by the leaves in the subtree rooted at v'_1 can obtain s_2 and with it s_3, s_4 and the KEM key pairs corresponding to these seeds.

The seed value s_1 of node v_1 needs to be encrypted to the nodes in the resolution of v'_0 which is just v'_0 itself and similarly s_3 of node v_3 needs to be encrypted to only v'_2. Then c.path will contain a new keypackage c.keypackage of client m binding $\mathsf{kem_pk}_0$ to m and three `UpdatePathNode` structures in c.nodes.

Commit content processing

Let c be a `Commit` structure representing the content of a Commit message and let m be a client of some group G with `state` structure m.s. We consider two cases of Commit message processing: (1) m is not the creator of the received Commit message and (2) m is the creator. If m is the creator, then the map $m.s$.com_confirm will contain the `Commit` structure at some counter value ctr. The client then uses this ctr to obtain the appropriate ratchet tree and commit_secret value saved in $m.s$.commit_pending. It then stores this 'pending' ratchet tree and commit_secret in $m.s$.ratchet_tree and $m.s$.commit_secret respectively. This completes the processing of a `Commit` structure successfully, in case m is its creator. If m is not the creator, i.e. there exists no ctr such that $m.s$.com_confirm[ctr] contains the received `Commit` structure, then client m will proceed to process the Commit message content c as follows. Let ID be the identity of the creator of the `Commit` structure.

Client m first parses the c.proposals field to determine if each `Proposal` structure it contains is valid and that each client is targeted by at most one of the contained `Proposal` structures. Moreover, it uses c.proposals to determine if c.path is supposed to be populated. Recall that c.path must be populated in case c.proposals contains a `Remove` or `Update` structure or if $|c.\mathsf{proposals}| = 0$.

If c.path is supposed to be populated but is not, the processing of the `Commit` structure fails. If c.path does not need to be populated ($|c.\text{proposals}| \neq 0$ and c.proposals only contains `Add` structures) and is not, then m applies the c.proposals to its local ratchet tree $m.s$.ratchet_tree and sets commit_secret to the all zero vector of KDF.Nh length.

If c.path is populated then m applies c.path to its ratchet tree and commit_secret as follows. Let m_{leaf} and v_0 denote m-s representative leaf in its ratchet tree and committer's representative leaf respectively.[21] If no such leaves exist the `Commit` structure processing fails. First the client checks that the `KeyPackage` structure c.path.keypackage is valid and that it indeed corresponds to the creator of the `Commit` structure ID. If it is, m proceeds to extract the KEM public and secret keys from the c.path.nodes component, otherwise the processing fails.

Since m is eligible to know only the KEM secret keys on the common path of v_0 and m_{leaf}, $\mathsf{commonPath}(m.s.\mathsf{ratchet_tree}, v_0, m_{leaf})$, client m determines the least common ancestor w between m_{leaf} and v_0 by computing:

$$w \leftarrow \mathsf{LCA}(\mathsf{m.s.ratchet_tree}, m_{leaf}, v_0). \tag{4.12}$$

By Equation 4.10 it is enough for m to obtain the seed value s corresponding to node w to obtain all KEM secret keys it is allowed to know (by invariant 7).

To obtain s from c.path, client m needs to (1) find a node whose KEM public key was used to encrypt s and (2) whose KEM secret key m knows. To do this, m starts by computing the direct path of v_0 by calling:

$$[v_1, v_2, \ldots, v_n] \leftarrow \mathsf{dPath}(m.s.\mathsf{ratchet_tree}, v_0) \tag{4.13}$$

and v_0-s co-path by executing:

$$[v'_0, v'_1, \ldots, v'_{n-1}] \leftarrow \mathsf{coPath}(m.s.\mathsf{ratchet_tree}, v_0). \tag{4.14}$$

Let i be such that $w = v_i$.[22] Then m takes v'_{i-1} (the unupdated child of w) and computes its resolution $\mathsf{Res}(v'_{i-1})$. The nodes contained within $\mathsf{Res}(v'_{i-1})$ are all nodes whose KEM public keys were used to encrypt s.

Client m then parses $\mathsf{Res}(v'_{i-1})$ until it finds a node whose KEM secret key it knows. If no such node exists, the processing fails since the resolution of v'_{i-1} covers all leaves in v'_{i-1}-s subtree (and m_{leaf} is a leaf in this subtree).

Let u be the node in $\mathsf{Res}(v'_{i-1})$ whose KEM secret key is known to m. Client

[21] Since we know the committer's identity ID, m would simply search its local ratchet tree $m.s$.ratchet_tree until it finds the non-blank leaf v_0 such that v_0.credential.ID.

[22] Such an i exists since w is on v_0-s direct path by definition of common path.

m then uses u.kem_sk to decrypt the ciphertext under u.kem_pk stored in c.path.nodes$[i-1]$.enc_path_secret and obtain the seed value s (corresponding to w). More specifically, since the ciphertext is formed using SetSeal, then m obtains s by executing:

$$s \leftarrow \mathsf{OpenSeal}(m.s, u.\mathsf{kem_sk}, c.\mathsf{path.nodes}[i-1].\mathsf{enc_path_secret}[j]) \qquad (4.15)$$

where c.path.nodes$[i-1]$.enc_path_secret$[j]$ = SetSeal$(m.s, u.$kem_pk$, s)$, for the appropriate j. The OpenSeal algorithm takes in a `state` structure state, KEM secret key kem_sk and a ciphertext c (produced by SetSeal) and outputs the message underlying c.

```
OpenSeal(state, kem_sk, c):
  (enc, c′) ← c
  KEM ← state.ciphersuite.KEM
  NAEAD ← state.ciphersuite.NAEAD
  ctx ← ⟨state.group_context⟩
  k ← KEM.Decap(kem_sk, enc)
  (key,nonce) ← KeySchedule(state, k)
  m ← NAEAD.Dec(key, nonce, ctx, c′)
  return m
```

Figure 4.52: OpenSeal algorithm specification. The keyschedule used is defined in Figure 4.47.

It then uses s to derive each KEM key pair on the simple path from w to the root using Equation 4.10 as well as the new commit_secret. It of course checks that the KEM public keys derived in such a way match the KEM public keys contained in c.path.nodes (corresponding to nodes on the simple path from w to the root).

Finally m updates its local ratchet tree as follows. For each node in $v \in [v_0, v_1, \ldots, v_n]$ client m replaces v.kem_pk with the corresponding new KEM public key contained in c.path.nodes and sets v.unmerged_leaves to $\emptyset$, except for $v = v_0$ (since it is a leaf and has v_0.unmerged_leaves $= \perp$ by Section 4.5.1). Then for each node v in commonPath$(m$.s.ratchet_tree$, v_0, m_{leaf})$, client m overwrites v.kem_sk to contain the new corresponding KEM secret key. Lastly, m sets the `Credential` structure of v_0-s leaf to the one contained in c.path.keypackage.

The output of a successfully processed `Commit` structure is then a `state` structure. This `state` structure is almost identical to $m.s$ before the processing of the `Commit` structure began. The only difference is that the $m.s$.ratchet_tree and $m.s$.commit_secret fields contain the new ratchet tree and commit_secret respectively (obtained from the `Commit` structure processing).

4.8 Key Schedule

A client m with state $m.s$ executes the MLS key schedule only at the beginning of a new epoch. More specifically, it is run right after a receives a Commit message, processes its frame and its `Commit` structure, which updates its $m.s$.ratchet_tree, $m.s$.commit_secret and $m.s$.group_context (see Section 4.7).

The MLS key schedule is a deterministic procedure. It takes the commit_secret extracted from the received Commit message (now stored in $m.s$.commit_secret), the init_secret derived in the previous epoch (presently stored in $m.s$.init_secret) and the new `GroupContext` structure produced as a result of processing the Commit message (now stored in $m.s$.group_context). Given these inputs, the MLS key schedule produces a new value for all the remaining group secrets welcome_secret, encryption_secret, sender_data_secret, confirmation_key and init_secret. The client m then stores these derived values in the corresponding field in its state $m.s$.

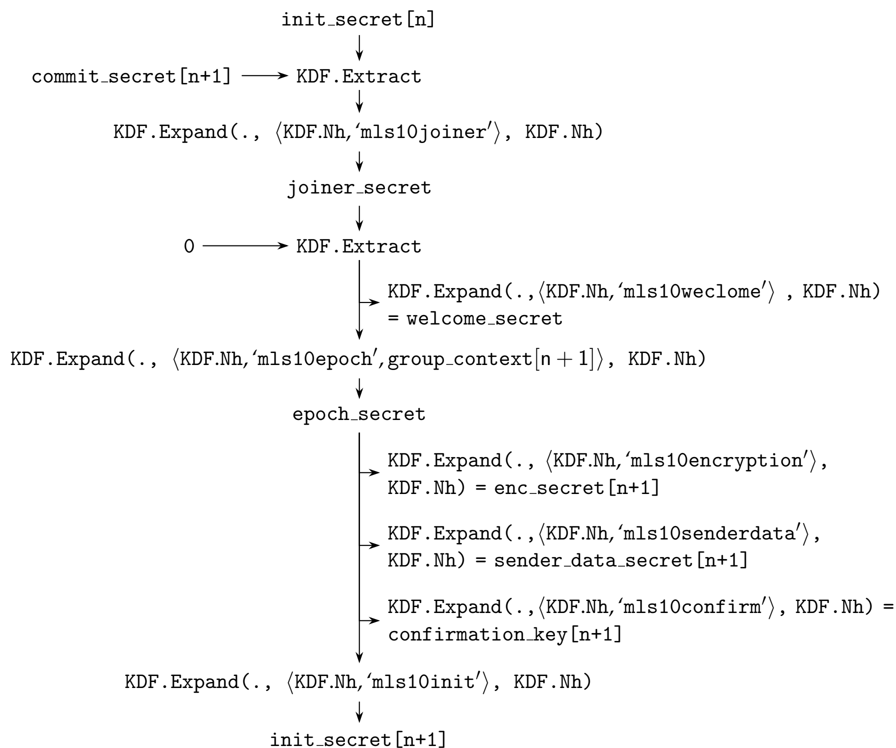


Figure 4.53: MLS Key schedule.

The entire process is summarized in Figure 4.53. The secrets in this figure have an index next to them, used to clarify which epoch the values belong to. That is if n is the epoch m was in prior to receiving the Commit mes-

sage, then we supply the Key Schedule with the init_secret derived in epoch n and the commit_secret and group_context derived in the new epoch n+1 (initiated by the Commit message processing). The MLS key schedule uses the key derivation scheme KDF stored in $m.s$.ciphersuite. The KDF.Extract takes its salt argument from the top and its randomness from the left. The KDF.Expand takes its pseudorandom key from the incoming arrow.

Note that (as explained in Section 4.4), given some client m with state $m.s$, $m.s$.group_context, $m.s$.init_secret and $m.s$.commit_secret are single values, not maps like suggested in Figure 4.53. We store single values instead of a map in order for the scheme to provide better forward-security (see Section 5).

Once the remaining group secrets have been derived, via the MLS key schedule, the new commit_secret is never used again. Therefore a client m, upon running the MLS key schedule will set $m.s$.commit_secret to $\perp$, in this way deleting its value. This is again done in order to facilitate forward-security of MLS (see Section 5). Note that the MLS key schedule overwrites the init_secret it used as input with the init_secret it derives. Since the $m.s$.group_context might as well be a public value (see Section 4.4), and all the other MLS key schedule inputs are overwritten, client m does not need to delete it from its state.[23]

In the sections to follow we explain the semantics of each of the remaining group secrets. In Section 4.9 we explain how the encryption_secret is utilised to initialise the new epoch's sending and receiving state in $m.s$.hand_ratchet_state and $m.s$.app_ratchet_state. In Section 4.10, we describe how the sender_data_secret is used to produce the enc_metadata field in the MLSmessage structure. Finally, in Section 4.11, we explain how the welcome_secret is used to welcome newly added members to the group.

4.9 Secret Tree

Secret Tree. A secret tree is an LBBT whose nodes contain a single bit-string. We adopt the same representation of secret trees as we did for ratchet trees. Namely within pseudocode, we assume that a secret tree is a list containing bit-strings.[24] As in the ratchet tree, bit-strings in leaf nodes are positioned at even-numbered indices, whereas bit-strings in parent nodes are held at odd-numbered indices; starting at the left-most node in the tree at position zero and running from left to right. Given this representation, we use the same functions as in the ratchet tree (see Section 4.5) to obtain the root of the

[23]If it were to erase $m.s$.group_context though, then m would need to save $m.s$.group_context.confirmed_transcript_hash in some field in its state. This is necessary because of the way the next `GroupContext` structure is computed, explained in Section 4.7.1.

[24]We simplify and make the list contain bit-strings instead of `node` structures, since the `node` structure would contain a single bit-string field.

tree ,sibling, left child, right child of any node and index of a given node. If a node's bit-string is not present, the $\perp$ symbol will be stored in place of it. A node whose bit-string is not present is considered blank in the secret tree.

Let G be a group. Then each member m of G, as part of Commit message processing, will construct a secret tree secret_tree (see Section 4.7.1). In particular, assume $m.s$ is the state of m after it ran the MLS key schedule to derive (and store) the new group secrets (corresponding to the new epoch initiated by the Commit message processing). To construct a secret tree, m calls $\mathsf{formSecretTree}(m.s.\mathsf{ciphersuite.KDF}, m.s.\mathsf{ratchet_tree}, m.s.\mathsf{encryption_secret})$, defined in Figure 4.54. It then sets secret_tree to the output of this call. Note that secret_tree is not a field in the `state` structure. This is because a secret tree's only purpose is to initialise the new epoch's sending and receiving state in $m.s$.hand_ratchet_state and $m.s$.app_ratchet_state, which is done immediately post secret tree construction.

```
formSecretTree(KDF,τ,encryption_secret):
  for i in range(|τ|)
    secret_tree[i] ← ⊥
  root ← Root(secret_tree)
  i ← node2index(secret_tree, root)
  secret_tree ← populate(KDF, encryption_secret, secret_tree, i)
  return secret_tree

populate(KDF, root_secret, secret_tree, i):
  secret_tree[i] ← root_secret
  left ← leftIndex(secret_tree, i)
  right ← rightIndex(secret_tree, i)
  if left ≠ ⊥:
    left_secret ← KDF.Expand(root_secret, ⟨KDF.Nh, 'mls10tree', ⟨left, 0⟩⟩ , KDF.Nh)
    secret_tree ← populate(KDF, left_secret, secret_tree, left)
  if right ≠ ⊥:
    right_secret ← KDF.Expand(root_secret, ⟨KDF.Nh, 'mls10tree', ⟨right, 0⟩⟩ , KDF.Nh)
    secret_tree ← populate(KDF, right_secret, secret_tree, right)
  return secret_tree
```

Figure 4.54: formSecretTree algorithm.

The formSecretTree algorithm takes in a key derivation function KDF, a ratchet tree τ and a bit-string encryption_secret and returns a secret tree that has the same number of leaves as τ. The secrets in the nodes of the output secret tree are derived recursively, starting from the root node which is assigned the encryption_secret. Namely if i is the index of some parent node n in the secret tree, then the secrets of the children nodes of n are defined as:

$$\mathsf{secret_tree[left]} \leftarrow \mathsf{KDF.Expand}(\mathsf{secret_tree[i]}, \langle \mathsf{KDF.Nh}, \text{'mls10tree'}, \langle \mathsf{left}, 0 \rangle \rangle, \mathsf{KDF.Nh}) \quad (4.16)$$

$$\mathsf{secret_tree[right]} \leftarrow \mathsf{KDF.Expand}(\mathsf{secret_tree[i]}, \langle \mathsf{KDF.Nh}, \text{'mls10tree'}, \langle \mathsf{right}, 0 \rangle \rangle, \mathsf{KDF.Nh}) \quad (4.17)$$

where left is the index of n-s left child (calculated by $\mathsf{leftIndex}(\mathsf{secret_tree}, \mathsf{i})$) and right is the index of n-s right child (calculated by $\mathsf{rightIndex}(\mathsf{secret_tree}, \mathsf{i})$).

Each group member in G is associated to a leaf secret in secret_tree. Which leaf secret a member a corresponds to is fully determined by the index of the node representing a in $m.s$.ratchet_tree. Concretely, if some member a is represented by the node at $m.s$.ratchet_tree$[i]$ (for some index i), then evaluating secret_tree$[i]$ returns the secret associated to a.

Given this secret_tree, m then uses the leaf secrets in secret_tree to initialise $m.s$.hand_ratchet_state[ep] and $m.s$.app_ratchet_state[ep], where ep is the new epoch being initiated, i.e. ep $= m.s$.group_context.epochID.[25] Concretely, m calls initialiseRatchets_step1($m.s$.ciphersuite.KDF, $m.s$.ratchet_tree, secret_tree) to obtain two maps ID2app and ID2hand, that take the identity of a member in G and map it to a bit-string (see Figure 4.55).

```
initialiseRatchets_step1(KDF, ratchet_tree, secret_tree):
  for i in range(|ratchet_tree|)
    if ratchet_tree[i].credential ≠ ⊥:
      ID ← ratchet_tree[i].credential.ID
      ID2app[ID] ← KDF.Expand(secret_tree[i], ⟨KDF.Nh, 'mls10application', ⟨i, 0⟩⟩ , KDF.Nh)
      ID2hand[ID] ← KDF.Expand(secret_tree[i], ⟨KDF.Nh, 'mls10handshake', ⟨i, 0⟩⟩ , KDF.Nh)
  return (ID2app, ID2hand)
```

Figure 4.55: initialiseRatchets_step1 algorithm.

The initialiseRatchets_step1 algorithm takes in a key derivation function KDF, a ratchet tree ratchet_tree and a secret tree secret_tree and outputs a pair of maps (ID2app, ID2hand), that map client identifiers to bit-strings. Each map contains a bit-string for each client ID represented by a leaf in ratchet_tree[i] for some index i. The bit-string is derived, from the leaf secret associated to ID in secret_tree, as follows:

$$\mathsf{KDF.Expand}(\mathsf{secret_tree}[i], \big\langle \mathsf{KDF.Nh}, \text{`mls10'} \| \mathsf{label}, \langle i, 0 \rangle \big\rangle, \mathsf{KDF.Nh}) \quad (4.18)$$

where label is set to 'application' for bit-strings stored in ID2app and 'handshake' for ID2hand.

Finally, client m uses these two maps to initialise $m.s$.hand_ratchet_state[ep] and $m.s$.app_ratchet_state[ep] by calling initialiseRatchets_step2($m.s$, ID2app, ID2hand), defined in Figure 4.56. It then overwrites its state $m.s$ with the output of this call.

The initialiseRatchets_step2 algorithm takes in a `state` structure state, and two maps ID2app and ID2hand, that map client identifiers to bit-strings, and produces a `state` structure. The algorithm sets the state.hand_ratchet_state and state.app_ratchet_state for epoch state.group_context.epochID to contain ID2hand and ID2app respectively along with a counter value 0 and an empty

[25] Recall that, since an MLS key schedule (and hence the secret tree) is only run at the beginning of a new epoch, $m.s$.group_context.epochID contains the new epoch's identifier.

list (since no NAEAD key-nonce pairs have been derived at initialisation time).

```
initialiseRatchets_step2(state, ID2app, ID2hand):
  ep ← state.group_context.epochID
  for ID in ID2app.keys:
    D ← []
    app_sts_R[ID] ← (ID2app[ID], 0, D)
    hand_sts_R[ID] ← (ID2hand[ID], 0, D)
    if state.credential.ID = ID:
      app_sts_S ← ID2app[ID]
      hand_sts_S ← ID2hand[ID]
  state.app_ratchet_state[ep] ← (app_sts_S, app_sts_R)
  state.hand_ratchet_state[ep] ← (hand_sts_S, hand_sts_R)
  return state
```

Figure 4.56: initialiseRatchets_step2 algorithm.

After m updates its state using the initialiseRatchets_step2 algorithm, it sets $m.s$.encryption_secret to $\perp$ (which deletes it). This is done, again, because the only purpose of the encryption_secret is to initialise the handshake and application sending and receiving state of the new epoch.

4.10 Handshake and Application message framing

In Section 4.7, we have seen that the Handshake and Application messages are represented by the same `MLSmessage` structure shown in Figure 4.57.

```
struct MLSmessage:
  GID
  epochID
  content_type
  user_ad
  enc_metadata
  enc_data
```

Figure 4.57: `MLSmessage` structure. Both handshake and application messages have this structure.

There we noted that the enc_metadata and enc_data are NAEAD ciphertexts, which allowed us to view the creation (and processing) of the `MLSmessage` structure as a two step procedure. Namely, a client creating an `MLSmessage` structure *msg* starts by creating the plaintext underlying the *msg*.enc_data (step 1 of `MLSmessage` structure creation). The client then uses the results of this step to form the *msg*.enc_data and all remaining fields in `MLSmessage` structure, the so called framing (step 2 of `MLSmessage` structure creation). Similarly a client processing an `MLSmessage` structure *msg*, first process the framing of an `MLSmessage` structure (step 1 of `MLSmessage` structure pro-

cessing) and then processes the plaintext underlying *msg*.enc_data (step 2 of MLSmessage structure processing) using the results of frame processing.

Section 4.7 explained the creation and processing of the plaintext underlying enc_data. In this section we explain how the 'framing' of the MLSmessage structure is created and processed. We start by defining the MLSSenderData structure, used to represent the metadata underlying the MLSmessage structure's enc_metadata and the symmetric hash ratchet, used to derive NAEAD key-nonce pairs. We then proceed to explain how the MLSmessage framing (and hence the MLSmessage structure) is created using the results of MLSCiphertextContent structure creation. Finally, we describe how the MLSmessage framing is processed upon MLSmessage structure receival.

4.10.1 Metadata

In Section 4.7.1 we have seen that the plaintext underlying the enc_data field is an (encoded) MLSCiphertextContent structure. Similarly, the metadata underlying the MLSmessage's enc_metadata is an (encoded) MLSSenderData structure defined in Figure 4.58.

```
struct MLSSenderData:
  sender
  generation
  reuse_guard
```

Figure 4.58: MLSSenderData structure representing the metadata MLS protects through NAEAD encryption.

Let *metadata* be an MLSSenderData structure and let *msg* be the MLSmessage structure containing the encryption of *metadata*. Then *metadata*.sender is a client identifier representing the creator (and sender) of *msg* and hence *metadata*. The *metadata*.generation is an integer value. If *msg* is used to represent a Handshake message, then *metadata*.generation contains the number of Handshake messages sent by *metadata*.sender in the epoch identified by *msg*.epoch prior to sending *msg*. Similarly, if *msg* is used to represent an Application message, then *metadata*.generation contains the number of Application messages sent by *metadata*.sender in the epoch identified by *msg*.epoch prior to sending *msg*. Finally the *metadata*.reuse_guard is a 32-bit-string, sampled uniformly at random by *metadata*.sender.

4.10.2 Symmetric hash ratchet

Let KDF be a key derivation function and NAE a nonce based authenticated encryption scheme with associated data. A symmetric hash ratchet is a se-

quence of elements, where each element (also called generation) is a bundle of three KDF.Expand calls.

Let e_i be the i-th element (generation) of the symmetric-key hash ratchet, for some $i \in \mathbb{N}_0$. Then all three KDF.Expand calls in e_i take, as the pseudorandom input, the same ratchet secret st_i and produce a new ratchet secret st_{i+1}, an NAE key k_i and an NAE nonce n_i. All KDF.Expand calls (in and across elements) that derive ratchet secrets, NAE keys and NAE nonces take KDF.Nh, NAE.Nk and NAE.Nn as the integer input respectively. The output st_{i+1} of the i-th element is then used as the pseudorandom input of the $i+1$-th element, for all $i \in \mathbb{N}_0$. We call st_0 the initial ratchet secret of the symmetric ratchet. Figure 4.59 illustrates two successive elements of a symmetric ratchet.

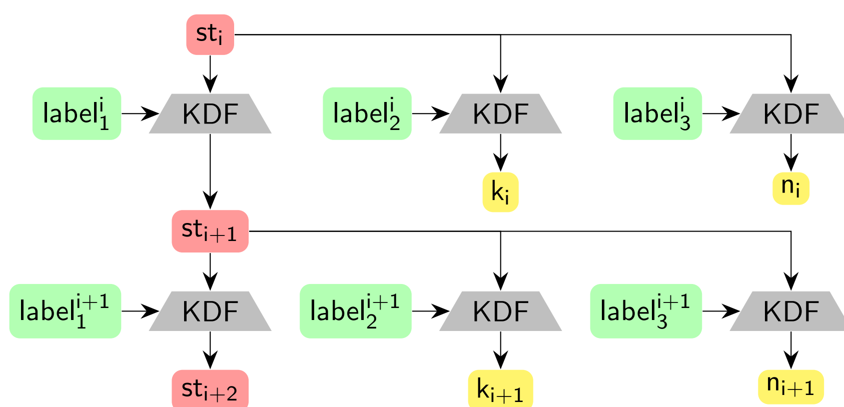


Figure 4.59: Symmetric hash ratchet. In this figure we use KDF as a shorthand for KDF.Expand, and we abstract away the length input for better readability.

A symmetric hash ratchet is then fully specified by the KDF and NAE scheme, the value of its initial ratchet secret st_0 and the context inputs supplied to each KDF.Expand call.

4.10.3 Framing creation

In this section we explain how the framing of the `MLSmessage` structure is created. Let a be a client creating an `MLSmessage` structure. Assume a already created the `MLSCiphertextContent` structure underlying the enc_data field, from which it obtains the created `MLSCiphertextContent` structure ptx and a string user_ad that a wanted only integrity protected. To form the rest of the `MLSmessage` structure (framing) a proceeds as follows. Client a starts by creating an empty `MLSmessage` structure msg by doing:

$$msg \leftarrow \texttt{MLSmessage}(\perp, \perp, \perp, \perp, \perp, \perp) \tag{4.19}$$

that it then proceeds to populate field-by-field. It first sets the first four fields of *msg* by calling form_AD($a.s$, ptx, user_ad, *msg*), defined in Figure 4.60, where $a.s$ is a-s state after creating the `MLSCiphertextContent` structure. Client a then sets *msg* to the output of this call.

```
form_AD(state, ptx, user_ad, msg):
  msg.GID ← state.group_context.GID
  msg.epochID ← state.group_context.epochID
  if ptx.commit ≠ ⊥:
     msg.content_type ← "commit"
  if ptx.proposal ≠ ⊥:
     msg.content_type ← "proposal"
  if ptx.application ≠ ⊥:
     msg.content_type ← "app"
  msg.user_ad ← user_ad
  return msg
```

Figure 4.60: form_AD algorithm.

The form_AD algorithm takes a `state` structure state, `MLSCiphertextContent` structure ptx, string user_ad and `MLSmessage` structure msg and returns msg with its first 4 fields overwritten. Namely the msg.GID and msg.epochID are set to the group identifier and epoch contained in state.group_context, the msg.content_type is set to contain the type of `MLSCiphertextContent` structure ptx is supposed to represent and msg.user_ad is set to the input user_ad.

As mentioned in Section 4.7, the *msg*.GID, *msg*.epochID, *msg*.content_type and *msg*.user_ad fields are used to create the associated data *ad*, for both *msg*.enc_data and *msg*.enc_metadata. Namely *ad* is computed as follows:

$$ad \leftarrow \langle msg.\mathsf{GID}, msg.\mathsf{epochID}, msg.\mathsf{content_type}, msg.\mathsf{user_ad}\rangle\,. \tag{4.20}$$

At this point, a has a formed plaintext ptx underlying *msg*.enc_data as well as the associated data *ad*. All that is left for a to do, in order to form *msg*.enc_data, is to derive the key and nonce pair it will use to encrypt *ptx* and integrity protect the associated data. These key and nonce pairs are derived from symmetric hash ratchets maintained by a.

Namely, assume client a is a member of a group G. Then for each epoch, client a maintains some information pertaining to $2 \cdot (|\mathsf{G}| + 1)$ distinct symmetric hash ratchets. Each of these $2 \cdot (|\mathsf{G}| + 1)$ (per epoch) symmetric hash ratchets correspond to a member of G. More specifically, for each member b in G, a will maintain 2 'receiving' symmetric ratchets, a Handshake and an Application hash ratchet, which it will use to derive decryption key-nonce pairs needed to process received Application and Handshake messages respectively. Note that a can also receive from itself, since the DS Broadcast service sends the message it receives to the entire group including its creator. Additionally, a maintains 2 more 'sending' symmetric ratchets, that

correspond to a (itself), which a uses to derive key-nonce pairs to encrypt and hence send messages to the group.

At the beginning of each epoch ep, client a uses the $a.s.\mathsf{app_ratchet_state}[\mathsf{ep}]$ ($a.s.\mathsf{hand_ratchet_state}[\mathsf{ep}]$) entries initialised according to Section 4.9, to set the initial ratchet secret of each of the $2 \cdot (|\mathsf{G}| + 1)$ symmetric ratchets it maintains. More concretely, let b be a member in G, and let $(\mathsf{st_S}, \mathsf{sts_R})$ be this initialised entry of $a.s.\mathsf{app_ratchet_state}[\mathsf{ep}]$ ($a.s.\mathsf{hand_ratchet_state}[\mathsf{ep}]$) at the beginning of epoch ep. Moreover, let $(\mathsf{st_R}, \mathsf{i_R}, \mathcal{D})$ be the entry at $\mathsf{sts_R}[\mathsf{b}]$. Then $\mathsf{st_R}$ is used as the initial ratchet secret of the 'receiving' Application (Handshake) symmetric ratchet of b (in epoch ep). If $a = b$ then $\mathsf{st_S}$ is used as the initial ratchet secret of the 'sending' Application (Handshake) symmetric ratchet of a.

Note that for $a = b$, the initialiseRatchets_step2 algorithm (defined in Figure 4.56), sets $\mathsf{st_S}$ and $\mathsf{st_R}$ with the same value, i.e. $\mathsf{st_R} = \mathsf{st_S}$. Hence, at the beginning of any epoch ep (before any messages are sent or received by a) the 'sending' and 'receiving' Application (Handshake) symmetric ratchets corresponding to a are guaranteed to be identical. However, this is only guaranteed at the start, since due to message reordering they may diverge.

Each of the $2 \cdot (|\mathsf{G}| + 1)$ symmetric ratchets uses $a.s.\mathsf{ciphersuite.KDF}$ and $a.s.\mathsf{ciphersuite.NAEAD}$ as its key derivation function KDF and nonce based authenticated encryption scheme NAE respectively. Let b be a member in G. Then the context a supplies to each KDF.Expand call in the j-th element (generation) of all 'receiving' (and 'sending' if $a = b$) Handshake and Application symmetric ratchets corresponding to b is:

$$\left\langle \mathsf{L}, \text{'mls10'} \| \mathsf{label}, \langle \mathsf{node2index}(a.s.\mathsf{ratchet_tree}, b), \mathsf{j} \rangle \right\rangle \tag{4.21}$$

where L and label are set to be KDF.Nh and 'secret', NAE.Nk and 'key' or NAE.Nn and 'nonce' if KDF.Expand derives a ratchet secret, key or nonce respectively.

Each time a sends a message to the group, a will use its 'sending' Handshake or Application symmetric ratchet to derive the key and nonce pair it will use for encryption. More concretely, if a sends its j-th message to the group (in its current epoch), then a will use the key-nonce pair of the j-th generation of its 'sending' Handshake (Application) symmetric ratchet for encryption in the j-th Handshake (Application) message. Each time a a receives a message from a client b in G, a will check the type of message (Application or Handshake message) and then use the'receiving' (Application or Handshake message) ratchet corresponding to b, to derive the appropriate key and nonce for decryption. Note however that due to possible reordering of messages, the generation this decryption key-nonce pair belong to will have to be communicated by the sender to the receiver, to ensure efficient

derivation. As we shall see, this is exactly what is carried by the generation field in the MLSSenderData structure representing the message's underlying metadata.

Given this, if each member of G uses its i-th generation key-nonce pair for encryption in the i-th message it sends (for any $i \in \mathbb{N}_0$), then all key-nonce pairs derived from any 'sending' or 'receiving' symmetric ratchet will be used only once. Of course, to do this a member could keep two counter values, to store the amount of Handshake and Application messages it has sent thus far (in the current epoch).

However if these counters are lost or overwritten by mistake (instead of only when a Handshake or Application message is sent), then a client might reuse a generation that has already been used, causing reuse of a key-nonce pair. To prevent this scenario, when a client sends its i-th Handshake (Application) message, upon deriving the key-nonce pair from the i-th generation from its 'sending' Handshake (Application) ratchet, the client must also generate a so called 'reuse-guard', which is a 32-bit-string r sampled uniformly at random. This reuse guard r is then xored with the first 32 bits of the nonce derived from the i-th 'sending' Handshake (Application) ratchet generation. Thus, the nonce used to encrypt in the i-th Handshake (Application) message, is the value resulting from this xor.

Of course since r is a sampled value, it will have to be communicated by the sender to the receiver, along with the symmetric ratchet generation used. Moreover, since the MLSmessage structure (used to represent Handshake and Application messages) does not contain any explicit information about its creator's identity, the identifier of the sender also need to be communicated across to the receivers, for them to know whose 'receiving' ratchet to use. This is exactly what is contained by the MLSSenderData structure representing the message's underlying metadata.

In order to provide forward-security of messages, the used ratchet secrets, keys and nonces must no longer be maintained by a. A ratchet secret is used if it has been taken as input by a KDF.Expand call that produced another ratchet secret, a key and a nonce. We consider a key and nonce to be used if they were employed to encrypt or decrypt a message.[26]

The exact information a must maintain pertaining to the $2 \cdot (|G| + 1)$ symmetric hash ratchets, to ensure forward-security but also successful encryption and decryption, is precisely captured by the fields $a.s.$hand_ratchet_state, $a.s.$app_ratchet_state, $a.s.$hand_generation and $a.s.$app_generation stored in a-s state $a.s$. The $a.s.$hand_generation and $a.s.$app_generation, as explained in Sec-

[26]This implies that a needs to only store the most recently derived ratchet secret, in each of its $2 \cdot (|G| + 1)$ symmetric hash ratchets.

tion 4.4, are integer values representing the total number of Handshake and Application messages sent by a in its current epoch.

As mentioned in Section 4.4, each entry in both $a.s$.app_ratchet_state and $a.s$.hand_ratchet_state is a sending and receiving state pair $(\mathsf{st_S}, \mathsf{sts_R})$. The sending state $\mathsf{st_S}$ is a bit-string that represents the ratchet secret most recently derived in the 'sending' Application (Handshake) ratchet corresponding to a. The receiving state $\mathsf{sts_R}$ is a map from client identifiers (representing group members) to a triple $(\mathsf{st_R}, \mathsf{i_R}, \mathcal{D})$. Let a client identified as ID be a member of a-s group G, and let $(\mathsf{st_R}, \mathsf{i_R}, \mathcal{D})$ be the entry contained at $a.s$.app_ratchet_state[ep] ($a.s$.hand_ratchet_state[ep]) for some epoch ep. Then $\mathsf{st_R}$ is a bit-string representing the most recently derived ratchet secret in the 'receiving' Application (Handshake) ratchet corresponding to client ID and $\mathsf{i_R}$ is the generation of $\mathsf{st_R}$. Finally, $\mathcal{D}$ is a list of NAEAD key-nonce pairs that were derived but not yet used (for decryption) in a generation preceding $\mathsf{i_R}$, in the 'receiving' Application (Handshake) ratchet corresponding to client ID. These key-nonce pairs can arise due to messages being delivered out of order.

The process of forming the msg.enc_data can then be summarised by a calling form_EncData($a.s$, msg.content_type, ad, $\langle$ptx$\rangle$, r), defined in Figure 4.61, where r is the reuse guard sampled by a. Client a then assigns the output of this call to its state $a.s$ and msg.enc_data.

```
form_EncData(state, type, a, m, r):
  NAE ← state.ciphersuite.NAEAD
  id ← state.credential.ID
  index ← node2index(state.ratchet_tree, id)
  ep ← state.group_context.epochID
  if type = "commit" or type="proposal":
    gen ← state.hand_generation
    (st_S, sts_R) ← state.hand_ratchet_state[ep]
  else:
    gen ← state.app_generation
    (st_S, sts_R) ← state.app_ratchet_state[ep]
  nonce ← KDF.Expand(st_S, ⟨NAE.Nn, "mls10nonce", index, i_S⟩, NAE.Nn)
  key ← KDF.Expand(st_S, ⟨NAE.Nk, "mls10key", index, i_S⟩, NAE.Nk)
  st_S ← KDF.Expand(st_S, ⟨KDF.Nh, "mls10secret", index, i_S⟩, KDF.Nh)
  n ← nonce[0, ..., 31]
  n ← n ⊕ r
  nonce ← n||nonce[32, ..., NAE.Nn]
  if type = "commit" or type="proposal":
    state.hand_ratchet_state[ep] ← (st_S, sts_R)
  else:
    state.app_ratchet_state[ep] ← (st_S, sts_R)
  c ← NAE.Enc(key, nonce, a, m)
  return (state, c)
```

Figure 4.61: form_EncData algorithm.

The form_EncData algorithm takes in a `state` structure state, a string type

which can take on three values $\mathsf{type} \in \{\text{“commit”}, \text{“proposal”}, \text{“app”}\}$, a bit-string ad, an `MLSmessage` structure m and a 32-bit-string r. It then outputs a new `state` structure and an NAEAD ciphertext c.

The client then finally goes on to form the $msg.\mathsf{enc_data}$ as follows. It starts by creating the `MLSSenderData` structure *meta* that will underlay this $msg.\mathsf{enc_data}$. As hinted before, *meta* will contain all data (reuse guard, used generation and the identity of the sender of *msg*) *a* needs to provide the recipients of *msg* with, to ensure efficient and correct derivation of the key-nonce pair used to encrypt (and hence used to decrypt) $msg.\mathsf{enc_data}$. Therefore *a* creates *meta* by doing:

$$meta \leftarrow \mathtt{MLSSenderData}(a.s.\mathsf{credential.ID}, a.s.\mathsf{generation}, \mathsf{r}) \tag{4.22}$$

where $\mathsf{generation}$ is set to be $a.s.\mathsf{app_generation}$ or $a.s.\mathsf{hand_generation}$ depending on whether *msg* is an Application or a Handshake message.

Now again, client *a* at this point has the associated data *ad* and *meta* underlying the $msg.\mathsf{enc_metadata}$. Therefore, *a* proceeds to form the ‘metadata’ key-nonce pair that it will use to encrypt *meta* and integrity protect *ad*. Namely the ‘metadata’ key-nonce pair (used to create messages in *a*-s current epoch ep) is computed from: (1) the $\mathsf{sender_data_secret}$ derived in epoch ep, which is stored in $a.s.\mathsf{sender_data_secret}[\mathsf{ep}]$ as explained in Section 4.8, and (2) the first $\mathsf{KDF.Nh}$ bits of $msg.\mathsf{enc_data}$, where KDF is the key derivation function stored in $a.s.\mathsf{ciphersuite.KDF}$.[27]

More concretely, the ‘metadata’ key and nonce used to form $msg.\mathsf{enc_metadata}$ are derived as follows:

$$key \leftarrow \mathsf{KDF.Expand}(\mathsf{s}, \langle \mathsf{NAE.Nk}, \text{‘mls10key’}, msg.\mathsf{enc_data}[0, \ldots, \mathsf{KDF.Nh}-1] \rangle, \mathsf{NAE.Nk}) \tag{4.23}$$
$$nonce \leftarrow \mathsf{KDF.Expand}(\mathsf{s}, \langle \mathsf{NAE.Nn}, \text{‘mls10nonce’}, msg.\mathsf{enc_data}[0, \ldots, \mathsf{KDF.Nh}-1] \rangle, \mathsf{NAE.Nn}) \tag{4.24}$$

where KDF and NAE are the key derivation function and nonce based authenticated encryption scheme stored in the fields $a.s.\mathsf{ciphersuite.KDF}$ and $a.s.\mathsf{ciphersuite.NAEAD}$ respectively, and s is $a.s.\mathsf{sender_data_secret}[\mathsf{ep}]$ (ep is *a*-s current epoch $a.s.\mathsf{group_context.epochID}$). Client *a* can now use *key*, *nonce*, *meta* and *ad* to populate $msg.\mathsf{enc_metadata}$ field. With this, *a* has formed its (Handshake or Application) message *msg*.

The process of *a* creating an `MLSmessage` structure can be summarised by *a* calling $\mathsf{create_Step2}(a.s, \mathsf{ptx}, \mathsf{user_ad})$, defined in Figure 4.62. The $\mathsf{create_Step2}$ algorithm takes a `state` structure, `MLSCiphertextContent` structure and string $\mathsf{user_ad}$ and outputs a `state` structure state and an `MLSmessage` struc-

[27] The [BBM+21] document when forming the underlying plaintext of $msg.\mathsf{enc_data}$ also padds it if the plaintext is less than $\mathsf{KDF.Nh}$ bit in length. It does not provide any further details about this padding.

ture *msg*. Client *a* then sets its *a.s* to the output `state` structure and sends *msg* to the group.

```
create_Step2(state, ptx, user_ad):
  ep ← state.group_context.epochID
  gid ← state.group_context.GID
  NAE ← state.ciphersuite.NAEAD
  if ptx.commit ≠ ⊥:
    type ← "commit"
  if ptx.proposal ≠ ⊥:
    type ← "proposal"
  if ptx.application ≠ ⊥:
    type ← "app"
  a ← ⟨gid, ep, type, user_ad⟩
  r ←$ {0,1}^32
  (state, enc_data) ← form_Ciphertext(state,type,a,⟨ptx⟩,r)
  metadata ← form_Metadata(state,type,a,enc_data,r)
  if type = "commit":
    state.com_ctr ++
    state.hand_generation ++
  if type="proposal":
    state.prop_ctr ++
    state.hand_generation ++
  else:
    state.app_generation ++
  msg ← MLSmessage(gid, ep, type, user_ad, enc_metadata, enc_data)
  return (state, msg)
```

```
form_Metadata(state, type, a, c, r):
  KDF ← state.ciphersuite.KDF
  NAE ← state.ciphersuite.NAEAD
  id ← state.credential.ID
  ep ← state.group_context.epochID
  if type = "commit" or type="proposal":
    gen ← state.hand_generation
    (st_S, sts_R) ← state.hand_ratchet_state[ep]
  else:
    gen ← state.app_generation
    (st_S, sts_R) ← state.app_ratchet_state[ep]
  meta ← ⟨id, gen, r⟩
  c_sample ← c[0, ..., KDF.Nh − 1]
  m_S ← state.sender_data_secret[ep]
  ctx1 ← ⟨NAE.Nk, mls10key, c_sample⟩
  ctx2 ← ⟨NAE.Nn, mls10nonce, c_sample⟩
  key ← KDF.Expand(m_S, ctx1, NAE.Nk)
  nonce ← KDF.Expand(m_S, ctx2, NAE.Nn)
  enc_metadata ← NAE.Enc(key, nonce, a, meta)
  return enc_metadata
```

Figure 4.62: create_Step2 algorithm.

4.10.4 Framing processing

In this section we explain how the framing of an `MLSmessage` structure is processed. Let *a* be a client, with state *a.s*, who received an `MLSmessage` structure *msg* representing a Handshake or Application message. Then *a* starts by ensuring that the group to which *msg* is sent to is its own, by checking $msg.\mathsf{GID} = a.s.\mathsf{group_context.GID}$, and that *msg*.content_type is one of the three strings it is expected to be ("commit", "propopsal", "app"). Moreover if *msg*.content_type = "commit" or *msg*.content_type = "proposal", then *a* asserts that the epoch in which this Commit or Proposal message *msg* was created matches its current epoch, i.e. it checks that $msg.\mathsf{epochID} = a.s.\mathsf{group_context.epochID}$. This ensures that the group secrets derived from processing a Commit or Proposal message define a future epoch, not an already existing one.[28]

Client *a* then proceeds to form the associated data used to form the NAEAD ciphertexts *msg*.enc_metadata and *msg*.enc_data as follows:

$$ad \leftarrow \langle msg.\mathsf{GID}, msg.\mathsf{epochID}, msg.\mathsf{content_type}, msg.\mathsf{user_ad}\rangle\,. \tag{4.25}$$

It then uses *ad* along with the sender_data_secret derived in epoch *msg*.epochID (stored in *a.s*.sender_data_secret[*msg*.epochID]) to compute the 'metadata' key-nonce pair used to produce *msg*.enc_metadata.

[28] This check is not needed for Application messages, since they are not used to modify the group or derive new group secrets.

More concretely, a derives the 'metadata' key key and nonce $nonce$ according to Equation 4.23, where $\mathsf{s} = a.s.\mathsf{sender_data_secret}[msg.\mathsf{epochID}]$ and $\mathsf{KDF} = a.s.\mathsf{ciphersuite.KDF}$ and $\mathsf{NAE} = a.s.\mathsf{ciphersuite.NAEAD}$.

```
process_Step1(state, msg):
  ep ← state.group_context.epochID
  gid ← state.group_context.GID
  NAE ← state.ciphersuite.NAEAD
  KDF ← state.ciphersuite.KDF
  if msg.GID ≠ gid :
    return (state, ⊥)
  if msg.content_type = "commit" or msg.content_type = "proposal":
    if msg.epochID ≠ ep :
      return (state, ⊥)
  a ← ⟨msg.GID, msg.epochID, msg.content_type, msg.user_ad⟩
  c_sample ← msg.enc_data[0, ..., KDF.Nh − 1]
  m_S ← state.sender_data_secret[msg.epochID]
  key ← KDF.Expand(m_S, ⟨NAE.Nk, mls10key, c_sample⟩ , NAE.Nk)
  nonce ← KDF.Expand(m_S, ⟨NAE.Nn, mls10nonce, c_sample⟩ , NAE.Nn)
  meta ← NAE.Dec(key, nonce, a, msg.enc_metadata)
  if meta = ⊥ :
    return (state, ⊥)
  ⟨ID, gen, r⟩ ← meta
  index ← node2index(state.ratchet_tree, ID)
  if msg.content_type = "app":
    (st_S, sts_R) ← state.app_ratchet_state[msg.epochID]
  else:
    (st_S, sts_R) ← state.hand_ratchet_state[msg.epochID]
  (st_R, i_R, D) ← sts_R[ID]
  while D[i] = ⊥ and i_R ≤ gen:
   nonce ← KDF.Expand(st_R, ⟨NAE.Nn, "mls10nonce", index, i_R⟩ , NAE.Nn)
   key ← KDF.Expand(st_R, ⟨NAE.Nk, "mls10key", index, i_R⟩ , NAE.Nk)
   st_R ← KDF.Expand(st_R, ⟨KDF.Nh, "mls10secret", index, i_R⟩ , KDF.Nh)
    D[i_R] ← (key, nonce)
    i_R + +
  if D[gen] = ⊥:
    sts_R[ID] ← (st_R, i_R, D)
    if msg.content_type = "app":
      state.app_ratchet_state[msg.epochID] ← (st_S, sts_R)
    else:
      state.hand_ratchet_state[msg.epochID] ← (st_S, sts_R)
    return (state, ⊥)
  (key, nonce) ← D[gen]
  n ← nonce[0, ..., 31]
  n ← n ⊕ r
  nonce ← n||nonce[32, ..., NAE.Nn]
  ⟨ptx⟩ ← NAE.Dec(key, nonce, a, msg.enc_data)
  if ptx ≠ ⊥:
    D[gen] ← ⊥
  sts_R[ID] ← (st_R, i_R, D)
  if msg.content_type = "app":
    state.app_ratchet_state[msg.epochID] ← (st_S, sts_R)
  else:
    state.hand_ratchet_state[msg.epochID] ← (st_S, sts_R)
  if ptx = ⊥:
    return (state, ⊥)
  return (state, (ID, msg.user_ad, msg.content_type, ptx))
```

Figure 4.63: process_Step1 algorithm.

Client a can then use key, $nonce$ and ad to decrypt msg.enc_metadata to obtain the underlying metadata $meta$. This underlying metadata $meta$ contains all the information a needs in order to derive the key-nonce pair used to form

msg.enc_data. Namely $meta$ contains a client identifier ID, integer gen and a 32-bit-string r, which client a uses in the following way to derive the key-nonce pair (used to form msg.enc_data).

Using the client identifier ID, a selects the 'receiving' Handshake or Application (depending on msg.content_type) symmetric ratchet corresponding to ID. Client a then derives a key-nonce pair (k, n) from the gen-th element (generation) in this 'receiving' Handshake (Application) symmetric ratchet (corresponding to ID). Finally, a forms a nonce n' such that:

$$\mathsf{n}'[0,\ldots,31] = \mathsf{n}[0,\ldots,31] \oplus \mathsf{r} \wedge \mathsf{n}'[32,\ldots,\mathsf{NAE.Nn}-1] = \mathsf{n}[32,\ldots,\mathsf{NAE.Nn}-1] \tag{4.26}$$

where KDF = $a.s$.ciphersuite.KDF and NAE = $a.s$.ciphersuite.NAEAD.[29] Client a then uses k, n′ and ad to decrypt the msg.enc_data and obtain the underlying plaintext ptx. If any of the checks or decryptions fail the framing processing fails. Otherwise the framing processing is considered successful, and a continues to process the extracted plaintext ptx.

The entire processing of the framing of an `MLSmessage` structure can be summarised by the process_Step1 algorithm, defined in Figure 4.63. It takes in a `state` structure and an `MLSmessage` structure msg and returns another `state` structure and a tuple grouping all information that needs to be passed to the second stage of processing (covered in Section 4.7). Therefore, client a upon receiving an `MLSmessage` structure msg (representing a Handshake or Application message), will call process_Step1($a.s$, msg) to obtain a new state that it will use to overwrite its current one $a.s$ and all the information it needs to pass on to the plaintext processing stage (or $\perp$ in case of frame processing failure).

4.11 Welcome message

As explained in Section 4.3 and 4.7.3, whenever a client creates a Commit message, it will also create a Welcome message if the Commit contains an Add Proposal. The Commit message is then sent to all the existing members of the group (clients who are part of the group at the time of creating the Commit message). Assuming the Commit message contains an Add Proposal, the committer creates a (single) Welcome message and sends it to each client suggested to be added directly (not via the Delivery Service). Each client suggested to be added by the Commit message is called a 'new joiner'. The Welcome message provides each 'new joiner' with all the information it needs to derive its group data (pertaining to the group it is joining) of the new epoch, being initiated by the Commit message.

[29] Note that for any nonce n, $\mathsf{n} = \mathsf{n}[0,\ldots,\mathsf{NAE.Nn}]$ holds for all nonce based authenticated encryption schemes with associated data NAE.

A Welcome message is represented by a `Welcome` structure, defined in Figure 4.64. It contains an integer version, representing the MLS version supported by the group and a `Ciphersuite` structure ciphersuite, representing the `Ciphersuite` structure supported by all members in the group, and a ratchet tree ratchet_tree. Moreover it contains a list of `EncryptedGroupSecrets` structures secrets, defined in Figure 4.64, and an NAEAD ciphertext encrypted_group_info. The NAEAD ciphertext encrypted_group_info is formed over an (encoded) `GroupInfo` structure defined in Figure 4.65 using a key-nonce pair derived from the new welcome_secret (see Section 4.8).

`struct Welcome`:	`struct EncryptedGroupSecrets`:
version	keypackage
cipher_suite	encrypted_group_secrets
ratchet_tree	
secrets	
encrypted_group_info	

Figure 4.64: `Welcome` structure and `EncryptedGroupSecrets` structure.

An `EncryptedGroupSecrets` structure consists of a `KeyPackage` structure keypackage and a ciphertext encrypted_group_secrets, where the ciphertext is a KEM ciphertext and NAEAD ciphertext pair. The `GroupSecrets` structure contains two bit-strings, path_secret and joiner_secret.

A `GroupInfo` structure consists of (1) a `GroupContext` structure group_context, (2) a MAC tag confirmation_tag, (3) a client identifier sender and (4) a signature signature, where signature is computed over fields (1),(2) and (3) using the secret signing key associated to sender.

`struct GroupInfo`:	`struct GroupSecrets`:
group_context	path_secret
confirmation_tag	joiner_secret
sender	
signature	

Figure 4.65: `GroupInfo` structure and `GroupSecrets` structure.

We next describe how a committer forms a `Welcome` structure (representing a Welcome message), after which we explain how a 'new joiner' goes about processing it to set its local group data.

4.11.1 Welcome message creation

In this section we explain how a client creating a Commit message, containing Add proposals, creates a (single) Welcome message that it sends to each 'new joiner'. Let a, with state $a.s$, be a client who created a Commit message c such that $c.\mathsf{proposals}$ contains Add `Proposal` structures. Let ℓ be the number of clients that would be added (new joiners) to the group with c. To form the `Welcome` structure (representing the Welcome message) a proceeds as follows. It starts by creating an empty `Welcome` structure wel by doing:

$$wel \leftarrow \texttt{Welcome}(\bot, \bot, \bot, \bot, \bot) \tag{4.27}$$

which it then proceeds to populate field-by-field. Namely, a sets $wel.\mathsf{version}$ and $wel.\mathsf{ciphersuite}$ to the MLS version and `Ciphersuite` structure contained in its state $a.s.\mathsf{version}$ and $a.s.\mathsf{ciphersuite}$ respectively (as that is the version and `Ciphersuite` structure supported by all group members).

Then, a goes on to form the `GroupInfo` structure $info$ and a list containing `GroupSecrets` structures $gscr$, s.t. $info$ underlies the $wel.\mathsf{encrypted_group_info}$ field and $gscr[\mathsf{i}]$ underlies $wel.\mathsf{secrets}[\mathsf{i}].\mathsf{encrypted_group_secrets}$ for all $0 \leq \mathsf{i} < \ell$ (one `GroupSecrets` structure for each 'new joiner'), respectively.

Namely, a first creates an empty `GroupInfo` structure $info$ by doing:

$$info \leftarrow \texttt{GroupInfo}(\bot, \bot, \bot, \bot). \tag{4.28}$$

It then forms a sequence of `KeyPackage` structures seq of length ℓ, such that (1) the `KeyPackage` structures at positions $0 \leq \mathsf{i}, \mathsf{j} < \ell$ are the same if and only if $\mathsf{i} = \mathsf{j}$ and (2) each `KeyPackage` structure belongs to one 'new joiner'.[30] The `KeyPackage` structures used to form this sequence are the ones present in the Add proposals, included in c.

This allows a to associate to each 'new joiner' b, an index $0 \leq \mathsf{i} < \ell$, that will corresponding to their position in this sequence. Then a will create a list of `GroupSecrets` structures $gscr$, such that each `GroupSecrets` structure it contains is empty, i.e.

$$gscr[\mathsf{i}] \leftarrow \texttt{GroupSecrets}(\bot, \bot). \tag{4.29}$$

For each $0 \leq \mathsf{i} < \ell$, the `GroupSecrets` structure $gscr[\mathsf{i}]$ will then correspond to the i-th 'new joiner'. Now client a will go on to populate each field of each `GroupSecrets` structure in $gscr$, as well as populate each field of $info$. It will do so by essentially 'simulating' the processing of its own Commit message c.

[30] The sequence can be arbitrary, but must follow the 2 rules set out.

Namely a creates a copy of its state st, and processes its own Commit message c onto this copy st (according to Sections 4.10.4, 4.7.1, 4.7.3, 4.8 and 4.9).[31] During the processing of c onto st, at the stage when the MLS key schedule is run to derive the group secrets, client a will store the joiner_secret into $gscr[\mathsf{i}].\mathsf{joiner_secret}$, for each $0 \leq \mathsf{i} < \ell$ (see Figure 4.53). Observe that the joiner_secret can be used to derive the welcome_secret, init_secret, enc_secret and confirmation_key of the epoch that would be initiated by c (see Figure 4.53).

After c has been processed onto st, a uses st to populate $info.\mathsf{group_context}$, $info.\mathsf{confirmation_tag}$ and $info.\mathsf{sender}$ with $st.\mathsf{group_context}$, $st.\mathsf{confirmation_tag}$ and $st.\mathsf{credential.ID}$ respectively. Note that the field $st.\mathsf{group_context}$ contains the `GroupContext` context structure of the new epoch (initiated by c) and the field $st.\mathsf{confirmation_tag}$ contains the MAC tag of the plaintext underlying c (see Section 4.7.1). It then forms the $info.\mathsf{signature}$ by doing:

$$info.\mathsf{signature} \leftarrow st.\mathsf{ciphersuite.DS.Sign}(st.\mathsf{sign_sk}, \mathsf{m}) \tag{4.30}$$

where $\mathsf{m} = \langle info.\mathsf{group_context}, info.\mathsf{confirmation_tag}, info.\mathsf{sender} \rangle$.

Now a proceeds to form the $wel.\mathsf{encrypted_group_info}$ using the populated $info$ and $st.\mathsf{welcome_secret}$. Namely a uses $st.\mathsf{welcome_secret}$ to derive a key-nonce pair as follows:

$$\mathsf{wel_key} \leftarrow \mathsf{KDF.Expand}(st.\mathsf{welcome_secret}, \text{`key'}, \mathsf{NAE.Nk}) \tag{4.31}$$

$$\mathsf{wel_nonce} \leftarrow \mathsf{KDF.Expand}(st.\mathsf{welcome_secret}, \text{`nonce'}, \mathsf{NAE.Nn}) \tag{4.32}$$

where KDF and NAE are st.ciphersuite.KDF and st.ciphersuite.NAEAD respectively. This key-nonce pair $(\mathsf{wel_key}, \mathsf{wel_nonce})$ is then used to encrypt $info$, by doing:

$$wel.\mathsf{encrypted_group_info} \leftarrow \mathsf{NAE.Enc}(\mathsf{wel_key}, \mathsf{wel_nonce}, \langle \text{""} \rangle, info) \tag{4.33}$$

where NAE is again st.ciphersuite.NAEAD.

At this point, for all $0 \leq \mathsf{i} < \ell$, only the $gscr[\mathsf{i}].\mathsf{joiner_secret}$ field has been populated. To see if $gscr[\mathsf{i}].\mathsf{path_secret}$, for all $0 \leq \mathsf{i} < \ell$, needs to be set to a non $\perp$ value, a checks if $c.\mathsf{path}$ is populated. If it is not, that means no new KEM secret keys have been produced in the ratchet tree. Hence all the KEM secret keys contained have been derived due to a Commit message before c. Of course if $c.\mathsf{path}$ is not populated, since MLS does not want the 'new joiners' to have any past secrets, then $gscr[\mathsf{i}].\mathsf{path_secret}$ is set to $\perp$ for all $0 \leq \mathsf{i} < \ell$. If however, $c.\mathsf{path}$ was populated, then a needs to provide each

[31] Note that we opt for the copying of the state approach, since it makes clear which the exact values used in forming wel are. In reality this can be done in a much more efficient way and the [BBM+21] does not constrain a client to copying.

'new joiner' with all the new KEM secret keys it is entitled to know by the ratchet tree invariants (see Section 4.5.1). Therefore, like in Section 4.7.3, a will send the appropriate seed to each 'new joiner', enabling the 'new joiner' to derive all KEM secret keys it is allowed to know.

More specifically, let a_{leaf} be the leaf that represents a in $st.\mathsf{ratchet_tree}$ and let b be the i-th 'new joiner' (for some $0 \leq \mathsf{i} < \ell$) represented by leaf node b_{leaf} in $st.\mathsf{ratchet_tree}$. Then client a identifies the lowest common ancestor u of a_{leaf} and b_{leaf} by computing:

$$u \leftarrow \mathsf{LCA}(\mathsf{st.ratchet_tree}, \mathsf{a_{leaf}}, \mathsf{b_{leaf}}). \tag{4.34}$$

Let s be the seed value corresponding to u. Then, client a sets $gscr[\mathsf{i}].\mathsf{path_secret}$ to contain s. This process is repeated for each of the ℓ 'new joiners', at the end of which, the $gscr$ list will contain fully populated `GroupSecrets` structures.

Now for each $0 \leq \mathsf{i} < \ell$ the $gscr[\mathsf{i}]$, which corresponds to the i-th 'new joiner' (in seq), must be encrypted using the KEM public key of the i-th 'new joiner'. Namely, let keypacks be a list of `KeyPackage` structures such that $\mathsf{keypacks}[\mathsf{i}]$ is the `KeyPackage` structure of the i-th 'new joiner' (contained in the Add `Proposal` structure in $c.\mathsf{proposals}$). Then for each $0 \leq \mathsf{i} < \ell$, a goes on to encrypt $gscr[\mathsf{i}]$ using $\mathsf{keypacks}[\mathsf{i}].\mathsf{kem_pk}$ as follows:

$$enc_gscr[\mathsf{i}] \leftarrow \mathsf{SetSeal}(st, \mathsf{keypacks}[\mathsf{i}].\mathsf{kem_pk}, gscr[\mathsf{i}]). \tag{4.35}$$

where SetSeal is defined in Figure 4.47.

It then uses the list of `KeyPackage` structures keypacks and the list of ciphertexts produced enc_gscr to form the list of `EncryptedGroupSecrets` structures $wel.\mathsf{secrets}$. Namely, for each $0 \leq \mathsf{i} < \ell$ if forms $wel.\mathsf{secrets}[\mathsf{i}]$ as follows:

$$wel.\mathsf{secrets}[\mathsf{i}] \leftarrow \mathtt{EncryptedGroupSecrets}(\mathsf{keypacks}[\mathsf{i}], enc_gscr[\mathsf{i}]). \tag{4.36}$$

Therefore, each new joiner will have exactly one entry in $wel.\mathsf{secrets}$, whose encrypted_group_secrets field it will be able to successfully decrypt (using its KEM secret key). Finally a sets the $wel.\mathsf{ratchet_tree}$ to the public state of the $st.\mathsf{ratchet_tree}$, i.e. get-PS$(st.\mathsf{ratchet_tree})$. With this a has finished populating each field in wel, and hence completed the creation of the Welcome message.

4.11.2 Welcome message processing

In this section we explain how a 'new joiner' processes a Welcome message upon receiving it. Let a, with state $a.s$, be a 'new joiner' who just received a `Welcome` structure wel representing a Welcome message. Recall from Section 4.4, $a.s$ will only have its client data fields populated, whilst all group data

fields will be empty (set to $\perp$).[32] The client then proceeds to process wel as follows.

It first identifies an index i such that the entry wel.secrets[i] contains a-s KeyPackage structure in wel.secrets[i].keypackage. Then a checks that the wel.secrets[i].keypackage.suite and wel.secrets[i].keypackage.version fields match wel.suite and wel.version respectively.[33] If they do not, then the processing of wel fails, hence let us assume that no miss-match occurred. Then, if wel is a correctly formed (according to 4.11.1) Welcome structure, a will be able to successfully decrypt wel.secrets[i].encrypted_group_secrets using its KEM secret key $a.s$.kem_sk (corresponding to the KEM public key contained in wel.secrets[i].keypackage.kem_pk) as follows:

$$gscr \leftarrow \mathsf{OpenSeal}(a.s, a.s.\mathsf{kem_sk}, wel.\mathsf{secrets[i].encrypted_group_secrets}) \tag{4.37}$$

where the OpenSeal algorithm is defined according to Figure 4.52. If decryption is not successful, then a of course stops processing wel. The $gscr$ obtained, upon successful decryption, is a GroupSecrets structure, which contains all the secrets a needs, in order to derive its local group data and hence successfully participate in group conversations in the new epoch.

Namely, a first uses $gscr$.joiner_secret to (according to the Key Schedule 4.53) derive its group secrets: $a.s$.init_secret, $a.s$.welcome_secret, $a.s$.encryption_secret, $a.s$.sender_data_secret and $a.s$.confirmation_key. It then uses $a.s$.welcome_secret to construct the key-nonce pair used to form wel.encrypted_group_info as follows:

$$\mathsf{wel_key} \leftarrow \mathsf{KDF.Expand}(a.s.\mathsf{welcome_secret}, \text{`key'}, \mathsf{NAE.Nk}) \tag{4.38}$$

$$\mathsf{wel_nonce} \leftarrow \mathsf{KDF.Expand}(a.s.\mathsf{welcome_secret}, \text{`nonce'}, \mathsf{NAE.Nn}) \tag{4.39}$$

where KDF and NAE are the same as wel.secrets[i].keypackage.suite.KDF and wel.secrets[i].keypackage.suite.NAEAD respectively.[34] Client a then goes on to use wel_key and wel_nonce to decrypt wel.encrypted_group_info by doing:

$$info \leftarrow \mathsf{NAE.Dec}(\mathsf{wel_key}, \mathsf{wel_nonce}, \langle\text{""}\rangle, wel.\mathsf{encrypted_group_info}) \tag{4.40}$$

where NAE is set to be wel.secrets[i].keypackage.suite.NAEAD.

[32]Note that at the beginning of our description, we assumed that all clients are members of at most one group at any point in time. Hence, a client needs to only maintain group data for at most one group.

[33]Recall that we assumed that all clients support the same and only one version and Ciphersuite structure. Hence in our simplification this check will always pass. In reality a client can have more than one KeyPackage structure stored in the Deliver Service, one for each version and Ciphersuite structure combination it supports. Therefore, it needs to ensure that the correct KeyPackage structure was used to suggest a to be added to the group.

[34]Note that in our MLS simplification is the same as $a.s$.ciphersuite is the same as wel.secrets[i].keypackage.suite.

If the decryption fails, then so does the processing of wel. If decryption succeeds $info$ is a `GroupInfo` structure, and a can proceed to verify its contained signature $info$.signature. More concretely, a will first inspect the $info$.sender field to know who created wel, and hence who created $info$.signature. It then uses the information in the ratchet tree wel.ratchet_tree to obtain the public signing key of $info$.sender. Before obtaining the public signing key however, a first verifies that the ratchet tree is valid. Namely it checks that: (1) invariants 3,4,5,6 hold (see Section 4.5.1), (2) each node with a populated `Credential` structure (not equal $\perp$) has a valid signature and (3) the $info$.group_context.tree_hash is valid w.r.t. wel.ratchet_tree. If wel.ratchet_tree is a valid ratchet tree (passes the above checks), then a goes on to find the leaf node representing $info$.sender in wel.ratchet_tree, otherwise wel processing fails.

Let $sender_{leaf}$ be the leaf representing the client, identified by $info$.sender. Now a can use the public signing key contained in the `Credential` structure $sender_{leaf}$.credential.sign_pk, to verify $info$.signature as follows:

$$b \leftarrow \mathsf{DS.Vfy}(sender_{leaf}.\mathsf{credential.sign_pk}, \mathsf{m}, info.\mathsf{signature}) \tag{4.41}$$

where $\mathsf{m} = \langle info.\mathsf{group_context}, info.\mathsf{confirmation_tag}, info.\mathsf{sender}\rangle$ and DS is the digital signature scheme contained in wel.secrets[i].keypackage.suite.DS.

If b = `false`, a stops processing wel. Otherwise, it continues to populate its group data using the information it obtained from $info$. Namely, it assigns $info$.group_context and $info$.confirmation_tag to $a.s$.group_context and $a.s$.confirmation_tag respectively.

The only thing left for a to do, in order to have all the group information it needs to participate in group conversations, is to populate the wel.ratchet_tree with secret KEM keys it is allowed to know. Recall that the $gscr$.path_secret can be set to $\perp$, indicating that the committer did not generate any new KEM keys along its simple path to the root, and hence it did not generate any KEM secret key a is allowed to know. Therefore, if $gscr.\mathsf{path_secret} = \perp$, a assigns the wel.ratchet_tree as is to its local ratchet tree $a.s$.ratchet_tree.

Otherwise, a goes on to find the leaf node in wel.ratchet_tree that represents a (contains a-s `Credential` structure). Let this representative leaf be a_{leaf}. Client a will now use $gscr$.path_secret (like existing members did when processing the `Commit` structure in Section 4.7.3) to derive the secret KEM keys of all nodes on the common path of a_{leaf} and $sender_{leaf}$ $\mathsf{commonPath}(wel.\mathsf{ratchet_tree}, a_{leaf}, sender_{leaf})$, using the Prg algorithm, defined in Figure 4.46. It then assigns these secret KEM keys to the appropriate nodes in the ratchet tree wel.ratchet_tree, and then sets it local ratchet tree $a.s$.ratchet_tree to contain this updated wel.ratchet_tree.

Finally a sets all the counters to 0, all the pending and confirmation maps

to be empty and derives the initial values of $a.s$.hand_ratchet_state[ep] and $a.s$.app_ratchet_state[ep] according to Section 4.9, where ep is the epoch in which a is part of the group, i.e. ep = $a.s$.group_context.epochID.

4.12 Initialise group

In previous sections we have seen how an **existing** group can change by having its members add, remove clients or update keys. However, thus far, we have avoided specifying (in detail) how a client creates a group to begin with. Namely, we have seen that the group information was propagated to 'new joiners' via Welcome messages, which were in turn constructed by some existing member incorporating part of its own group data. This then begs the question: 'How do the very first members of the group initialise their group data?' This is exactly the problem the initGroupData algorithm, first mentioned in step 4 of the MLS protocol overview (Section 4.3), solves.

Namely, consider a client a, with state $a.s$, who wishes to create a group, identified by the group identifier GID, containing clients $a_1, a_2, \ldots, a_n$ (and itself) for some $n \in \mathbb{N}_0$. In order for a to create a group, (according to Section 4.3) a will need to follow steps 3 and 4, captured by Figures 4.11 and 4.13. Assume that a finished step 3 and hence has selected the `Ciphersuite` structure suite_G and MLS version v_G of the group. Now client a will call initGroupData($a.s$,suite_G,v_G,GID), defined in Figure 4.66. It will then assign the output of this algorithm to its state $a.s$.

```
initGroupData(state, suite, v, GID):
  n ← node(state.kem_pk, state.kem_sk, state.credential, ⊥)
  state.ratchet_tree ← [n]
  root ← Root(state.ratchet_tree)
  epochID ← 0
  tree_hash ← TH(suite.H, state.ratchet_tree,root)
  confirmed_transcript_hash ← ⟨""⟩
  state.group_context ← GroupContext(GID, epochID, tree_hash, confirmed_transcript_hash)
  state.confirmation_tag ← ⟨""⟩
  state.init_secret ←$ {0,1}^suite.KDF.Nh
  return state
```

Figure 4.66: initGroupData algorithm.

The initGroupData algorithm takes in a `state` structure state, a `Ciphersuite` structure suite, an integer (representing the MLS version) v and a group identifier GID and produces another `state` structure. The produced `state` structure, is almost the same as state, except that some of its group data fields are not empty (⊥) but instead, initialised to contain information of a one-member group.

Namely the state.ratchet_tree now contains a ratchet tree with a single node,

representing the member to whom the state belongs to. The state.group_context is initialised with a GroupContext structure that contains the group identifier GID, epoch identifier 0, a tree hash of the single node ratchet tree state.ratchet_tree and a zero length octet string as the confirmed_transcript_hash (since no Commit messages have been processed yet). Moreover, because no Commit messages have been processed, the initGroupData algorithm sets the state.confirmed_transcript_hash to a a zero length octet string as well. Finally, in order for the MLS Key Schedule (see Figure 4.53) to be able to compute future group secrets, the initGroupData algorithm sets the initial init_secret to a KDF.Nh-bit-string sampled uniformly at random (where KDF = suite.KDF).

Therefore, at this point a has essentially formed a group with itself under the group identifier GID. In order to actually form the group it wants, namely a group containing clients a and $a_1, a_2, \ldots, a_n$, client a will simply create an Add Proposal message (according to Sections 4.7 and 4.10) for each a_i, where $i \in \{1, \ldots, n\}$. Since at this point a is the only existing member of group GID, a does not send the Add Proposal messages to the DS Broadcast service, and instead immediately forms a Commit message c by setting c.proposals to include each of the n Add Proposal structures it created (according to Sections 4.7 and 4.10).

Client a again does not send c out (since a is still the only existing member), but instead processes c immediately upon creation, to derive a new state. This new state now contains group data pertaining to a group containing a along with $a_1, a_2, \ldots, a_n$. At the same time a creates a single Welcome message (according to Section 4.11.1), which it then sends to a_i for all $i \in \{1, \ldots, n\}$. The 'new joiners' $a_1, a_2, \ldots, a_n$ then process the Welcome message according to Section 4.11.2 to derive their own local group data, pertaining to the group GID with members $a, a_1, a_2, \ldots, a_n$.

4.13 Comparison to MLSv11

This section covers the differences between our description and the MLS description given in the specification document [BBM+21].

We exclude the following sections from our description:

- External initialization [BBM+21, Section 8.1]
- Pre-Shared Keys [BBM+21, Section 8.2]
- Exporters [BBM+21, Section 8.6]
- Resumption secret [BBM+21, Section 8.7]
- State authentication keys [BBM+21, Section 8.8]
- Linking a New Group to an Existing Group [BBM+21, Section 10.1]

- Proposal types that are not Adds, Removes or Updates [BBM+21, Sections 11.1.4 - 11.1.8]
- External commits [BBM+21, Section 11.2.1]

Furthermore, we only consider one type of credential as described in 4.1. The `KeyPackage` structure is flattened, in the sense that all the 'must have' extensions (`Capabilities` and `Lifetime` structure) are embedded directly into the `KeyPackage`. Moreover, the types of these extensions are not specified in this flattened view, since the types are 'only' useful for parsing the `KeyPackage` structure in practice (and do not contribute to the security of MLS). In our description we also assume that an MLS client can be a member of at most one group, i.e. in any run of the MLS protocol, all groups formed will be disjoint. Moreover, we assume that all clients support one and the same `Ciphersuite` structure and MLS version. This simplifies the state, as well as the logic of all clients, and provides the same security guarantees as the original MLS protocol. The entire `state` structure, in Section 4.4, was introduced by us, since we believe it facilitates a more fine-grained understanding of MLS and its security properties. Moreover, the `node` structure (used to represent a node in the ratchet tree, see Section 4.5) in our description, does not contain a 'parent hash' field; as its purpose was not made clear by the [OBR+21, BBM+21] documents, nor the MLS mailing list. We consider the purpose of the 'parent hash' field an open problem, for possible future work.

Next to this, in Section 4.3, we assumed that it is the client's job to delete `KeyPackage` structures it used to form a group, whereas MLS just suggests for this deletion to occur, but does not specify the entity responsible for this deletion. The AS and DS KeyPackage API, is also something we introduced, to make the diagrams in Section 4.3, more exact. The MLS protocol itself does not put any restraints on this of course. Further, we assumed that the Handshake and Application messages are represented solely by the `MLSmessage` structure, which ensures that the underlying data is protected by NAEAD encryption. However, MLS also allows for Handshake messages to be transferred in an `MLSPlaintext` structure, which essentially groups the `MLSCiphertextContent` structure, associated data and its metadata, as is (no ciphertexts are formed from these data fields). Moreover, the list of proposals contained in the Commit message contains a list of so called '`ProposalOrRef`' structures, instead of `Proposal` structures. The `ProposalOrRef` structure is, as the name suggests either a `Proposal` structure or a digest of a `Proposal` structure. This, can be used to make the payload of the Commit message smaller, but does not contribute to the understanding of MLS, and hence we avoid describing this extra complexity.

Lastly, we assumed that the same `GroupContext` structure is used to form: (1) the signature signature contained in the `MLSCiphertextContent`, (2) the

MAC tag confirmation_tag in the MLSCiphertextContent and (3) the ciphertexts enc_path_secret contained in the UpdatePathNode. However, in MLS, each of these uses a different GroupContext structure. The usefulness of this change in GroupContext structure remains an open problem, left for future work. In addition to this, in Section 4.10, we assumed that the associated data used to form enc_data and enc_metadata is the same, whereas MLS excludes the user_ad field from the associated data used to form enc_metadata. This choice, as well, is left as an open problem. Finally, the Welcome structure, described in Section 4.11, contained a ratchet tree ratchet_tree. However, in MLS this information is either carried as an extra 'ratchet tree' extension in the KeyPackage structure (included in the Welcome message), or is posted (publicly) at the DS service for new-joiners to fetch. Again, this alteration made no difference to the security of MLS, but made structures overall simpler.

Bearing all these alterations in mind, we believe our description still captures the main aspects of the MLS protocol whilst excluding cumbersome details of extension-like functionalities.

4.14 MLS protocol security

In Section 3.3 we listed the security properties an SGM protocol is expected to provide. In this section we give a brief summary of how MLS tries to achieve each of these properties.

Confidentiality. The plaintext of Handshake and Application messages (represented by the MLSmessage structure) is, as discussed in Section 4.7.1, represented by the MLSCiphertextContent structure. The confidentiality of Handshake and Application messages' plaintext is achieved by having the plaintext (MLSCiphertextContent structure) be encrypted to form the NAEAD ciphertext enc_data (using an NAEAD key-nonce pair only known to group members) of the MLSmessage structure.

The plaintext of Welcome messages (represented by the Welcome structure) is considered to be all the GroupInfo and GroupSecrets structures combined. The confidentiality of Welcome messages' plaintext is ensured by having the: (1) GroupSecrets structures encrypted under the public KEM key of new joiners (according to the SetSeal algorithm) and having the (2) GroupInfo structure encrypted using an NAEAD key-nonce pair derived from the welcome_secret, only known by group members.

Message and Sender Authenticity. We consider two aspects to authenticity. The first aspect allows members to detect if a message has been tampered with and if it was sent by some member in the group (but not which mem-

ber). The second allows members to detect message tampering and the exact client it was sent by.

For Handshake and Application messages' plaintext, the first form is provided by the NAEAD ciphertext enc_data, over the underlying plaintext, and the fact that only group members know the NAEAD keys and nonces used. Note that, because all members start with the same initial values in their hand_ratchet_state and app_ratchet_state, all members are able to compute all NAEAD keys and nonces. Hence the NAEAD key-nonce pairs are not cryptographically bound to a specific group member. The second form for Handshake and Application messages' plaintext is instead provided by the digital signature contained as part of the `MLSCiphertextContent` structure.

For Welcome messages' plaintext the first form of authenticity is ensured by: (1) having the `GroupSecrets` structures (containing secrets only known to group members) be encrypted under the public KEM key of new joiners, and having the (2) `GroupInfo` structure encrypted using an NAEAD key-nonce pair derived from the welcome_secret, only known by group members. The second form is provided by the signature contained in the `GroupInfo` structure.

Anonymity. The anonymity of senders, for Handshake and Application messages, is ensured by: (1) having the sender identity (which is part of the metadata), be encrypted to form the enc_metadata field, using a key-nonce pair derived from the metadata_secret (known only by group members) of the `MLSCiphertextContent` structure (used to represent Handshake and Application messages) and (2) having the signature of the sender be encrypted as well (to form enc_data). For Welcome messages the sender holds, because the `GroupInfo` structure containing the sender's identity and signature is encrypted to form an NAEAD ciphertext (using a key-nonce pair derived from the welcome_secret, only known to group members).

Forward-security (FS). Let a be a client whose state is compromised. Then because: (1) all commit_secrets before a's most recent update remain secret, (2) the MLS key schedule and secret tree delete all values used to derive a fresh initial handshake and application state (see Section 4.8 and 4.9) and (3) the client stores only the ratchet state and key-nonce pairs that are not used in its handshake and application state (see Section 4.10), then forward security holds for messages (w.r.t. confidentiality and authenticity [35]).

Post-compromise security (PCS). Let $a_1, a_2, \ldots, a_n$ for some $n \in \mathbb{N}$, be all the clients whose state was leaked to some adversary. Then, after each a_i (for all $i \in \{1, \ldots, n\}$) (1) proposed an update (that was committed and

[35] We will see in Chapter 5 that forward secure anonymity, for MLS, does not hold.

processed) or (2) created and processed a Commit message (with a non-empty path field) or (3) has been proposed for removal (that was committed and processed), then the messages become secure again. In more detail, consider a_i (for some $i \in \{1, \ldots, n\}$).

Assume a_i proposed an update of its keys (that has been committed). Then, the processing of this update (according to Section 4.7.2) would cause all the KEM key pairs on a_i's direct path to be blanked, and its own node would now contain a fresh (randomly sampled) KEM key pair; such that only a_i knows the fresh KEM secret key. Recall, that a client can only know the KEM secret keys that lie on its (representative leaf's) simple path to the root. Therefore, all the secret KEM keys (pertaining to a_i) known to the adversary, will be overwritten to be either blank or a value derived in a non-deterministic manner. Hence, the adversary would have no knowledge of any secret KEM key pairs (pertaining to a_i), and would therefore be unable to derive new commit_secrets (using a_i's state).

Assume a_i is proposed for removal (that has been committed). Then, the processing of this remove proposal (according to Section 4.7.2) would cause all the KEM key pairs on a_i's simple path to be blanked. Therefore, all the secret KEM keys (pertaining to a_i) known to the adversary, will be overwritten to be blank. Hence, similarly as above, the adversary would have no knowledge of any secret KEM key pairs (pertaining to a_i), and would therefore be unable to derive new commit_secrets (using a_i's state). Finally, assume a_i created a Commit message (with a non-empty path field). Then, the processing of this Commit message (according to Section 4.7.3) would cause all the KEM key pairs on a_i's simple path to be overwritten with new values, derived recursively using Prg (see Equation 4.10 and Figure 4.46) from a randomly sampled seed. Therefore, all the secret KEM keys (pertaining to a_i) known to the adversary, will be overwritten with a value derived in a non-deterministic manner. Hence, again, the adversary would be unable to derive new commit_secrets (using a_i's state).

Now, since the commit_secret is necessary for the derivation of new group secrets (which are in turn used to derive the handshake and application ratchet state and key-nonce pairs), then the messages would regain security.

Chapter 5

Building blocks of MLS

This chapter starts by presenting the MLS protocol, described in Chapter 4, as a modular construction of five components: continuous group key agreement (CGKA), key-evolving group symmetric encryption (KEGSE) scheme, PRF-PRNG scheme, a digital signature scheme (DS) and a message authentication code (MAC), inspired by [ACD18]. Subsequently, we provide intuition for the security goals and purpose, behind each of the (main) non-standard components (CGKA, PRF-PRNG and KEGSE); and explain how they interact with one another within MLS. Since the CGKA component has been formally studied by many prior works [ACDT19, AJM20, ACC^{+}19] and DS schemes and MACs are standard cryptographic primitives, we then turn our attention to formalising the PRF-PRNG and KEGSE components for the remainder of the chapter.

5.1 Modular construction of MLS

In this section we describe an alternative way of viewing the MLS protocol, described in Chapter 4. Namely, we see that we can cast the MLS protocol as an SGM protocol composed of five components: continuous group key agreement (CGKA), key-evolving group symmetric encryption (KEGSE) scheme, PRF-PRNG scheme, a digital signature scheme (DS) and a message authentication code (MAC), see Figure 5.1. Next, we describe the goals and security we want from each of the non-standard primitives (CGKA, PRF-PRNG and KEGSE) and explain which parts of the MLS protocol (we described) belong to which primitive. How each of these MLS based primitives achieves their security goals, can be easily extracted from Section 4.14, which explicitly states the parts of MLS (and hence primitives) which contribute to its security properties. Finally, we explain the data flow between these different primitives.

5.1.1 CGKA

The CGKA scheme can be seen as the group analogue of a continuous key agreement (CKA) scheme, described in Section 3.4.3. On a high level, the goal of a CGKA scheme is to, in an asynchronous scenario for each epoch, provide all members of a group with fresh secret random values *scr*. For members to derive these shared fresh secrets, a CGKA scheme provides its clients with methods for: creating a group, proposing a client for addition or removal to the rest of the group and proposing an update of its own key material. Moreover, to enable clients to choose which of the proposed operations comes into effect, the CGKA scheme also allows members to commit a set of proposed operations, as well as welcome any new clients added by the commitment. Each commitment c will, in turn, cause the group to change according to the set of proposals committed by c, thereby initiating a new epoch. More notably, each commitment c will result in a fresh secret value *scr* to be derived by each group member.

A CGKA scheme is considered secure if all the following properties hold; (1) given the transcript of messages exchanged between group members, the random secrets *scr* underlying those messages should look uniformly random and independent to an eavesdropping adversary (confidentiality), (2) if the state of any group member is compromised, then previous secret values remain secure (forward security), and (3) if every group member whose state was compromised, creates a commitment, proposes an update or is proposed for removal (which is committed and processed) then secret values *scr* become secure again (post-compromise security).

Given this, we can view the Welcome messages (see Section 4.11) and Handshake message content (see Section 4.7.3 and 4.7.2), along with the ratchet tree (see Section 4.5), commit_secret and `GroupContext` structure (see Section 4.6) as forming an MLS based instantiation of a CGKA scheme (see Figure 5.1). Namely, a client proposes to add, remove or update its keys via a `Proposal` structure, and commits a set of proposals by creating a corresponding `Commit` structure. If the `Commit` structure contains an Add `Proposal` structure, then the committer also forms and sends a `Welcome` structure to the 'new joiners'. Each time an existing group members ('new joiner') processes a `Commit` (`Welcome`) structure, its state changes, and in particular, its ratchet tree, group_context and commit_secret change. Out of these values only the commit_secret is (1) secret, (2) the same across all members of the group, and (3) is derived in a non-deterministic way. Hence, the commit_secret acts as the fresh random secret value *scr* in the MLS based CGKA.

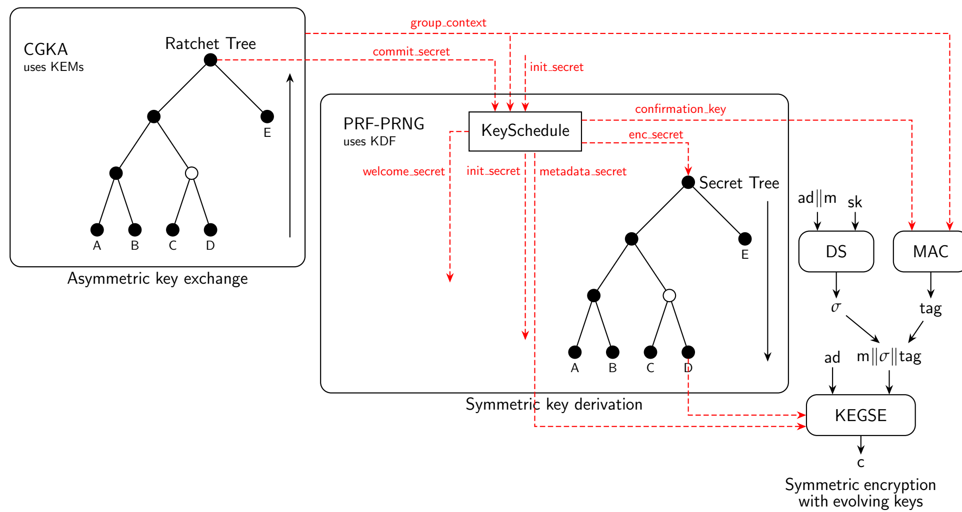


Figure 5.1: Sketch of the modular construction of the MLS protocol and the data flow between its components.

5.1.2 PRF-PRNG

A PRF-PRNG scheme, is the primitive defined in [ACD18], whose summary can be found in Section 3.4.3. The goal of a PRF-PRNG scheme is to essentially serve as a high entropy pool, from which high entropy values can be obtained. The MLS key schedule (see Section 4.8) and the secret tree (see Section 4.9), can be viewed as an MLS based instantiation of a PRF-PRNG scheme (see Figure 5.1).

More concretely, the init_secret can be seen as the state of the PRF-PRNG since, each time the MLS based PRF-PRNG is run (and hence the key schedule is run), an old init_secret is consumed and a new one is produced. Moreover, the tuple containing the group_context, commit_secret and ratchet tree can be seen as the input value of the MLS based PRF-PRNG scheme. The group_context and commit_secret are part of the input, since the key schedule takes them (along with the init_secret) as its input. The ratchet tree is part of the input, to let the secret tree know how many leaf nodes are contained in the ratchet tree (since according to Section 4.9, the secret and ratchet trees in a given epoch have the same structure, modulo the node contents). Note that, the commit_secret, group_context and ratchet tree can not be part of the state instead, because they are not modified by the key schedule or the secret tree, but by the CGKA component instead. The output values of the MLS based PRF-PRNG scheme are a concatenation of: (1) all the group_secrets (excluding the init_secret), derived by the key schedule, and (2) all the handshake and application secrets derived from the leaf node secrets of the secret tree.

5.1.3 KEGSE

A KEGSE scheme can be seen as the group analogue of a forward-secure authenticated encrpytion scheme with associated data (FS-AEAD), described in Section 3.4.3. A KEGSE scheme is a stateful primitive that essentially models an SGM scheme (and hence protocol) within a single epoch. Thus, the goal of a KEGSE scheme is to allow a (static) group of clients to communicate with each other securely and asynchronously. Moreover, a KEGSE scheme is a symmetric primitive, since each group member starts with the same set of shared secrets (in its state), that it then evolves over time whenever it needs to send or receive a message to or from the group.

A KEGSE scheme is considered secure, if all the following properties hold; given an active adversary, (1) the messages exchanged within the group are indistinguishable from one another (confidentiality), (2) legitimate messages are unforgable, i.e can not be formed by anyone apart from the group members (authenticity), (3) senders of messages are indistinguishable from one another (anonymity), and (4) upon state compromise of a client, all messages sent and received by the client prior to state compromise remain secure, in the sense of confidentiality, authenticity and anonymity (forward-security).

Given this, we can view the framing of Handshake and Application messages (see Section 4.10) along with the metadata_secret produced by the MLS key schedule (see Section 4.8) and the handshake and application secrets (and state), initialised from leaf secrets in the secret tree using the initialiseRatchets_step1 and initialiseRatchets_step2 algorithms (see Section 4.9) as forming an MLS based instantiation of a KEGSE scheme.

5.1.4 Inter-component data flow

As explained in Section 5.1.1, the goal of a CGKA scheme is to enable all group members, who communicate asynchronously, to derive (the same) fresh secret random value *scr*. This fresh secret *scr* is seen as a high entropy source, that can be further processed by a PRF-PRNG scheme to derive more high entropy secrets. These PRF-PRNG derived secrets can then, in turn, be used to refresh the state of the KEGSE scheme.

Note that this is exactly the data flow that occurs in MLS, displayed in Figure 5.1. Namely each time a client processes a Commit message (or Welcome message), it modifies its ratchet tree and group_context in some way (depending on the proposals committed). This change in the ratchet tree then propagates to the root (according to 4.7.3), from which a new commit_secret is derived (which has the same value across all members).

This new commit_secret, along with the new group_context and ratchet tree (PRF-PRNG input), is then used by the key schedule (PRF-PRNG), along

with its init_secret (PRF-PRNG state), to produce the welcome_secret, metadata_secret, confirmation_key and enc_secret (short for encryption_secret) and a new init_secret (PRF-PRNG state). Thereafter, the enc_secret is used to derive a secret tree, whose leaves are further processed (according to the initialiseRatchets_step1 algorithm, in Figure 4.55) to derive handshake and application secrets. We can then view the handshake, application secrets and group secrets (derived by the MLS key schedule, excluding the init_secret), as forming a high-entropy pool of secrets.

Finally, the MLS based KEGSE scheme will use the metadata_secret, handshake and application secrets to refresh its state. More concretely, it will use the handshake and application secrets to initialise its handshake and application ratchet state (according to the initialiseRatchets_step2 algorithm, in Figure 4.56). Moreover, it will take the metadata_secret, in attempts to ensure sender anonymity.

5.2 PRF-PRNG

We start this section by defining the syntax of a PRF-PRNG, which matches the one given in [ACD18]. We then give the concrete MLS based instantiation of a PRF-PRNG scheme by combining the MLS key schedule and secret tree (as explained in Section 5.1.2). The security game of the PRF-PRNG matches the one provided by [ACD18] (whose intuition we provided in 3.4.3) and hence we omit its definition.

5.2.1 PRF-PRNG Syntax

Definition 5.1 *A PRF-PRNG scheme* P *specifies two deterministic algorithms* $\mathsf{P.Init}$ *and* $\mathsf{P.Up}$*. Associated to* P *is a keyspace* $\mathsf{P}.\mathcal{K}$*. The algorithms are defined as follows:*

- $\mathsf{P.Init}$ algorithm takes in a key $\mathsf{k} \in \mathsf{P}.\mathcal{K}$ and produces a state st, denoted as $\mathsf{st} \leftarrow \mathsf{P.Init}(\mathsf{k})$.
- $\mathsf{P.Up}$ algorithm takes in a state st and an input $\mathcal{I}$ and produces a new state st' and an output $\mathcal{R}$, denoted as $(\mathsf{st}', \mathcal{R}) \leftarrow \mathsf{P.Up}(\mathsf{st}, \mathcal{I})$.

5.2.2 Instantiating a PRF-PRNG scheme

In this section we give a concrete instantiation of a PRF-PRNG scheme based on the MLS protocol. Let KDF be a key derivation function. Then $\mathsf{P} = \mathsf{MLS\text{-}PRF\text{-}PRNG}[\mathsf{KDF}]$ is a PRF-PRNG scheme with $\mathsf{P}.\mathcal{K} = \{0,1\}^{\mathsf{KDF.Nh}}$, whose algorithms are defined as shown in Figure 5.2. As explained in 5.1.2, the state of P is the secret bit-string init_secret, the input values $\mathcal{I}$ are tuples consisting of a `GroupContext` structure, bit-string commit_secret and a

ratchet tree. The output values $\mathcal{R}$ are a concatenation of the following bitstrings: (1) all the group_secrets (excluding the init_secret), derived by the key schedule, and (2) all the handshake and application secrets derived from the leaf node secrets of the secret tree, according to the initialiseRatchets_step1 algorithm (defined in Figure 4.55).

```
P-Init(k):
  init_secret ← k
  return init_secret

P-Up(st, I)
  init_secret ← st
  (group_context, commit_secret, ratchet_tree) ← I
  (init_secret, (wel_secret, enc_secret, meta_secret, confirm_key)) ← KeySchedule(group_context, commit_secret, init_secret)
  secret_tree ← formSecretTree(KDF,ratchet_tree,enc_secret)
  (ID2app, ID2hand) ← initialiseRatchets_step1(KDF, ratchet_tree, secret_tree)
  R ← wel_secret‖meta_secret‖confirm_key
  for ID in ID2app.keys:
    R ← R‖ID2app[ID]‖ID2hand[ID]
  return (init_secret, R)

KeySchedule(group_context, commit_secret, init_secret):
  joiner_secret ← KDF.Expand(KDF.Extract(commit_secret, init_secret), ⟨KDF.Nh, mls10joiner⟩, KDF.Nh)
  wel_secret ← KDF.Expand(KDF.Extract(0,joiner_secret),⟨KDF.Nh,'mls10welcome'⟩, KDF.Nh)
  epoch_secret ← KDF.Expand(KDF.Extract(0,joiner_secret), ⟨KDF.Nh,'mls10epoch'⟩, KDF.Nh)
  meta_secret ← KDF.Expand(epoch_secret, ⟨KDF.Nh,'mls10senderdata'⟩, KDF.Nh)
  enc_secret ← KDF.Expand(epoch_secret, ⟨KDF.Nh,'mls10encryption'⟩, KDF.Nh)
  confirm_key ← KDF.Expand(epoch_secret, ⟨KDF.Nh,'mls10confirm'⟩, KDF.Nh)
  init_secret ← KDF.Expand(epoch_secret, ⟨KDF.Nh,'mls10init'⟩, KDF.Nh)
  return (init_secret, (wel_secret, enc_secret, meta_secret, confirm_key))
```

Figure 5.2: Instantiating PRF-PRNG based on MLS. The formSecretTree and initialiseRatchets_step1 are the algorithms we defined in Figures 4.54 and 4.55 respectively.

5.3 Key-Evolving Group Symmetric Encryption Scheme (KEGSE) with Associated Data

We start this section by defining the syntax of a KEGSE scheme. We then define what we mean by a KEGSE scheme to be correct and secure. We consider two notions of KEGSE security in this work. Informally the basic security properties a secure KEGSE scheme must provide is authenticity, confidentiality and forward security of messages. This basic security is captured by the game $\mathsf{G}^{\text{fs-gaead}}$ shown in Figure 5.4. The stronger notion that we consider (which implies the basic one) is captured by the game $\mathsf{G}^{\text{fs-anonim-gaead}}$ shown in Figure 5.5. Apart from capturing the basic security properties (confidentiality, authenticity and forward security of messages) it also captures the notion of sender anonymity and the forward security of sender anonymity. We consider this property, since the MLS protocol also aims to provide metadata protection, that in part requires sender anonymity to hold. The work by Chan and Rogaway [CR19] introduces the concept of anonymous nonce-based authenticated encryption schemes (anAE), which provide the same security guarantees as the traditional nonce-based authenticated encryption (NAE) schemes as well as sender privacy, which in turn

implies anonymity. However their notion of privacy does not include an adversary who is allowed to compromise honest clients. Therefore we believe that this work is the first to study forward secure anonymity as a property in any scheme.

After defining both security notions of a KEGSE scheme we divert our attention to a simpler primitive termed as a key evolving symmetric encryption (KESE) scheme with associated data. This simpler primitive models a KEGSE scheme with a single sender. We define two security notions for such a scheme, which essentially capture all the properties the basic KEGSE security notion captures, of course accommodating for the single sender scenario. The first security notion captures the security of a single sender and multiple receivers, whereas the second notion captures the security of a scenario with a single sender and single receiver. The later security notion has been studied before, however we deem the existing definitions inadequate and provide attacks illustrating the definitional problems. We provide an MLS based instantiation of a KESE scheme and prove it secure against our second KESE security definition. The security proof for the MLS based KESE scheme in the single sender multiple receivers scenario is omitted as the approach would be identical to the one taken for the first security notion.

We then define a KEGSE instantiation in terms of a KESE scheme and argue that if a KESE scheme is secure in the single sender multiple receivers scenario, then the KESE based KEGSE scheme is secure in the basic sense. We end this section by showing that a KEGSE instantiation in terms of any KESE scheme (and hence including the MLS based one) does not meet the stronger KEGSE security definition by providing an adversary who breaks the FS-anonymity property. In this way we show that MLS does not meet all the requirements we believe it strives to achieve based on the architecture document [OBR+21]. We end this chapter by giving a KEGSE instantiation based on puncturable function families we trust meets the stronger KEGSE security.

5.3.1 KEGSE Syntax

A KEGSE scheme consists of three algorithms: initialization algorithm, send algorithm and receive algorithm. The initialization algorithm takes in an initial metadata secret k, list of group member identifiers ids, encryption secrets scrts (one for each group member) and an index i such that $\mathsf{ids}[\mathsf{i}]$ is the identifier of the initialised client; it then outputs the initial state st of client $\mathsf{ids}[\mathsf{i}]$. The sending algorithm takes in the state of the sender st, a message to be encrypted m and associated data a and outputs a new state st' and ciphertext c. The receive algorithm takes in a state st of the receiver, associated data a and a ciphertext c and outputs a new state st', the identity of the sender of this ciphertext sid, the underlying message m and a message counter i.

Since messages may arrive out of order or be dropped, the message counter i ensures that, when a message m is delivered to the recipient, the recipient is able to place it in the correct spot in relation to the other messages already received. This allows the receiver id to sequence the messages in the same order in which they were sent by the sender sid. Therefore, each message and ciphertext has a corresponding counter i.

Definition 5.2 *A key-evolving symmetric group encryption scheme with associated data* KEGSE *specifies algorithms* $\mathsf{KEGSE.Init}$, $\mathsf{KEGSE.Send}$, $\mathsf{KEGSE.Recieve}$. *Associated to* KEGSE *is a set of metadata secrets* $\mathsf{KEGSE.MK}$, *a set of message encryption secrets* $\mathsf{KEGSE.DK}$, *a set of identifiers* $\mathsf{KEGSE.ID}$ *and a set of random values* $\mathsf{KEGSE}.\mathcal{R}$. *Algorithms* $\mathsf{KEGSE.Recieve}$ *and* $\mathsf{KEGSE.Init}$ *are deterministic and* $\mathsf{KEGSE.Send}$ *is probabilistic. The algorithms are defined as follows:*

- $\mathsf{KEGSE.Init}$ algorithm takes in a metadata secret $\mathsf{k} \in \mathsf{KEGSE.MK}$, a list of identifiers $\mathsf{ids} \subseteq \mathsf{KEGSE.ID}$, a list of encryption secrets $\mathsf{scrts} \subseteq \mathsf{KEGSE.DK}$ such that $|\mathsf{ids}| = |\mathsf{scrts}|$ and an integer $0 \leq \mathsf{i} < |\mathsf{ids}|$; it outputs a state st denoted as $\mathsf{st} \leftarrow \mathsf{KEGSE.Init}(\mathsf{k}, \mathsf{ids}, \mathsf{scrts}, \mathsf{i})$.
- $\mathsf{KEGSE.Send}$ algorithm takes in a state st a message $\mathsf{m} \in \{0,1\}^*$ and associated data $\mathsf{a} \in \{0,1\}^*$ and outputs a state and a ciphertext pair $(\mathsf{st}', \mathsf{c})$ denoted as $(\mathsf{st}', \mathsf{c}) \leftarrow\!\$\ \mathsf{KEGSE.Send}(\mathsf{st}, \mathsf{m}, \mathsf{a})$.
- $\mathsf{KEGSE.Recieve}$ algorithm takes a state st, associated data $\mathsf{a} \in \{0,1\}^*$ and a ciphertext c and produces a new state st', counter i, identifier sid and message $\mathsf{m} \in \{0,1\}^* \cup \{\bot\}$ denoted as $(\mathsf{st}', \mathsf{i}, \mathsf{sid}, \mathsf{m}) \leftarrow \mathsf{KEGSE.Recieve}(\mathsf{st}, \mathsf{a}, \mathsf{c})$.

5.3.2 KEGSE Correctness

In this section we define what we mean by a correct KEGSE scheme. Informally a KEGSE scheme is correct if any ciphertext sent to the group can be immediately decrypted by each group member no matter the order of delivery. The game $\mathsf{G}^{\mathsf{kegse\text{-}corr}}$, defined in Figure 5.3, captures this intuition by initialising a set trans representing all ciphertexts transmitted within the group. More specifically it contains elements of the form $(\mathsf{id}, \mathsf{rid}, \mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c})$ where id is the identifier of the sending client, rid is the identifier of the receiver, i is the message counter corresponding to message m, c is the ciphertext produced and a the associated data.

The game calls $\mathcal{A}_1$ to obtain the identifiers of the clients forming the group ids and checks that no client is repeated twice. It also samples the metadata secret k used by the group and the encryption secrets scrts, one for each client in the group. It then uses these values to initialise each client via $\mathsf{KEGSE.Init}$ in the group and saves the produced initial state of each client in the st map. Given a client id $\mathsf{send_ctr}[\mathsf{id}]$ is a monotonically increasing integer starting at 0 that corresponds to the number of messages sent by client id so

far in the game. It also maintains a win flag win which is initialised as `false` and is set to `true` if and only if the adversary manages to break the correctness of the scheme. This occurs if and only if a ciphertext that was transmitted (recorded in trans set) fails to be decrypted to the underlying message, message counter and sender via KEGSE.Recieve. The adversary $\mathcal{A}_2$ is granted access to a transmit oracle which crafts legitimate ciphertexts and deliver oracle which receives only ciphertexts created via the transmit oracle. Note that the attacker is allowed to deliver messages out of order but that the decryption should still remain correct.

```
G^{kegse-corr}_{KEGSE,A}:

    trans ← ∅, scrts ← []
    k ←$ KEGSE.MK
    win ← false
    (A_1, A_2) ← A
    (state, ids) ←$ A_1()
    if ∃i, j ∈ range(|ids|) : i ≠ j ∧ ids[i] = ids[j]:
        return false
    for id in ids:
        s ←$ KEGSE.DK
        scrts ← scrts + [s]
    for id in ids:
    for i in range(|ids|):
        st[ids[i]] ← KEGSE.Init(k, ids, scrts, i)
        send_ctr[ids[i]] ← 0
    A_2^{transmit,deliver}(state)
    return win

transmit(id,a,m):

    (st[id],c) ←$ KEGSE.Send(st[id],m,a)
    for rid in ids:
        trans ← trans ∪ {(id, rid, send_ctr[id], a, m, c)}
    send_ctr[id] ++
    return c

deliver(id,a,c):

    req ∃! sid, i, m : (sid, id, i, a, m, c) ∈ trans
    (st[id], i', sid', m') ← KEGSE.Recieve(st[id], a, c)
    if i', m', sid' ≠ i, m, sid:
        win ← true
    else:
        trans ⇐ (sid', id, i', a, m', c)
    return (i', sid', m')
```

Figure 5.3: The KEGSE correctness game and its oracles.

Correctness A KEGSE scheme is (ϵ, t, q)-correct if for all (t, q)-adversaries $\mathcal{A}$:

$$\Pr\left[\mathsf{G}^{\text{kegse-corr}}_{\text{KEGSE},\mathcal{A}} \right] \leq \epsilon$$

where an attacker is a pair of algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, and is parametrised over its running time t and number of queries q it makes throughout the game. The adversary $\mathcal{A}_1$ returns a list of identifiers $\mathsf{ids} \subseteq \mathsf{KEGSE.ID}$ along with some state state that contains information about its computation; state could include the random coins used by $\mathcal{A}_1$, the list of identifiers ids returned by $\mathcal{A}_1$, etc. The adversary $\mathcal{A}_2$ takes in the state information state forwarded by $\mathcal{A}_1$ and uses it to interact with its oracles transmit and deliver.[1]

5.3.3 KEGSE Security

In this section we define two notions of security for a KEGSE scheme. The first security notion (FS-GAEAD) captures confidentiality, authenticity and forward security of messages exchanged within a static (no members added

[1] We consider a KEGSE scheme to provide perfect correctness if it is $(0, \infty, \infty)$-correct.

or removed) group of clients. The second notion (FS-ANONIM-GAEAD) captures all properties FS-GAEAD security captures with the addition of sender anonymity and forward secure sender anonymity.

FS-GAEAD security Consider game $\mathsf{G}^{\text{fs-gaead}}$ of Figure 5.4, associated to a key-evolving (stateful) group symmetric encryption scheme with associated data KEGSE, and to an adversary $\mathcal{A}$ (pair of algorithms $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$) which captures the basic security properties (confidentiality, authenticity and forward security of messages). The advantage of $\mathcal{A}$ in breaking the FS-GAEAD security of KEGSE is defined as:

$$\mathsf{Adv}^{\text{fs-gaead}}_{\mathsf{KEGSE}}(\mathcal{A}) = 2 \cdot \Pr\left[\, \mathsf{G}^{\text{fs-gaead}}_{\mathsf{KEGSE},\mathcal{A}} \,\right] - 1. \quad (5.1)$$

The game samples a random challenge bit b, calls $\mathcal{A}_1$ to obtain the identifiers of the clients forming the group ids and checks that no client is repeated twice. It initialises all the variables it needs to keep track of during execution and then requires the adversary $\mathcal{A}_2$ to guess the bit b. More specifically since we want the security game to capture forward-security of messages we need to allow state compromise to occur via corr oracle. To capture authenticity and confidentiality, $\mathcal{A}_2$ must be given the power to inject malicious ciphertexts via a Dec oracle and to call a left-or-right (LoR) oracle. However these requirements interfere with each other (when a receiver of messages in transition is compromised, these messages lose all security since the receiver needs to keep state required to process the incoming messages; an adversary can trivially forge and decrypt all future ciphertexts when it leaks the state of a client). In order for these trivial attacks to be prevented the game keeps track of all ciphertexts in transmission and challenge ciphertexts via sets trans and chall respectively. The game also maintains $\mathsf{comp}_{\mathsf{fut}}$ and $\mathsf{comp}_{\mathsf{past}}$ which for each client keep track of compromised message counters due to state leaks. For a given client id the $\mathsf{comp}_{\mathsf{fut}}[\mathsf{id}]$ captures all the future messages yet to be sent by id that are trivially forgable and decryptable upon state compromise of any group member. It is enough for $\mathsf{comp}_{\mathsf{fut}}[\mathsf{id}]$ to store a single integer representing the smallest of all future compromised message counters since sending occurs in order. The $\mathsf{comp}_{\mathsf{past}}[\mathsf{id}]$ is the set of all message counters corresponding to messages that are sent by id that is yet to be received by some compromised group member. The $\mathsf{comp}_{\mathsf{past}}[\mathsf{id}]$ is a set and not an integer because a receiver may get messages out of order, whereas sending always occurs in order.

Similarly to the correctness game, $\mathsf{G}^{\text{fs-gaead}}$ samples the metadata secret k and the encryption secrets sts and uses these values to initialise each client in the group via KEGSE.Init. It then saves the produced initial state of each client in the st map. For each client id it maintains that client's message counter in send_ctr[id] and initialises it to 0 since no messages have been sent by any

group member at the beginning of the game. Given a sender id, a receiver id′ and a message counter i the not_used[id][id′][i] is set to true if the information maintained by client id′ needed to receive the i-th message of id has not been used (deleted) from its state. If it is used (deleted) then not_used[id][id′][i] is set to true. At the beginning of the game since no messages have been received yet by any client (and hence all clients have all the information in their state to decrypt all messages) not_used[id][id′][i] is set to true for all id, id′, i. Maintaining this variable ensures that deletion of certain key material occurs (and hence future security can hold) and that the scheme is secure against replay attacks.

```
G^{fs-gaead}_{KEGSE,A}:

    b ←$ {0,1}
    trans, chall ← ∅, scrts ← []
    comp_fut[.] ← ∞
    comp_past[.] ← ∅
    k ←$ KEGSE.MK
    (A_1, A_2) ← A
    (state, ids) ←$ A_1()
    if ∃i, j ∈ range(|ids|) : i ≠ j ∧ ids[i] = ids[j]:
        return false
    for id in ids:
        s ←$ KEGSE.DK
        scrts ← scrts + [s]
    for i in range(|ids|):
        st[ids[i]] ← KEGSE.Init(k, ids, scrts, i)
        send_ctr[ids[i]] ← 0
        for id′ in ids:
            not_used[ids[i]][id′][.] ← true
    b′ ←$ A_2^{LoR,Dec,corr}(state)
    return b′ = b

LoR(a,r, id,m_0,m_1):

    req |m_0| = |m_1| and comp_fut[id] = ∞
    (st[id], c) ← KEGSE.Send(st[id], m_b, a;r)
    for id′ in ids:
        trans ← (id, id′, send_ctr[id], a, m_b, c)
        if m_0 ≠ m_1:
            chall ← (id, id′, send_ctr[id], a, m_b, c)
    send_ctr[id] ++
    return c

corr(id):

    req ∄id′, i, a, m, c : (id′, id, i, a, m, c) ∈ chall
    for id′ ∈ ids :
        comp_fut[id′] ← send_ctr[id′]
        S ← {i | (∃a, m, c, r : (id′, id, i, a, m, c) ∈ trans)}
        comp_past[id′] ← comp_past[id′] ∪ S
    return st[id]

Dec(id,a,c):

    (st[id], i, sid, m) ← KEGSE.Receive(st[id],a,c)
    if m = ⊥:
        return ⊥
    if not_used[sid][id][i] and ((sid, id, i, a, m, c) ∈ trans or i ≥ comp_fut[sid] or i ∈ comp_past[sid]):
        trans ⇐ (sid, id, i, a, m, c)
        chall ⇐ (sid, id, i, a, m, c)
        not_used[sid][id][i] ← false
        return ⊥
    if b=1:
        return m
    else:
        return ⊥
```

Figure 5.4: The FS-GAEAD security game and its oracles.

The LoR oracle takes in associated data a, nonce r, an identifier id representing the sender and two messages m_0, m_1 and returns an encryption c of m_b created by calling the KEGSE.Send algorithm. It also increments the message counter send_ctr of the sender id and records for each member id' in the group $(id, id', send_ctr[id], a, m_b, c)$ into trans and possibly chall if the ciphertext is a challenge ciphertext ($m_0 \neq m_1$) indicating that the ciphertext c corresponding to counter send_ctr[id], message m_b, associated data a has been sent by id to id′. An adversary is allowed to call the LoR if messages m_0 and m_1 are of the same length and if no compromise occurred. The second requirement does not weaken the adversary $\mathcal{A}$ because $\mathcal{A}$ can leak the state of all clients via the corr oracle and hence obtain all the information it needs to simulate the LoR oracle for any client. We put this requirement in place in order to make it easier to prove a scheme secure against the definition.

The Dec oracle takes in a receiver id, associated data a and ciphertext c and

returns a message $\mathsf{m} \neq \bot$ if and only if c is a malicious (not produced by LoR) non-trivial (false on the second if condition) ciphertext that is successfully decrypted by KESE.Receive and $\mathsf{b} = 1$. A ciphertext c is trivial if and only if the receiver id did not delete the information needed to decrypt the i-th ciphertext sent by sid ($\mathsf{not_used}[\mathsf{sid}][\mathsf{id}][\mathsf{i}]$ holds) and the ciphertext c was either created by the LoR oracle ($(\mathsf{sid}, \mathsf{id}, \mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c}) \in \mathsf{trans}$) or it corresponds to a message counter that is considered to be compromised ($\mathsf{i} \geq \mathsf{comp}_{\mathsf{fut}}[\mathsf{sid}]$ or $\mathsf{i} \in \mathsf{comp}_{\mathsf{past}}[\mathsf{sid}]$)). If the ciphertext is trivial Dec removes the record (if any) $(\mathsf{sid}, \mathsf{id}, \mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c})$ from the trans and chall set since the ciphertext c was transmitted to client id and sets $\mathsf{not_used}[\mathsf{sid}][\mathsf{id}][\mathsf{i}]$ to false. Doing this ensures that a secure KEGSE scheme requires each client to delete all information pertaining to any message the client received.

Finally the corr oracle takes in a client identifier id and returns that client's state st[id]. It can only be called if all challenge ciphertexts have been delivered to id since id's state would trivially have to contain information needed to decrypt this challenge ciphertext successfully. The corr oracle also sets the $\mathsf{comp}_{\mathsf{fut}}$ and $\mathsf{comp}_{\mathsf{past}}$ variables appropriately to indicate which message counters (and hence which messages and ciphertexts) should be considered as compromised after leaking the state of id. More specifically for each group member id' it sets $\mathsf{comp}_{\mathsf{fut}}[\mathsf{id}']$ to the current message counter of client id' indicated by $\mathsf{send_ctr}[\mathsf{id}']$ since with this state leak the adversary has the exact same information as the honest sender. [2] The corr oracle also adds all the message counters corresponding to a message not yet received by id to $\mathsf{comp}_{\mathsf{past}}[\mathsf{id}']$.

Definition 5.3 *A key-evolving (stateful) group symmetric encryption scheme with associated data* KEGSE *is (ϵ, t, q)-FS-GAEAD-secure if for all (t, q)-attackers $\mathcal{A}$:*

$$\mathsf{Adv}^{\mathsf{fs\text{-}gaead}}_{\mathsf{KEGSE}}(\mathcal{A}) \leq \epsilon$$

where $\mathcal{A}$ is a pair of two algorithms $(\mathcal{A}_1, \mathcal{A}_2)$ parametrised over its running time t and a value q which represents the maximum number of queries made to the LoR *oracle. Furthermore for each query (a,i,c,r) made to the* Dec *oracle $i < q$ must hold.*

FS-ANONIM-GAEAD security Consider game $\mathsf{G}^{\mathsf{fs\text{-}anonim\text{-}gaead}}$ shown in Figure 5.5, associated to a key-evolving (stateful) group symmetric encryption scheme with associated data KEGSE, and to an adversary $\mathcal{A}$. The advantage

[2] Note that since the LoR oracle is disabled upon the first state compromise, the send_ctr will never change after the first compromise either. This further implies that we do not need to take care that $\mathsf{comp}_{\mathsf{fut}}[\mathsf{id}']$ for some id' is set to the minimum message counter known to the adversary.

of $\mathcal{A}$ in breaking the FS-GAEAD security of KEGSE is defined as:

$$\mathsf{Adv}^{\text{fs-anonim-gaead}}_{\mathsf{KEGSE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\text{fs-anonim-gaead}}_{\mathsf{KEGSE},\mathcal{A}}\right] - 1. \tag{5.2}$$

The game is very similar to $\mathsf{G}^{\text{fs-gaead}}$ with the only differences highlighted in gray. These differences are all that needs to be introduced in order to capture sender anonymity and future anonymity properties. The LoR oracle now takes in two sender identifier and message pairs $(\mathsf{id}_0, \mathsf{m}_0)$ and $(\mathsf{id}_1, \mathsf{m}_1)$ instead of a single sender and two messages. Intuitively the idea is that given a ciphertext an adversary is not only unable to distinguish which message (out of two possible messages) is the one underlying the observed ciphertext but is also unable to distinguish who was the sender of this ciphertext (out of two possible senders). This is of course assuming that the sender ids are of the same length. Thus, a call to the LoR oracle is now allowed if the sender identifiers queried are of the same length (apart from the requirements already present from the FS-GAEAD security notion). Consequently a ciphertext is considered a challenge ciphertexts if it challenges the confidentiality of messages ($\mathsf{m}_0 \neq \mathsf{m}_1$) or if it challenges the anonymity (privacy) of a sender ($\mathsf{id}_0 \neq \mathsf{id}_1$).

The notion of forward secure sender anonymity, extends from the forward security of messages intuitively. Namely if the state of some client id is leaked then we require that all ciphertexts client id sent and received leak as much about the sender as if no client state leak occurred. Of course if more than one state leak occurs then a ciphertext c does not leak the sender's identity if all the clients whose state has been leaked have received c. Since a KEGSE scheme requires message counters (in order to for example allow an instant messaging application to position chat messages within a group correctly) a client would need to maintain some information in its state pertaining to its own sending counter. Hence because we allow the state of any client id to be leaked we need some additional variables to prevent trivial wins by $\mathcal{A}$ who obtains such message counter information from a leaked state. Namely the $\mathsf{G}^{\text{fs-anonim-gaead}}$ game maintains two additional maps left and right which map a client identifier id to an integer representing the amount of times the client id occurred in the left position and right position of the LoR oracle respectively. Then in order to allow the corruption of a client id we require that the amount of times id has occurred in the left position and right position of the LoR oracle is the same. Note that this requirement allows for stronger adversaries than a requirement that demands all clients send the same amount of ciphertexts prior to compromise. The Dec oracle does not need to change in any way. An adversary $\mathcal{A}$ wins if it manages to guess the challenge bit the game $\mathsf{G}^{\text{fs-anonim-gaead}}$ samples at the very beginning.

Definition 5.4 *A key-evolving group symmetric encryption scheme with associated data* KEGSE *is* (ϵ, t, q)*-FS-ANONIM-GAEAD-secure if for all* (t, q)*-attackers* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\mathsf{fs\text{-}anonim\text{-}gaead}}_{\mathsf{KEGSE}}(\mathcal{A}) \leq \epsilon$$

where $\mathcal{A}$ *is a pair of two algorithms* $(\mathcal{A}_1, \mathcal{A}_2)$ *parametrised over its running time t and a value* q *which represents the maximum number of queries made to the* LoR *oracle. Furthermore for each query (a,i,c,r) made to the* Dec *oracle* $i < q$ *must hold.*

```
G^{fs-anonim-gaead}_{KEGSE,A}:

  b ←$ {0,1}
  trans, chall ← ∅, scrts← []
  comp_fut[.] ← ∞
  comp_past[.] ← ∅
  k ←$ KEGSE.MK
  (A_1, A_2) ← A
  (state, ids) ←$ A_1()
  if ∃i,j ∈ range(|ids|) : i ≠ j ∧ ids[i] = ids[j]:
      return false
  for id in ids:
      s ←$ KEGSE.DK
      scrts ← scrts + [s]
  for i in range(|ids|):
      st[ids[i]] ← KEGSE.Init(k, ids, scrts, i)
      left[ids[i]], right[ids[i]] ← 0
      send_ctr[ids[i]] ← 0
      for id' in ids:
          not_used[ids[i]][id'][.] ← true
  b' ←$ A_2^{LoR,Dec,corr}(state)
  return b' = b

LoR(a,r, (id_0, m_0), (id_1, m_1)):

  req |m_0| = |m_1| and |id_0| = |id_1| and comp_fut[id_0] = ∞
  (st[id_b], c) ← KEGSE.Send(st[id_b], m_b,a;r)
  for id in ids:
      trans ← (id_b, id, send_ctr[id_b], a, m_b, c)
      if (id_0, m_0) ≠ (id_1, m_1):
          chall ← (id_b, id, send_ctr[id_b], a, m_b, c)
  left[id_0] ++, right[id_1] ++
  send_ctr[id_b] ++
  return c

corr(id):

  req ∄id', i, a, m, c : (id', id, i, a, m, c) ∈ chall and left[id] = right[id]
  for id' ∈ ids :
      comp_fut[id'] ← send_ctr[id']
      S ← {i | (∃a, m, c, r : (id', id, i, a, m, c) ∈ trans)}
      comp_past[id'] ← comp_past[id'] ∪ S
  return st[id]

Dec(id,a,c):

  (st[id], i, sid, m) ← KEGSE.Receive(st[id],a,c)
  if m = ⊥:
      return ⊥
  if not_used[sid][id][i] and ((sid, id, i, a, m, c) ∈ trans or i ≥ comp_fut[sid] or i ∈ comp_past[sid]):
      trans ←− (sid, id, i, a, m, c)
      chall ←− (sid, id, i, a, m, c)
      not_used[sid][id][i] ← false
      return ⊥
  if b=1:
      return m
  else:
      return ⊥
```

Figure 5.5: The FS-ANONIM-GAEAD security game and its oracles. The code added on top of the FS-GAEAD security game is highlighted in gray.

5.3.4 Key-Evolving (Stateful) Symmetric Encryption Scheme with Associated Data (KESE)

A key-evolving (stateful) symmetric encryption scheme with associated data (KESE) scheme can be considered as a single-sender variant of a KEGSE scheme.

KESE Syntax

Informally a KESE scheme also consists of three algorithms: initialization, send and receive algorithm. The initialization algorithm takes in a secret k and a client identifier id and outputs the initial sending state $\mathsf{st_S}$ and the initial receiving state $\mathsf{st_R}$ corresponding to client id. If client id wishes to

send a message m securely, it will call the sending algorithm that takes in the sending state $\mathsf{st_S}$ corresponding to the client id wishing to send, message m, associated data a and a nonce r and outputs a new sending state $\mathsf{st'_S}$ and ciphertext c. The sender will then transmit the produced ciphertext c along with the associated data a, message counter i corresponding to c and the nonce r it used to form c into the network. The client receiving messages sent by client id will call upon delivery call the receive algorithm that takes in the receiving state $\mathsf{st_R}$ corresponding to sender id. The receiver will also provide the delivered associated data a, message counter i, ciphertext c and a nonce r to the receive algorithm which in turn outputs a new receiver state $\mathsf{st'_R}$ and the underlying message m. The message counter has the same meaning as in a KEGSE scheme.

Definition 5.5 *A key-evolving (stateful) symmetric encryption scheme with associated data* KESE *specifies algorithms* KESE.Init, KESE.Send, KESE.Recieve. *Associated to* KESE *is a set of initialization secrets* $\mathsf{KESE}.\mathcal{K}$*, a set of identifiers* KESE.ID *and a set of random values* $\mathsf{KESE}.\mathcal{R}$*. All algorithms are deterministic and are defined as follows:*

- KESE.Init algorithm takes a secret $\mathsf{k} \in \mathsf{KESE}.\mathcal{K}$ and identifier $\mathsf{id} \in \mathsf{KESE.ID}$ and outputs a sender state $\mathsf{st_S}$ and receiver state $\mathsf{st_R}$ which is denoted as $(\mathsf{st_S}, \mathsf{st_R}) \leftarrow \mathsf{KESE.Init}(\mathsf{k}, \mathsf{id})$.
- KESE.Send algorithm takes in a sender state $\mathsf{st_S}$, a message $\mathsf{m} \in \{0,1\}^*$, associated data $\mathsf{a} \in \{0,1\}^*$ and nonce $\mathsf{r} \in \mathsf{KESE}.\mathcal{R}$ and outputs a new sender state $\mathsf{st'_S}$ and a ciphertext c denoted as $(\mathsf{st'_S}, \mathsf{c}) \leftarrow \mathsf{KESE.Send}(\mathsf{st_S}, \mathsf{m}, \mathsf{a}, \mathsf{r})$.
- KESE.Receive takes as input a receiver state $\mathsf{st_R}$, associated data $\mathsf{a} \in \{0,1\}^*$, integer i, ciphertext c and nonce $\mathsf{r} \in \mathsf{KESE}.\mathcal{R}$ and produces a new receiver state $\mathsf{st'_R}$ and a message $\mathsf{m} \in \{0,1\}^* \cup \{\perp\}$ which is denoted as $(\mathsf{st'_R}, \mathsf{m}) \leftarrow \mathsf{KESE.Receive}(\mathsf{st_R}, \mathsf{a}, \mathsf{i}, \mathsf{c}, \mathsf{r})$.

KESE Correctness

Informally a KESE scheme is correct if any ciphertext sent by the sender can be immediately decrypted by a receiver no matter the order of delivery. The game $\mathsf{G}^{\text{kese-corr}}$ captures this notion of correctness formally and is very similar to the $\mathsf{G}^{\text{kegse-corr}}$. All the differences stem from the fact that KESE represents a single sender variant of a KEGSE scheme and that the KESE.Receive algorithm, unlike the KEGSE.Receive algorithm takes in a message counter i and nonce r, and just produces the underlying message. The adversary $\mathcal{A}_1$ needs to specify an identifier id which represents the client who will send ciphertexts and identifiers $\mathsf{rcv_ids}$ representing the receiving clients. The set trans now contains records of the form $(\mathsf{id}, \mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c}, \mathsf{r})$ instead of $(\mathsf{id}, \mathsf{rid}, \mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c})$ since there is only a single sender to receive from and an explicit nonce r in KESE. The sender and receiver state pair $(\mathsf{st_S}, \mathsf{st_R})$ cor-

responding to id is obtained by calling KESE.Init. The sender then has its initial state set to st_S whilst all the receiving clients have their initial state set to st_R. All other parts of the game easily extend from the explanation given for the $G^{kegse\text{-}corr}$ game.

$G^{kese\text{-}corr}_{KESE,\mathcal{A}}$:

```
(A1, A2) ← A
(state, id, rcv_ids) ←$ A1()
trans ← ∅
send_ctr ← 0
if ∃i, j ∈ range(|rcv_ids|) : i ≠ j ∧ rcv_ids[i] = rcv_ids[j]:
    return false
win ← false
k ←$ KESE.K
(stS, stR) ← KESE.Init(k, id)
for id' in rcv_ids :
    st[id'] ← stR
A2^{transmit,deliver}(state)
return win
```

transmit(a,m,r):

```
(stS, c) ← KESE.Send(stS, m, a, r)
for id in rcv_ids:
    trans ←+ (id, send_ctr,a,m,c,r)
return (send_ctr,c)
```

deliver(id, a,i,c,r):

```
req ∃! m : (id, i, a, m, c, r) ∈ trans
(stR, m') ← KESE.Receive(stR, a, i, c, r)
if m' ≠ m:
    win ← true
else:
    trans ←- (id, send_ctr, a, m, c, r)
return m'
```

Figure 5.6: The correctness game of a key-evolving symmetric encryption scheme with associated data.

Correctness A key-evolving (stateful) symmetric encryption scheme with associated data KESE is correct if for all attackers $\mathcal{A}$:

$$\Pr\left[\, G^{kese\text{-}corr}_{KESE,\mathcal{A}} \,\right] = 0$$

where an adversary $\mathcal{A}$ is a tuple of two sub-algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$. The adversary $\mathcal{A}_1$ returns an identifier id $\in$ KESE.ID along with some state state that contains information about its computation; state could include the random coins used by $\mathcal{A}_1$, the identifier id returned by $\mathcal{A}_1$, etc. The adversary $\mathcal{A}_2$ takes in the state information state forwarded by $\mathcal{A}_1$ and uses it to query oracles transmit and deliver.

KESE Security

In this section we define what we mean by a secure KESE scheme in a scenario where there may be multiple receivers and a scenario with a single receiver (see Figure 5.7). Intuitively, the former requires that the unidirectional communication from the single sender to the multiple receivers is confidential, authentic and forward secure in the presence of an adaptive adversary, captured by game $G^{fs\text{-}aead\text{-}broadcast}$ in Figure 5.8. Similarly a secure KESE scheme in the later scenario ensures that the unidirectional communication from the single sender to the single receiver has the same security properties, captured by game $G^{fs\text{-}aead}$ in Figure 5.9.

We then further define the selective security variant of $G^{fs\text{-}aead}$ as given by

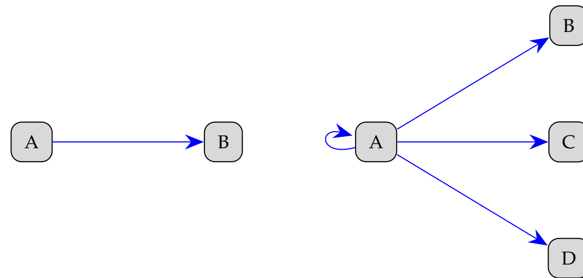


Figure 5.7: KESE scheme in the single sender and single receiver scenario (left) and KESE scheme in a single sender and multiple receivers (right) example where client A acts as the single sender in both cases.

game $\mathsf{G}^{\text{fs-aead-select}}$, shown in Figure 5.10. In the selective security variant the adversary needs to select the soonest position in the key evolution it wishes to leak pos. The game checks that the adversary does not misbehave by ensuring that the soonest key evolution state leaked occurs at position pos in the key evolution, not before, not after. If misbehaviour is detected the game discards the adversary's answer b' and instead samples the guess bit b' uniformly and independently at random itself.

We then show a reduction from breaking FS-AEAD selective security of a KESE scheme to breaking its FS-AEAD adaptive security. This in turn, will allow us to prove the MLS based instantiation in the 'simpler' selective variant instead (which would then imply FS-AEAD security of the MLS based KESE), in Section 5.3.4. The approach to proving that the KESE instantiation is FS-AEAD-broadcast secure would be exactly the same. Namely, we would define a selective variant of $\mathsf{G}^{\text{fs-aead-broadcast}}$, where pos is now the soonest position in the key evolution the adversary wishes to leak **across all receivers**. We prove the FS-AEAD security, since the games are a bit shorter and easier to understand than when the exact same argument is made for the FS-AEAD-broadcast security of the scheme.

Prior to our work, Bellare and Yee [BY01] captured forward security of symmetric schemes w.r.t. confidentiality, but not authenticity (in the single sender single receiver scenario). Alwen et al. in [ACD18] define an FS-AEAD primitive, analogous to a KESE scheme, and attempt to capture both confidentiality and authenticity and forward security of messages sent (in the single sender single receiver scenario). In appendix A.1 we argue that their notion is inadequate by defining a game equivalent to theirs in order to fit our KESE syntax and prove that the MLS based KESE scheme is not secure in their notion. The attacks found clearly show that the problem lies in their security definition rather than the MLS based KESE scheme.

FS-AEAD broadcast security Consider the game $\mathsf{G}^{\text{fs-aead-broadcast}}$ associated to a key-evolving symmetric scheme with associated data KESE and an ad-

versary $\mathcal{A}$, defined in Figure 5.8. The game captures the same basic properties as game $\mathsf{G}^{\text{fs-gaead}}$ for the scenario containing a single sender and possibly multiple receivers, and is hence a trivial simplification of it. The advantage of $\mathcal{A}$ in breaking the FS-AEAD broadcast security of KESE is defined as:

$$\mathsf{Adv}^{\text{fs-aead-broadcast}}_{\mathsf{KESE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\text{fs-aead-broadcast}}_{\mathsf{KESE},\mathcal{A}} \right] - 1. \tag{5.3}$$

```
G^{fs-aead-broadcast}_{KESE,A}:
    b <-$ {0,1}
    trans, chall <- ∅
    comp_fut <- ∞
    comp_past <- ∅
    k <-$ KESE.K
    (A_1, A_2) <- A
    (state, id, rcv_ids) <-$ A_1()
    (st_S, st_R) <- KESE.Init(k, id)
    send_ctr <- 0
    if ∃i,j ∈ range(|rcv_ids|) : i ≠ j ∧ rcv_ids[i] = rcv_ids[j]:
        return false
    for id' in rcv_ids:
        not_used[id'][.] <- true
        st[id'] <- st_R
    b' <-$ A_2^{LoR,Dec,corr-S,corr-R}(state)
    return b' = b

corr-S:
    comp_fut <- send_ctr
    return st_S

corr-R(id):
    req ∄i, a, m, c : (id, i, a, m, c) ∈ chall
    comp_fut <- send_ctr
    S <- {i | (∃a, m, c, r : (id, i, a, m, c, r) ∈ trans)}
    comp_past <- comp_past ∪ S
    return st[id]

LoR(a,m_0,m_1,r):
    req |m_0| = |m_1| and comp_fut = ∞ :
    (st_S, c) <- KESE.Send(st_S, m_b, a, r)
    for id in rcv_ids:
        trans <+ (id, send_ctr,a,m_b,c,r)
        if m_0 ≠ m_1 :
            chall <+ (id, send_ctr,a,m_b,c,r)
    send_ctr++
    return c

Dec(id,a,i,c,r):
    (st[id], m) <- KESE.Receive(st[id], a, i, c, r)
    if m = ⊥:
        return ⊥
    if not_used[id][i] and ((id,i,a,m,c,r) ∈ trans or i ≥ comp_fut or i ∈ comp_past):
        trans <- (id,i,a,m,c,r)
        chall <- (id,i,a,m,c,r)
        not_used[id][i] <- false
        return ⊥
    if b=1:
        return m
    else:
        return ⊥
```

Figure 5.8: The FS-AEAD broadcast security game and its oracles.

Definition 5.6 *A key-evolving symmetric encryption scheme with associated data* KESE *is (ϵ, t, q)-FS-AEAD-broadcast-secure if for all (t, q)-attackers $\mathcal{A}$:*

$$\mathsf{Adv}^{\text{fs-aead-broadcast}}_{\mathsf{KESE}}(\mathcal{A}) \leq \epsilon$$

where an adversary $\mathcal{A}$ is a tuple of two sub-algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ parametrised over its running time t and a value $q \in \mathbb{N}_0$. The adversary $\mathcal{A}_1$ returns the identifier of the sender id $\in$ KESE.ID*, the identifiers of the receivers* rcv_ids $\subseteq$ KESE.ID *along with some state* state *that contains information about its computation;* state *could include the random coins used by $\mathcal{A}_1$, id,rcv_ids returned by $\mathcal{A}_1$, etc. The adversary $\mathcal{A}_2$ takes in the state information* state *forwarded by $\mathcal{A}_1$ and uses it to query oracles LoR,Dec,corr-S and corr-R such that it makes at most* q *queries to the* LoR *oracle and for each query (a,i,c,r) made to the* Dec *oracle $i < q$ holds.*

FS-AEAD security Consider game $\mathsf{G}^{\text{fs-aead}}$ shown in Figure 5.9, associated to a key-evolving (stateful) symmetric encryption scheme with associated data KESE, and to an adversary $\mathcal{A}$. The advantage of $\mathcal{A}$ in breaking the FS-AEAD security of KESE is defined as $\mathsf{Adv}^{\text{fs-aead}}_{\mathsf{KESE}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A}} \right] - 1$.

```
G^{fs-aead}_{KESE,A}:
    b ←$ {0,1}
    trans, chall ← ∅
    comp_send ← ∞
    comp_rcv ← false
    not_used[.] ← true
    k ←$ KESE.K
    (A_1, A_2) ← A
    (state,id) ←$ A_1()
    send_ctr ← 0
    (st_S, st_R) ← KESE.Init(k,id)
    b' ←$ A_2^{LoR,Dec,corr-S,corr-R}(state)
    return b' = b

corr-S:
    comp_send ← send_ctr
    return st_S

corr-R:
    req chall = ∅
    comp_rcv ← true
    return st_R

LoR(a,m_0,m_1,r):
    req |m_0| = |m_1| and ¬comp_rcv and comp_send = ∞
    (st_S, c) ← KESE.Send(st_S, m_b, a, r)
    trans ←+ (send_ctr,a,m_b,c,r)
    if m_0 ≠ m_1 :
        chall ←+ (send_ctr,a,m_b,c,r)
    send_ctr + +
    return c

Dec(a,i,c,r):
    (st_R, m) ← KESE.Receive(st_R, a, i, c, r)
    if m = ⊥:
        return ⊥
    if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
        trans ←- (i,a,m,c,r)
        chall ←- (i,a,m,c,r)
        not_used[i] ← false
        return ⊥
    if b=1:
        return m
    else:
        return ⊥
```

Figure 5.9: The FS-AEAD security game.

The only differences between this game and the previous ones is that the variables $\mathsf{comp_{past}}$ and $\mathsf{comp_{fut}}$ are now replaced by $\mathsf{comp_{rcv}}$ and $\mathsf{comp_{send}}$. The $\mathsf{comp_{send}}$ now contains exactly those message counters that are compromised in case of sender's state leak. The $\mathsf{comp_{rcv}}$ is a boolean set to true if and only if the receiver is compromised. Moreover the corruption oracle is now split into two oracles corr-S and corr-R, explicitly representing the corruption of the sender and receiver respectively. Because of us separating senders and receiver's compromise, the LoR oracle is now disabled in case of either receiver or sender compromise. Observe that this requirement is fine as the adversary again can simulate the LoR oracle (since the first client compromise) because it can leak the state of the sender as many times as it wishes.

Definition 5.7 *A key-evolving (stateful) symmetric encryption scheme with associated data* KESE *is (ϵ, t, q)-FS-AEAD-secure if for all (t, q)-attackers $\mathcal{A}$:*

$$\mathsf{Adv}^{\mathsf{fs\text{-}aead}}_{\mathsf{KESE}}(\mathcal{A}) \leq \epsilon$$

where an attacker is parametrised over its running time t and a value $q \in \mathbb{N}_0$ which represents the maximum number of queries made to the LoR *oracle. Furthermore for each query (a,i,c,r) made to the* Dec *oracle $i < q$ must hold.*

FS-AEAD selective security Consider game $\mathsf{G}^{\mathsf{fs\text{-}aead\text{-}select}}$ shown in Figure 5.10, associated to a key-evolving (stateful) symmetric encryption scheme with associated data KESE, to an adversary $\mathcal{A}$ and $\mathsf{q} \in \mathbb{N}_0$. The adversary must declare the earliest state leaked in the key evolution by specifying its position pos in the key evolution. If the adversary does not wish

to leak the state, then the adversary specifies $\mathsf{pos} = \mathsf{q} + 1$. The advantage of $\mathcal{A}$ in breaking the FS-AEAD selective security of KESE is defined as $\mathsf{Adv}^{\text{fs-aead-select}}_{\mathsf{KESE,q}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathsf{G}^{\text{fs-aead-select}}_{\mathsf{KESE},\mathcal{A},\mathsf{q}}\right] - 1$.

```
G^{fs-aead-select}_{KESE,A,q}:
  b ←$ {0,1}
  trans, chall ← ∅
  comp_send ← ∞
  comp_rcv ← false
  not_used[.] ← true
  k ←$ KESE.K
  (A_1, A_2) ← A
  (state,id,pos) ←$ A_1()
  send_ctr ← 0
  soonest_rcv ← q + 1
  max_rcv ← 0
  soonest_send ← q + 1
  (st_S, st_R) ← KESE.Init(k,id)
  b' ←$ A_2^{LoR,Dec,corr-S,corr-R}(state)
  soonest ← min(soonest_rcv, soonest_send)
  if soonest ≠ pos:
    b' ←$ {0,1}
  return b' = b

corr-S:
  soonest_send ← send_ctr
  comp_send ← send_ctr
  return st_S

corr-R:
  req chall = ∅
  if ¬comp_rcv:
    soonest_rcv ← max_rcv
  comp_rcv ← true
  return st_R

LoR(a,m_0,m_1,r):
  req |m_0| = |m_1| and ¬comp_rcv and comp_send = ∞
  (st_S, c) ← KESE.Send(st_S, m_b, a, r)
  trans ←+ (send_ctr,a,m_b,c,r)
  if m_0 ≠ m_1 :
    chall ←+ (send_ctr,a,m_b,c,r)
  send_ctr + +
  return c

Dec(a,i,c,r):
  (st_R, m) ← KESE.Receive(st_R, a, i, c, r)
  if max_rcv < i:
    max_rcv ← i
  if m = ⊥:
    return ⊥
  if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
    trans ⇐ (i,a,m,c,r)
    chall ⇐ (i,a,m,c,r)
    not_used[i] ← false
    return ⊥
  if b=1:
    return m
  else:
    return ⊥
```

Figure 5.10: The FS-AEAD selective security game. The code added on top of the game $\mathsf{G}^{\text{fs-aead}}$ is highlighted in gray.

Definition 5.8 *A key-evolving (stateful) symmetric encryption scheme with associated data* KESE *is* (ϵ, t, q)*-FS-AEAD-selective-secure if for all* (t, q)*-attackers* $\mathcal{A}$*:*

$$\mathsf{Adv}^{\text{fs-aead-select}}_{\mathsf{KESE,q}}(\mathcal{A}) \leq \epsilon$$

where an adversary $\mathcal{A}$ *is a tuple of two sub-algorithms* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *parametrised over its running time* t *and a value* $q \in \mathbb{N}_0$*. The adversary* $\mathcal{A}_1$ *returns an identifier* $\mathsf{id} \in \mathsf{KESE.ID}$ *and an integer* $0 \leq \mathsf{pos} \leq \mathsf{q} + 1$ *along with some state* state *that contains information about its computation;* state *could include the random coins used by* $\mathcal{A}_1$*, id,pos returned by* $\mathcal{A}_1$*, etc. The adversary* $\mathcal{A}_2$ *takes in the state information* state *forwarded by* $\mathcal{A}_1$ *and uses it to query oracles LoR,Dec,corr-S and corr-R such that it makes at most* q *queries to the LoR oracle and for each query (a,i,c,r) made to the Dec oracle* $i < q$ *holds.*

Theorem 5.9 *Let KESE be* (ϵ, t, q)*-FS-AEAD-selective-secure. Then KESE is* (ϵ', t, q)*-FS-AEAD-secure where* $\epsilon' = (q + 2)\epsilon$*.*

Proof. Consider adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ playing in game $\mathsf{G}^{\text{fs-aead}}$ and simulating game $\mathsf{G}^{\text{fs-aead-select}}$ to adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, defined in Figure 5.11.

```
Adversary B_1:
  pos ←$ {0,1,...,q,q+1}
  (state,id) ←$ A_1()
  return (state,id,pos)

Adversary B_2^{LoR,Dec,corr-S,corr-R}(state):
  b' ←$ A_2^{LoRSim, DecSim, corr-SSim,corr-RSim}(state)
  return b'

corr-SSim:
  return corr-S

corr-RSim:
  return corr-R

LoRSim(a,m_0,m_1,r):
  return ← LoR(a,m_0,m_1,r)

DecSim(a,i,c,r):
  return Dec(a,i,c,r)
```

Figure 5.11: Adversary $\mathcal{B}$ playing in game $\mathsf{G}^{\text{fs-aead}}$-select for proof of Theorem 5.9.

The adversary $\mathcal{B}$ starts by guessing the soonest state $\mathcal{A}$ will reveal in the key evolution pos. Since $\mathcal{A}$ is parametrised over the range of queries it can ask to the LoR and Dec oracle q and the receiver and sender states are only updated by calls to these oracles, then any state in the key evolution must correspond to a position $0 \leq i \leq q$. Therefore if $\mathcal{A}$ leaks a state via corr-R or corr-S then $0 \leq \text{pos} \leq q$. If $\mathcal{A}$ does not leak a state then $\text{pos} = q + 1$. Therefore $\text{pos} \in \{0, 1, \ldots, q, q + 1\}$ and $\Pr[\text{pos} = k] = \frac{1}{q+2}$ for all $k \in \{0, 1, \ldots, q, q + 1\}$. If $\mathcal{B}$ guesses the soonest state reveal correctly then it perfectly simulates the game $\mathsf{G}^{\text{fs-aead}}$ to $\mathcal{A}$ and wins its selective game if $\mathcal{A}$ wins its adaptive game. If $\mathcal{B}$ does not guess correctly, then the game $\mathsf{G}^{\text{fs-aead-select}}$ will detect this misbehaviour, discard b′ provided by $\mathcal{B}$ and instead sample b′ locally and output $b' = b$. Therefore $\mathcal{B}$ wins if and only if $\mathcal{A}$ wins and pos=soonest or pos $\neq$ soonest and $b' = b$ when sampled by the game.

Let E denote the event pos=soonest. The probability of E occurring is:

$$\Pr[\mathsf{E}] = \sum_{k \in \{0,1,\ldots,q,q+1\}} \Pr[\text{pos} = k \wedge \text{soonest} = k] \tag{5.4}$$

$$= \sum_{k \in \{0,1,\ldots,q,q+1\}} \Pr[\text{pos} = k] \cdot \Pr[\text{soonest} = k] \tag{5.5}$$

$$= \frac{1}{q+2} \sum_{k \in \{0,1,\ldots,q,q+1\}} \Pr[\text{soonest} = k] = \frac{1}{q+2} \tag{5.6}$$

This then implies that $\Pr[\neg\mathsf{E}] = \frac{q+2-1}{q+2}$.

Since $\Pr[b = b' \mid \mathsf{E}] = \Pr\left[\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A}}\right]$ and $\Pr[b = b' \mid \neg\mathsf{E}] = \frac{1}{2}$ we have:

$$\mathsf{Adv}^{\text{fs-aead-select}}_{\mathsf{KESE}}(\mathcal{B}) = 2 \cdot \Pr[b = b'] - 1 \tag{5.7}$$

$$= 2 \cdot (\Pr[b = b' \mid \mathsf{E}] \cdot \Pr[\mathsf{E}] + \Pr[b = b' \mid \neg\mathsf{E}] \cdot \Pr[\neg\mathsf{E}]) - 1 \tag{5.8}$$

$$= 2 \cdot (\Pr[b = b' \mid \mathsf{E}] \cdot \frac{1}{q+2} + \Pr[b = b' \mid \neg\mathsf{E}] \cdot \frac{q+2-1}{q+2}) - 1 \tag{5.9}$$

$$= 2 \cdot (\Pr\left[\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A}}\right] \cdot \frac{1}{q+2} + \frac{1}{2} \cdot \frac{q+2-1}{q+2}) - 1 \tag{5.10}$$

$$= 2 \cdot \Pr\left[\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A}}\right] \cdot \frac{1}{q+2} - \frac{1}{q+2} \tag{5.11}$$

$$= \frac{1}{q+2}(2 \cdot \Pr\left[\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A}}\right] - 1) \tag{5.12}$$

$$= \frac{1}{q+2}\mathsf{Adv}^{\text{fs-aead}}_{\mathsf{KESE}}(\mathcal{A}) \tag{5.13}$$

□

Instantiating a KESE scheme

We start this section by giving an instantiation of a key-evolving (stateful) symmetric encryption scheme with associated data based on the Handshake and Application message framing of MLS (see Section 4.10). We then simplify this MLS extracted instantiation and obtain a key-evolving symmetric encryption scheme with associated data scheme that we prove to be FS-AEAD secure (see Figure 5.10).

KESE scheme based on MLS Let KDF be a key derivation function and NAE be a nonce based authenticated encryption scheme with associated data. Then KESE=MLS-ORG-KESE[KDF,NAE] is a key-evolving symmetric encryption scheme with associated data with $\mathsf{KESE}.\mathcal{K} = \{0,1\}^{\mathsf{KDF.Nh}}$, $\mathsf{KESE.ID} = \{a, b, \ldots, z\}^*$ and $\mathsf{KESE}.\mathcal{R} = \{0,1\}^{\mathsf{NAE.Nn}}$ whose algorithm definitions are given in Figure 5.12.

```
KESE.Init(k, id):
  v ← k
  i_S ← 0
  D[.] ← ⊥
  st_S ← (v, i_S, id)
  st_R ← (v, i_S, id, D)
  return (st_S, st_R)

KESE.Send(st_S,m,a,r):
  (v, i_S, id) ← st_S
   nonce ← KDF.Expand(v, ⟨NAE.Nn, "mls10nonce", id, i_S⟩ , NAE.Nn)
   key ← KDF.Expand(v, ⟨NAE.Nk, "mls10key", id, i_S⟩ , NAE.Nk)
   v ← KDF.Expand(v, ⟨KDF.Nh, "mls10secret", id, i_S⟩ , KDF.Nh)
  nonce ← nonce ⊕ r
  c ← NAE.Enc(key, nonce, a, m)
  st_S ← (v, i_S + 1, index)
  return (st_S, c)

KESE.Receive(st_R,a,i,c,r):
  (v, i_R, id, D) ← st_R
  while D[i] = ⊥ and i_R < i:
      nonce ← Expand(v, ⟨NAE.Nn, "mls10nonce", index, i_R⟩ , NAE.Nn)
      key ← Expand(v, ⟨NAE.Nk, "mls10key", index, i_R⟩ , NAE.Nk)
      v ← Expand(v, ⟨KDF.Nh, "mls10secret", index, i_R⟩ , KDF.Nh)
    D[i_R] ← (key, nonce)
    i_R + +
  if D[i] = ⊥:
    st_R ← (v, i_R, id, D)
    return (st_R, ⊥)
  (key, nonce) ← D[i]
  nonce ← nonce ⊕ r
  m ← NAE.Dec(key, nonce, a, c)
  if m ≠ ⊥:
    D[i] ← ⊥
  st_R ← (v, i_R, id, D)
  return (st_R, m)
```

Figure 5.12: MLS based key-evolving (stateful) symmetric encryption scheme with associated data KESE = MLS-ORG-KESE[KDF,NAE].

MLS based KESE simplified In this section we present a simplification of the KESE scheme described in section 5.3.4. Namely, the simplification, given in Figure 5.13, substitutes the Expand algorithm of a KDF scheme, with a function family F, to derive the key,nonce and v bit-strings. The code that differs between the original, in Figure 5.12, and simplified instantiation, in Figure 5.13, is highlighted in yellow.

```
KESE.Init(k, id):
  v ← k
  i_S ← 0
  D[.] ← ⊥
  st_S ← (v, i_S, id)
  st_R ← (v, i_S, id, D)
  return (st_S, st_R)

KESE.Send(st_S,m,a,r):
  (v, i_S, id) ← st_S
  knv ← F.Ev(v, ⟨id, i_S⟩)
  key ← knv[0, ..., NAE.Nk − 1]
  nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
  v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
  nonce ← nonce ⊕ r
  c ← NAE.Enc(key, nonce, a, m)
  st_S ← (v, i_S + 1, id)
  return (st_S, c)

KESE.Receive(st_R,a,i,c,r):
  (v, i_R, id, D) ← st_R
  while D[i] = ⊥ and i_R ≤ i:
    knv ← F.Ev(v, ⟨id, i_R⟩)
    key ← knv[0, ..., NAE.Nk − 1]
    nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
    v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
    D[i_R] ← (key, nonce)
    i_R + +
  if D[i] = ⊥:
    st_R ← (v, i_R, id, D)
    return (st_R, ⊥)
  (key, nonce) ← D[i]
  nonce ← nonce ⊕ r
  m ← NAE.Dec(key, nonce, a, c)
  if m ≠ ⊥:
    D[i] ← ⊥
  st_R ← (v, i_R, id, D)
  return (st_R, m)
```

Figure 5.13: Simplified MLS based key-evolving (stateful) symmetric encryption scheme with associated data KESE = MLS-KESE[F,NAE].

Let NAE be a nonce-based authenticated encryption scheme with associated data. Let F be a function family F with F.Out = NAE.Nk+ NAE.Nn + F.KI. Then KESE = MLS-KESE[F, NAE] is a key-evolving (stateful) symmetric encryption scheme with associated data, defined in Figure 5.9, with KESE.$\mathcal{K}$ $= \{0,1\}^{\mathsf{F.KI}}$, KESE.ID $= \{a, b, \ldots, z\}^*$ and KESE.$\mathcal{R} = \{0,1\}^{\mathsf{NAE.Nn}}$.

Theorem 5.10 *Let* NAE *be a* (ϵ_{mae}, t_1)*-MAE secure nonce-based authenticated encryption scheme with associated data and let* F *be a* (ϵ_{prf}, t_2)*-PRF-secure function family with* F.Out = NAE.Nk+ NAE.Nn + F.KI. *Let* KESE = MLS-KESE[F, NAE]. *Then for any* $q \in \mathbb{N}_0$ KESE *is* (ϵ, t, q)*-FS-AEAD-selective-secure where* $\epsilon = 2(q+1) \cdot \epsilon_{prf} + \epsilon_{mae}$, $t \approx \max(t_1, t_2)$.

Proof. The proof is a hybrid argument over PRF security of F followed by a single step of MAE security of NAE. Let NAE be a (ϵ_{mae}, t_1)-MAE secure nonce-based authenticated scheme with associated data and let F be a (ϵ_{prf}, t_2)-PRF-secure function family with F.Out = NAE.Nk+ NAE.Nn + F.KI. Let KESE = MLS-KESE[F, NAE] and $\mathsf{t}, \mathsf{q} \in \mathbb{N}_0$. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary running in time t making at most q queries to the LoR oracle and query the Dec oracle with i values less than q.

Consider games G^0-G^{q+1} associated to F, NAE and $\mathcal{A}$ and q defined in Figure 5.14. Observe that game $\mathsf{G}^0_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}$ is equivalent to $\mathsf{G}^{\text{fs-aead-select}}_{\mathsf{KESE},\mathcal{A},\mathsf{q}}$ since counters i_S and i_R are monotonically increasing throughout the game and start with values $\mathsf{i_S} = \mathsf{i_R} = 0$ and pos $\in \{0, 1, \ldots, \mathsf{q}, \mathsf{q}+1\}$. Therefore we have:

$$\Pr\left[\,\mathsf{G}^0_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\,\right] = \Pr\left[\,\mathsf{G}^{\text{fs-aead}}_{\mathsf{KESE},\mathcal{A},\mathsf{q}}\,\right]. \tag{5.14}$$

```
G^j_{F,NAE,A,q}:
  b ←$ {0,1}
  f[.] ← ⊥
  trans, chall← ∅
  comp_send ← ∞
  comp_rcv ← false
  not_used[.] ← true
  k ←$ {0,1}^F.Kl
  (A1, A2) ← A
  (state,id,pos) ←$ A1()
  send_ctr ← 0
  soonest_rcv ← q + 1
  max_rcv ← 0
  soonest_send ← q + 1
  v ← k
  i_S, i_R ← 0
  D[.] ← ⊥
  st_S ← (v, i_S, id)
  st_R ← (v, i_R, id, D)
  b' ←$ A2^{LoR,Dec,corr-S,corr-R}(state)
  soonest ← min(soonest_rcv, soonest_send)
  if soonest ≠ pos:
    b' ←$ {0,1}
  return b' = b

LoR(a,m0,m1,r):
  req |m0| = |m1| and ¬comp_rcv and comp_send = ∞
  (v, i_S, id) ← st_S
  if f[i_S] = ⊥:
    if i_S ≥ j:
      knv ← F.Ev(v, ⟨id, i_S⟩)
    else:
      knv ←$ {0,1}^F.Out
    if i_S ≥ pos:
      knv ← F.Ev(v, ⟨id, i_S⟩)
    key ← knv[0, ..., NAE.Nk − 1]
    nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
    v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
    f[i_S] ← (key, nonce, v)
  else:
    (key, nonce, v) ← f[i_S]
  nonce ← nonce ⊕ r
  c ← NAE.Enc(key, nonce, a, m_b)
  st_S ← (v, i_S + 1, id)
  trans ←+ (send_ctr,a,m_b,c,r)
  if m0 ≠ m1 :
    chall ←+ (send_ctr,a,m_b,c,r)
  send_ctr + +
  return c
```

```
Dec(a,i,c,r):
  (v, i_R, id, D) ← st_R
  while D[i] = ⊥ and i_R ≤ i:
    if f[i_R] = ⊥:
      if i_R ≥ j:
        knv ← F.Ev(v, ⟨id, i_R⟩)
      else:
        knv ←$ {0,1}^F.Out
      if i_R ≥ pos:
        knv ← F.Ev(v, ⟨id, i_R⟩)
      key ← knv[0, ..., NAE.Nk − 1]
      nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
      v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
      f[i_R] ← (key, nonce, v)
    else:
      (key, nonce, v) ← f[i_R]
    D[i_R] ← (key, nonce); i_R + +
  if D[i] = ⊥:
    st_R ← (v, i_R, id, D)
    m ← ⊥
  else:
    (key, nonce) ← D[i]
    nonce ← nonce ⊕ r
    m ← NAE.Dec(key, nonce, a, c)
    if m ≠ ⊥:
      D[i] ← ⊥
    st_R ← (v, i_R, id, D)
  if max_rcv < i:
    max_rcv ← i
  if m = ⊥:
    return ⊥
  if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
    trans ←− (i,a,m,c,r)
    chall ←− (i,a,m,c,r)
    not_used[i] ← false
    return ⊥
  if b=1:
    return m
  else:
    return ⊥

corr-SSim:
  soonest_send ← send_ctr
  comp_send ← send_ctr
  return st_S

corr-RSim:
  req chall = ∅
  if ¬comp_rcv:
    soonest_rcv ← max_rcv
  comp_rcv ← true
  return st_R
```

Figure 5.14: Game G^j (for some $j \in \mathbb{N}_0$) associated to a function family F, nonce-based authenticated encryption scheme with associated data NAE, an integer $q \in \mathbb{N}_0$ and an adversary $\mathcal{A}$. The code highlighted in gray is the code added by expanding the algorithms of KESE=MLS-KESE[F,NAE] in the $G^{\text{fs-aead-select}}_{\text{KESE},\mathcal{A},q}$.

Consider adversary $\mathcal{B}$ playing in game G^{prf}_{F} and using $\mathcal{A}$ to win its game as defined in Figure 5.15. Let h be the value sampled by $\mathcal{B}$ at the start of its execution. Then for any choice of $h \in \{0, 1, \ldots, q\}$ adversary $\mathcal{B}$ always assigns a random bit-string to knv when LoRSim (DecSim) is queried on a message (ciphertext) corresponding to counter i_S (i_R) such that $i_S < h$ and $i_S < \text{pos}$ ($i_R < h$ and $i_R < \text{pos}$) i.e. when $i_S < \min(h, \text{pos})$ ($i_R < \min(h, \text{pos})$). Observe that the bit-string knv is a concatenation of the NAE key, NAE nonce and F key bit-string respectively $\text{knv} = \text{key}\|\text{nonce}\|\text{v}$. Therefore if $i_S < \min(h, \text{pos})$ ($i_R < \min(h, \text{pos})$) then key, nonce and v bit-strings are assigned a random bit-string as well. If LoRSim (DecSim) is queried on a message (ciphertext)

corresponding to i_S (i_R) such that $i_S > \mathsf{h}$ or $i_S \geq \mathsf{pos}$ ($i_R > \mathsf{h}$ or $i_R \geq \mathsf{pos}$) then $\mathcal{B}$ instead assigns $\mathsf{F.Ev}(\mathsf{v}, \langle \mathsf{id}, i_S \rangle)$ ($\mathsf{F.Ev}(\mathsf{v}, \langle \mathsf{id}, i_R \rangle)$). The consistency between the answers of the two oracles is maintained with the map f. Now if LoRSim or DecSim is queried on a counter i_S (i_R) such that $i_S = \mathsf{h}$ and $i_S < \mathsf{pos}$ ($i_R = \mathsf{h}$ and $i_R < \mathsf{pos}$) then $\mathcal{B}$ queries its Comp oracle and assigns its reply to knv. If the Comp oracle returns outputs of a random function then $\mathcal{B}$ perfectly simulates the view of $\mathcal{A}$ in game $\mathsf{G}^{\mathsf{h}+1}_{\mathsf{F,NAE,q}}$, meaning:

$$\Pr\left[\, \mathsf{G}^{\mathsf{k}+1}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}} \,\right] = \Pr\left[\, d' = 1 \mid d = 0, \mathsf{h} = \mathsf{k} \,\right] \tag{5.15}$$

holds for all $\mathsf{k} \in \{0, 1, \ldots, \mathsf{q}\}$ where d denotes the challenge bit in the game $\mathsf{G}^{\mathsf{prf}}_{\mathsf{F},\mathcal{B}}$ and d' denotes $\mathcal{B}$'s guess.

Let E and E' be the events that denote $\mathsf{soonest} = \mathsf{pos}$ occurred in the original game G^{h} and in $\mathcal{B}$'s simulation respectively. Observe that up to event E', $\mathcal{B}$ is able to perfectly simulate the G^{h} game to $\mathcal{A}$. Thus, if $\mathcal{A}$ is started (with the same random coins) in G^{h}, it will behave the same (at least) until E happens. So, all events on which the occurrence of $\mathsf{pos} = \mathsf{soonest}$ might depend on are the same in both the original and simulated game. Consequently, E occurs if and only if E' occurs and hence we have $\Pr[\,\mathsf{E}\,] = \Pr[\,\mathsf{E}'\,]$. If Comp returns answers of function F.Ev we consider two cases: (1) $\mathcal{A}$ not misbehaving (E' occurred) and (2) misbehaving ($\neg\mathsf{E}'$ occurred).

First assume $\mathcal{A}$ does not misbehave. If Comp returns answers of function F.Ev and $i_S = \mathsf{h}$ and $i_S < \mathsf{pos}$ ($i_R = \mathsf{h}$ and $i_R < \mathsf{pos}$) then the bit-string v corresponding to $i_S + 1$ (similarly $i_R + 1$) is computed as follows:

$$\mathsf{key} \| \mathsf{nonce} \| \mathsf{v} = \mathsf{F.Ev}(\mathsf{k_f}, \langle \mathsf{id}, i_R \rangle) \tag{5.16}$$

where $\mathsf{k_f} \in \{0,1\}^{\mathsf{F.Kl}}$ is the key sampled uniformly at random in the setup of game $\mathsf{G}^{\mathsf{prf}}_{\mathsf{F},\mathcal{B}}$. Therefore v corresponding to $i_S + 1$ ($i_R + 1$) is independent of v corresponding to i_S (i_R) in $\mathcal{B}$'s simulation. But in game $\mathsf{G}^{\mathsf{h}}_{\mathsf{F,NAE,q}}$ v corresponding to i_S (i_R) is used in place of $\mathsf{k_f}$ in 5.16 to derive $\langle \mathsf{key}, \mathsf{nonce}, \mathsf{v} \rangle$ corresponding to the next counter $i_S + 1$ ($i_R + 1$). However since we assume that the earliest state $\mathcal{A}$ leaks corresponds to $i_S, i_R = \mathsf{pos}$ and it does not misbehave, then $\mathcal{A}$ can only obtain values of v for $i_S, i_R \geq \mathsf{pos}$. But the Comp oracle is only ever called if $i_S = \mathsf{h}$ and $i_S < \mathsf{pos}$ ($i_R = \mathsf{h}$ and $i_R < \mathsf{pos}$) holds. Since $i_S < \mathsf{pos}$, meaning v corresponding to i_S is not leaked by a well behaved $\mathcal{A}$, and v corresponding to i_S does not affect any oracles' outputs then $\mathcal{A}$ has no way of detecting this independence between bit-strings v corresponding to i_S (i_R) and $i_S + 1$ ($i_R + 1$). Therefore if Comp returns answers of function F.Ev and pos=soonest, then $\mathcal{B}$ perfectly simulates the view of $\mathcal{A}$

in game $G^h_{F,NAE,q}$ meaning:

$$\Pr\left[\mathsf{d}=1 \mid d=1, \mathsf{h}=\mathsf{k}, \mathsf{E}'\right] = \Pr\left[G^k_{F,NAE,\mathcal{A},q} \mid \mathsf{E}\right] \tag{5.17}$$

holds for all $\mathsf{k} \in \{0,1,\ldots,\mathsf{q}\}$ where d denotes the challenge bit in the game $G^{prf}_{F,\mathcal{B}}$ and d' denotes $\mathcal{B}$'s guess.

```
Adversary B^Comp:
  b <-$ {0,1}
  h <-$ {0,1,...,q}
  f[.] <- ⊥
  trans, chall <- ∅
  comp_send <- ∞
  comp_rcv <- false
  not_used[.] <- true
  k <-$ {0,1}^F.Kl
  (A1,A2) <- A
  (state,id,pos) <-$ A1()
  send_ctr <- 0
  soonest_rcv <- q+1
  max_rcv <- 0
  soonest_send <- q+1
  v <- k
  i_S, i_R <- 0
  D[.] <- ⊥
  st_S <- (v, i_S, id)
  st_R <- (v, i_R, id, D)
  b' <-$ A2^{LoRSim, DecSim, corr-SSim, corr-RSim}(state)
  soonest <- min(soonest_rcv, soonest_send)
  if soonest ≠ pos:
    b' <-$ {0,1}
  if b' = b :
    d' <- 1
  else:
    d' <- 0
  return d'

LoRSim(a,m0,m1,r):
  req |m0| = |m1| and ¬comp_rcv and comp_send = ∞
  (v, i_S, id) <- st_S
  if f[i_S] = ⊥:
    if i_S > h:
      knv <- F.Ev(v, ⟨id, i_S⟩)
    if i_S = h:
      knv <- Comp(⟨id, i_S⟩)
    if i_S < h:
      knv <-$ {0,1}^F.Out
    if i_S ≥ pos:
      knv <- F.Ev(v, ⟨id, i_S⟩)
    key <- knv[0,...,NAE.Nk − 1]
    nonce <- knv[NAE.Nk,...,NAE.Nk + NAE.Nn − 1]
    v <- knv[NAE.Nk + NAE.Nn,...,|knv| − 1]
    f[i_S] <- (key, nonce, v)
  else:
    (key, nonce, v) <- f[i_S]
  nonce <- nonce ⊕ r
  c <- NAE.Enc(key, nonce, a, m_b)
  st_S <- (v, i_S + 1, id)
  trans <-+ (send_ctr,a,m_b,c,r)
  if m0 ≠ m1 :
    chall <-+ (send_ctr,a,m_b,c,r)
  send_ctr++
  return c

DecSim(a,i,c,r):
  (v, i_R, id, D) <- st_R
  while D[i] = ⊥ and i_R ≤ i:
    if f[i_R] = ⊥:
      if i_R > h:
        knv <- F.Ev(v, ⟨id, i_R⟩)
      if i_R = h:
        knv <- Comp(⟨id, i_R⟩)
      if i_R < h:
        knv <-$ {0,1}^F.Out
      if i_R ≥ pos:
        knv <- F.Ev(v, ⟨id, i_R⟩)
      key <- knv[0,...,NAE.Nk − 1]
      nonce <- knv[NAE.Nk,...,NAE.Nk + NAE.Nn − 1]
      v <- knv[NAE.Nk + NAE.Nn,...,|knv| − 1]
      f[i_R] <- (key, nonce, v)
    else:
      (key, nonce, v) <- f[i_R]
    D[i_R] <- (key, nonce); i_R++
  if D[i] = ⊥:
    st_R <- (v, i_R, id, D)
    m <- ⊥
  else:
    (key, nonce) <- D[i]
    nonce <- nonce ⊕ r
    m <- NAE.Dec(key, nonce, a, c)
    if m ≠ ⊥:
      D[i] <- ⊥
    st_R <- (v, i_R, id, D)
  if max_rcv < i:
    max_rcv <- i
  if m = ⊥:
    return ⊥
  if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
    trans <-+ (i,a,m,c,r)
    chall <-+ (i,a,m,c,r)
    not_used[i] <- false
    return ⊥
  if b=1:
    return m
  else:
    return ⊥

corr-SSim:
  soonest_send <- send_ctr
  comp_send <- send_ctr
  return st_S

corr-RSim:
  req chall = ∅
  if ¬comp_rcv:
    soonest_rcv <- max_rcv
  comp_rcv <- true
  return st_R
```

Figure 5.15: Adversary $\mathcal{B}$ playing in game G^{prf}_F for proof of Theorem 5.10. The code that differs from game G^j for $j \in \mathbb{N}_0$ is highlighted in yellow.

If $\mathcal{A}$ misbehaves we do not care to simulate the oracles perfectly since the game $G^h_{F,NAE,q}$ discards $\mathcal{A}$'s answer and samples its own b' instead. There-

fore if Comp returns answers of F.Ev and pos $\neq$ soonest, then $\mathcal{B}$ allows $\mathcal{A}$ to win with the same probability as in game $G^{h}_{F,NAE,q}$ given that misbehaviour occurred, meaning:

$$\Pr\left[\, d = 1 \,|\, d = 1, h = k, \neg E' \,\right] = \Pr\left[\, G^{k}_{F,NAE,\mathcal{A},q} \,|\, \neg E \,\right] \tag{5.18}$$

holds for all $k \in \{0, 1, \ldots, q\}$. Since $\Pr[\,E\,] = \Pr[\,E'\,]$ then combining Equations 5.17 and 5.18 we have:

$$\Pr\left[\, G^{k}_{F,NAE,\mathcal{A},q} \,\right] = \Pr\left[\, d' = 1 \,|\, d = 1, h = k \,\right] \tag{5.19}$$

holding for all $k \in \{0, 1, \ldots, q\}$.

We can now use Equation 5.19 and 5.15 to rewrite the advantage of $\mathcal{B}$ in G^{prf}_{F} as follows:

$$\begin{aligned}
\mathsf{Adv}^{prf}_{F}(\mathcal{B}) &= \Pr\left[\, d' = 1 \,|\, d = 1 \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0 \,\right] && (5.20)\\
&= \sum_{k=0}^{q} \Pr[\,h = k\,] \cdot (\Pr\left[\, d' = 1 \,|\, d = 1, h = k \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0, h = k \,\right]) && (5.21)\\
&= \frac{1}{q+1} \sum_{k=0}^{q} (\Pr\left[\, d' = 1 \,|\, d = 1, h = k \,\right] - \Pr\left[\, d' = 1 \,|\, d = 0, h = k \,\right]) && (5.22)\\
&= \frac{1}{q+1} \cdot (\sum_{k=0}^{q} (\Pr\left[\, G^{k}_{F,NAE,\mathcal{A},q} \,\right] - \Pr\left[\, G^{k+1}_{F,NAE,\mathcal{A},q} \,\right])) && (5.23)\\
&= \frac{1}{q+1} (\Pr\left[\, G^{0}_{F,NAE,\mathcal{A},q} \,\right] - \Pr\left[\, G^{q+1}_{F,NAE,\mathcal{A},q} \,\right]) && (5.24)
\end{aligned}$$

Now consider game G^{q+2} associated to F, NAE, $q \in \mathbb{N}_0$ and attacker $\mathcal{A}$ defined in Figure 5.16. We now prove that the game $G^{q+2}_{F,NAE,\mathcal{A},q}$ is equivalent to game $G^{q+1}_{F,NAE,\mathcal{A},q}$. Since the only code that differs between the two games is the code highlighted in gray it suffices to prove that the Dec oracle behaves the same in both games. By Dec oracle behaving in the same way in both games we mean that all variables and outputs obtain the same value on the same input when using the same random coins.

Namely let (a,i,c,r) be some input the Dec oracle is queried on for the first time. Since this is the first query to the Dec oracle all variables in both games have the same values at the start of Dec's execution. The proof is split into two cases. In the first case assume that $\exists m' : (i, a, m', c, r) \in \mathsf{trans}$ is false. Then the Dec oracle never executes the gray highlighted code and hence it produces the same output as the Dec oracle in game G^{q+1}. Now assume that $\exists m' : (i, a, m', c, r) \in \mathsf{trans}$ is true. This statement is true if and only if $c = \mathsf{NAE.Enc}(\mathsf{key}, \mathsf{nonce}, a, m')$ where $(\mathsf{key}, \mathsf{nonce} \oplus r, v) = f[i]$ for some bit-strings $v, \mathsf{key}, \mathsf{nonce}$. We consider two cases: (1) $\mathcal{D}[i] = \perp$ and (2) $\mathcal{D}[i] = (\mathsf{key}, \mathsf{nonce} \oplus r)$.

```
G^(q+2)_{F,NAE,A,q}:
  b ←$ {0,1}
  f[.] ← ⊥
  trans, chall← ∅
  comp_send ← ∞
  comp_rcv ← false
  not_used[.] ← true
  k ←$ {0,1}^F.Kl
  (A1, A2) ← A
  (state,id,pos) ←$ A1()
  send_ctr ← 0
  soonest_rcv ← q + 1
  max_rcv ← 0
  soonest_send ← q + 1
  v ← k
  i_S, i_R ← 0
  D[.] ← ⊥
  st_S ← (v, i_S, id)
  st_R ← (v, i_R, id, D)
  b' ←$ A2^{LoR,Dec,corr-S,corr-R}(state)
  soonest ← min(soonest_rcv, soonest_send)
  if soonest ≠ pos:
    b' ←$ {0,1}
  return b' = b

LoR(a,m0,m1,r):
  req |m0| = |m1| and ¬comp_rcv and comp_send = ∞
  (v, i_S, id) ← st_S
  if f[i_S] = ⊥:
    if i_S ≥ q + 1:
      knv ← F.Ev(v, ⟨id, i_S⟩)
    else:
      knv ←$ {0,1}^F.Out
    if i_S ≥ pos:
      knv ← F.Ev(v, ⟨id, i_S⟩)
    key ← knv[0, ..., NAE.Nk − 1]
    nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
    v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
    f[i_S] ← (key, nonce, v)
  else:
    (key, nonce, v) ← f[i_S]
  nonce ← nonce ⊕ r
  c ← NAE.Enc(key, nonce, a, m_b)
  st_S ← (v, i_S + 1, id)
  trans ←+ (send_ctr,a,m_b,c,r)
  if m0 ≠ m1 :
    chall ←+ (send_ctr,a,m_b,c,r)
  send_ctr + +
  return c

corr-R:
  req chall = ∅
  if ¬comp_rcv:
    soonest_rcv ← max_rcv
  comp_rcv ← true
  return st_R
```

```
Dec(a,i,c,r):
  (v, i_R, id, D) ← st_R
  while D[i] = ⊥ and i_R ≤ i:
    if f[i_R] = ⊥:
      if i_R ≥ q + 1:
        knv ← F.Ev(v, ⟨id, i_R⟩)
      else:
        knv ←$ {0,1}^F.Out
      if i_R ≥ pos:
        knv ← F.Ev(v, ⟨id, i_R⟩)
      key ← knv[0, ..., NAE.Nk − 1]
      nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
      v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
      f[i_R] ← (key, nonce, v)
    else:
      (key, nonce, v) ← f[i_R]
    D[i_R] ← (key, nonce); i_R + +
  if D[i] = ⊥:
    st_R ← (v, i_R, id, D)
    m ← ⊥
  else:
    (key, nonce) ← D[i]
    nonce ← nonce ⊕ r
    if ∃m' : (i, a, m', c, r) ∈ trans:                         [gray]
      if m' ≠ ⊥ :                                              [gray]
        D[i] ← ⊥                                               [gray]
      st_R ← (v, i_R, id, D)                                   [gray]
      if max_rcv < i:                                          [gray]
        max_rcv ← i                                            [gray]
      if m' = ⊥:                                               [gray]
        return ⊥                                               [gray]
      if not_used[i] and ((i, a, m', c, r) ∈ trans or i ≥ comp_send or comp_rcv):   [gray]
        trans ⇐ (i, a, m', c, r)                               [gray]
        chall ⇐ (i, a, m', c, r)                               [gray]
        not_used[i] ← false                                    [gray]
      return ⊥                                                 [gray]
    m ← NAE.Dec(key, nonce, a, c)
    if m ≠ ⊥:
      D[i] ← ⊥
    st_R ← (v, i_R, id, D)
  if max_rcv < i:
    max_rcv ← i
  if m = ⊥:
    return ⊥
  if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
    trans ⇐ (i,a,m,c,r)
    chall ⇐ (i,a,m,c,r)
    not_used[i] ← false
    return ⊥
  if b=1:
    return m
  else:
    return ⊥

corr-S:
  soonest_send ← send_ctr
  comp_send ← send_ctr
  return st_S
```

Figure 5.16: Game G^{q+2} associated to a function family F, nonce-based authenticated encryption scheme with associated data NAE, integer $q \in \mathbb{N}_0$ and an adversary $\mathcal{A}$. The code added on top of the game G^{q+1} is highlighted in gray.

First assume $\mathcal{D}[\mathsf{i}] = \bot$ then the gray code is never executed since the condition **if** $\mathcal{D}[i] = \bot$ is true. Hence game G^{q+1} and G^{q+2} return the same value trivially. Now assume $\mathcal{D}[\mathsf{i}] = (\mathsf{key}, \mathsf{nonce} \oplus \mathsf{r})$. Then in game G^{q+1} we have:

$$\mathsf{NAE.Dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{a}, \mathsf{c}) = \mathsf{NAE.Dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{a}, \mathsf{NAE.Enc}(\mathsf{key}, \mathsf{nonce}, \mathsf{a}, \mathsf{m}')).$$

By the correctness property of NAE we have that $\mathsf{m}' = \mathsf{NAE.Dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{a}, \mathsf{c})$ in game G^{q+1}.

Since the code up until and including the line $\mathsf{nonce} \leftarrow \mathsf{nonce} \oplus \mathsf{r}$ in the Dec

oracle is the same in both games then variables used by the code after that line in game G^{q+1} are the same as those used by the gray code in G^{q+2}. Now because $\mathsf{m}' = \mathsf{NAE.Dec}(\mathsf{key}, \mathsf{nonce}, \mathsf{a}, \mathsf{c})$ and $\mathsf{m}' \neq \perp$ in game G^{q+1} and both games have the same variable values after line $\mathsf{nonce} \leftarrow \mathsf{nonce} \oplus \mathsf{r}$ then it is easy to see that the Dec oracle in both games behaves in the same way on input (a,i,c,r).

Because the Dec oracle in both games behaves in the same way on the first query, then all variables have the same values after this first query terminates. Then by induction on the number of queries we have the above argument extend to all possible queries to the Dec oracle not only the first. Therefore games G^{q+1} and G^{q+2} are equivalent, meaning:

$$\Pr\left[\mathsf{G}^{q+1}_{\mathsf{F},\mathsf{NAE},\mathcal{A},q}\right] = \Pr\left[\mathsf{G}^{q+2}_{\mathsf{F},\mathsf{NAE},\mathcal{A},q}\right]. \tag{5.25}$$

Now consider adversary $\mathcal{G}$ playing in game $\mathsf{G}^{\mathsf{mae}}_{\mathsf{NAE}}$ as defined in Figure 5.17. We now argue that each of the oracles available to $\mathcal{A}$ in game $\mathsf{G}^{q+2}_{\mathsf{F},\mathsf{NAE},q}$ are perfectly simulated by $\mathcal{G}$ when pos=soonest. We only care that the simulation is perfect in this case because we only use $\mathcal{A}$'s answer b' if pos=soonest.

Since $\mathsf{pos} \in \{0, 1, \ldots, q+1\}$ the LoRSim and DecSim oracles now sample keys for $i_S, i_R < \mathsf{pos}$ independently and uniformly at random by querying oracle New. The nonce and v bit-strings are sampled independently and uniformly at random locally by $\mathcal{G}$. Note that the New oracle does not return the uniform key and hence the key bit-string for $i_S, i_R < \mathsf{pos}$ is set to $\perp$ indicating that $\mathcal{G}$ does not know the key yet. For $i_S, i_R \geq \mathsf{pos}$ the bit-string $\mathsf{knv} = \mathsf{key} \| \mathsf{nonce} \| \mathsf{v}$ is derived using F.Ev locally and hence $\mathcal{G}$ knows all values needed to locally encrypt messages corresponding to $i_S, i_R \geq \mathsf{pos}$.

In the LoRSim oracle these values are then used to encrypt either message m_0 or m_1. If the encryption is being made when $i_S, i_R < \mathsf{pos}$ then $\mathcal{G}$ queries its Enc oracle to obtain a ciphertext. Adversary $\mathcal{G}$ not knowing key bit-strings for $i_S, i_R < \mathsf{pos}$ is unimportant since it entrusts the encryption to its Enc oracle in this case. If the encryption is being made when $i_S, i_R \geq \mathsf{pos}$ then $\mathcal{G}$ uses bit b it sampled at the start of its execution to produce a ciphertext locally using the key and nonce corresponding to $i_S \geq \mathsf{pos}$. Moreover the bit b sampled locally causes no change in the simulation because it is guaranteed that $\mathsf{m}_0 = \mathsf{m}_1$ for all $i_S \geq \mathsf{pos}$ if soonest=pos.

Namely for soonest=pos to be true either the first receiver state st_R or first sender state st_S leaked must correspond to position pos. After the first corruption (either of sender or receiver) calls to the LoR oracle are prohibited and hence after the first state leak no honestly crafted ciphertexts are sent (chall and trans sets never have elements added anymore) and the sender state st_S is never updated. The receiver state st_R can only be leaked once all the ciphertexts in the chall set have been delivered by calling the Dec oracle.

Therefore if soonest=pos holds then it must be the case that $m_0 = m_1$ for all $i_S \geq$ pos. Hence the LoR oracle of $\mathcal{A}$ is perfectly simulated by LoRSim.

```
Adversary G^{New,Enc,Dec,Corr}:
  b ←$ {0,1}
  f[.] ← ⊥
  trans, chall ← ∅
  comp_send ← ∞
  comp_rcv ← false
  not_used[.] ← true
  k ←$ {0,1}^{F.Kl}
  (A_1, A_2) ← A
  (state,id,pos) ←$ A_1()
  send_ctr ← 0
  soonest_rcv ← q + 1
  max_rcv ← 0
  soonest_send ← q + 1
  v ← k
  i_S, i_R ← 0
  D[.] ← ⊥
  st_S ← (v, i_S, id)
  st_R ← (v, i_R, id, D)
  b' ←$ A_2^{LoRSim, DecSim, corr-SSim, corr-RSim}(state)
  soonest ← min(soonest_rcv, soonest_send)
  if soonest ≠ pos:
    b' ←$ {0,1}
  return b'

LoRSim(a,m_0,m_1,r):
  req |m_0| = |m_1| and ¬comp_rcv and comp_send = ∞
  (v, i_S, id) ← st_S
  if f[i_S] = ⊥:
    if i_S ≥ q + 1:
      knv ← F.Ev(v, ⟨id, i_S⟩)
      key ← knv[0, ..., NAE.Nk − 1]
      nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
      v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
    else:
      New()                          [highlighted]
      key ← ⊥                        [highlighted]
      nonce ←$ {0,1}^{NAE.Nn}        [highlighted]
      v ←$ {0,1}^{F.Kl}              [highlighted]
    if i_S ≥ pos:
      knv ← F.Ev(v, ⟨id, i_S⟩)
      key ← knv[0, ..., NAE.Nk − 1]
      nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
      v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
    f[i_S] ← (key, nonce, v)
  else:
    (key, nonce, v) ← f[i_S]
  nonce ← nonce ⊕ r
  if i_S ≥ pos:
    c ← NAE.Enc(key, nonce, a, m_b)
  if i_S < pos:                      [highlighted]
    c ← Enc(i_S, nonce, m_0, m_1, a) [highlighted]
  st_S ← (v, i_S + 1, id)
  trans ←+ (send_ctr, a, m_b, c, r)
  if m_0 ≠ m_1 :
    chall ←+ (send_ctr, a, m_b, c, r)
  send_ctr + +
  return c

corr-RSim:
  req chall = ∅
  if ¬comp_rcv:
    soonest_rcv ← max_rcv
    (v, i_R, id, D) ← st_R           [highlighted]
    for i in range(|D|):             [highlighted]
      if i < pos:                    [highlighted]
        (key, nonce) ← D[i]          [highlighted]
        key ← Corr(i)                [highlighted]
        D[i] ← (key, nonce)          [highlighted]
        f[i] ← (key, nonce, v)       [highlighted]
    st_R ← (v, i_R, id, D)           [highlighted]
  comp_rcv ← true
  return st_R
```

```
DecSim(a,i,c,r):
  (v, i_R, id, D) ← st_R
  while D[i] = ⊥ and i_R ≤ i:
    if f[i_R] = ⊥:
      if i_R ≥ q + 1:
        knv ← F.Ev(v, ⟨id, i_R⟩)
        key ← knv[0, ..., NAE.Nk − 1]
        nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
        v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
      else:
        New()                        [highlighted]
        key ← ⊥                      [highlighted]
        nonce ←$ {0,1}^{NAE.Nn}      [highlighted]
        v ←$ {0,1}^{F.Kl}            [highlighted]
      if i_R ≥ pos:
        knv ← F.Ev(v, ⟨id, i_R⟩)
        key ← knv[0, ..., NAE.Nk − 1]
        nonce ← knv[NAE.Nk, ..., NAE.Nk + NAE.Nn − 1]
        v ← knv[NAE.Nk + NAE.Nn, ..., |knv| − 1]
      f[i_R] ← (key, nonce, v)
    else:
      (key, nonce, v) ← f[i_R]
    D[i_R] ← (key, nonce); i_R + +
  if D[i] = ⊥:
    st_R ← (v, i_R, id, D)
    m ← ⊥
  else:
    (key, nonce) ← D[i]
    nonce ← nonce ⊕ r
    if ∃m' : (i, a, m', c, r) ∈ trans:
      if m' ≠ ⊥ :
        D[i] ← ⊥
      st_R ← (v, i_R, id, D)
      if max_rcv < i:
        max_rcv ← i
      if m' = ⊥:
        return ⊥
      if not_used[i] and ((i, a, m', c, r) ∈ trans or i ≥ comp_send or comp_rcv):
        trans ⇐ (i, a, m', c, r)
        chall ⇐ (i, a, m', c, r)
        not_used[i] ← false
      return ⊥
    if i ≥ pos or key ≠ ⊥:
      m ← NAE.Dec(key, nonce, a, c)
    else:                            [highlighted]
      m ← Dec(i, nonce, c, a)        [highlighted]
    if m ≠ ⊥:
      D[i] ← ⊥
    st_R ← (v, i_R, id, D)
  if max_rcv < i:
    max_rcv ← i
  if m = ⊥:
    return ⊥
  if not_used[i] and ((i,a,m,c,r) ∈ trans or i ≥ comp_send or comp_rcv):
    trans ⇐ (i,a,m,c,r)
    chall ⇐ (i,a,m,c,r)
    not_used[i] ← false
    return ⊥
  if b=1:
    return m
  else:
    return ⊥

corr-SSim:
  soonest_send ← send_ctr
  comp_send ← send_ctr
  return st_S
```

Figure 5.17: Adversary $\mathcal{G}$ playing in game G^{mae}_{NAE} for proof of Theorem 5.10. The code that differs from game G^{q+2} is highlighted in yellow.

The DecSim oracle trivially simulates the Dec oracle in game $\mathsf{G}^{q+2}_{\mathsf{F,NAE,q}}$ perfectly since for all keys that $\mathcal{G}$ does not know ($i_R < \mathsf{pos}$ and $\mathsf{key} \neq \bot$) it calls its Dec oracle and for all the keys it does know $i_R \geq \mathsf{pos}$ it decrypts the ciphertext locally. Observe the extra game G^{q+2} here is crucial, since the Dec oracle of the $\mathsf{G}^{\mathsf{mae}}$ game returns $\bot$ if $(\mathsf{i}, \mathsf{nonce}, \mathsf{c}, \mathsf{a}) \in \mathsf{trans}$. Therefore we needed the added gray code in G^{q+2} to ensure that the state of the receiver is the same in the game G^{q+2} (hence in G^{q+1} game as well) and in the simulation.

The corr-S is trivially perfectly simulated by corr-SSim. The corr-R oracle is perfectly simulated by corr-RSim because all the keys not known (set to $\bot$) in the map $\mathcal{D}$ maintained in the receiver state, are obtained by $\mathcal{G}$ calling its Corr oracle once corr-RSim is called. Note that the keys corresponding to a ciphertext stored in the chall set are the only ones that can not be obtained by calling the Corr oracle. However since corr-RSim can only be called once chall is empty and hence the keys corresponding to these challange ciphertexts are erased, this exactly captures what $\mathcal{A}$ expects to see after leaking $\mathsf{st_R}$. Therefore all oracles are perfectly simulated in case soonest=pos and hence $\mathcal{G}$ wins if and only if $\mathcal{A}$ wins if soonest=pos. If soonest $\neq$ pos then $\mathcal{G}$ samples b' locally and outputs it as its guess. Therefore if soonest $\neq$ pos occurs $\mathcal{G}$ wins with the same probability as the adversary $\mathcal{A}$ in game $\mathsf{G}^{q+2}_{\mathsf{F,NAE,q}}$ given soonest $\neq$ pos. Therefore we have:

$$\Pr\left[\mathsf{G}^{\mathsf{mae}}_{\mathsf{NAE},\mathcal{G}}\right] = \Pr\left[\mathsf{G}^{q+2}_{\mathsf{F,NAE,q},\mathcal{A}}\right]. \tag{5.26}$$

Now using Equations 5.14, 5.24, 5.25 and 5.26 we can rewrite the advantage of $\mathcal{A}$ winning in game $\mathsf{G}^{\mathsf{fs\text{-}aead\text{-}select}}_{\mathsf{KESE,q}}$ as follows:

$$\mathsf{Adv}^{\mathsf{fs\text{-}aead\text{-}select}}_{\mathsf{KESE,q}}(\mathcal{A}) = 2\cdot\Pr\left[\mathsf{G}^{\mathsf{fs\text{-}aead\text{-}select}}_{\mathsf{KESE},\mathcal{A},\mathsf{q}}\right] - 1 = 2\cdot\Pr\left[\mathsf{G}^{0}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right] - 1 = \tag{5.27}$$

$$2\cdot(\Pr\left[\mathsf{G}^{0}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right] - \Pr\left[\mathsf{G}^{q+1}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right] + \Pr\left[\mathsf{G}^{q+1}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right]) - 1 = \tag{5.28}$$

$$2\cdot(\Pr\left[\mathsf{G}^{0}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right] - \Pr\left[\mathsf{G}^{q+1}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right] + \Pr\left[\mathsf{G}^{q+1}_{\mathsf{F,NAE},\mathcal{A},\mathsf{q}}\right]) - 1 = \tag{5.29}$$

$$2\cdot((\mathsf{q}+1)\cdot\mathsf{Adv}^{\mathsf{prf}}_{\mathsf{F}}(\mathcal{B}) + \frac{\mathsf{Adv}^{\mathsf{mae}}_{\mathsf{NAE}}(\mathcal{G})+1}{2}) - 1 = \tag{5.30}$$

$$2(\mathsf{q}+1)\cdot\epsilon_{prf} + \epsilon_{mae}. \tag{5.31}$$

□

5.3.5 KEGSE instantiation

MLS based KEGSE instantiation

In this section, we give an instantiation of a key-evolving group symmetric encryption scheme with associated data based on the MLS protocol's framing. Let KDF be a key derivation function and NAE be a nonce based authenticated encryption scheme with associated data. Then KEGSE=MLS-ORG-KEGSE[KDF,NAE,KESE] is a key-evolving group symmetric encryption scheme such that $\mathsf{KEGSE.MK} = \{0,1\}^{\mathsf{KDF.Nh}}$, $\mathsf{KEGSE.DK} = \mathsf{KESE}.\mathcal{K}$,

KEGSE.ID = KESE.ID and KEGSE.$\mathcal{R}$ = KESE.$\mathcal{R}$. Its algorithm definitions are given in Figure 5.18.

```
KEGSE.Init(k, ids, scrts, i):
  m_S ← k
  ME ← ids[i]
  i_S ← 0
  for j = 0,...,n :
    (s, r) ← KESE.Init(scrts[j], ids[j])
    sts_R[j] ← r
    if i=j:
      sts_S ← s
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return st

KEGSE.Send(st,m,a;r):
  (ME, m_S, sts_S, sts_R, i_S) ← st
  (sts_S, c_1) ← KESE.Send(sts_S, m, a, r)
  ptx ← ⟨ME, i_S, r⟩
  key ← KDF.Expand(m_S, ⟨NAE.Nk, mls10key, c_1⟩, NAE.Nk)
  nonce ← KDF.Expand(m_S, ⟨NAE.Nn, mls10nonce, c_1⟩, NAE.Nn)
  c_2 ← NAE.Enc(key, nonce, a, ptx)
  c ← (c_1, c_2)
  i_S ++
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return (st, c)

KEGSE.Receive(st,a,c):
  (ME, m_S, sts_S, sts_R, i_S) ← st
  (c_1, c_2) ← c
  key ← KDF.Expand(m_S, ⟨NAE.Nk, mls10key, c_1⟩, NAE.Nk)
  nonce ← KDF.Expand(m_S, ⟨NAE.Nn, mls10nonce, c_1⟩, NAE.Nn)
  ⟨sID, i, r⟩ ← NAE.Dec(key, nonce, a, c_2)
  (sts_R[sID], m) ← KESE.Receive(sts_R[sID], a, i, c_1, r)
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return (st, i, m)
```

Figure 5.18: KEGSE instantiation using MLS.

MLS based KEGSE simplified

We now present a simplification of the KEGSE scheme described in Figure 5.18. The code that differs between the original, in Figure 5.18, and the simplified instantiation, in Figure 5.19, is highlighted in yellow.

```
KEGSE.Init(k, ids, scrts, i):
  m_S ← k
  ME ← ids[i]
  i_S ← 0
  for j = 0,...,n :
    (s, r) ← KESE.Init(scrts[j], ids[j])
    sts_R[j] ← r
    if i=j:
      sts_S ← s
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return st

KEGSE.Send(st,m,ad;r):
  (ME, m_S, sts_S, sts_R, i_S) ← st
  (sts_S, c_1) ← KESE.Send(sts_S, m, ad, r)
  ptx ← ⟨ME, i_S, r⟩
  kn ← F.Ev(m_S, ⟨c_1⟩)
  key ← kn[0,...,NAE.Nk − 1]
  nonce ← kn[NAE.Nk,...,NAE.Nk + NAE.Nn − 1]
  c_2 ← NAE.Enc(key, nonce, ad, ptx)
  c ← (c_1, c_2)
  i_S ++
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return (st, c)

KEGSE.Receive(st,ad,c):
  (ME, m_S, sts_S, sts_R, i_S) ← st
  (c_1, c_2) ← c
  kn ← F.Ev(m_S, ⟨c_1⟩)
  key ← kn[0,...,NAE.Nk − 1]
  nonce ← kn[NAE.Nk,...,NAE.Nk + NAE.Nn − 1]
  ⟨sID, i, r⟩ ← NAE.Dec(key, nonce, ad, c_2)
  (sts_R[sID], m) ← KESE.Receive(sts_R[sID], ad, i, c_1, r)
  st ← (ME, m_S, sts_S, sts_R, i_S)
  return (st, i, m)
```

Figure 5.19: KEGSE instantiation using MLS simplified.

Let NAE be a nonce-based authenticated encryption scheme with associated data. Let F be a function family F with F.Out = NAE.Nk+ NAE.Nn. Then KEGSE = MLS-KEGSE[F, NAE, KESE] is the key-evolving (stateful) symmetric encryption scheme with associated data with KEGSE.MK = $\{0,1\}^{\mathsf{F.Kl}}$, KEGSE.DK = KESE.$\mathcal{K}$, KEGSE.ID = KESE.ID and KEGSE.$\mathcal{R}$ = KESE.$\mathcal{R}$ whose algorithm definitions are given in Figure 5.19.

FS-GAEAD security intuition We now give an informal argument behind KEGSE = MLS-KEGSE[F, NAE, KESE] being FS-GAEAD secure if KESE is FS-AEAD broadcast secure. Note that we make no assumptions on the function family F and nonce based authenticated encryption scheme NAE as MLS uses them to achieve anonymity, not the basic properties captured by FS-GAEAD security. Let $n \in \mathbb{N}$ and let a KEGSE scheme be run between n clients identified as $\{id_1, id_2, \ldots, id_n\}$ respectively. The KEGSE security game, in Figure 5.4, starts with each client id_i initialising its state by calling the KEGSE.Init algorithm, where $i \in \{1, \ldots, n\}$. The KESGE.Init in KEGSE = MLS-KEGSE[F, NAE, KESE] (defined in Figure 5.19) calls KESE.Init for each of the n identifiers in order to generate one sender state $\mathsf{st_S}$ and n receiver states $\mathsf{sts_R}$ (allowing for the sender to receive from itself). Since each client calls KEGSE.Init on the same k,ids and scrts input values, the underlying KESE.Init calls (which take the entries of scrts and ids as input) will generate the same output.

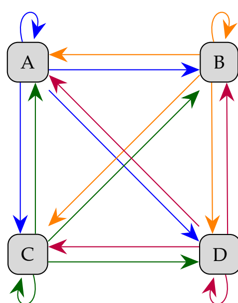


Figure 5.20: KEGSE session run between clients A,B,C,D consisting of 4 independent KESE sessions where KEGSE = MLS-KEGSE[F, NAE, KESE]. We use different colors to annotate each KESE session. Note that the self-loop is present for each sender since it sends messages to the entire group including itself.

Therefore each client id_i's sender state $\mathsf{st_S}$ will have its corresponding responder state $\mathsf{st_R}$ saved by each client id_j where $j \in \{1, 2, \ldots, n\}$. Consequently each time id_i sends a message to the rest of the group, a client receiving id_i's message will use the $\mathsf{st_R}$ corresponding to id_i to decrypt it. This is exactly what occurs in a KESE scheme that is FS-AEAD broadcast secure. Moreover reciever states maintained by each client progress independently of one another. Therefore a single KEGSE session between n clients can be

seen as n independent KESE sessions in a single sender multiple receivers scenario. Consequently if KESE is FS-AEAD broadcast secure, that means that each of these n independent KESE sessions is secure, making the whole KEGSE session secure. Figure 5.20 showcases this argument pictorially for $n = 4$ and identifiers $\{A, B, C, D\}$.

FS-ANONIM-GAEAD insecure We now show that the MLS based KEGSE scheme instantiation KEGSE = MLS-KEGSE[F, NAE, KESE] for some family function F, nonce based authenticated encryption scheme with associated data NAE and a key-evolving symmetric encryption scheme KESE is insecure, in the FS-ANONIM-GAEAD sense, by providing an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, defined in Figure 5.21, who wins the game $G^{\text{fs-anonim-gaead}}_{\text{KEGSE}}$ with probability 1.

Intuitively, the MLS based KEGSE scheme does not provide forward-secure anonymity, because the metadata secret m_S, is never refreshed. Thus, if an adversary (at some point in time) leaks the state of some client (sender or receiver), the metadata secret m_S, contained in the leaked state, would be the same one used to form all c_2 ciphertexts (see Figure 5.19). Hence, assuming the adversary recorded previous ciphertexts (which are themselves, pairs of ciphertexts (c_1, c_2)), it can use the leaked m_S along with c_1, to derive the key-nonce pair used to form c_2. Thus, the adversary would be able to decrypt all c_2 ciphertexts, and hence leak each identity.

```
Adversary A_1():
  id_0 <-$ KEGSE.ID
  id_1 <-$ KEGSE.ID/{id_0}
  ids <- [id_0, id_1]
  return (ids,ids)

Adversary A_2^{LoR,Dec,corr}(state):
  r <-$ {0,1}^{KESE.R}
  a <- 0110101
  m <- 1001010
  id_0 <- state[0], id_1 <- state[1]
  c <- LoR(a, r, (id_0, m), (id_1, m))
  c' <- LoR(a, r, (id_1, m), (id_0, m))
  m' <- Dec(id_0, a, c))
  m' <- Dec(id_0, a, c'))
  st <- corr(id_0)
  (ME, m_S, st_S, st_R, i_S) <- st
  (c_1, c_2) <- c
  kn <- F.Ev(m_S, ⟨c_1⟩)
  key <- kn[0, ..., NAE.Nk - 1]
  nonce <- kn[NAE.Nk, ..., NAE.Nk + NAE.Nn - 1]
  ⟨sid, i, r⟩ <- NAE.Dec(key, nonce, ad, c_2)
  if sid = id_0:
    return 0
  else:
    return 1
```

Figure 5.21: Adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ playing in game $G^{\text{fs-anonim-gaead}}_{\text{KEGSE}}$ (defined in Figure 5.5) such that KEGSE = MLS-KEGSE[F, NAE, KESE].

The adversary $\mathcal{A}$, given in Figure 5.21, follows exactly the intuition laid out

above. Namely, it uses $\mathcal{A}_1$ to form a group of two clients id_0 and id_1 (of course the group can be larger than this). It then uses $\mathcal{A}_2$ to win the game $\mathsf{G}^{\text{fs-anonim-gaead}}_{\text{KEGSE}}$ as follows. First, $\mathcal{A}_2$ selects some random seed r, associated data a, and message m, which it uses to query the LoR oracle two times on $(\mathsf{a}, \mathsf{r}, (\mathsf{id}_0, \mathsf{m}), (\mathsf{id}_1, \mathsf{m}))$ and $(\mathsf{a}, \mathsf{r}, (\mathsf{id}_1, \mathsf{m}), (\mathsf{id}_0, \mathsf{m}))$. It then uses the ciphertexts obtained from these LoR oracle queries, c and c', along with id_0, to query the Dec oracle. This is necessary, in order for $\mathcal{A}_2$ to pass the requirement imposed by the corr oracle. Finally, $\mathcal{A}_2$ now leaks the state of id_0 and uses the obtained m_5 value to, as described above, decrypt c_2 of c and break the FS-ANONIM-GAEAD security of the MLS based KEGSE. The advantage of this attack is, trivially, 1.

PPRF based KEGSE

Finally we propose a KEGSE scheme based on puncturable function families G and nonce-based authenticated encryption schemes NAE we believe is secure in the FS-ANONIM-GAEAD sense. The idea is that a client who wishes to send a message to the group first samples an input value $r \leftarrow_{\$} \{0,1\}^{\mathsf{G.In}}$. If $\mathsf{G.Ev} : \{0,1\}^{\mathsf{G.Kl}} \times \{0,1\}^{\mathsf{G.In}} \to \{0,1\}^{\mathsf{NAE.Nk}+\mathsf{NAE.Nn}}$ then the client can use this sampled r and its current G key k to obtain obtain a NAE key nonce pair. It can then use this NAE key nonce pair to encrypt its message along with its identifier to form a ciphertext c.

In order to ensure forward security of messages and anonimity holds, the sender would then puncture its current key k on the value it sampled r, and replace k by the punctured key. This way, if an adversary leaks a client's state, the G key it obtains can not be used to obtain any of the NAE key nonce pairs used to encrypt any of the messages the compromised client received. Finally, the sender would then transmit c along with r it sampled. The later needs to be transmitted in order for the receiver to be able to efficiently compute the NAE key and nonce pair used to encrypt c. The reciever then itself punctures its local G key on the input value r it used to process c.

Note that transmitting r in the clear then does not effect the forward secure anonymity of the scheme because all clients need to receive all challenge ciphertexts prior to having their state exposed to the adversary. Therefore an adversary trying to detect the identity of a sender based on whether the key is punctured or not on a certain input value r, will trivially fail, as all clients received the ciphertext corresponding to r. Of course, because r is sampled, there may be an instance where two clients sample the same input value. Since the receiver punctures the current G key immediately upon receiving a ciphertext from some client, this implies that the receiver will not be able to decrypt a legitimate ciphertext from one of the two r colliding clients.

Therefore a sufficiently large sampling space KEGSE.$\mathcal{R}$ must be provided in order for these collisions to be minimised.

More formally let NAE be a nonce-based authenticated encryption scheme with associated data and let G be a punctured function family such that $\mathsf{G.Out} = \mathsf{NAE.Nk} + \mathsf{NAE.Nn}$. Then $\mathsf{KEGSE} = \mathsf{PPRF\text{-}KEGSE}[\mathsf{G}, \mathsf{NAE}]$ is a key-evolving (stateful) group symmetric encryption scheme with $\mathsf{KEGSE.MK} = \{0,1\}^{\mathsf{G.Kl}}$ and $\mathsf{KEGSE}.\mathcal{R} = \{0,1\}^{\mathsf{G.In}}$ whose algorithm definitions are given in Figure 5.22.

```
KEGSE.Init(k, ids, scrts, i):
  ctr ← 0
  ME ← ids[i]
  st ← (ME, k, ctr)
  return st

KEGSE.Send(st,m,a;r):
  (ME, k, ctr) ← st
  kn ← G.Ev(k, r)
  if kn = ⊥ :
    return (st,⊥)
  key ← kn[0, . . . , NAE.Nk − 1]
  nonce ← kn[NAE.Nk, . . . , NAE.Nk + NAE.Nn − 1]
  k′ ← G.Punct(k, r)
  c′ ← NAE.Enc(key, nonce, ⟨a, r⟩ , ⟨ME, ctr, m⟩)
  c ← (r, c′)
  ctr + +
  st ← (ME, k′, ctr)
  return (st, c)

KEGSE.Receive(st,a,c):
  (ME, k, ctr) ← st
  if c = ⊥ :
    return (st,⊥,⊥,⊥)
  (r, c′) ← c
  kn ← G.Ev(k, r)
  if kn = ⊥ :
    return (st,⊥,⊥,⊥)
  key ← kn[0, . . . , NAE.Nk − 1]
  nonce ← kn[NAE.Nk, . . . , NAE.Nk + NAE.Nn − 1]
  k′ ← G.Punct(k, r)
  ⟨id, i, m⟩ ← NAE.Dec(key, nonce, ⟨a, r⟩ , c′)
  st ← (ME, k′, ctr)
  return (st, i, id, m)
```

Figure 5.22: PPRF-based KEGSE instantiation.

Chapter 6

Conclusions

In this chapter we give a summary of this work as well as some ideas for future research directions.

6.1 Summary

We started this work by defining the syntax and correctness game of a secure group messaging (SGM) protocol. Thereafter, we explained on a high level the properties an SGM scheme must satisfy as well as how they compare to the secure messaging protocol, which model secure asynchronous communication between two parties (instead of a dynamic group). In particular, we gave a summary of the work by Alwen et al. [ACD18], which cast the Double ratchet protocol as an SM protocol composed of 3 components: a CKA scheme, PRF-PRNG scheme and FS-AEAD scheme. We, and many other works, use this as inspiration to modularise the MLS protocol as well.

Subsequently, we describe a (simplification) of the MLS protocol and throughout it provide pseudocode, in hopes, to yield a better understanding of what goes into implementing MLS. We also comment on the security of MLS w.r.t. the security properties we laid out in the beginning. Once armed with the MLS description, we cast it in a more modular light, by showing that it consists of three main components: CGKA, PRF-PRNG and KEGSE scheme; along with a MAC and DS scheme. Since the CGKA component has been studied extensively in prior works and the PRF-PRNG scheme is the same primitive used in SM composition of Alwen et al. we divert our attention to formalising and analysing the KEGSE component.

In defining KEGSE security, we provide a formal definition of a novel property called forward-secure (sender) anonymity; which, intuitively, demands that upon client compromise the adversary gains no knowledge about the sender of a received message by the compromised client. We then go on to

show that the MLS based construction of KEGSE does not satisfy this property, and propose an alternative KEGSE scheme based on PPRFs, which we believe to satisfy this strong security property. To capture the exact security the MLS based KEGSE satisfies, we define a security notion that completely excludes the anonymity property.

To prove the MLS based KEGSE scheme secure in this 'more basic' sense, we decompose the MLS based KEGSE further into (1) a KESE scheme (the latter similar to FS-AEAD primitive used in [ACD18]), and (2) nonce based authenticated encryption (NAEAD) schemes with associated data. Following this, we show that the security definition of FS-AEAD schemes in [ACD18] is wrong, fix it and prove the MLS-based instantiation of a KESE scheme secure in the fixed game. Finally, we give a proof sketch of the MLS-based KEGSE scheme being secure in the 'more basic' sense assuming the security of the underlying KESE scheme.

6.2 Future work

Throughout researching MLS, many questions regarding some choices and assumptions made about the MLS protocol arose, some of which we presented in Section 4.13. We now recap these ambiguities and expand on them.

Namely, the first thing to question, is the assumption made on the ordering of Commit messages by the DS Broadcast service. More concretely, in Section 4.2, we noted that MLS assumes, that the DS broadcast service provides clients with a consistent view of which Commit message defines each epoch. However, the [OBR+21] document, did not provide an adequate way of realising this assumption, and hence we believe that group splitting attacks (see [ACJM20]) are a big concern.

Further, we are not convinced that the MLS key schedule's (see Section 4.8) complexity is necessary. Moreover, we do not see why a secret tree (see Section 4.9) that contains n leaves (representing n members), can not be substituted by n parallel PRF calls, which take the encryption_secret as their key and the unique identity of each member, as the input.

In addition to this, as pointed out in Section 4.13, the purpose of the parent hash field in the nodes of a ratchet tree is left unclear, as well as, the need for evolving the `GroupContext` structure during Commit message creation and processing. Moreover, we consider the purpose of the confirmation_tag (assuming that the `MLSmessage` structure is used to transfer the underlying `Commit` structure) an open problem.

In Section 4.10.3, we explained that a client sending a message, will sample a random reuse guard, in order to protect against a scenario in which a client

reuses a generation of its 'sending' symmetric ratchets. How, probable such a scenario is, given that a client needs to only maintain a monotonically incriminating counter, is debatable. Nonetheless, this reuse attack is yet to be modelled formally, and is left as possible future work.

In Section 4.7.1 and 4.10, we have seen signatures being encrypted using an NAEAD scheme (to protect sender anonymity), a paradigm not studied prior in literature, and which we consider an open problem. Moreover, the current literature about MLS does not consider various insider security notions, which we deem as an open problem as well. Finally, proving that the PPRF-based KEGSE scheme we proposed in Section 5.3.5 is indeed secure in the FS-ANONIM-GAEAD sense is left for future work.

Appendix A

Appendix

A.1 KESE security definition motivation

$\underline{\mathsf{G}^{\text{fs-aead}}_{\text{KESE}}(\mathcal{A})}$:

$\mathsf{k} \leftarrow_\$ \mathcal{K}$
$(\mathcal{A}_1, \mathcal{A}_2) \leftarrow \mathcal{A}$
$(\mathsf{id}, \mathsf{state}) \leftarrow_\$ \mathcal{A}_1$
$\mathsf{b} \leftarrow_\$ \{0,1\}$
win, corr $\leftarrow$ `false`
trans, comp, chall $\leftarrow \emptyset$
send_ctr $\leftarrow 0$
$(\mathsf{st_S}, \mathsf{st_R}) \leftarrow$ KESE.Init(k,id)
$\mathsf{b}' \leftarrow_\$ \mathcal{A}_2^{\text{LoR,Dec,corr-S,corr-R}}(\mathsf{state})$
return win **or** $\mathsf{b}' = \mathsf{b}$

<u>**delete**(i)</u>:

if $\exists \mathsf{a}, \mathsf{m}, \mathsf{c}, \mathsf{r} : (\mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c}, \mathsf{r}) \in$ trans:
  trans,comp,chall $\xleftarrow{-}$ (i,a,m,c,r)

<u>**corr-R**</u>:
req chall $= \emptyset$
end $(\mathsf{st_R}, \mathsf{st_S})$

<u>**corr-S**</u>:
corr $\leftarrow$ `true`
return $\mathsf{st_S}$

<u>**transmit**(a,m,r)</u>:

$(\mathsf{st_S}, \mathsf{c}) \leftarrow \mathsf{KESE.Send}(\mathsf{st_S}, \mathsf{a}, \mathsf{m_b}, \mathsf{r})$
trans $\xleftarrow{+}$ (send_ctr,a,m,c,r)
if corr:
  comp $\xleftarrow{+}$ (send_ctr,a,$\mathsf{m_b}$,c,r)
send_ctr $++$
return c

<u>**chall**(a,$\mathsf{m_0}$,$\mathsf{m_1}$,r)</u>:

req $|\mathsf{m_0}| = |\mathsf{m_1}|$ **and** $\neg$corr
$(\mathsf{st_S}, \mathsf{c}) \leftarrow \mathsf{KESE.Send}(\mathsf{st_S}, \mathsf{a}, \mathsf{m_b}, \mathsf{r})$
trans $\xleftarrow{+}$ (send_ctr,a,$\mathsf{m_b}$,c,r)
if $\mathsf{m_0} \neq \mathsf{m_1}$:
  chall $\xleftarrow{+}$ (send_ctr,a,$\mathsf{m_b}$,c,r)
send_ctr $++$
return c

<u>**Deliver**(a,i,c,r)</u>:

req $\exists \mathsf{m} : (\mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c}, \mathsf{r}) \in$ trans
$(\mathsf{st_R}, \mathsf{m}') \leftarrow \mathsf{KESE.Recieve}(\mathsf{st_R}, \mathsf{a}, \mathsf{i}, \mathsf{c}, \mathsf{r})$
if $(\mathsf{i}, \mathsf{a}, \mathsf{m}', \mathsf{c}, \mathsf{r}) \in$ chall:
  $\mathsf{m}' \leftarrow \perp$
delete(i)
return m'

<u>**Inject**(a,i,c,r)</u>:

req $\nexists \mathsf{m} : (\mathsf{i}, \mathsf{a}, \mathsf{m}, \mathsf{c}, \mathsf{r}) \in$ trans
$(\mathsf{st_R}, \mathsf{m}') \leftarrow \mathsf{KESE.Recieve}(\mathsf{st_R}, \mathsf{a}, \mathsf{i}, \mathsf{c}, \mathsf{r})$
if $\mathsf{m}' \neq \perp$ **and** $\nexists \mathsf{a}', \mathsf{m}', \mathsf{c}', \mathsf{r}' : (\mathsf{i}, \mathsf{a}', \mathsf{m}', \mathsf{c}', \mathsf{r}') \in$ comp:
  win $\leftarrow$ `true`
delete(i)
return m'

Figure A.1: The FS-AEAD security game and its oracles.

This section contains the deferred argument for our KESE security definition given by the game in Figure 5.9 and against the security definition provided by [ACD18]. In order to properly compare the two security notions we need to make the syntax of the schemes the same. Their primitive uses two initialization algorithms, one for the sender and another for the receiver state, which we collapse into one to fit our syntax. Therefore their FS-Init-S and FS-Init-R become KESE.Init, their FS-Send (we add r as input) corresponds to KESE.Send and FS-Recieve corresponds to KESE.Recieve. Then their security game simplified is shown in Figure A.1. The simplification comes from the fact that we collapse the Record oracle from their game into transmit and chall oracle.[1]

$\underline{\mathcal{A}_1}$:

id $\leftarrow\!\!\$$ KESE.ID
return id

$\underline{\mathcal{A}_2^{\text{LoR,Dec,corr-S,corr-R}}(\text{state})}$:

$\text{st}_S \leftarrow$ corr-S
(v, i_S, index) $\leftarrow \text{st}_S$
a $\leftarrow$ 01101011
m $\leftarrow$ 01101000
r $\leftarrow\!\!\$$ KESE.$\mathcal{R}$
$\langle \text{key}, \text{nonce}, \text{v} \rangle \leftarrow \text{F.Ev}(\text{v}, \langle \text{id}, i_S \rangle)$
nonce $\leftarrow$ nonce $\oplus$ r
c $\leftarrow$ NAE.Enc(key, nonce, a, m)
m $\leftarrow$ Inject(a, i, c, r)
return 0

Figure A.2: Adversary breaking FS-AEAD security given in Figure A.1. The bit-strings chosen here are random and the attack would work with any other choice.

We now show that the MLS based KESE scheme we proved secure against our security definition in Figure 5.9 is insecure against the game given in Figure A.1. Namely we define two adversaries that break their security of the MLS based KESE scheme. The two adversaries clearly showcase that the the problem lies in the security definition provided by [ACD18], not the MLS based KESE instantiation.

The first mistake they make in their security game is that upon client compromise, a ciphertext is only added to the comp set (containing compromised ciphertexts) if it was created by the transmit oracle. However upon state compromise of a client, an adversary is able to craft its own ciphertexts locally and has no need for the transmit oracle. Of course this kind of ciphertext should not be considered a break of security and should be added to the comp set. But the game only populates comp if transmit is called. This flaw is exactly what allows the adversary defined in Figure A.2 to win the game.

The second problem is that their deletions are not conditional. Namely if

[1] The simplifications we give here do not change their security definition but are purely for better readability.

a ciphertext corresponding to some i were to fail decryption (it decrypted to $m = \bot$) intuitively a KESE scheme would not erase the key material that was used to try and decrypt this i-th ciphertext. However since their game simply deletes the record corresponding to i this allows an adversary $\mathcal{A}$ defined in Figure A.3 to break the scheme by essentially first sending a bad ciphertext for i (that fails decryption) and then send the honestly crafted i-th ciphertext.

$\underline{\mathcal{A}_1:}$

id $\leftarrow_\$$ KESE.ID
return id

$\underline{\mathcal{A}_2^{\mathsf{LoR,Dec,corr\text{-}S,corr\text{-}R}}(\mathsf{state}):}$

a $\leftarrow$ 01101011
m $\leftarrow$ 01101000
r $\leftarrow_\$$ KESE.$\mathcal{R}$
c $\leftarrow$ transmit(a,m,r)
sample ciphertext c′
Inject(a,0,c′,r)
Inject(a,0,c,r)
return 0

Figure A.3: Adversary breaking FS-AEAD security given in Figure A.1. The bit-strings chosen here are random and the attack would work with any other choice.

A.2 FS-GAEAD definition motivation

The second flaw in the FS-AEAD security definition of [ACD18] discussed in Appendix A.1 is identical to the flaw in the FS-GAEAD security of [ACDT21] (which is supposed to capture the same security intuition as our FS-GAEAD game given in Figure 5.4). Namely their inj-AM oracle mistakenly deletes a record from the AM map even if the decryption of a ciphertext was unsuccessful. This allows us to attack the security of an FS-GAEAD scheme (Init,Send,Rcv) defined in terms of another FS-GAEAD scheme $(\mathsf{Init}', \mathsf{Send}', \mathsf{Rcv}')$ as shown in Figure A.4.

$\underline{\mathsf{Init}(k_e, n, \mathsf{ID}):}$

$s \leftarrow \mathsf{Init}'(k_e, n, \mathsf{ID})$
$\mathsf{pos} \leftarrow 1$
$v \leftarrow (s, \mathsf{pos}, \mathsf{ID})$
return v

$\underline{\mathsf{Send}(v, a, m):}$

$(s, \mathsf{pos}, \mathsf{ID}) \leftarrow v$
$a' \leftarrow (a, \mathsf{pos}, \mathsf{ID})$
$(s, e') \leftarrow \mathsf{Send}'(v, a', m)$
$\mathsf{pos}++$
$e \leftarrow (\mathsf{ID}, \mathsf{pos}, e')$
$v \leftarrow (s, \mathsf{pos}, \mathsf{ID})$
return (v, e)

$\underline{\mathsf{Rcv}(v, a, e):}$

$(s, \mathsf{pos}, \mathsf{ID}) \leftarrow v$
$(\mathsf{S}, i, e') \leftarrow e$
$a' \leftarrow (a, \mathsf{pos}, \mathsf{ID})$
$(s, \mathsf{S}', i', m) \leftarrow \mathsf{Rcv}'(s, a', e')$
$v \leftarrow (s, \mathsf{pos}, \mathsf{ID})$
return (v, S, i, m)

Figure A.4: FS-GAEAD scheme construction.

Intuitively if $(\mathsf{Init}', \mathsf{Send}', \mathsf{Rcv}')$ is a secure FS-GAEAD scheme (according to

the definition of [ACDT21]) then (Init,Send,Rcv) should also be a secure FS-GAEAD scheme. However the adversary defined in Figure A.5 manages to win the game exactly because of the non-conditional deletion flaw.

$\underline{\mathcal{A}^{\text{init, send,corr,no-del}}_{\text{dlv-AM,inj-AM,chall}}:}$

sample associated data a
sample message m
$\mathsf{S} \leftarrow_{\$} \mathsf{G}$
$\mathsf{e} \leftarrow \mathsf{send}(\mathsf{S},\mathsf{a},\mathsf{m})$
sample ciphertext e'
$\mathsf{R} \leftarrow_{\$} \mathsf{G} \setminus \{\mathsf{S}\}$
inj-AM(a,e',R)
inj-AM(a,e,R)
return 0

Figure A.5: Adversary breaking FS-GAEAD security given in [ACDT21] of the scheme given in Figure A.4.

Bibliography

[ACC$^+$19] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. Cryptology ePrint Archive, Report 2019/1489, 2019. https://ia.cr/2019/1489.

[ACD18] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. Cryptology ePrint Archive, Report 2018/1037, 2018. https://eprint.iacr.org/2018/1037.

[ACDT19] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. https://eprint.iacr.org/2019/1189.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. Cryptology ePrint Archive, Report 2021/1083, 2021. https://ia.cr/2021/1083.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography — TCC 2020*, volume 12552 of *LNCS*, Cham, 12 2020. Springer International Publishing.

[AJM20] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of mls. Cryptology ePrint Archive, Report 2020/1327, 2020. https://ia.cr/2020/1327.

[BBLW21] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-09, Internet Engineering Task Force, May 2021. Work in Progress.

[BBM+21] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. March 2021. https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html.

[BBN19] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[BDdK+21] Colin Boyd, Gareth T. Davies, Bor de Kock, Kai Gellert, Tibor Jager, and Lise Millerjord. Symmetric key exchange with full forward security and robust synchronization. Cryptology ePrint Archive, Report 2021/702, 2021. https://ia.cr/2021/702.

[BFG+19] Jacqueline Brendel, Marc Fischlin, Felix Günther, Christian Janson, and Douglas Stebila. Towards post-quantum security for signal's x3dh handshake. Cryptology ePrint Archive, Report 2019/1356, 2019. https://eprint.iacr.org/2019/1356.

[BSJ+16] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. Cryptology ePrint Archive, Report 2016/1028, 2016. https://ia.cr/2016/1028.

[BST13] Mihir Bellare, Igors Stepanovs, and Stefano Tessaro. Poly-many hardcore bits for any one-way function and a framework for differing-inputs obfuscation. Cryptology ePrint Archive, Report 2013/873, 2013. https://ia.cr/2013/873.

[BY01] Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. Cryptology ePrint Archive, Report 2001/035, 2001. https://ia.cr/2001/035.

[CGCD+16] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. https://eprint.iacr.org/2016/1013.

[CR19] John Chan and Phillip Rogaway. Anonymous ae. Cryptology ePrint Archive, Report 2019/1033, 2019. https://ia.cr/2019/1033.

[Gmb21] Wire Swiss GmbH. Wire Security Whitepaper. July 2021. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf.

[Gre16] Andy Greenberg. You Can All Finally Encrypt Facebook Messenger, So Do It. April 2016. https://www.wired.com/2016/10/facebook-completely-encrypted-messenger-update-now/.

[HKKP21] Keitaro Hashimoto, Shuichi Katsumata, Kris Kwiatkowski, and Thomas Prest. An efficient and generic construction for signal's handshake (x3dh): Post-quantum, state leakage secure, and deniable. Cryptology ePrint Archive, Report 2021/616, 2021. https://eprint.iacr.org/2021/616.

[Iqb17] Mansoor Iqbal. WhatsApp Revenue and Usage Statistics (2021). August 2017. https://www.businessofapps.com/data/whatsapp-statistics/.

[Lun18] Joshua Lund. Signal partners with Microsoft to bring end-to-end encryption to Skype. January 2018. https://signal.org/blog/skype-partnership/.

[Mar16] Moxie Marlinspike. Open Whisper Systems partners with Google on end-to-end encryption for Allo. May 2016. https://signal.org/blog/allo/.

[OBR+21] Emad Omara, Benjamin Beurdouche, Eric Rescorla, Srinivas Inguva, Albert Kwon, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Technical report, March 2021. https://messaginglayersecurity.rocks/mls-architecture/draft-ietf-mls-architecture.html.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[Siga] Signal. Signal specification and libraries. https://signal.org/docs/.

[Sigb] Signal. The X3DH Key Agreement Protocol. https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[Tan21a] H. Tankovska. Daily engagement rate of selected mobile social media apps among Android users in the United States as of June 2020. January 2021. https://www.statista.com/statistics/290492/mobile-media-apps-daily-engagement-rate-of-us-users/.

[Tan21b] H. Tankovska. Most popular global mobile messenger apps as of January 2021, based on number of monthly active users. January 2021. https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/.

[Vib] Viber. Viber encryption overview. https://www.viber.com/app/uploads/viber-encryption-overview.pdf.

[Wha16] Whatsapp. WhatsApp encryption overview. April 2016. https://scontent.whatsapp.net/v/t39.8562-34/122249142_469857720642275_2152527586907531259_n.pdf/WA_Security_WhitePaper.pdf?ccb=1-5&_nc_sid=2fbf2a&_nc_ohc=xR2mDEs-IOMAX9QFKNn&_nc_ht=scontent.whatsapp.net&oh=08d9de6e2694f73469e53a1e6d6aa0ca&oe=6147A299.