Workgroup: Network Working Group
Internet-Draft: draft-ietf-mls-protocol-14
Published: 3 May 2022
Intended Status: Informational
Expires: 4 November 2022
Authors: R. Barnes (Cisco)
B. Beurdouche (Inria & Mozilla)
R. Robert (Facebook)
J. Millican (Facebook)
E. Omara (Google)
K. Cohn-Gordon (University of Oxford)

The Messaging Layer Security (MLS) Protocol

Abstract

Messaging applications are increasingly making use of end-to-end security mechanisms to ensure that messages are only accessible to the communicating endpoints, and not to any servers involved in delivering messages. Establishing keys to provide such protections is challenging for group chat settings, in which more than two clients need to agree on a key but may not be online at the same time. In this document, we specify a key establishment protocol that provides efficient asynchronous group key establishment with forward secrecy and post-compromise security for groups in size ranging from two to thousands.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/mlswg/mls-protocol.

Status of This Memo

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of MLS and has not yet seen significant security analysis. It should not be used as a basis for building production systems.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/mlswg/mls-protocol. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the MLS mailing list.

A group of users who want to send each other encrypted messages needs a way to derive shared symmetric encryption keys. For two parties, this problem has been studied thoroughly, with the Double Ratchet emerging as a common solution [doubleratchet] [signal]. Channels implementing the Double Ratchet enjoy fine-grained forward secrecy as well as post-compromise security, but are nonetheless efficient enough for heavy use over low-bandwidth networks.

For a group of size greater than two, a common strategy is to unilaterally broadcast symmetric "sender" keys over existing shared symmetric channels, and then for each member to send messages to the group encrypted with their own sender key. Unfortunately, while this

improves efficiency over pairwise broadcast of individual messages and provides forward secrecy (with the addition of a hash ratchet), it is difficult to achieve post-compromise security with sender keys. An adversary who learns a sender key can often indefinitely and passively eavesdrop on that member's messages. Generating and distributing a new sender key provides a form of post-compromise security with regard to that sender. However, it requires computation and communications resources that scale linearly with the size of the group.

In this document, we describe a protocol based on tree structures that enable asynchronous group keying with forward secrecy and post-compromise security. Based on earlier work on "asynchronous ratcheting trees" [art], the protocol presented here uses an asynchronous key-encapsulation mechanism for tree structures. This mechanism allows the members of the group to derive and update shared keys with costs that scale as the log of the group size.

1.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-13

- *TLS syntax updates (including variable-header-length vectors) (*)
- *Stop generating redundant PKE key pairs. (*)
- *Move validation of identity change to the AS
- *Add message/mls MIME type registration
- *Split LeafNode from KeyPackage (*)
- *Remove endpoint_id (*)
- *Reorganize to make section layout more sane
- *Forbid proposals by reference in external commits (*)
- *Domain separation for KeyPackage and Proposal references (*)
- *Downgrade MUST to SHOULD for commit senders including all valid commits
- *Stronger parent hashes for authenticated identities (*)
- *Move wire_format to a separate tagged-union structure MLSMessage
- *Generalize tree extend/truncate algorithms

* Add algorithms for link-based trees

* Forbid self-Update entirely (*)

* Consolidate resumption PSK cases (*)

* 384 Ciphersuite Addition

* Remove explicit version pin on HPKE (*)

* Remove the requirement for Add in external commit (*)

* Use smaller, fixed-size hash-based identifiers (*)

* Be explicit that Credentials can attest to multiple identities (*)

draft-12

* Use the GroupContext to derive the joiner_secret (*)

* Make PreSharedKeys non optional in GroupSecrets (*)

* Update name for this particular key (*)

* Truncate tree size on removal (*)

* Use HPKE draft-08 (*)

* Clarify requirements around identity in MLS groups (*)

* Signal the intended wire format for MLS messages (*)

* Inject GroupContext as HPKE info instead of AAD (*)

* Clarify extension handling and make extension updatable (*)

* Improve extensibility of Proposals (*)

* Constrain proposal in External Commit (*)

* Remove the notion of a 'leaf index' (*)

* Add group_context_extensions proposal ID (*)

* Add RequiredCapabilities extension (*)

* Use cascaded KDF instead of concatenation to consolidate PSKs (*)

* Use key package hash to index clients in message structs (*)

* Don't require PublicGroupState for external init (*)

* Make ratchet tree section clearer.

* Handle non-member sender cases in MLSPlaintextTBS

* Clarify encoding of signatures with NIST curves

* Remove OPEN ISSUEs and TODOs

* Normalize the description of the zero vector

draft-11

* Include subtree keys in parent hash (*)

* Pin HPKE to draft-07 (*)

* Move joiner secret to the end of the first key schedule epoch (*)

* Add an AppAck proposal

* Make initializations of transcript hashes consistent

draft-10

* Allow new members to join via an external Commit (*)

* Enable proposals to be sent inline in a Commit (*)

* Re-enable constant-time Add (*)

* Change expiration extension to lifetime extension (*)

* Make the tree in the Welcome optional (*)

* PSK injection, re-init, sub-group branching (*)

* Require the initial init_secret to be a random value (*)

* Remove explicit sender data nonce (*)

* Do not encrypt to joiners in UpdatePath generation (*)

* Move MLSPlaintext signature under the confirmation tag (*)

* Explicitly authenticate group membership with MLSPLaintext (*)

* Clarify X509Credential structure (*)

* Remove unneeded interim transcript hash from GroupInfo (*)

* IANA considerations

- Derive an authentication secret

- Use Extract/Expand from HPKE KDF

- Clarify that application messages MUST be encrypted

draft-09

- Remove blanking of nodes on Add (*)

- Change epoch numbers to uint64 (*)

- Add PSK inputs (*)

- Add key schedule exporter (*)

- Sign the updated direct path on Commit, using "parent hashes" and one signature per leaf (*)

- Use structured types for external senders (*)

- Redesign Welcome to include confirmation and use derived keys (*)

- Remove ignored proposals (*)

- Always include an Update with a Commit (*)

- Add per-message entropy to guard against nonce reuse (*)

- Use the same hash ratchet construct for both application and handshake keys (*)

- Add more ciphersuites

- Use HKDF to derive key pairs (*)

- Mandate expiration of ClientInitKeys (*)

- Add extensions to GroupContext and flesh out the extensibility story (*)

- Rename ClientInitKey to KeyPackage

draft-08

- Change ClientInitKeys so that they only refer to one ciphersuite (*)

- Decompose group operations into Proposals and Commits (*)

- Enable Add and Remove proposals from outside the group (*)

- *Replace Init messages with multi-recipient Welcome message (*)
- *Add extensions to ClientInitKeys for expiration and downgrade resistance (*)
- *Allow multiple Proposals and a single Commit in one MLSPlaintext (*)

draft-07

- *Initial version of the Tree based Application Key Schedule (*)
- *Initial definition of the Init message for group creation (*)
- *Fix issue with the transcript used for newcomers (*)
- *Clarifications on message framing and HPKE contexts (*)

draft-06

- *Reorder blanking and update in the Remove operation (*)
- *Rename the GroupState structure to GroupContext (*)
- *Rename UserInitKey to ClientInitKey
- *Resolve the circular dependency that draft-05 introduced in the confirmation MAC calculation (*)
- *Cover the entire MLSPlaintext in the transcript hash (*)

draft-05

- *Common framing for handshake and application messages (*)
- *Handshake message encryption (*)
- *Convert from literal state to a commitment via the "tree hash" (*)
- *Add credentials to the tree and remove the "roster" concept (*)
- *Remove the secret field from tree node values

draft-04

- *Updating the language to be similar to the Architecture document
- *ECIES is now renamed in favor of HPKE (*)
- *Using a KDF instead of a Hash in TreeKEM (*)

draft-03

- *Added ciphersuites and signature schemes (*)
- *Re-ordered fields in UserInitKey to make parsing easier (*)
- *Fixed inconsistencies between Welcome and GroupState (*)
- *Added encryption of the Welcome message (*)

draft-02

- *Removed ART (*)
- *Allowed partial trees to avoid double-joins (*)
- *Added explicit key confirmation (*)

draft-01

- *Initial description of the Message Protection mechanism. (*)
- *Initial specification proposal for the Application Key Schedule using the per-participant chaining of the Application Secret design. (*)
- *Initial specification proposal for an encryption mechanism to protect Application Messages using an AEAD scheme. (*)
- *Initial specification proposal for an authentication mechanism of Application Messages using signatures. (*)
- *Initial specification proposal for a padding mechanism to improving protection of Application Messages against traffic analysis. (*)
- *Inversion of the Group Init Add and Application Secret derivations in the Handshake Key Schedule to be ease chaining in case we switch design. (*)
- *Removal of the UserAdd construct and split of GroupAdd into Add and Welcome messages (*)
- *Initial proposal for authenticating handshake messages by signing over group state and including group state in the key schedule (*)
- *Added an appendix with example code for tree math
- *Changed the ECIES mechanism used by TreeKEM so that it uses nonces generated from the shared secret

draft-00

*Initial adoption of draft-barnes-mls-protocol-01 as a WG item.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Client: An agent that uses this protocol to establish shared cryptographic state with other clients. A client is defined by the cryptographic keys it holds.

Group: A group represents a logical collection of clents that share a common secret value at any given time. Its state is represented as a linear sequence of epochs in which each epoch depends on its predecessor.

Epoch: A state of a group in which a specific set of authenticated clients hold shared cryptographic state.

Member: A client that is included in the shared state of a group, hence has access to the group's secrets.

Key Package: A signed object describing a client's identity and capabilities, and including a hybrid public-key encryption (HPKE [RFC9180]) public key that can be used to encrypt to that client, and which other clients can use to introduce the client to a new group.

Signature Key: A signing key pair used to authenticate the sender of a message.

Handshake Message: An MLSPlaintext or MLSCiphertext message carrying an MLS Proposal or Commit object, as opposed to application data.

Application Message: An MLSCiphertext message carrying application data.

Terminology specific to tree computations is described in Section 5.1.

In general, symmetric values are referred to as "keys" or "secrets" interchangeably. Either term denotes a value that MUST be kept confidential to a Client. When labelling individual values, we typically use "secret" to refer to a value that is used derive

further secret values, and "key" to refer to a value that is used with an algorithm such as HMAC or an AEAD algorithm.

2.1. Presentation Langauge

We use the TLS presentation language [RFC8446] to describe the structure of protocol messages. In addition to the base syntax, we add two additional features, the ability for fields to be optional and the ability for vectors to have variable-size length headers.

2.1.1. Optional Value

An optional value is encoded with a presence-signaling octet, followed by the value itself if present. When decoding, a presence octet with a value other than 0 or 1 MUST be rejected as malformed.

```
struct {
    uint8 present;
    select (present) {
        case 0: struct{};
        case 1: T value;
    }
} optional<T>;
```

2.1.2. Variable-size Vector Headers

In the TLS presentation language, vectors are encoded as a sequence of encoded elements prefixed with a length. The length field has a fixed size set by specifying the minimum and maximum lengths of the encoded sequence of elements.

In MLS, there are several vectors whose sizes vary over significant ranges. So instead of using a fixed-size length field, we use a variable-size length using a variable-length integer encoding based on the one in Section 16 of [RFC9000]. (They differ only in that the one here requires a minimum-size encoding.) Instead of presenting min and max values, the vector description simply includes a V. For example:

```
struct {
    uint32 fixed<0..255>;
    opaque variable<V>;
} StructWithVectors;
```

Such a vector can represent values with length from 0 bytes to 2^30 bytes. The variable-length integer encoding reserves the two most

significant bits of the first byte to encode the base 2 logarithm of the integer encoding length in bytes. The integer value is encoded on the remaining bits, in network byte order. The encoded value MUST use the smallest number of bits required to represent the value. When decoding, values using more bits than necessary MUST be treated as malformed.

This means that integers are encoded on 1, 2, or 4 bytes and can encode 6-, 14-, or 30-bit values respectively.

Prefix	Length	Usable Bits	Min	Max
00	1	6	0	63
01	2	14	64	16383
10	4	30	16384	1073741823
11	invalid	-	-	-

Table 1: Summary of Integer Encodings

Vectors that start with "11" prefix are invalid and MUST be rejected.

For example, the four byte sequence 0x9d7f3e7d decodes to 494878333; the two byte sequence 0x7bbd decodes to 15293; and the single byte 0x25 decodes to 37.

The following figure adapts the pseudocode provided in [RFC9000] to add a check for minimum-length encoding:

```
ReadVarint(data):
  // The length of variable-length integers is encoded in the
  // first two bits of the first byte.
  v = data.next_byte()
  prefix = v >> 6
  length = 1 << prefix

  // Once the length is known, remove these bits and read any
  // remaining bytes.
  v = v & 0x3f
  repeat length-1 times:
    v = (v << 8) + data.next_byte()
  return v

  // Check that the encoder used the minimum bits required
  if length > 1 && v < (1 << (length - 1)):
    raise an exception
```

The use of variable-size integers for vector lengths allows vectors to grow very large, up to 2^30 bytes. Implementations should take

care not to allow vectors to overflow available storage. To facilitate debugging of potential interoperatbility problems, implementations should provide a clear error when such an overflow condition occurs.

3. Operating Context

MLS is designed to operate in the context described in [I-D.ietf-mls-architecture]. In particular, we assume that the following services are provided:

* A Delivery Service that routes MLS messages among the participants in the protocol. The following types of delivery are typically required:

  - Pre-publication of KeyPackage objects for clients

  - Broadcast delivery of Proposal and Commit messages to members of a group

  - Unicast delivery of Welcome messages to new members of a group

* An Authentication Service that enables group members to authenticate the credentials presented by other group members.

4. Protocol Overview

The core functionality of MLS is continuous group authenticated key exchange (AKE). As with other authenticated key exchange protocols (such as TLS), the participants in the protocol agree on a common secret value, and each participant can verify the identity of the other participants. MLS provides group AKE in the sense that there can be more than two participants in the protocol, and continuous group AKE in the sense that the set of participants in the protocol can change over time.

The core organizing principles of MLS are *groups* and *epochs*. A group represents a logical collection of clients that share a common secret value at any given time. The history of a group is divided into a linear sequence of epochs. In each epoch, a set of authenticated *members* agree on an *epoch secret* that is known only to the members of the group in that epoch. The set of members involved in the group can change from one epoch to the next, and MLS ensures that only the members in the current epoch have access to the epoch secret. From the epoch secret, members derive further shared secrets for message encryption, group membership authentication, etc.

The creator of an MLS group creates the group's first epoch unilaterally, with no protocol interactions. Thereafter, the members

of the group advance their shared cryptographic state from one epoch to another by exchanging MLS messages:

- A *KeyPackage* object describes a client's capabilities and provides keys that can be used to add the client to a group.

- A *Proposal* message proposes a change to be made in the next epoch, such as adding or removing a member

- A *Commit* message initiates a new epoch by instructing members of the group to implement a collection of proposals

- A *Welcome* message provides a new member to the group with the information to initialize their state for the epoch in which they were added or in which they want to add themselves to the group

KeyPackage and Welcome messages are used to initiate a group or introduce new members, so they are exchanged between group members and clients not yet in the group.

Proposal and Commit messages are sent from one member of a group to the others. MLS provides a common framing layer for sending messages within a group: An *MLSPlaintext* message provides sender authentication for unencrypted Proposal and Commit messages. An *MLSCiphertext* message provides encryption and authentication for both Proposal/Commit messages as well as any application data.

4.1. Cryptographic State and Evolution

The cryptographic state at the core of MLS is divided into three areas of responsibility:

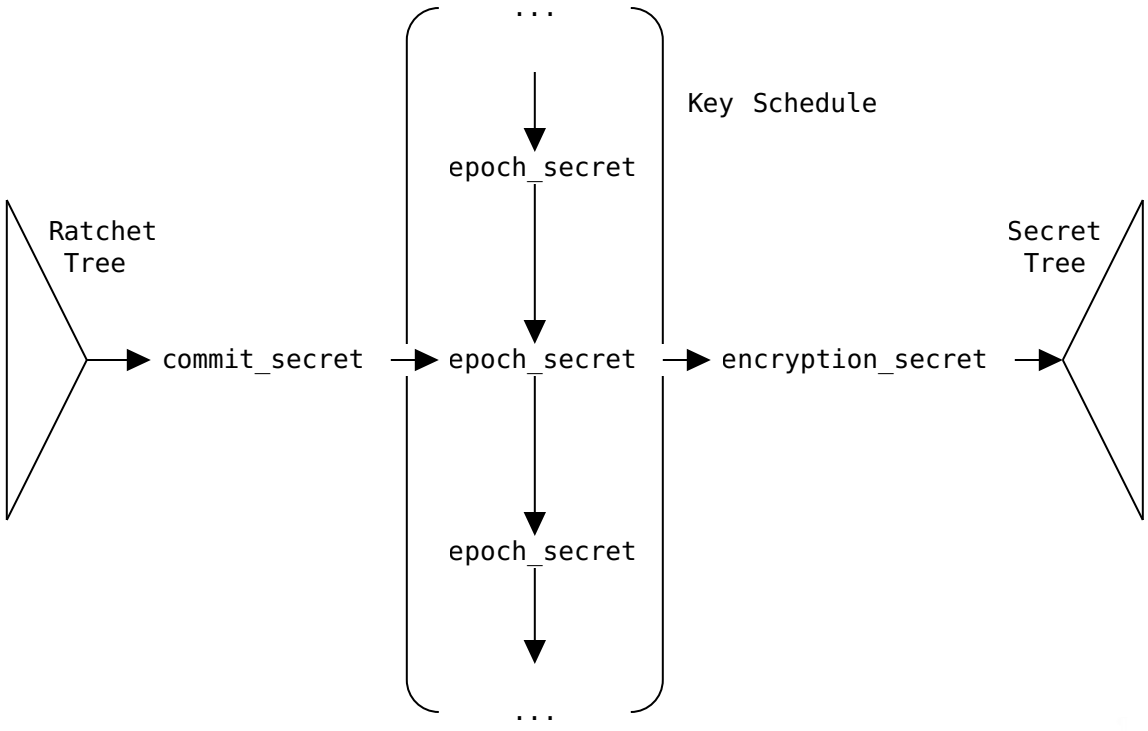


Figure 1: Overview of MLS group evolution

* A *ratchet tree* that represents the membership of the group, providing group members a way to authenticate each other and efficiently encrypt messages to subsets of the group. Each epoch has a distinct ratchet tree. It seeds the *key schedule*.

* A *key schedule* that describes the chain of key derivations used to progress from epoch to epoch (mainly using the *init_secret* and *epoch_secret*); and to derive a variety of other secrets (see Table 3) used during the current epoch. One of these (the *encryption_secret*) is the root of *secret_tree*.

* A *secret tree* derived from the key schedule that represents shared secrets used by the members of the group to provide confidentiality and forward secrecy for MLS messages. Each epoch has a distinct secret tree.

Each member of the group maintains a view of these facets of the group's state. MLS messages are used to initialize these views and keep them in sync as the group transitions between epochs.

Each new epoch is initiated with a Commit message. The Commit instructs existing members of the group to update their views of the ratchet tree by applying a set of Proposals, and uses the updated ratchet tree to distribute fresh entropy to the group. This fresh entropy is provided only to members in the new epoch, not to members

who have been removed, so it maintains the confidentiality of the epoch secret (in other words, it provides post-compromise security with respect to those members).

For each Commit that adds member(s) to the group, there is a single corresponding Welcome message. The Welcome message provides all the new members with the information they need to initialize their views of the key schedule and ratchet tree, so that these views are equivalent to the views held by other members of the group in this epoch.

In addition to defining how one epoch secret leads to the next, the key schedule also defines a collection of secrets that are derived from the epoch secret. For example:

- An *encryption secret* that is used to initialize the secret tree for the epoch.

- A *confirmation key* that is used to confirm that all members agree on the shared state of the group.

- A *resumption secret* that members can use to prove their membership in the group, e.g., in the case of branching a subgroup.

Finally, an *init secret* is derived that is used to initialize the next epoch.

4.2. Example Protocol Execution

There are three major operations in the lifecycle of a group:

- Adding a member, initiated by a current member;

- Updating the leaf secret of a member;

- Removing a member.

Each of these operations is "proposed" by sending a message of the corresponding type (Add / Update / Remove). The state of the group is not changed, however, until a Commit message is sent to provide the group with fresh entropy. In this section, we show each proposal being committed immediately, but in more advanced deployment cases an application might gather several proposals before committing them all at once. In the illustrations below, we show the Proposal and Commit messages directly, while in reality they would be sent encapsulated in MLSPlaintext or MLSCiphertext objects.

Before the initialization of a group, clients publish KeyPackages to a directory provided by the Service Provider.

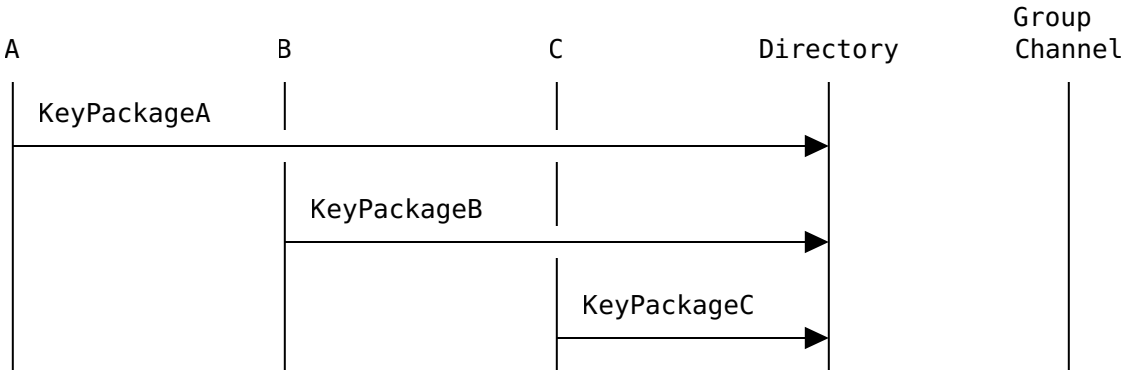


Figure 2: Clients A, B, and C publish KeyPackages to the directory

When a client A wants to establish a group with B and C, it first initializes a group state containing only itself and downloads KeyPackages for B and C. For each member, A generates an Add and Commit message adding that member, and broadcasts them to the group. It also generates a Welcome message and sends this directly to the new member (there's no need to send it to the group). Only after A has received its Commit message back from the server does it update its state to reflect the new member's addition.

Upon receiving the Welcome message, the new member will be able to read and send new messages to the group. However, messages sent before they were added to the group will not be accessible.

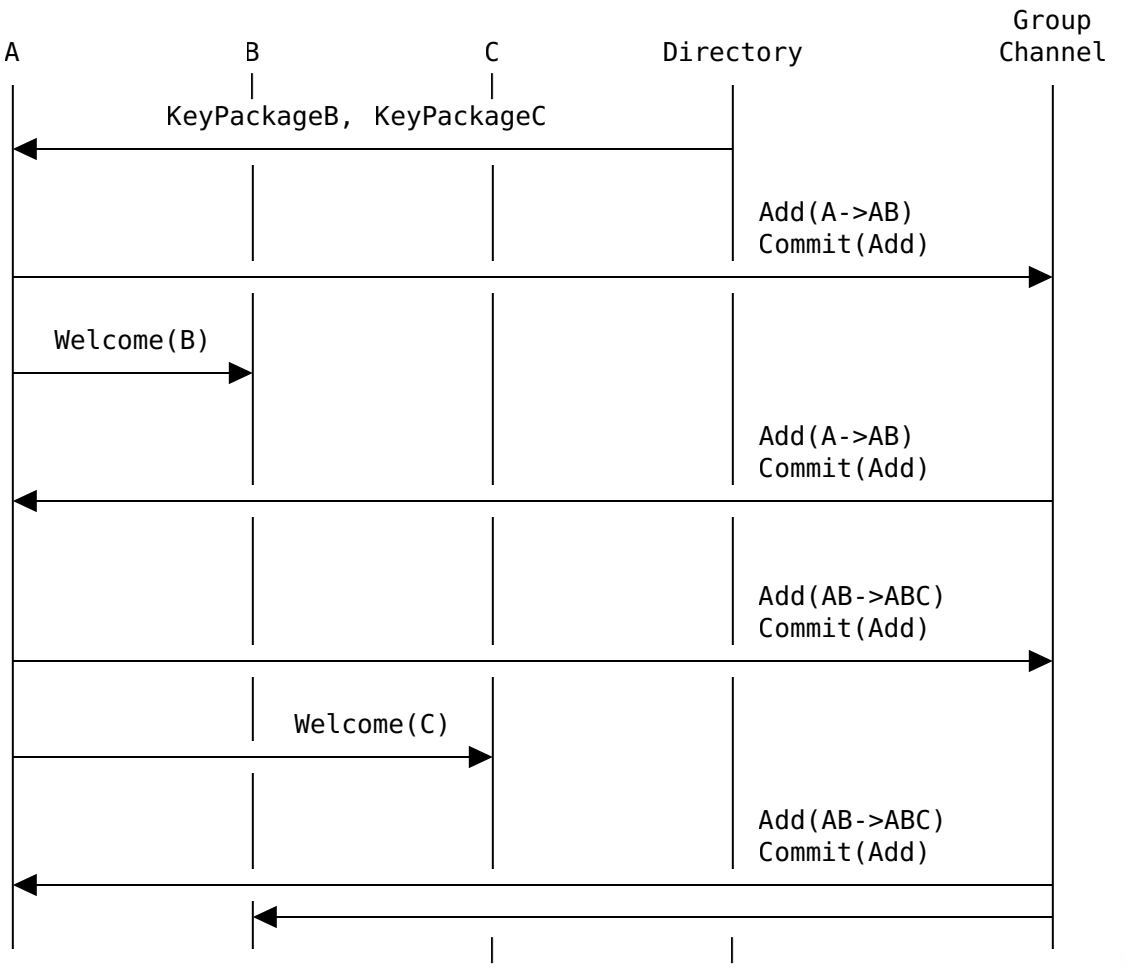


Figure 3: Client A creates a group with clients B and C

Subsequent additions of group members proceed in the same way. Any member of the group can download a KeyPackage for a new client and broadcast Add and Commit messages that the current group will use to update their state, and a Welcome message that the new client can use to initialize its state and join the group.

To enforce the forward secrecy and post-compromise security of messages, each member periodically updates the keys that represent them to the group. A member does this by sending a Commit (possibly with no proposals), or by sending an Update message that is committed by another member. Once the other members of the group have processed these messages, the group's secrets will be unknown to an attacker that had compromised the sender's prior leaf secret.

Update messages should be sent at regular intervals of time as long as the group is active, and members that don't update should eventually be removed from the group. It's left to the application to determine an appropriate amount of time between Updates.

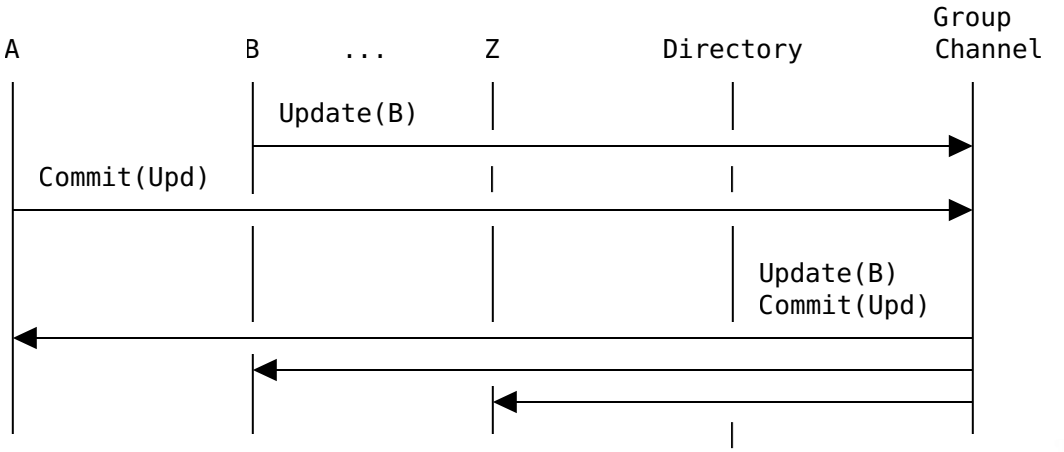


Figure 4: Client B proposes to update its key, and client A commits the proposal. As a result, the keys for both B and A updated, so the group has post-compromise security with respect to both of them.

Members are removed from the group in a similar way. Any member of the group can send a Remove proposal followed by a Commit message. The Commit message provides new entropy to all members of the group except the removed member. This new entropy is added to the epoch secret for the new epoch, so that it is not known to the removed member. Note that this does not necessarily imply that any member is actually allowed to evict other members; groups can enforce access control policies on top of these basic mechanism.

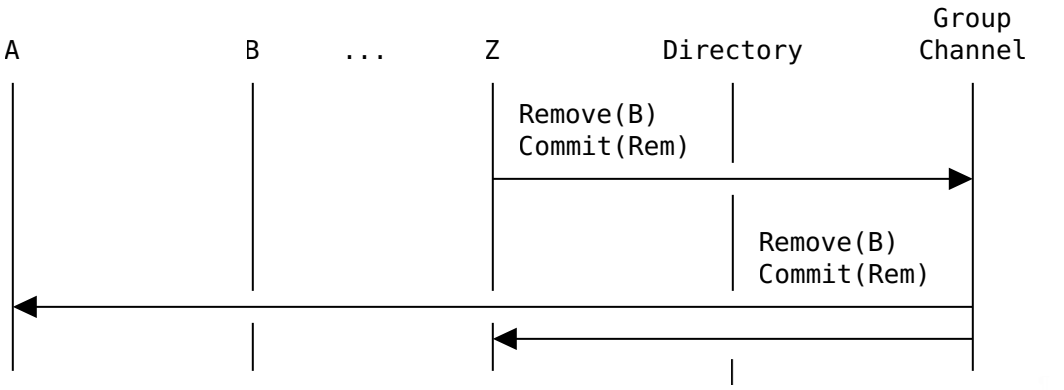


Figure 5: Client Z removes client B from the group

4.3. Relationships Between Epochs

A group has a single linear sequence of epochs. Groups and epochs are generally independent of one-another. However, it can sometimes be useful to link epochs cryptographically, either within a group or

across groups. MLS derives a resumption pre-shared key (PSK) from each epoch to allow entropy extracted from one epoch to be injected into a future epoch. This link guarantees that members entering the new epoch agree on a key if and only if they were members of the group during the epoch from which the resumption key was extracted.

MLS supports two ways to tie a new group to an existing group. Re-initialization closes one group and creates a new group comprising the same members with different parameters. Branching starts a new group with a subset of the original group's participants (with no effect on the original group). In both cases, the new group is linked to the old group via a resumption PSK.

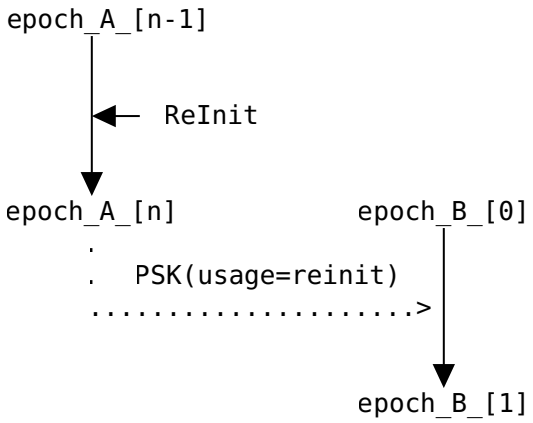


Figure 6: Reinitializing a group

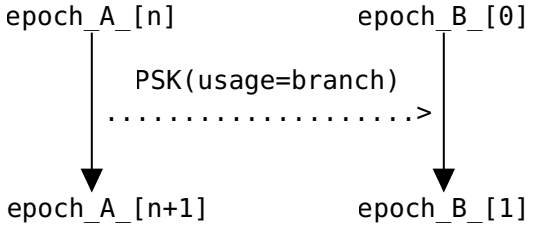


Figure 7: Branching a group

Applications may also choose to use resumption PSKs to link epochs in other ways. For example, the following figure shows a case where a resumption PSK from epoch n is injected into epoch n+k. This demonstrates that the members of the group at epoch n+k were also members at epoch n, irrespective of any changes to these members' keys due to Updates or Commits.

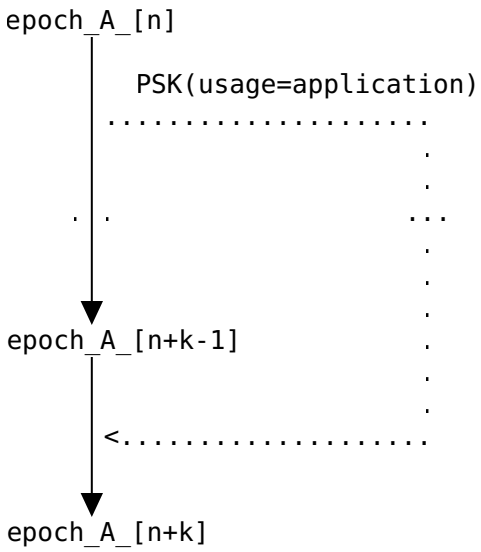


Figure 8: Reinjecting entropy from an earlier epoch

5. Ratchet Tree Concepts

The protocol uses "ratchet trees" for deriving shared secrets among a group of clients. A ratchet tree is an arrangement of secrets and key pairs among the members of a group in a way that allows for secrets to be efficiently updated to reflect changes in the group.

Ratchet trees allow a group to efficiently remove any member by encrypting new entropy to a subset of the group. A ratchet tree assigns shared keys to subgroups of the overall group, so that, for example, encrypting to all but one member of the group requires only log(N) encryptions, instead of the N-1 encryptions that would be needed to encrypt to each participant individually (where N is the number of members in the group).

This remove operation allows MLS to efficiently achieve post-compromise security. In an Update proposal or a full Commit message, an old (possibly compromised) representation of a member is efficiently removed from the group and replaced with a freshly generated instance.

5.1. Ratchet Tree Terminology

Trees consist of *nodes*. A node is a *leaf* if it has no children, and a *parent* otherwise; note that all parents in our trees have precisely two children, a *left* child and a *right* child. A node is the *root* of a tree if it has no parents, and *intermediate* if it has both children and parents. The *descendants* of a node are that node's children, and the descendants of its children, and we say a tree *contains* a node if that node is a descendant of the root of the tree, or if the node

itself is the root of the tree. Nodes are *siblings* if they share the same parent.

A *subtree* of a tree is the tree given by any node (the *head* of the subtree) and its descendants. The *size* of a tree or subtree is the number of leaf nodes it contains. For a given parent node, its *left subtree* is the subtree with its left child as head (respectively *right subtree*).

All trees used in this protocol are left-balanced binary trees. A binary tree is *full* (and *balanced*) if its size is a power of two and for any parent node in the tree, its left and right subtrees have the same size.

A binary tree is *left-balanced* if for every parent, either the parent is balanced, or the left subtree of that parent is the largest full subtree that could be constructed from the leaves present in the parent's own subtree. Given a list of n items, there is a unique left-balanced binary tree structure with these elements as leaves.

(Note that left-balanced binary trees are the same structure that is used for the Merkle trees in the Certificate Transparency protocol [I-D.ietf-trans-rfc6962-bis].)

The *direct path* of a root is the empty list, and of any other node is the concatenation of that node's parent along with the parent's direct path. The *copath* of a node is the node's sibling concatenated with the list of siblings of all the nodes in its direct path, excluding the root.

For example, in the below tree:

*The direct path of C is (W, V, X)

*The copath of C is (D, U, Z)

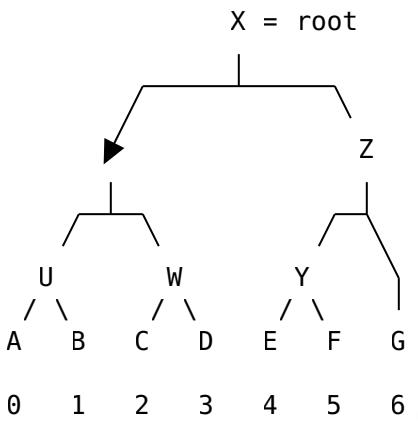


Figure 9: A complete tree with seven members

A tree with n leaves has 2*n - 1 nodes. For example, the above tree has 7 leaves (A, B, C, D, E, F, G) and 13 nodes.

Each leaf is given an *index* (or *leaf index*), starting at 0 from the left to n-1 at the right.

Finally, a node in the tree may also be *blank*, indicating that no value is present at that node (i.e. no keying material). This is often the case when a leaf was recently removed from the tree.

There are multiple ways that an implementation might represent a ratchet tree in memory. For example, left-balanced binary trees can be represented as an array of nodes, with node relationships computed based on nodes' indices in the array. Or a more traditional representation of linked node objects may be used. Appendix B and Appendix C provide some details on how to implement the tree operations required for MLS in these representations. MLS places no requirements on implementations' internal representations of ratchet trees. An implementation MAY use any tree representation and associated algorithms, as long as they produce correct protocol messages.

5.2. Views of a Ratchet Tree

We generally assume that each participant maintains a complete and up-to-date view of the public state of the group's ratchet tree, including the public keys for all nodes and the credentials associated with the leaf nodes.

No participant in an MLS group knows the private key associated with every node in the tree. Instead, each member is assigned to a leaf of the tree, which determines the subset of private keys it knows. The credential stored at that leaf is one provided by the member.

In particular, MLS maintains the members' views of the tree in such a way as to maintain the *tree invariant*:

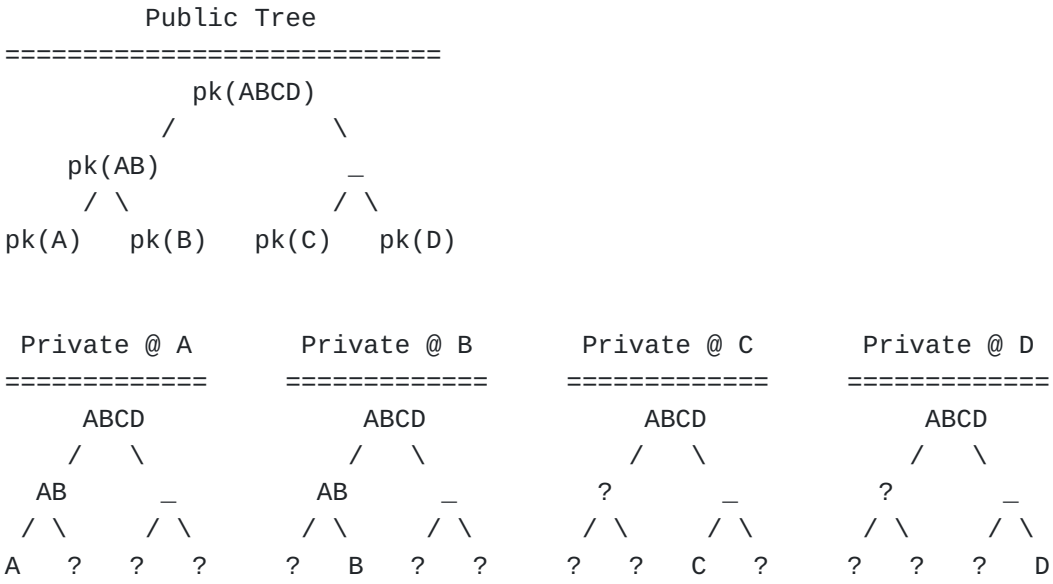
The private key for a node in the tree is known to a member of the group only if the node's subtree contains that member's leaf.

In other words, if a node is not blank, then it holds a public key. The corresponding private key is known only to members occupying leaves below that node.

The reverse implication is not true: A member may not know the private keys of all the intermediate nodes they're below. Such a member has an *unmerged* leaf. Encrypting to an intermediate node requires encrypting to the node's public key, as well as the public

keys of all the unmerged leaves below it. A leaf is unmerged when it is first added, because the process of adding the leaf does not give it access to all of the nodes above it in the tree. Leaves are "merged" as they receive the private keys for nodes, as described in Section 8.4.

For example, consider a four-member group (A, B, C, D) where the node above the right two members is blank. (This is what it would look like if A created a group with B, C, and D.) Then the public state of the tree and the views of the private keys of the tree held by each participant would be as follows, where _ represents a blank node, ? represents an unknown private key, and pk(X) represents the public key corresponding to the private key X:



Note how the tree invariant applies: Each member knows only their own leaf, and the private key AB is known only to A and B.

5.3. Ratchet Tree Nodes

A particular instance of a ratchet tree includes the same parameters that define an instance of HPKE, namely:

- A Key Encapsulation Mechanism (KEM), including a DeriveKeyPair function that creates a key pair for the KEM from a symmetric secret
- A Key Derivation Function (KDF), including Extract and Expand functions
- An AEAD encryption scheme

Each non-blank node in a ratchet tree contains up to five values:

- A public key
- A private key (only within the member's direct path, see below)
- A credential (only for leaf nodes)
- An ordered list of "unmerged" leaves (see Section 5.2)
- A hash of certain information about the node's parent, as of the last time the node was changed (see Section 8.8).

The *resolution* of a node is an ordered list of non-blank nodes that collectively cover all non-blank descendants of the node. The resolution of a non-blank node with no unmerged leaves is just the node itself. More generally, the resolution of a node is effectively a depth-first, left-first enumeration of the nearest non-blank nodes below the node:

- The resolution of a non-blank node comprises the node itself, followed by its list of unmerged leaves, if any
- The resolution of a blank leaf node is the empty list
- The resolution of a blank intermediate node is the result of concatenating the resolution of its left child with the resolution of its right child, in that order

For example, consider the following subtree, where the _ character represents a blank node and unmerged leaves are indicated in square brackets:

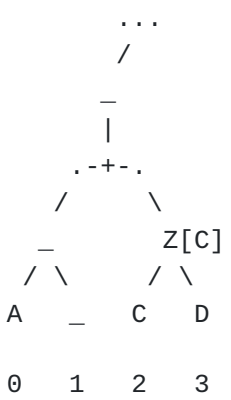


Figure 10: A tree with blanks and unmerged leaves

In this tree, we can see all of the above rules in play:

- The resolution of node Z is the list [Z, C]
- The resolution of leaf 1 is the empty list []
- The resolution of top node is the list [A, Z, C]

Every node, regardless of whether the node is blank or populated, has a corresponding *hash* that summarizes the contents of the subtree below that node. The rules for computing these hashes are described in Section 8.7.

6. Cryptographic Objects

6.1. Ciphersuites

Each MLS session uses a single ciphersuite that specifies the following primitives to be used in group key computations:

- HPKE parameters:
  - A Key Encapsulation Mechanism (KEM)
  - A Key Derivation Function (KDF)
  - An AEAD encryption algorithm
- A hash algorithm
- A MAC algorithm
- A signature algorithm

MLS uses HPKE for public-key encryption [RFC9180]. The DeriveKeyPair function associated to the KEM for the ciphersuite maps octet strings to HPKE key pairs. As in HPKE, MLS assumes that an AEAD algorithm produces a single ciphertext output from AEAD encryption (aligning with [RFC5116]), as opposed to a separate ciphertext and tag.

Ciphersuites are represented with the CipherSuite type. HPKE public keys are opaque values in a format defined by the underlying protocol (see the Cryptographic Dependencies section of the HPKE specification for more information).

```
opaque HPKEPublicKey<V>;
```

The signature algorithm specified in the ciphersuite is the mandatory algorithm to be used for signatures in MLSMessageAuth and the tree signatures. It MUST be the same as the signature algorithm specified in the credentials in the leaves of the tree (including the leaf node information in KeyPackages used to add new members).

Like HPKE public keys, signature public keys are represented as opaque values in a format defined by the cipher suite's signature scheme.

```
opaque SignaturePublicKey<V>;
```

For ciphersuites using Ed25519 or Ed448 signature schemes, the public key is in the format specified in [RFC8032]. For ciphersuites using ECDSA with the NIST curves (P-256, P-384, or P-521), the public key is the output of the uncompressed Elliptic-Curve-Point-to-Octet-String conversion according to [SECG].

To disambiguate different signatures used in MLS, each signed value is prefixed by a label as shown below:

```
SignWithLabel(SignatureKey, Label, Content) =
    Signature.Sign(SignatureKey, SignContent)

VerifyWithLabel(VerificationKey, Label, Content) =
    Signature.Verify(VerificationKey, SignContent)
```

Where SignContent is specified as:

```
struct {
    opaque label<V> = "MLS 1.0 " + Label;
    opaque content<V> = Content;
} SignContent;
```

Here, the functions Signature.Sign and Signature.Verify are defined by the signature algorithm.

The ciphersuites are defined in section Section 18.1.

6.2. Hash-Based Identifiers

Some MLS messages refer to other MLS objects by hash. For example, Welcome messages refer to KeyPackages for the members being welcomed, and Commits refer to Proposals they cover. These identifiers are computed as follows:

```
opaque HashReference[16];

HashReference KeyPackageRef;
HashReference LeafNodeRef;
HashReference ProposalRef;
```

```
MakeKeyPackageRef(value) = KDF.expand(
  KDF.extract("", value), "MLS 1.0 KeyPackage Reference", 16)

MakeLeafNodeRef(value) = KDF.expand(
  KDF.extract("", value), "MLS 1.0 Leaf Node Reference", 16)

MakeProposalRef(value) = KDF.expand(
  KDF.extract("", value), "MLS 1.0 Proposal Reference", 16)
```

For a KeyPackageRef, the value input is the encoded KeyPackage, and the ciphersuite specified in the KeyPackage determines the KDF used. For a LeafNodeRef, the value input is the LeafNode object for the leaf node in question. For a ProposalRef, the value input is the MLSMessageContentAuth carrying the proposal. In the latter two cases, the KDF is determined by the group's ciphersuite.

6.3. Credentials

Each member of a group presents a credential that associates an identity with the member's key material. This information is verified according to the Authentication Service in use for a group.

A Credential can provide multiple identifiers for the client. It is up to the application to decide which identifier or identifiers to use at the application level. For example, a certificate in an X509Credential may attest to several domain names or email addresses in its subjectAltName extension. An application may decide to present all of these to a user, or if it knows a "desired" domain name or email address, it can check that the desired identifier is among those attested. Using the terminology from [RFC6125], a Credential provides "presented identifiers", and it is up to the application to supply a "reference identifier" for the authenticated client, if any.

Credentials MAY also include information that allows a relying party to verify the identity / signing key binding, such as a signature from a trusted authority.

```
// See IANA registry for registered values
uint16 CredentialType;

struct {
    opaque cert_data<V>;
} Certificate;

struct {
    CredentialType credential_type;
    select (credential_type) {
        case basic:
            opaque identity<V>;

        case x509:
            Certificate chain<V>;
    };
} Credential;
```

A BasicCredential is a bare assertion of an identity, without any additional information. The format of the encoded identity is defined by the application.

For an X.509 credential, each entry in the chain represents a single DER-encoded X.509 certificate. The chain is ordered such that the first entry (chain[0]) is the end-entity certificate and each subsequent certificate in the chain MUST be the issuer of the previous certificate. The public key encoded in the subjectPublicKeyInfo of the end-entity certificate MUST be identical to the signature_key in the LeafNode containing this credential.

The signatures used in this document are encoded as specified in [RFC8446]. In particular, ECDSA signatures are DER-encoded and EdDSA signatures are defined as the concatenation of r and s as specified in [RFC8032].

Each new credential that has not already been validated by the application MUST be validated against the Authentication Service. Applications SHOULD require that a client present the same set of identifiers throughout its presence in the group, even if its Credential is changed in a Commit or Update. If an application allows clients to change identifiers over time, then each time the client presents a new credential, the application MUST verify that the set of identifiers in the credential is acceptable to the application for this client.

6.3.1. Uniquely Identifying Clients

MLS implementations will presumably provide applications with a way to request protocol operations with regard to other clients (e.g., removing clients). Such functions will need to refer to the other clients using some identifier. MLS clients have a few types of identifiers, with different operational properties.

The Credentials presented by the clients in a group authenticate application-level identifiers for the clients. These identifiers may not uniquely identify clients. For example, if a user has multiple devices that are all present in an MLS group, then those devices' clients could all present the user's application-layer identifiers.

Internally to the protocol, group members are uniquely identified by their leaves, expressed as LeafNodeRef objects. These identifiers are unstable: They change whenever the member sends a Commit, or whenever an Update proposal from the member is committed.

MLS provides two unique client identifiers that are stable across epochs:

- The index of a client among the leaves of the tree
- The epoch_id field in the key package

The application may also provide application-specific unique identifiers in the extensions field of KeyPackage or LeafNode objects.

7. Message Framing

Handshake and application messages use a common framing structure. This framing provides encryption to ensure confidentiality within the group, as well as signing to authenticate the sender within the group.

The main structure is MLSMessageContent, which contains the content of the message. This structure is authenticated using MLSMessageAuth (see Section 7.1). The two structures are combined in MLSMessageContentAuth, which can then be encoded/decoded from/to MLSPlaintext or MLSCiphertext, which are then included in the MLSMessage structure.

MLSCiphertext represents a signed and encrypted message, with protections for both the content of the message and related metadata. MLSPlaintext represents a message that is only signed, and not encrypted. Applications MUST use MLSCiphertext to encrypt application messages and SHOULD use MLSCiphertext to encode handshake messages, but MAY transmit handshake messages encoded as MLSPlaintext objects

in cases where it is necessary for the Delivery Service to examine such messages.

```
enum {
    reserved(0),
    mls10(1),
    (255)
} ProtocolVersion;

enum {
    reserved(0),
    application(1),
    proposal(2),
    commit(3),
    (255)
} ContentType;

enum {
    reserved(0),
    member(1),
    external(2),
    new_member(3),
    (255)
} SenderType;

struct {
    SenderType sender_type;
    switch (sender_type) {
        case member:
            LeafNodeRef member_ref;
        case external:
            uint32 sender_index;
        case new_member:
            struct{};
    }
} Sender;

enum {
  reserved(0),
  mls_plaintext(1),
  mls_ciphertext(2),
  mls_welcome(3),
  mls_group_info(4),
  mls_key_package(5),
  (255)
} WireFormat;

struct {
    opaque group_id<V>;
    uint64 epoch;
    Sender sender;
    opaque authenticated_data<V>;
```

```
    ContentType content_type;
    select (MLSMessageContent.content_type) {
        case application:
          opaque application_data<V>;
        case proposal:
          Proposal proposal;
        case commit:
          Commit commit;
    }
} MLSMessageContent;

struct {
    ProtocolVersion version = mls10;
    WireFormat wire_format;
    select (MLSMessage.wire_format) {
        case mls_plaintext:
            MLSPlaintext plaintext;
        case mls_ciphertext:
            MLSCiphertext ciphertext;
        case mls_welcome:
            Welcome welcome;
        case mls_group_info:
            GroupInfo group_info;
        case mls_key_package:
            KeyPackage key_package;
    }
} MLSMessage;
```

External sender types are sent as MLSPlaintext, see Section 13.1.9 for their use.

The following structure is used to fully describe the data transmitted in plaintexts or ciphertexts.

```
struct {
    WireFormat wire_format;
    MLSMessageContent content;
    MLSMessageAuth auth;
} MLSMessageContentAuth;
```

7.1. Content Authentication

MLSMessageContent is authenticated using the MLSMessageAuth structure.

```
struct {
    opaque mac_value<V>;
} MAC;

struct {
    // SignWithLabel(., "MLSMessageContentTBS", MLSMessageContentTBS)
    opaque signature<V>;
    select (MLSMessageContent.content_type) {
        case commit:
            // MAC(confirmation_key,
            //     GroupContext.confirmed_transcript_hash)
            MAC confirmation_tag;
        case application:
        case proposal:
            struct{};
    }
} MLSMessageAuth;
```

The signature is computed using SignWithLabel with label "MLSMessageContentTBS" and with a content that covers the message content and the wire format that will be used for this message. If the sender's sender_type is Member, the content also covers the GroupContext for the current epoch, so that signatures are specific to a given group and epoch.

The sender MUST use the private key corresponding to the following signature key depending on the sender's sender_type:

- *member: The signature key contained in the Credential at the leaf with the sender's LeafNodeRef

- *external: The signature key contained in the Credential at the index indicated by the sender_index in the external_senders group context extension (see Section Section 13.1.9.1). In this case, the content_type of the message MUST NOT be commit, since only members of the group or new joiners can send Commit messages.

- *new_member: The signature key depends on the content_type:

  - -proposal: The signature key in the credential contained in KeyPackage in the Add proposal (see Section Section 13.1.9).

  - -commit: The signature key in the credential contained in the KeyPackage in the Commit's path (see Section Section 9.3).

```
struct {
    ProtocolVersion version = mls10;
    WireFormat wire_format;
    MLSMessageContent content;
    select (MLSMessageContentTBS.content.sender.sender_type) {
        case member:
        case new_member:
            GroupContext context;

        case external:
            struct{};
    }
} MLSMessageContentTBS;
```

Recipients of an MLSMessage MUST verify the signature with the key depending on the sender_type of the sender as described above.

The confirmation tag value confirms that the members of the group have arrived at the same state of the group.

A MLSMessageAuth is said to be valid when both the signature and confirmation_tag fields are valid.

7.2. Encoding and Decoding a Plaintext

Plaintexts are encoded using the MLSPlaintext structure.

```
struct {
    MLSMessageContent content;
    MLSMessageAuth auth;
    select(MLSPlaintext.content.sender.sender_type) {
        case member:
            MAC membership_tag;
        case external:
        case new_member:
            struct{};
    }
} MLSPlaintext;
```

The membership_tag field in the MLSPlaintext object authenticates the sender's membership in the group. For messages sent by members, it MUST be set to the following value:

```
struct {
  MLSMessageContentTBS content_tbs;
  MLSMessageAuth auth;
} MLSMessageContentTBM;
```

```
membership_tag = MAC(membership_key, MLSMessageContentTBM)
```

When decoding a MLSPlaintext into a MLSMessageContentAuth, the application MUST check membership_tag, and MUST check that the MLSMessageAuth is valid.

7.3. Encoding and Decoding a Ciphertext

Ciphertexts are encoded using the MLSCiphertext structure.

```
struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<V>;
    opaque encrypted_sender_data<V>;
    opaque ciphertext<V>;
} MLSCiphertext;
```

encrypted_sender_data and ciphertext are encrypted using the AEAD function specified by the ciphersuite in use, using as input the structures MLSSenderData and MLSCiphertextContent.

7.3.1. Content Encryption

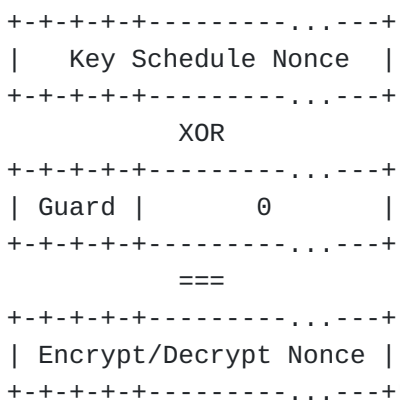
The ciphertext content is encoded using the MLSCiphertextContent structure.

```
struct {
    select (MLSCiphertext.content_type) {
        case application:
          opaque application_data<V>;

        case proposal:
          Proposal proposal;

        case commit:
          Commit commit;
    }

    MLSMessageAuth auth;
    opaque padding<V>;
} MLSCiphertextContent;
```

In the MLS key schedule, the sender creates two distinct key ratchets for handshake and application messages for each member of the group. When encrypting a message, the sender looks at the ratchets it derived for its own member and chooses an unused generation from either the handshake or application ratchet depending on the content type of the message. This generation of the ratchet is used to derive a provisional nonce and key.

Before use in the encryption operation, the nonce is XORed with a fresh random value to guard against reuse. Because the key schedule generates nonces deterministically, a client must keep persistent state as to where in the key schedule it is; if this persistent state is lost or corrupted, a client might reuse a generation that has already been used, causing reuse of a key/nonce pair.

To avoid this situation, the sender of a message MUST generate a fresh random 4-byte "reuse guard" value and XOR it with the first four bytes of the nonce from the key schedule before using the nonce for encryption. The sender MUST include the reuse guard in the reuse_guard field of the sender data object, so that the recipient of the message can use it to compute the nonce to be used for decryption.



The Additional Authenticated Data (AAD) input to the encryption contains an object of the following form, with the values used to identify the key and nonce:

```
struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<V>;
} MLSCiphertextContentAAD;
```

When decoding a MLSCiphertextContent, the application MUST check that the MLSMessageAuth is valid.

7.3.2. Sender Data Encryption

The "sender data" used to look up the key for the content encryption is encrypted with the ciphersuite's AEAD with a key and nonce derived from both the sender_data_secret and a sample of the encrypted content. Before being encrypted, the sender data is encoded as an object of the following form:

```
struct {
    LeafNodeRef sender;
    uint32 generation;
    opaque reuse_guard[4];
} MLSSenderData;
```

MLSSenderData.sender is assumed to be a member sender type. When constructing an MLSSenderData from a Sender object, the sender MUST verify Sender.sender_type is member and use Sender.sender for MLSSenderData.sender.

The reuse_guard field contains a fresh random value used to avoid nonce reuse in the case of state loss or corruption, as described in Section 7.3.1.

The key and nonce provided to the AEAD are computed as the KDF of the first KDF.Nh bytes of the ciphertext generated in the previous section. If the length of the ciphertext is less than KDF.Nh, the whole ciphertext is used without padding. In pseudocode, the key and nonce are derived as:

```
ciphertext_sample = ciphertext[0..KDF.Nh-1]

sender_data_key = ExpandWithLabel(sender_data_secret, "key",
                      ciphertext_sample, AEAD.Nk)
sender_data_nonce = ExpandWithLabel(sender_data_secret, "nonce",
                      ciphertext_sample, AEAD.Nn)
```

The Additional Authenticated Data (AAD) for the SenderData ciphertext is the first three fields of MLSCiphertext:

```
struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
} MLSSenderDataAAD;
```

When parsing a SenderData struct as part of message decryption, the recipient MUST verify that the LeafNodeRef indicated in the sender field identifies a member of the group.

8. Ratchet Tree Operations

The ratchet tree for an epoch describes the membership of a group in that epoch, providing public-key encryption (HPKE) keys that can be used to encrypt to subsets of the group as well as information to authenticate the members. In order to reflect changes to the membership of the group from one epoch to the next, corresponding changes are made to the ratchet tree. In this section, we describe the content of the tree and the required operations.

8.1. Parent Node Contents

As discussed in Section 5.3, the nodes of a ratchet tree contain several types of data describing individual members (for leaf nodes) or subgroups of the group (for parent nodes). Parent nodes are simpler:

```
struct {
    HPKEPublicKey encryption_key;
    opaque parent_hash<V>;
    uint32 unmerged_leaves<V>;
} ParentNode;
```

The encryption_key field contains an HPKE public key whose private key is held only by the members at the leaves among its descendants. The parent_hash field contains a hash of this node's parent node, as described in Section 8.8. The unmerged_leaves field lists the leaves under this parent node that are unmerged, according to their indices among all the leaves in the tree.

8.2. Leaf Node Contents

A leaf node in the tree describes all the details of an individual client's appearance in the group, signed by that client. It is also used in client KeyPackage objects to store the information that will be needed to add a client to a group.

```
enum {
    reserved(0),
    key_package(1),
    update(2),
    commit(3),
    (255)
} LeafNodeSource;

struct {
    ProtocolVersion versions<V>;
    CipherSuite ciphersuites<V>;
    ExtensionType extensions<V>;
    ProposalType proposals<V>;
    CredentialType credentials<V>;
} Capabilities;

struct {
    uint64 not_before;
    uint64 not_after;
} Lifetime;

// See IANA registry for registered values
uint16 ExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<V>;
} Extension;

struct {
    HPKEPublicKey encryption_key;
    SignaturePublicKey signature_key;
    Credential credential;
    Capabilities capabilities;

    LeafNodeSource leaf_node_source;
    select (leaf_node_source) {
        case key_package:
            Lifetime lifetime;

        case update:
            struct {}

        case commit:
            opaque parent_hash<V>;
    }

    Extension extensions<V>;
    // SignWithLabel(., "LeafNodeTBS", LeafNodeTBS)
    opaque signature<V>;
```

```
} LeafNode;

struct {
    HPKEPublicKey encryption_key;
    SignaturePublicKey signature_key;
    Credential credential;
    Capabilities capabilities;

    LeafNodeSource leaf_node_source;
    select (leaf_node_source) {
        case key_package:
            Lifetime lifetime;

        case update:
            struct{};

        case commit:
            opaque parent_hash<V>;
    }

    Extension extensions<V>;

    select (leaf_node_source) {
        case key_package:
            struct{};

        case update:
            opaque group_id<V>;

        case commit:
            opaque group_id<V>;
    }
} LeafNodeTBS;
```

The encryption_key field conains an HPKE public key whose private key is held only by the member occupying this leaf (or in the case of a LeafNode in a KeyPackage object, the issuer of the KeyPackage). The credential contains authentication information for this member, as described in Section 6.3.

The capabilities field indicates what protocol versions, ciphersuites, extensions, credential types, and non-default proposal types are supported by a client. Proposal and extension types defined in this document are considered "default" and thus need not be listed, while any credential types the application wishes to use MUST be listed. Extensions that appear in the extensions field of a LeafNode MUST be included in the extensions field of the capabilities

field, and the credential type used in the LeafNode MUST be included in the credentials field of the capabilities field.

The leaf_node_source field indicates how this LeafNode came to be added to the tree. This signal tells other members of the group whether the leaf node is required to have a lifetime or parent_hash, and whether the group_id is added as context to the signature. Whether these fields can be computed by the client represented by the LeafNode depends on when the LeafNode was created. For example, a KeyPackage is created before the client knows which group it will be used with, so its signature can't bind to a group_id.

In the case where the leaf was added to the tree based on a pre-published KeyPackage, the lifetime field represents the times between which clients will consider a LeafNode valid. These times are represented as absolute times, measured in seconds since the Unix epoch (1970-01-01T00:00:00Z). Applications MUST define a maximum total lifetime that is acceptable for a LeafNode, and reject any LeafNode where the total lifetime is longer than this duration.

In the case where the leaf node was inserted into the tree via a Commit message, the parent_hash field contains the parent hash for this leaf node (see Section 8.8).

The LeafNodeTBS structure covers the fields above the signature in the LeafNode. In addition, when the leaf node was created in the context of a group (the update and commit cases), the group ID of the group is added as context to the signature.

LeafNode objects stored in the group's ratchet tree are updated according to the evolution of the tree. Each modification of LeafNode content MUST be reflected by a change in its signature. This allows other members to verify the validity of the LeafNode at any time, particularly in the case of a newcomer joining the group.

8.3. Leaf Node Validation

The validity of a LeafNode needs to be verified at a few stages:

* When a LeafNode is downloaded in a KeyPackage, before it is used to add the client to the group

* When a LeafNode is received by a group member in an Add, Update, or Commit message

* When a client joining a group receives LeafNode objects for the other members of the group in the group's ratchet tree

The client verifies the validity of a LeafNode using the following steps:

* Verify that the credential in the LeafNode is valid according to the authentication service and the client's local policy. These actions MUST be the same regardless of at what point in the protocol the LeafNode is being verified with the following exception: If the LeafNode is an update to another LeafNode, the authentication service MUST additionally validate that the set of identities attested by the credential in the new LeafNode is acceptable relative to the identities attested by the old credential.

* Verify that the signature on the LeafNode is valid using the public key in the LeafNode's credential

* Verify that the LeafNode is compatible with the group's parameters. If the GroupContext has a required_capabilities extension, then the required extensions, proposals, and credential types MUST be listed in the LeafNode's capabilities field.

* Verify that the credential type is supported by all members of the group, as specified by the capabilities field of each member's LeafNode, and that the capabilities field of this LeafNode indicates support for all the credential types currently in use by other members.

* Verify the lifetime field:

  - When validating a LeafNode in a KeyPackage before sending an Add proposal, the current time MUST be within the lifetime range. A KeyPackage containing a LeafNode that is expired or not yet valid MUST NOT be sent in an Add proposal.

  - When receiving an Add or validating a tree, checking the lifetime is RECOMMENDED, if it is feasible in a given application context. Because of the asynchronous nature of MLS, the lifetime may have been valid when the leaf node was proposed for addition, even if it is expired at these later points in the protocol.

* Verify that the leaf_node_source field has the appropriate value for the context in which the LeafNode is being validated (as defined in Section 8.2).

* Verify that the following fields in the LeafNode are unique among the members of the group (including any other members added in the same Commit):

  - encryption_key

-signature_key

*Verify that the extensions in the leaf node are supported. The ID for each extension in the extensions field MUST be listed in the field capabilities.extensions of the LeafNode.

8.4. Ratchet Tree Evolution

In order to provide forward secrecy and post-compromise security, whenever a member initiates an epoch change (i.e., commits; see Section 13.2), they refresh the key pairs of their leaf and of all nodes on their leaf's direct path (all nodes for which they know the secret key).

The member initiating the epoch change generates the fresh key pairs using the following procedure. The procedure is designed in a way that allows group members to efficiently communicate the fresh secret keys to other group members, as described in Section 8.9.

Recall the definition of resolution from Section 5.3. To begin with, the generator of the UpdatePath updates its leaf and its leaf's *filtered direct path* with new key pairs. The filtered direct path of a node is obtained from the node's direct path by removing all nodes whose child on the nodes's copath has an empty resolution (any unmerged leaves of the copath child count towards its resolution). Such a removed node does not need a key pair, since after blanking it, its resolution consists of a single node on the filtered direct path. Using the key pair of the node in the resolution is equivalent to using the key pair of the removed node.

*Blank all the nodes on the direct path from the leaf to the root.

*Generate a fresh HPKE key pair for the leaf.

*Generate a sequence of path secrets, one for each node on the leaf's filtered direct path, as follows. In this setting, path_secret[0] refers to the first parent node in the filtered direct path, path_secret[1] to the second parent node, and so on.

```
path_secret[0] is sampled at random
path_secret[n] = DeriveSecret(path_secret[n-1], "path")
```

*Compute the sequence of HPKE key pairs (node_priv,node_pub), one for each node on the leaf's direct path, as follows.

```
node_secret[n] = DeriveSecret(path_secret[n], "node")
node_priv[n], node_pub[n] = KEM.DeriveKeyPair(node_secret[n])
```

The node secret is derived as a temporary intermediate secret so that each secret is only used with one algorithm: The path secret is used as an input to DeriveSecret and the node secret is used as an input to DeriveKeyPair.

For example, suppose there is a group with four members, with C an unmerged leaf at Z:

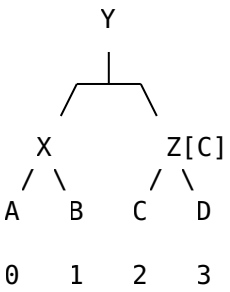
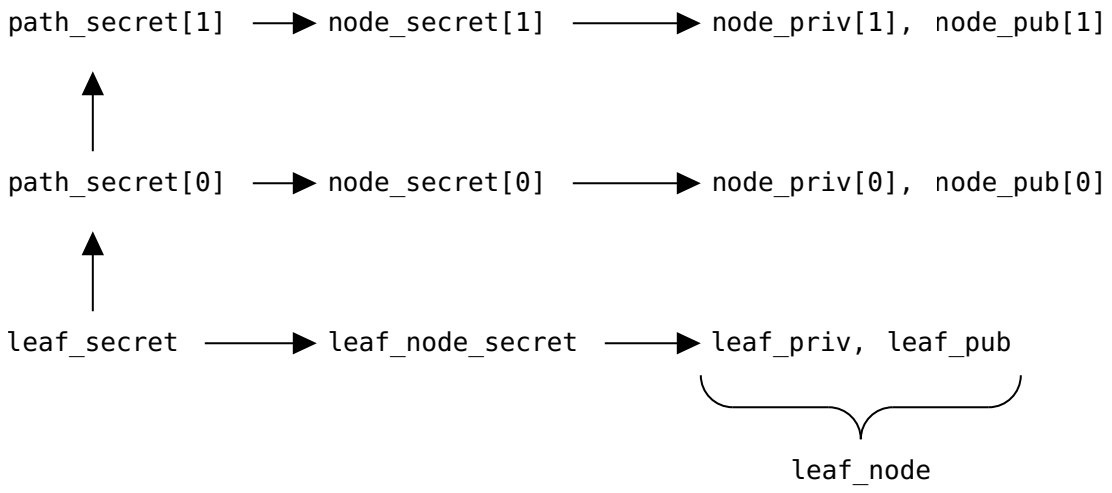
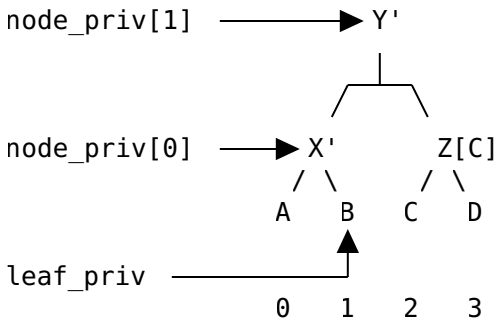


Figure 11: A full tree with one unmerged leaf

If member B subsequently generates an UpdatePath based on a secret "leaf_secret", then it would generate the following sequence of path secrets:



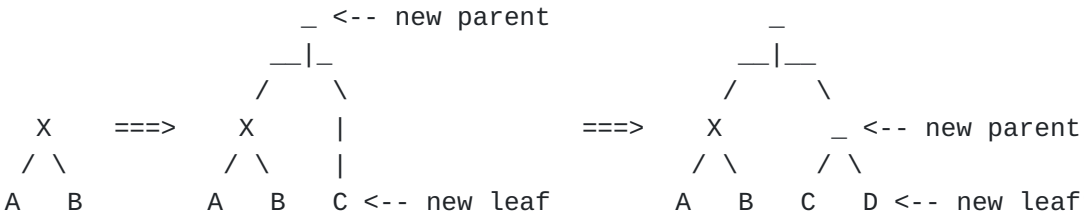
After applying the UpdatePath, the tree will have the following structure, where lp and np[i] represent the leaf_priv and node_priv values generated as described above:



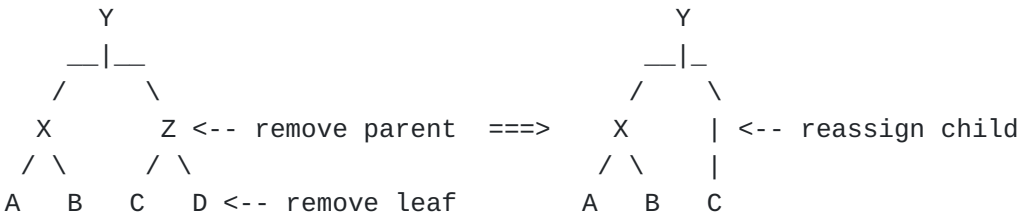
8.5. Adding and Removing Leaves

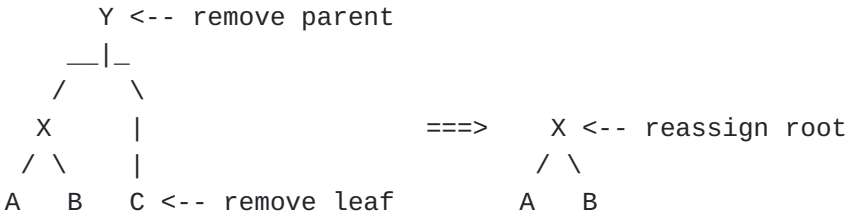
In addition to the path-based updates to the tree described above, it is also necessary to add and remove leaves of the tree in order to reflect changes to the membership of the group (see Section 13.1.1 and Section 13.1.3). Leaves are always added and removed at the right edge of the tree: Either a new rightmost leaf is added, or the rightmost leaf is removed. Nodes' parent/child node relationships are then updated to maintain the tree's left-balanced structure. These operations are also known as *extending* and *truncating* the tree.

To add a new leaf: Add leaf L as the new rightmost leaf of the tree. Add a blank parent node P whose right child is L. P is attached to the tree as the right child of the only appropriate node to make the updated tree left-balanced (or set it as a new root). The former right child of P's parent becomes P's left child (or the old root becomes P's left child if P is the new root).



To remove the rightmost leaf: Remove the rightmost leaf node L and its parent node P. If P was the root of the tree, P's left child is now the root of the tree. Otherwise, set the right child of P's parent to be P's left child.





Note that in the rest of the protocol, the rightmost leaf will only be removed when it is blank.

Concrete algorithms for these operations on array-based and link-based trees are provided in Appendix B and Appendix C. The concrete algorithms are non-normative. An implementation MAY use any algorithm that produces the correct tree in its internal representation.

8.6. Synchronizing Views of the Tree

After generating fresh key material and applying it to ratchet forward their local tree state as described in the Section 8.4, the generator must broadcast this update to other members of the group in a Commit message, who apply it to keep their local views of the tree in sync with the sender's. More specifically, when a member commits a change to the tree (e.g., to add or remove a member), it transmits an UpdatePath containing a set of public keys and encrypted path secrets for intermediate nodes in the filtered direct path of its leaf. The other members of the group use these values to update their view of the tree, aligning their copy of the tree to the sender's.

An UpdatePath contains the following information for each node in the filtered direct path of the sender's leaf, including the root:

- The public key for the node
- Zero or more encrypted copies of the path secret corresponding to the node

The path secret value for a given node is encrypted for the subtree corresponding to the parent's non-updated child, that is, the child on the copath of the sender's leaf node. There is one encryption of the path secret to each public key in the resolution of the non-updated child.

The recipient of an UpdatePath processes it with the following steps:

1. Compute the updated path secrets.

    * Identify a node in the filtered direct path for which the recipient is in the subtree of the non-updated child.

    * Identify a node in the resolution of the copath node for which the recipient has a private key.

    * Decrypt the path secret for the parent of the copath node using the private key from the resolution node.

    * Derive path secrets for ancestors of that node using the algorithm described above.

    * The recipient SHOULD verify that the received public keys agree with the public keys derived from the new path_secret values.

2. Merge the updated path secrets into the tree.

    * Blank all nodes on the direct path of the sender's leaf.

    * For all nodes on the filtered direct path of the sender's leaf,

        - Set the public key to the received public key.

        - Set the list of unmerged leaves to the empty list.

        - Store the updated hash of the next node on the filtered direct path (represented as a ParentNode struct), going from root to leaf, so that each hash incorporates all the non-blank nodes above it. The root node always has a zero-length hash for this value.

    * For nodes where a path secret was recovered in step 1 ("Compute the updated path secrets"), compute and store the node's updated private key.

For example, in order to communicate the example update described in the previous section, the sender would transmit the following values:

Public Key	**Ciphertext(s)**
node_pub[1]	E(pk(Z), path_secret[1]), E(pk(C), path_secret[1])
node_pub[0]	E(pk(A), path_secret[0])

Table 2

In this table, the value node_pub[i] represents the public key derived from node_secret[i], pk(X) represents the current public key of node X, and E(K, S) represents the public-key encryption of the path secret S to the public key K (using HPKE).

After processing the update, each recipient MUST delete outdated key material, specifically:

- The path secrets used to derive each updated node key pair.

- Each outdated node key pair that was replaced by the update.

8.7. Tree Hashes

To allow group members to verify that they agree on the public cryptographic state of the group, this section defines a scheme for generating a hash value (called the "tree hash") that represents the contents of the group's ratchet tree and the members' leaf nodes. The tree hash of a tree is the tree hash of its root node, which we define recursively, starting with the leaves.

The tree hash of a leaf node is the hash of leaf's LeafNodeHashInput object which might include a LeafNode object depending on whether or not it is blank.

```
struct {
    uint32 leaf_index;
    optional<LeafNode> leaf_node;
} LeafNodeHashInput;
```

Now the tree hash of any non-leaf node is recursively defined to be the hash of its ParentNodeHashInput. This includes an optional ParentNode object depending on whether the node is blank or not.

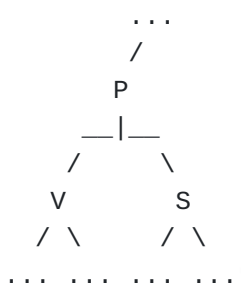
```
struct {
    optional<ParentNode> parent_node;
    opaque left_hash<V>;
    opaque right_hash<V>;
} ParentNodeHashInput;
```

The left_hash and right_hash fields hold the tree hashes of the node's left and right children, respectively.

8.8. Parent Hash

The parent_hash field in ratchet tree nodes carries information to authenticate the information in the ratchet tree. Parent hashes chain together so that the signature on a leaf node, by covering the leaf node's parent hash, indirectly includes information about the structure of the tree at the time the leaf node was last updated.

Consider a ratchet tree with a non-blank parent node P and children V and S.



The parent hash of P changes whenever an UpdatePath object is applied to the ratchet tree along a path from a leaf U traversing node V (and hence also P). The new "Parent hash of P (with copath child S)" is obtained by hashing P's ParentHashInput struct.

```
struct {
    HPKEPublicKey encryption_key;
    opaque parent_hash<V>;
    opaque original_sibling_tree_hash<V>;
} ParentHashInput;
```

The field encryption_key contains the HPKE public key of P. If P is the root, then the parent_hash field is set to a zero-length octet string. Otherwise, parent_hash is the Parent Hash of the next node after P on the filtered direct path of U. This way, P's Parent Hash fixes the new HPKE public key of each node V on the path from P to the root. Note that the path from P to the root may contain some blank nodes that are not fixed by P's Parent Hash. However, for each node that has an HPKE key, this key is fixed by P's Parent Hash.

Finally, original_sibling_tree_hash is the tree hash of S in the ratchet tree modified as follows:

* Extend the subtree of S by adding blank leaves until it is full, i.e., until its number of leaves is a power of 2 (see Section 8.5).

*For each leaf L in P.unmerged_leaves, blank L and remove it from the unmerged_leaves sets of all parent nodes.

Observe that original_sibling_tree_hash does not change between updates of P. This property is crucial for the correctness of the protocol.

For example, in the following tree:

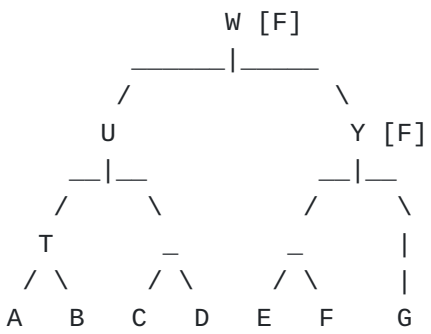
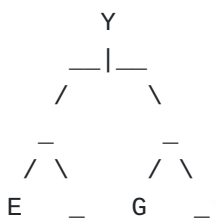


Figure 12: A tree illustrating parent hash computations.

With P = W and S = Y, original_sibling_tree_hash is the tree hash of the following tree:



Because W.unmerged_leaves includes F, F is blanked and removed from Y.unmerged_leaves.

Note that no recomputation is needed if the tree hash of S is unchanged since the last time P was updated. This is the case for computing or processing a Commit whose UpdatePath traverses P, since the Commit itself resets P. (In other words, it is only necessary to recompute the original sibling tree hash when validating group's tree on joining.) More generally, if none of the entries in P.unmerged_leaves is in the subtree under S (and thus no nodes were truncated), then the original tree hash at S is the tree hash of S in the current tree.

If it is necessary to recompute the original tree hash of a node, the efficiency of recomputation can be improved by caching intermediate tree hashes, to avoid recomputing over the subtree when the subtree is included in multiple parent hashes. A subtree hash can be reused

as long as the intersection of the parent's unmerged leaves with the subtree is the same as in the earlier computation.

8.8.1. Using Parent Hashes

The Parent Hash of P appears in three types of structs. If V is itself a parent node then P's Parent Hash is stored in the parent_hash field of the structs ParentHashInput and ParentNode of the node before P on the filtered direct path of U. (The ParentNode struct is used to encapsulate all public information about that node that must be conveyed to a new member joining the group as well as to define its Tree Hash.)

If, on the other hand, V is the leaf U and its LeafNode has leaf_node_source set to commit, then the Parent Hash of P (with V's sibling as copath child) is stored in the parent_hash field. This is true in particular of the LeafNode object sent in the leaf_node field of an UpdatePath. The signature of such a LeafNode thus also attests to which keys the group member introduced into the ratchet tree and to whom the corresponding secret keys were sent. This helps prevent malicious insiders from constructing artificial ratchet trees with a node V whose HPKE secret key is known to the insider yet where the insider isn't assigned a leaf in the subtree rooted at V. Indeed, such a ratchet tree would violate the tree invariant.

8.8.2. Verifying Parent Hashes

Parent hashes are verified at two points in the protocol: When joining a group and when processing a Commit.

The parent hash in a node U is valid with respect to a parent node P if the following criteria hold:

- *U is a descendant of P in the tree
- *The nodes between U and P in the tree are all blank
- *The parent_hash field of U is equal to the parent hash of P with copath child S, where S is the child of P that is not on the path from U to P.

A parent node P is "parent-hash valid" if it can be chained back to a leaf node in this way. That is, if there is leaf node L and a sequence of parent nodes P_1, ..., P_N such that P_N = P and each step in the chain is authenticated by a parent hash: L's parent hash is valid with respect to P_1, P_1's parent hash is valid with respect to P_2, and so on.

When joining a group, the new member MUST authenticate that each non-blank parent node P is parent-hash valid. This can be done "bottom

up" by building chains up from leaves and verifying that all non-blank parent nodes are covered by exactly one such chain, or "top down" by verifying that there is exactly one descendant of each non-blank parent node for which the parent node is parent-hash valid.

When processing a Commit message that includes an UpdatePath, clients MUST recompute the expected value of parent_hash for the committer's new leaf and verify that it matches the parent_hash value in the supplied leaf_node. After being merged into the tree, the nodes in the UpdatePath form a parent-hash chain from the committer's leaf to the root.

8.9. Update Paths

As described in Section 13.2, each MLS Commit message may optionally transmit a LeafNode and parent node values along its direct path. The path contains a public key and encrypted secret value for all intermediate nodes in the filtered direct path from the leaf to the root. The path is ordered from the closest node to the leaf to the root; each node MUST be the parent of its predecessor.

```
struct {
    opaque kem_output<V>;
    opaque ciphertext<V>;
} HPKECiphertext;

struct {
    HPKEPublicKey encryption_key;
    HPKECiphertext encrypted_path_secret<V>;
} UpdatePathNode;

struct {
    LeafNode leaf_node;
    UpdatePathNode nodes<V>;
} UpdatePath;
```

For each UpdatePathNode, the resolution of the corresponding copath node MUST be filtered by removing all new leaf nodes added as part of this MLS Commit message. The number of ciphertexts in the encrypted_path_secret vector MUST be equal to the length of the filtered resolution, with each ciphertext being the encryption to the respective resolution node.

The HPKECiphertext values are computed as

```
kem_output, context = SetupBaseS(node_public_key, group_context)
ciphertext = context.Seal("", path_secret)
```

where node_public_key is the public key of the node that the path secret is being encrypted for, group_context is the current GroupContext object for the group, and the functions SetupBaseS and Seal are defined according to [RFC9180].

Decryption is performed in the corresponding way, using the private key of the resolution node.

9. Key Schedule

Group keys are derived using the Extract and Expand functions from the KDF for the group's ciphersuite, as well as the functions defined below:

```
ExpandWithLabel(Secret, Label, Context, Length) =
    KDF.Expand(Secret, KDFLabel, Length)

DeriveSecret(Secret, Label) =
    ExpandWithLabel(Secret, Label, "", KDF.Nh)
```

Where KDFLabel is specified as:

```
struct {
    uint16 length = Length;
    opaque label<V> = "MLS 1.0 " + Label;
    opaque context<V> = Context;
} KDFLabel;
```

The value KDF.Nh is the size of an output from KDF.Extract, in bytes. In the below diagram:

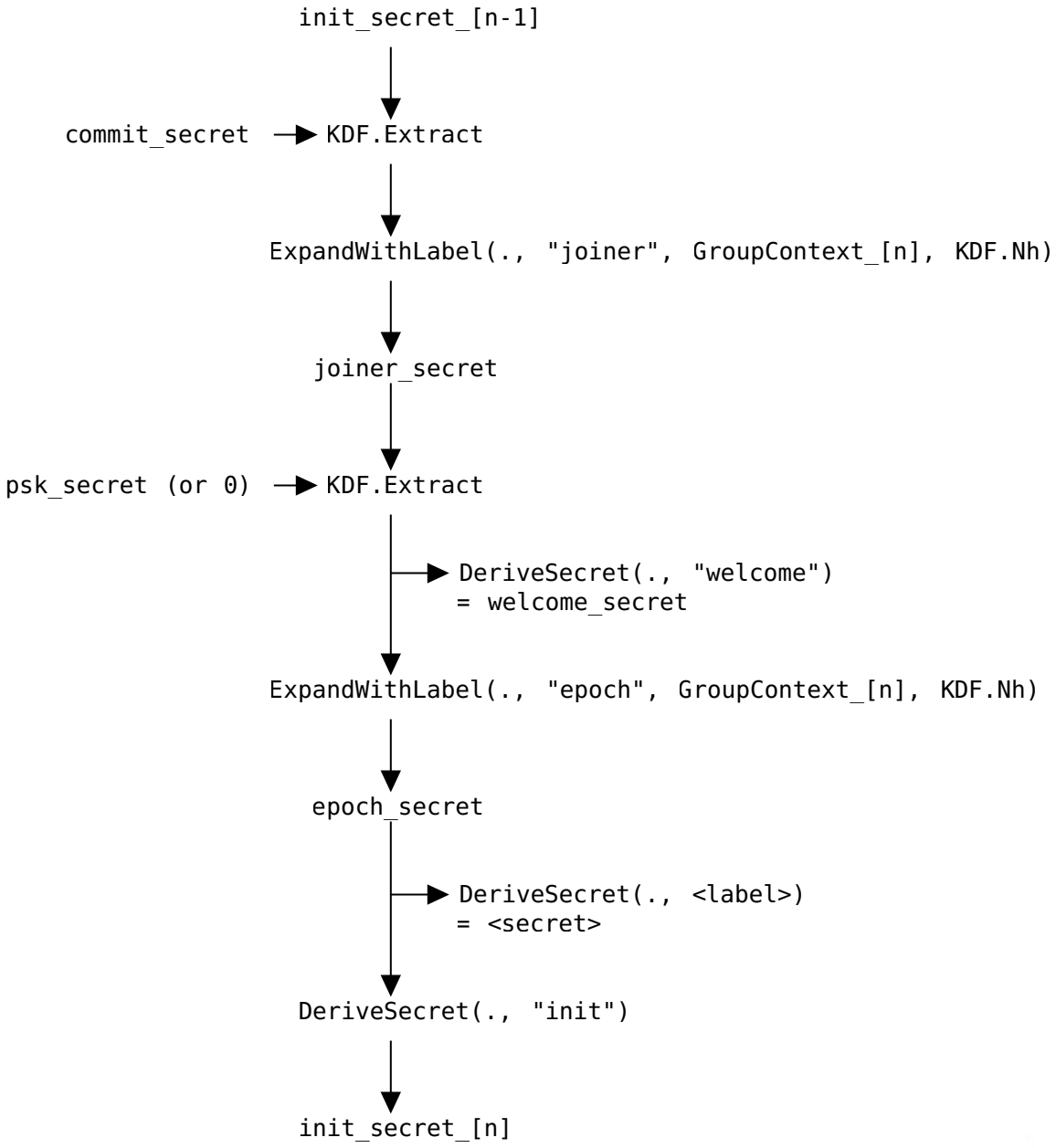
- *KDF.Extract takes its salt argument from the top and its Input Key Material (IKM) argument from the left
- *DeriveSecret takes its Secret argument from the incoming arrow
- *0 represents an all-zero byte string of length KDF.Nh.

When processing a handshake message, a client combines the following information to derive new epoch secrets:

- *The init secret from the previous epoch
- *The commit secret for the current epoch

*The GroupContext object for current epoch

Given these inputs, the derivation of secrets for an epoch proceeds as shown in the following diagram:



A number of values are derived from the epoch secret for different purposes:

Label	Secret	Purpose
"sender data"	sender_data_secret	Deriving keys to encrypt sender data
"encryption"	encryption_secret	

Label	Secret	Purpose
		Deriving message encryption keys (via the secret tree)
"exporter"	exporter_secret	Deriving exported secrets
"external"	external_secret	Deriving the external init key
"confirm"	confirmation_key	Computing the confirmation MAC for an epoch
"membership"	membership_key	Computing the membership MAC for an MLSPlaintext
"resumption"	resumption_psk	Proving membership in this epoch (via a PSK injected later)
"authentication"	epoch_authenticator	Confirming that two clients have the same view of the group

Table 3: Epoch-derived secrets

The external_secret is used to derive an HPKE key pair whose private key is held by the entire group:

```
external_priv, external_pub = KEM.DeriveKeyPair(external_secret)
```

The public key external_pub can be published as part of the GroupInfo struct in order to allow non-members to join the group using an external commit.

9.1. Group Context

Each member of the group maintains a GroupContext object that summarizes the state of the group:

```
struct {
    opaque group_id<V>;
    uint64 epoch;
    opaque tree_hash<V>;
    opaque confirmed_transcript_hash<V>;
    Extension extensions<V>;
} GroupContext;
```

The fields in this state have the following semantics:

- The group_id field is an application-defined identifier for the group.
- The epoch field represents the current version of the group.

* The tree_hash field contains a commitment to the contents of the group's ratchet tree and the credentials for the members of the group, as described in Section 8.7.

* The confirmed_transcript_hash field contains a running hash over the messages that led to this state.

* The extensions field contains the details of any protocol extensions that apply to the group.

When a new member is added to the group, an existing member of the group provides the new member with a Welcome message. The Welcome message provides the information the new member needs to initialize its GroupContext.

Different changes to the group will have different effects on the group state. These effects are described in their respective subsections of Section 13.1. The following general rules apply:

* The group_id field is constant.

* The epoch field increments by one for each Commit message that is processed.

* The tree_hash is updated to represent the current tree and credentials.

* The confirmed_transcript_hash field is updated with the data for an MLSPlaintext message encoding a Commit message as described below.

* The extensions field changes when a GroupContextExtensions proposal is committed.

9.2. Transcript Hashes

The transcript hashes computed in MLS represent a running hash over all Proposal and Commit messages that have ever been sent in a group. Commit messages are included directly. Proposal messages are indirectly included via the Commit that applied them. Both types of message are included by hashing the MLSPlaintext in which they were sent.

The confirmed_transcript_hash is updated with an MLSMessageContent and MLSMessageAuth containing a Commit in two steps:

```
struct {
    WireFormat wire_format;
    MLSMessageContent content; //with content.content_type == commit
    opaque signature<V>;
} MLSMessageCommitContent;

struct {
    MAC confirmation_tag;
} MLSMessageCommitAuthData;
```

```
interim_transcript_hash_[0] = ""; // zero-length octet string

confirmed_transcript_hash_[n] =
    Hash(interim_transcript_hash_[n] ||
        MLSMessageCommitContent_[n]);

interim_transcript_hash_[n+1] =
    Hash(confirmed_transcript_hash_[n] ||
        MLSMessageCommitAuthData_[n]);
```

Thus the confirmed_transcript_hash field in a GroupContext object represents a transcript over the whole history of MLSMessage Commit messages, up to the confirmation_tag field of the most recent Commit. The confirmation tag is then included in the transcript for the next epoch. The interim transcript hash is computed by new members using the confirmation_tag of the GroupInfo struct, while existing members can compute it directly.

As shown above, when a new group is created, the interim_transcript_hash field is set to the zero-length octet string.

9.3. External Initialization

In addition to initializing a new epoch via KDF invocations as described above, an MLS group can also initialize a new epoch via an asymmetric interaction using the external key pair for the previous epoch. This is done when an new member is joining via an external commit.

In this process, the joiner sends a new init_secret value to the group using the HPKE export method. The joiner then uses that init_secret with information provided in the GroupInfo and an external Commit to initialize their copy of the key schedule for the new epoch.

```
kem_output, context = SetupBaseS(external_pub, "")
init_secret = context.export("MLS 1.0 external init secret", KDF.Nh)
```

Members of the group receive the kem_output in an ExternalInit proposal and preform the corresponding calculation to retrieve the init_secret value.

```
context = SetupBaseR(kem_output, external_priv, "")
init_secret = context.export("MLS 1.0 external init secret", KDF.Nh)
```

In both cases, the info input to HPKE is set to the GroupInfo for the previous epoch, encoded using the TLS serialization.

9.4. Pre-Shared Keys

Groups which already have an out-of-band mechanism to generate shared group secrets can inject those into the MLS key schedule to seed the MLS group secrets computations by this external entropy.

Injecting an external PSK can improve security in the case where having a full run of updates across members is too expensive, or if the external group key establishment mechanism provides stronger security against classical or quantum adversaries.

Note that, as a PSK may have a different lifetime than an update, it does not necessarily provide the same Forward Secrecy (FS) or Post-Compromise Security (PCS) guarantees as a Commit message. Unlike the key pairs populated in the tree by an Update or Commit, which are always freshly generated, PSKs may be pre-distributed and stored. This creates the risk that a PSK may be compromised in the process of distribution and storage. The security that the group gets from injecting a PSK thus depends on both the entropy of the PSK and the risk of compromise. These factors are outside of the scope of this document, but should be considered by application designers relying on PSKs.

Each PSK in MLS has a type that designates how it was provisioned. External PSKs are provided by the application, while resumption PSKs are derived from the MLS key schedule and used in cases where it is necessary to authenticate a member's participation in a prior epoch.

The injection of one or more PSKs into the key schedule is signaled in two ways: Existing members are informed via PreSharedKey proposals covered by a Commit, and new members added in the Commit are informed via GroupSecrets object in the Welcome message corresponding to the Commit. To ensure that existing and new members compute the same PSK

input to the key schedule, the Commit and GroupSecrets objects MUST indicate the same set of PSKs, in the same order.

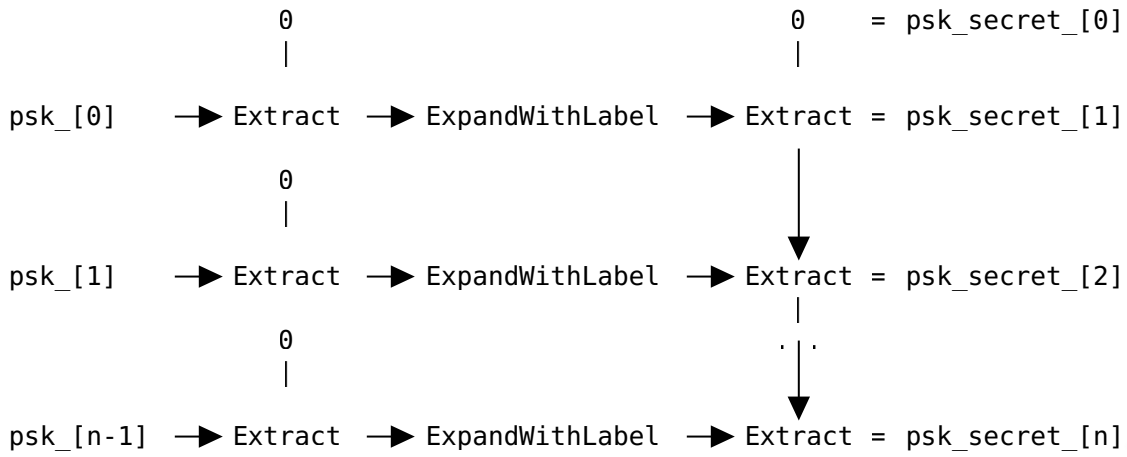
```
enum {
  reserved(0),
  external(1),
  resumption(2),
  (255)
} PSKType;

enum {
  reserved(0),
  application(1),
  reinit(2),
  branch(3),
} ResumptionPSKUsage;

struct {
  PSKType psktype;
  select (PreSharedKeyID.psktype) {
    case external:
      opaque psk_id<V>;

    case resumption:
      ResumptionPSKUsage usage;
      opaque psk_group_id<V>;
      uint64 psk_epoch;
  }
  opaque psk_nonce<V>;
} PreSharedKeyID;

struct {
    PreSharedKeyID psks<V>;
} PreSharedKeys;
```

On receiving a Commit with a PreSharedKey proposal or a GroupSecrets object with the psks field set, the receiving Client includes them in the key schedule in the order listed in the Commit, or in the psks field respectively. For resumption PSKs, the PSK is defined as the resumption_psk of the group and epoch specified in the PreSharedKeyID object. Specifically, psk_secret is computed as follows:

```
struct {
    PreSharedKeyID id;
    uint16 index;
    uint16 count;
} PSKLabel;
```

```
psk_extracted_[i] = KDF.Extract(0, psk_[i])
psk_input_[i] = ExpandWithLabel(psk_extracted_[i], "derived psk",
                  PSKLabel, KDF.Nh)

psk_secret_[0] = 0
psk_secret_[i] = KDF.Extract(psk_input_[i-1], psk_secret_[i-1])
psk_secret     = psk_secret_[n]
```

Here 0 represents the all-zero vector of length KDF.Nh. The index field in PSKLabel corresponds to the index of the PSK in the psk array, while the count field contains the total number of PSKs. In other words, the PSKs are chained together with KDF.Extract invocations (labelled "Extract" for brevity in the diagram), as follows:



In particular, if there are no PreSharedKey proposals in a given Commit, then the resulting psk_secret is psk_secret_[0], the all-zero vector.

9.5. Exporters

The main MLS key schedule provides an exporter_secret which can be used by an application as the basis to derive new secrets called exported_value outside the MLS layer.

```
MLS-Exporter(Label, Context, key_length) =
      ExpandWithLabel(DeriveSecret(exporter_secret, Label),
                      "exporter", Hash(Context), key_length)
```

Each application SHOULD provide a unique label to MLS-Exporter that identifies its use case. This is to prevent two exported outputs from being generated with the same values and used for different functionalities.

The exported values are bound to the group epoch from which the exporter_secret is derived, hence reflects a particular state of the group.

It is RECOMMENDED for the application generating exported values to refresh those values after a Commit is processed.

9.6. Resumption PSK

The main MLS key schedule provides a resumption_psk that is used as a PSK to inject entropy from one epoch into another. This functionality is used in the reinitialization and branching processes described in Section 12.2 and Section 12.3, but may be used by applications for other purposes.

Some uses of resumption PSKs might call for the use of PSKs from historical epochs. The application SHOULD specify an upper limit on the number of past epochs for which the resumption_psk may be stored.

9.7. Epoch Authenticators

The main MLS key schedule provides a per-epoch epoch_authenticator. If one member of the group is being impersonated by an active attacker, the epoch_authenticator computed by their client will differ from those computed by the other group members.

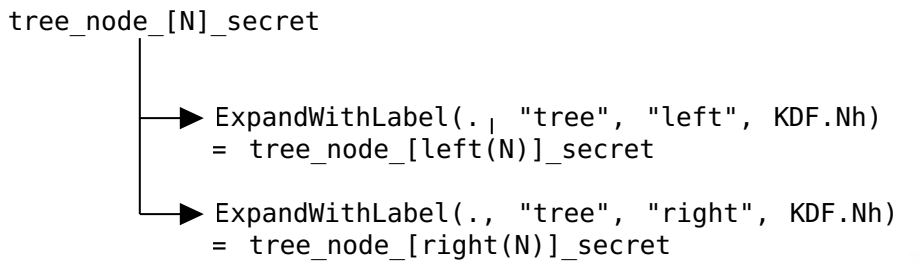
This property can be used to construct defenses against impersonation attacks that are effective even if members' signature keys are compromised. As a trivial example, if the users of the clients in an MLS group were to meet in person and reliably confirm that their epoch authenticator values were equal (using some suitable user interface), then each user would be assured that the others were not being impersonated in the current epoch. As soon as the epoch changed, though, they would need to re-do this confirmation. The state of the group would have changed, possibly introducing an attacker.

More generally, in order for the members of an MLS group to obtain concrete authentication protections using the epoch_authenticator, they will need to use it in some secondary protocol (such as the face-to-face protocol above). The details of that protocol will then determine the specific authentication protections provided to the MLS group.

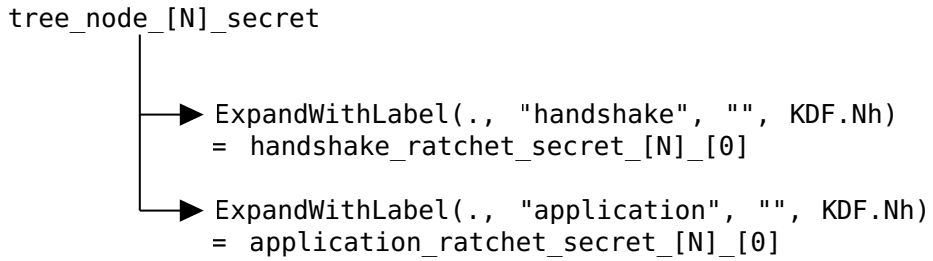
10. Secret Tree

For the generation of encryption keys and nonces, the key schedule begins with the encryption_secret at the root and derives a tree of secrets with the same structure as the group's ratchet tree. Each leaf in the Secret Tree is associated with the same group member as the corresponding leaf in the ratchet tree.

If N is a parent node in the Secret Tree then the secrets of the children of N are defined as follows (where left(N) and right(N) denote the children of N):



The secret in the leaf of the Secret Tree is used to initiate two symmetric hash ratchets, from which a sequence of single-use keys and nonces are derived, as described in Section 10.1. The root of each ratchet is computed as:



10.1. Encryption Keys

As described in Section 7, MLS encrypts three different types of information:

- Metadata (sender information)
- Handshake messages (Proposal and Commit)
- Application messages

The sender information used to look up the key for content encryption is encrypted with an AEAD where the key and nonce are derived from

both sender_data_secret and a sample of the encrypted message content.

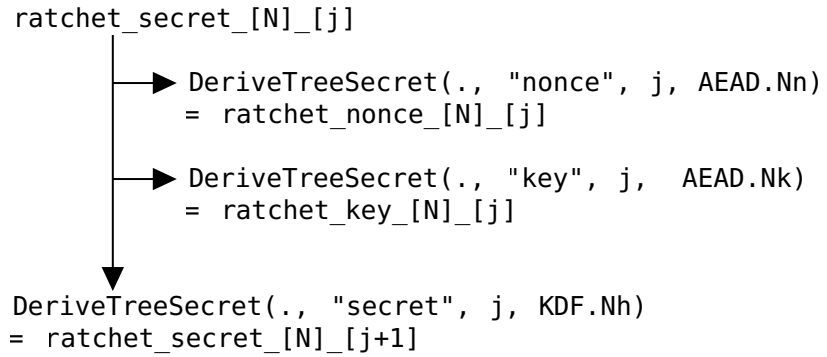
For handshake and application messages, a sequence of keys is derived via a "sender ratchet". Each sender has their own sender ratchet, and each step along the ratchet is called a "generation".

A sender ratchet starts from a per-sender base secret derived from a Secret Tree, as described in Section 10. The base secret initiates a symmetric hash ratchet which generates a sequence of keys and nonces. The sender uses the j-th key/nonce pair in the sequence to encrypt (using the AEAD) the j-th message they send during that epoch. Each key/nonce pair MUST NOT be used to encrypt more than one message.

Keys, nonces, and the secrets in ratchets are derived using DeriveTreeSecret. The context in a given call consists of the current position in the ratchet.

```
DeriveTreeSecret(Secret, Label, Generation, Length) =
    ExpandWithLabel(Secret, Label, Generation, Length)
```

Where Generation is encoded as a uint32.



Here, AEAD.Nn and AEAD.Nk denote the lengths in bytes of the nonce and key for the AEAD scheme defined by the ciphersuite.

10.2. Deletion Schedule

It is important to delete all security-sensitive values as soon as they are *consumed*. A sensitive value S is said to be *consumed* if

- S was used to encrypt or (successfully) decrypt a message, or if
- a key, nonce, or secret derived from S has been consumed. (This goes for values derived via DeriveSecret as well as ExpandWithLabel.)

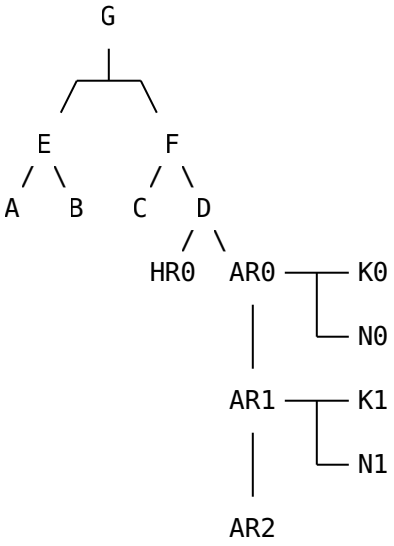
Here, S may be the init_secret, commit_secret, epoch_secret, encryption_secret as well as any secret in a Secret Tree or one of the ratchets.

As soon as a group member consumes a value they MUST immediately delete (all representations of) that value. This is crucial to ensuring forward secrecy for past messages. Members MAY keep unconsumed values around for some reasonable amount of time to handle out-of-order message delivery.

For example, suppose a group member encrypts or (successfully) decrypts an application message using the j-th key and nonce in the ratchet of leaf node L in some epoch n. Then, for that member, at least the following values have been consumed and MUST be deleted:

- the commit_secret, joiner_secret, epoch_secret, encryption_secret of that epoch n as well as the init_secret of the previous epoch n-1,
- all node secrets in the Secret Tree on the path from the root to the leaf with node L,
- the first j secrets in the application data ratchet of node L and
- application_ratchet_nonce_[L]_[j] and application_ratchet_key_[L]_[j].

Concretely, suppose we have the following Secret Tree and ratchet for participant D:



Then if a client uses key K1 and nonce N1 during epoch n then it must consume (at least) values G, F, D, AR0, AR1, K1, N1 as well as the key schedule secrets used to derive G (the encryption_secret), namely init_secret of epoch n-1 and commit_secret, joiner_secret, epoch_secret of epoch n. The client MAY retain (not consume) the values K0 and N0 to allow for out-of-order delivery, and SHOULD retain AR2 for processing future messages.

11. Key Packages

In order to facilitate the asynchronous addition of clients to a group, key packages are pre-published that provide some public information about a user. A KeyPackage object specifies:

1. A protocol version and ciphersuite that the client supports,

2. a public key that others can use to encrypt a Welcome message to this client, (an "init key") and

3. the content of the leaf node that should be added to the tree to represent this client.

KeyPackages are intended to be used only once and SHOULD NOT be reused except in case of last resort. (See Section 17.4). Clients MAY generate and publish multiple KeyPackages to support multiple ciphersuites.

The value for init_key MUST be a public key for the asymmetric encryption scheme defined by cipher_suite, and it MUST be unique among the set of KeyPackages created by this client. Likewise, the leaf_node field MUST be valid for the ciphersuite, including both the encryption_key and signature_key fields. The whole structure is signed using the client's signature key. A KeyPackage object with an invalid signature field MUST be considered malformed.

The signature is computed by the function SignWithLabel with a label KeyPackage and a content comprising of all of the fields except for the signature field.

```
struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey init_key;
    LeafNode leaf_node;
    Extension extensions<V>;
    // SignWithLabel(., "KeyPackageTBS", KeyPackageTBS)
    opaque signature<V>;
} KeyPackage;

struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey init_key;
    LeafNode leaf_node;
    Extension extensions<V>;
} KeyPackageTBS;
```

If a client receives a KeyPackage carried within an MLSMessage object, then it MUST verify that the version field of the KeyPackage has the same value as the version field of the MLSMessage. The version field in the KeyPackage provides an explicit signal of the intended version to the other members of group when they receive the KeyPackage in an Add proposal.

The field leaf_node.capabilities indicates what protocol versions, ciphersuites, credential types, and non-default proposal/extension types are supported by the client. (Proposal and extension types defined in this document are considered "default" and not listed.) This information allows MLS session establishment to be safe from downgrade attacks on the parameters described (as discussed in Section 12), while still only advertising one version / ciphersuite per KeyPackage.

The field leaf_node.leaf_node_source of the LeafNode in a KeyPackage MUST be set to key_package.

Extension included in the extensions or leaf_node.extensions fields MUST be included in the leaf_node.capabilities field.

11.1. KeyPackage Validation

The validity of a KeyPackage needs to be verified at a few stages:

- When a KeyPackage is downloaded by a group member, before it is used to add the client to the group

- When a KeyPackage is received by a group member in an Add message

The client verifies the validity of a KeyPackage using the following steps:

- Verify that the ciphersuite and protocol version of the KeyPackage match those in use in the group.
- Verify the leaf_node field of the KeyPackage according to the process defined in Section 8.3.
- Verify that the signature on the KeyPackage is valid using the public key in leaf_node.credential.
- Verify that the value of leaf_node.encryption_key is different from the value of the init_key field.

11.2. KeyPackage Identifiers

Within MLS, a KeyPackage is identified by its hash (see, e.g., Section 13.2.3.2). The external_key_id extension allows applications to add an explicit, application-defined identifier to a KeyPackage.

```
opaque external_key_id<V>;
```

12. Group Creation

A group is always created with a single member, the "creator". The other members are added when the creator effectively sends itself an Add proposal and commits it, then sends the corresponding Welcome message to the new participants. These processes are described in detail in Section 13.1.1, Section 13.2, and Section 13.2.3.2.

The creator of a group MUST take the following steps to initialize the group:

- Fetch KeyPackages for the members to be added, and select a version and ciphersuite according to the capabilities of the members. To protect against downgrade attacks, the creator MUST use the capabilities information in these KeyPackages to verify that the chosen version and ciphersuite is the best option supported by all members.
- Initialize a one-member group with the following initial values:
  - Ratchet tree: A tree with a single node, a leaf containing an HPKE public key and credential for the creator
  - Group ID: A value set by the creator

- Epoch: 0

- Tree hash: The root hash of the above ratchet tree

- Confirmed transcript hash: The zero-length octet string

- Interim transcript hash: The zero-length octet string

- Init secret: A fresh random value of size KDF.Nh

- Extensions: Any values of the creator's choosing

* For each member, construct an Add proposal from the KeyPackage for that member (see Section 13.1.1)

* Construct a Commit message that commits all of the Add proposals, in any order chosen by the creator (see Section 13.2)

* Process the Commit message to obtain a new group state (for the epoch in which the new members are added) and a Welcome message

* Transmit the Welcome message to the other new members

The recipient of a Welcome message processes it as described in Section 13.2.3.2. If application context informs the recipient that the Welcome should reflect the creation of a new group (for example, due to a branch or reinitialization), then the recipient MUST verify that the epoch value in the GroupInfo is equal to 1.

In principle, the above process could be streamlined by having the creator directly create a tree and choose a random value for first epoch's epoch secret. We follow the steps above because it removes unnecessary choices, by which, for example, bad randomness could be introduced. The only choices the creator makes here are its own KeyPackage and the leaf secret from which the Commit is built.

12.1. Required Capabilities

The configuration of a group imposes certain requirements on clients in the group. At a minimum, all members of the group need to support the ciphersuite and protocol version in use. Additional requirements can be imposed by including a required_capabilities extension in the GroupContext.

```
struct {
    ExtensionType extension_types<V>;
    ProposalType proposal_types<V>;
    CredentialType credential_types<V>;
} RequiredCapabilities;
```

This extension lists the extensions, proposals, and credential types that must be supported by all members of the group. The "default" proposal and extension types defined in this document are assumed to be implemented by all clients, and need not be listed in RequiredCapabilities in order to be safely used.

For new members, support for required capabilities is enforced by existing members during the application of Add commits. Existing members should of course be in compliance already. In order to ensure this continues to be the case even as the group's extensions can be updated, a GroupContextExtensions proposal is invalid if it contains a required_capabilities extension that requires non-default capabilities not supported by all current members.

12.2. Reinitialization

A group may be reinitialized by creating a new group with the same membership and different parameters, and linking it to the old group via a resumption PSK. The members of a group reinitialize it using the following steps:

1. A member of the old group sends a ReInit proposal (see Section 13.1.5)

2. A member of the old group sends a Commit covering the ReInit proposal

3. A member of the old group sends a Welcome message for the new group that matches the ReInit

    * The group_id, version, and cipher_suite fields in the Welcome message MUST be the same as the corresponding fields in the ReInit proposal.

    * The epoch in the Welcome message MUST be 1

    * The Welcome MUST specify a PreSharedKey of type resumption with usage reinit. The group_id must match the old group, and the epoch must indicate the epoch after the Commit covering the ReInit.

    * The psk_nonce included in the PreSharedKeyID of the resumption PSK MUST be a randomly sampled nonce of length KDF.Nh, for the KDF defined by the new group's ciphersuite.

Note that these three steps may be done by the same group member or different members. For example, if a group member sends a commit with an inline ReInit proposal (steps 1 and 2), but then goes offline,

another group member may send the corresponding Welcome. This flexibility avoids situations where a group gets stuck between steps 2 and 3.

Resumption PSKs with usage reinit MUST NOT be used in other contexts. A PreSharedKey proposal with type resumption and usage reinit MUST be considered invalid.

12.3. Sub-group Branching

A new group can be formed from a subset of an existing group's members, using the same parameters as the old group.

A member can create a sub-group by performing the following steps:

1. Determine a subset of existing members that should be a part of the sub-group.

2. Create a new tree for the sub-group by fetching a new KeyPackage for each existing member that should be included in the sub-group.

3. Create a Welcome message that includes a PreSharedKey of type resumption with usage branch.

A client receiving a Welcome including a PreSharedKey of type resumption with usage branch MUST verify that the new group reflects a subgroup branched from the referenced group.

* The version and ciphersuite values in the Welcome MUST be the same as those used by the old group.

* Each LeafNode in a new subgroup MUST match some LeafNode in the original group. In this context, a pair of LeafNodes is said to "match" if the identifiers presented by their respective credentials are considered equivalent by the application.

In addition, to avoid key re-use, the psk_nonce included in the PreSharedKeyID object MUST be a randomly sampled nonce of length KDF.Nh.

Resumption PSKs with usage branch MUST NOT be used in other contexts. A PreSharedKey proposal with type resumption and usage branch MUST be considered invalid.

13. Group Evolution

Over the lifetime of a group, its membership can change, and existing members might want to change their keys in order to achieve post-

compromise security. In MLS, each such change is accomplished by a two-step process:

1. A proposal to make the change is broadcast to the group in a Proposal message

2. A member of the group or a new member broadcasts a Commit message that causes one or more proposed changes to enter into effect

In cases where the Proposal and Commit are sent by the same member, these two steps can be combined by sending the proposals in the commit.

The group thus evolves from one cryptographic state to another each time a Commit message is sent and processed. These states are referred to as "epochs" and are uniquely identified among states of the group by eight-octet epoch values. When a new group is initialized, its initial state epoch is 0x0000000000000000. Each time a state transition occurs, the epoch number is incremented by one.

13.1. Proposals

Proposals are included in an MLSMessageContent by way of a Proposal structure that indicates their type:

```
// See IANA registry for registered values
uint16 ProposalType;

struct {
    ProposalType msg_type;
    select (Proposal.msg_type) {
        case add:                       Add;
        case update:                    Update;
        case remove:                    Remove;
        case psk:                       PreSharedKey;
        case reinit:                    ReInit;
        case external_init:             ExternalInit;
        case app_ack:                   AppAck;
        case group_context_extensions: GroupContextExtensions;
    };
} Proposal;
```

On receiving an MLSMessageContent containing a Proposal, a client MUST verify the signature inside MLSMessageAuth. If the signature verifies successfully, then the Proposal should be cached in such a way that it can be retrieved by hash (as a ProposalOrRef object) in a later Commit message.

13.1.1. Add

An Add proposal requests that a client with a specified KeyPackage be added to the group. The proposer of the Add MUST verify the validity of the KeyPackage, as specified in Section 11.1.

```
struct {
    KeyPackage key_package;
} Add;
```

An Add is applied after being included in a Commit message. The position of the Add in the list of proposals determines the leaf node where the new member will be added. For the first Add in the Commit, the corresponding new member will be placed in the leftmost empty leaf in the tree, for the second Add, the next empty leaf to the right, etc. If no empty leaf exists, the tree is extended to the right.

- Validate the KeyPackage as specified in Section 11.1. The leaf_node_source field in the LeafNode MUST be set to key_package.
- Identify the leaf L for the new member: if there are empty leaves in the tree, L is the leftmost empty leaf. Otherwise, the tree is extended to the right by one leaf node and L is the new leaf.
- For each non-blank intermediate node along the path from the leaf L to the root, add L's leaf index to the unmerged_leaves list for the node.
- Set the leaf node L to a new node containing the LeafNode object carried in the leaf_node field of the KeyPackage in the Add.

13.1.2. Update

An Update proposal is a similar mechanism to Add with the distinction that it replaces the sender's LeafNode in the tree instead of adding a new leaf to the tree.

```
struct {
    LeafNode leaf_node;
} Update;
```

A member of the group applies an Update message by taking the following steps:

- Validate the LeafNode as specified in Section 8.3. The leaf_node_source field MUST be set to update.
- Verify that the encryption_key value in the LeafNode is different from the corresponding field in the LeafNode being replaced.
- Replace the sender's LeafNode with the one contained in the Update proposal
- Blank the intermediate nodes along the path from the sender's leaf to the root

13.1.3. Remove

A Remove proposal requests that the member with LeafNodeRef removed be removed from the group.

```
struct {
    LeafNodeRef removed;
} Remove;
```

A member of the group applies a Remove message by taking the following steps:

- Identify a leaf node matching removed. This lookup MUST be done on the tree before any non-Remove proposals have been applied (the "old" tree in the terminology of Section 13.2), since proposals such as Update can change the LeafNode stored at a leaf. Let L be this leaf node.
- Replace the leaf node L with a blank node
- Blank the intermediate nodes along the path from L to the root
- Truncate the tree by removing leaves from the right side of the tree until the rightmost leaf node is not blank.

13.1.4. PreSharedKey

A PreSharedKey proposal can be used to request that a pre-shared key be injected into the key schedule in the process of advancing the epoch.

```
struct {
    PreSharedKeyID psk;
} PreSharedKey;
```

The psktype of the pre-shared key MUST be external and the psk_nonce MUST be a randomly sampled nonce of length KDF.Nh. When processing a Commit message that includes one or more PreSharedKey proposals, group members derive psk_secret as described in Section 9.4, where the order of the PSKs corresponds to the order of the PreSharedKey proposals in the Commit.

13.1.5. ReInit

A ReInit proposal represents a request to reinitialize the group with different parameters, for example, to increase the version number or to change the ciphersuite. The reinitialization is done by creating a completely new group and shutting down the old one.

```
struct {
    opaque group_id<V>;
    ProtocolVersion version;
    CipherSuite cipher_suite;
    Extension extensions<V>;
} ReInit;
```

A member of the group applies a ReInit proposal by waiting for the committer to send the Welcome message that matches the ReInit, according to the criteria in Section 12.2.

If a ReInit proposal is included in a Commit, it MUST be the only proposal referenced by the Commit. If other non-ReInit proposals have been sent during the epoch, the committer SHOULD prefer them over the ReInit proposal, allowing the ReInit to be resent and applied in a subsequent epoch. The version field in the ReInit proposal MUST be no less than the version for the current group.

13.1.6. ExternalInit

An ExternalInit proposal is used by new members that want to join a group by using an external commit. This proposal can only be used in that context.

```
struct {
  opaque kem_output<V>;
} ExternalInit;
```

A member of the group applies an ExternalInit message by initializing the next epoch using an init secret computed as described in Section 9.3. The kem_output field contains the required KEM output.

13.1.7. AppAck

An AppAck proposal is used to acknowledge receipt of application messages. Though this information implies no change to the group, it is structured as a Proposal message so that it is included in the group's transcript by being included in Commit messages.

```
struct {
    LeafNodeRef sender;
    uint32 first_generation;
    uint32 last_generation;
} MessageRange;

struct {
    MessageRange received_ranges<V>;
} AppAck;
```

An AppAck proposal represents a set of messages received by the sender in the current epoch. Messages are represented by the sender and generation values in the MLSCiphertext for the message. Each MessageRange represents receipt of a span of messages whose generation values form a continuous range from first_generation to last_generation, inclusive.

AppAck proposals are sent as a guard against the Delivery Service dropping application messages. The sequential nature of the generation field provides a degree of loss detection, since gaps in the generation sequence indicate dropped messages. AppAck completes this story by addressing the scenario where the Delivery Service drops all messages after a certain point, so that a later generation is never observed. Obviously, there is a risk that AppAck messages could be suppressed as well, but their inclusion in the transcript means that if they are suppressed then the group cannot advance at all.

The schedule on which sending AppAck proposals are sent is up to the application, and determines which cases of loss/suppression are detected. For example:

- The application might have the committer include an AppAck proposal whenever a Commit is sent, so that other members could know when one of their messages did not reach the committer.

* The application could have a client send an AppAck whenever an application message is sent, covering all messages received since its last AppAck. This would provide a complete view of any losses experienced by active members.

* The application could simply have clients send AppAck proposals on a timer, so that all participants' state would be known.

An application using AppAck proposals to guard against loss/suppression of application messages also needs to ensure that AppAck messages and the Commits that reference them are not dropped. One way to do this is to always encrypt Proposal and Commit messages, to make it more difficult for the Delivery Service to recognize which messages contain AppAcks. The application can also have clients enforce an AppAck schedule, reporting loss if an AppAck is not received at the expected time.

13.1.8. GroupContextExtensions

A GroupContextExtensions proposal is used to update the list of extensions in the GroupContext for the group.

```
struct { Extension extensions<V>; } GroupContextExtensions;
```

A member of the group applies a GroupContextExtensions proposal with the following steps:

* If the new extensions include a required_capabilities extension, verify that all members of the group support the required capabilities (including those added in the same commit, and excluding those removed).

* Remove all of the existing extensions from the GroupContext object for the group and replacing them with the list of extensions in the proposal. (This is a wholesale replacement, not a merge. An extension is only carried over if the sender of the proposal includes it in the new list.)

Note that once the GroupContext is updated, its inclusion in the confirmation_tag by way of the key schedule will confirm that all members of the group agree on the extensions in use.

13.1.9. External Proposals

Add and Remove proposals can be constructed and sent to the group by a party that is outside the group. For example, a Delivery Service might propose to remove a member of a group who has been inactive for a long time, or propose adding a newly-hired staff member to a group representing a real-world team. Proposals originating outside the

group are identified by a external or new_member SenderType in MLSPlaintext.

ReInit proposals can also be sent to the group by a external sender, for example to enforce a changed policy regarding MLS version or ciphersuite.

The new_member SenderType is used for clients proposing that they themselves be added. For this ID type the sender value MUST be zero and the Proposal type MUST be Add. The MLSPlaintext MUST be signed with the private key corresponding to the KeyPackage in the Add message. Recipients MUST verify that the MLSPlaintext carrying the Proposal message is validly signed with this key.

The external SenderType is reserved for signers that are pre-provisioned to the clients within a group. It can only be used if the external_senders extension is present in the group's group context extensions.

An external proposal MUST be sent as an MLSPlaintext object, since the sender will not have the keys necessary to construct an MLSCiphertext object.

13.1.9.1. External Senders Extension

The external_senders extension is a group context extension that contains credentials of senders that are permitted to send external proposals to the group.

```
Credential external_senders<V>;
```

13.2. Commit

A Commit message initiates a new epoch for the group, based on a collection of Proposals. It instructs group members to update their representation of the state of the group by applying the proposals and advancing the key schedule.

Each proposal covered by the Commit is included by a ProposalOrRef value, which identifies the proposal to be applied by value or by reference. Commits that refer to new Proposals from the committer can be included by value. Commits for previously sent proposals from anyone (including the committer) can be sent by reference. Proposals sent by reference are specified by including the hash of the MLSPlaintext in which the proposal was sent (see Section 6.2).

```
enum {
  reserved(0),
  proposal(1)
  reference(2),
  (255)
} ProposalOrRefType;

struct {
  ProposalOrRefType type;
  select (ProposalOrRef.type) {
    case proposal:  Proposal proposal;
    case reference: ProposalRef reference;
  }
} ProposalOrRef;

struct {
    ProposalOrRef proposals<V>;
    optional<UpdatePath> path;
} Commit;
```

A group member that has observed one or more valid proposals within an epoch MUST send a Commit message before sending application data. This ensures, for example, that any members whose removal was proposed during the epoch are actually removed before any application data is transmitted.

The sender of a Commit MUST include all valid proposals that it has received during the current epoch. Invalid proposals include, for example, proposals with an invalid signature or proposals that are semantically invalid, such as an Add when the sender does not have the application-level permission to add new users. Proposals with a non-default proposal type MUST NOT be included in a commit unless the proposal type is supported by all the members of the group that will process the Commit (i.e., not including any members being added or removed by the Commit).

If there are multiple proposals that apply to the same leaf, or multiple PreSharedKey proposals that reference the same PreSharedKeyID, the committer MUST choose one and include only that one in the Commit, considering the rest invalid. The committer MUST NOT include any Update proposals generated by the committer, since they would be duplicative with the path field in the Commit. The committer MUST prefer any Remove received, or the most recent Update for the leaf if there are no Removes. If there are multiple Add proposals containing KeyPackages that the committer considers to represent the same client or a client already in the group (for example, identical KeyPackages or KeyPackages sharing the same

Credential), the committer again chooses one to include and considers the rest invalid. The committer MUST consider invalid any Add or Update proposal if the Credential in the contained KeyPackage shares the same signature key with a Credential in any leaf of the group, or if the LeafNode in the KeyPackage shares the same encryption_key with another LeafNode in the group.

The Commit MUST NOT combine proposals sent within different epochs. Due to the asynchronous nature of proposals, receivers of a Commit SHOULD NOT enforce that all valid proposals sent within the current epoch are referenced by the next Commit. In the event that a valid proposal is omitted from the next Commit, and that proposal is still valid in the current epoch, the sender of the proposal MAY resend it after updating it to reflect the current epoch.

A member of the group MAY send a Commit that references no proposals at all, which would thus have an empty proposals vector. Such a Commit resets the sender's leaf and the nodes along its direct path, and provides forward secrecy and post-compromise security with regard to the sender of the Commit. An Update proposal can be regarded as a "lazy" version of this operation, where only the leaf changes and intermediate nodes are blanked out.

By default, the path field of a Commit MUST be populated. The path field MAY be omitted if (a) it covers at least one proposal and (b) none of the proposals covered by the Commit are of "path required" types. A proposal type requires a path if it cannot change the group membership in a way that requires the forward secrecy and post-compromise security guarantees that an UpdatePath provides. The only proposal types defined in this document that do not require a path are:

*add

*psk

*app_ack

*reinit

New proposal types MUST state whether they require a path. If any instance of a proposal type requires a path, then the proposal type requires a path. This attribute of a proposal type is reflected in the "Path Required" field of the proposal type registry defined in Section 18.3.

Update and Remove proposals are the clearest examples of proposals that require a path. An UpdatePath is required to evict the removed member or the old appearance of the updated member.

In pseudocode, the logic for validating the path field of a Commit is as follows:

```
pathRequiredTypes = [
    update,
    remove,
    external_init,
    group_context_extensions
]

pathRequired = false

for i, id in commit.proposals:
    proposal = proposalCache[id]
    assert(proposal != null)

    pathRequired = pathRequired ||
                   (proposal.msg_type in pathRequiredTypes)

if len(commit.proposals) == 0 || pathRequired:
    assert(commit.path != null)
```

To summarize, a Commit can have three different configurations, with different uses:

1. An "empty" Commit that references no proposals, which updates the committer's contribution to the group and provides PCS with regard to the committer.

2. A "partial" Commit that references proposals that do not require a path, and where the path is empty. Such a commit doesn't provide PCS with regard to the committer.

3. A "full" Commit that references proposals of any type, which provides FS with regard to any removed members and PCS for the committer and any updated members.

13.2.1. Creating a Commit

When creating or processing a Commit, three different ratchet trees and their associated GroupContexts are used:

1. "Old" refers to the ratchet tree and GroupContext for the epoch before the commit. The old GroupContext is used when signing the MLSPlainText so that existing group members can verify the signature before processing the commit.

2. "Provisional" refers to the ratchet tree and GroupContext constructed after applying the proposals that are referenced by the Commit. The provisional GroupContext uses the epoch number for the new epoch, and the old confirmed transcript hash. This is used when creating the UpdatePath, if the UpdatePath is needed.

3. "New" refers to the ratchet tree and GroupContext constructed after applying the proposals and the UpdatePath (if any). The new GroupContext uses the epoch number for the new epoch, and the new confirmed transcript hash. This is used when deriving the new epoch secrets, and is the only GroupContext that newly-added members will have.

A member of the group creates a Commit message and the corresponding Welcome message at the same time, by taking the following steps:

* Construct an initial Commit object with the proposals field populated from Proposals received during the current epoch, and an empty path field.

* Generate the provisional ratchet tree and GroupContext by applying the proposals referenced in the initial Commit object, as described in Section 13.1. Update proposals are applied first, followed by Remove proposals, and then finally Add proposals. Add proposals are applied in the order listed in the proposals vector, and always to the leftmost unoccupied leaf in the tree, or the right edge of the tree if all leaves are occupied.

  - Note that the order in which different types of proposals are applied should be updated by the implementation to include any new proposals added by negotiated group extensions.

  - PreSharedKey proposals are processed later when deriving the psk_secret for the Key Schedule.

* Decide whether to populate the path field: If the path field is required based on the proposals that are in the commit (see above), then it MUST be populated. Otherwise, the sender MAY omit the path field at its discretion.

* If populating the path field: Create an UpdatePath using the provisional ratchet tree and GroupContext. Any new member (from an add proposal) MUST be excluded from the resolution during the computation of the UpdatePath. The leaf_node for this UpdatePath MUST have leaf_node_source set to commit. Note that the LeafNode in the UpdatePath effectively updates an existing LeafNode in the

group and thus MUST adhere to the same restrictions as LeafNodes used in Update proposals (aside from leaf_node_source).

- Assign this UpdatePath to the path field in the Commit.

- Apply the UpdatePath to the tree, as described in Section 8.6, creating the new ratchet tree. Define commit_secret as the value path_secret[n+1] derived from the path_secret[n] value assigned to the root node.

* If not populating the path field: Set the path field in the Commit to the null optional. Define commit_secret as the all-zero vector of length KDF.Nh (the same length as a path_secret value would be). In this case, the new ratchet tree is the same as the provisional ratchet tree.

* Derive the psk_secret as specified in Section 9.4, where the order of PSKs in the derivation corresponds to the order of PreSharedKey proposals in the proposals vector.

* Construct an MLSMessageContent object containing the Commit object. Sign the MLSMessageContent using the old GroupContext as context.

  - Use the MLSMessageContent to update the confirmed transcript hash and generate the new GroupContext.

  - Use the init_secret from the previous epoch, the commit_secret and the psk_secret as defined in the previous steps, and the new GroupContext to compute the new joiner_secret, welcome_secret, epoch_secret, and derived secrets for the new epoch.

  - Use the confirmation_key for the new epoch to compute the confirmation_tag value, and the membership_key for the old epoch to compute the membership_tag value in the MLSPlaintext.

  - Calculate the interim transcript hash using the new confirmed transcript hash and the confirmation_tag from the MLSMessageAuth.

* Construct a GroupInfo reflecting the new state:

  - Group ID, epoch, tree, confirmed transcript hash, interim transcript hash, and group context extensions from the new state

  - The confirmation_tag from the MLSMessageAuth object

  - Other extensions as defined by the application

- Optionally derive an external keypair as described in Section 9 (required for External Commits, see Section 13.2.3.1)

- Sign the GroupInfo using the member's private signing key

- Encrypt the GroupInfo using the key and nonce derived from the joiner_secret for the new epoch (see Section 13.2.3.2)

* For each new member in the group:

  - Identify the lowest common ancestor in the tree of the new member's leaf node and the member sending the Commit

  - If the path field was populated above: Compute the path secret corresponding to the common ancestor node

  - Compute an EncryptedGroupSecrets object that encapsulates the init_secret for the current epoch and the path secret (if present).

* Construct a Welcome message from the encrypted GroupInfo object, the encrypted key packages, and any PSKs for which a proposal was included in the Commit. The order of the psks MUST be the same as the order of PreSharedKey proposals in the proposals vector.

* If a ReInit proposal was part of the Commit, the committer MUST create a new group with the parameters specified in the ReInit proposal, and with the same members as the original group. The Welcome message MUST include a PreSharedKeyID with the following parameters:

  - psktype: resumption

  - usage: reinit

  - group_id: The group ID for the current group

  - epoch: The epoch that the group will be in after this Commit

13.2.2. Processing a Commit

A member of the group applies a Commit message by taking the following steps:

* Verify that the epoch field of the enclosing MLSMessageContent is equal to the epoch field of the current GroupContext object

* Verify that the signature on the MLSMessageContent message as described in Section Section 7.1.

* Verify that all PreSharedKey proposals in the proposals vector have unique PreSharedKeyIDs and are available.

* Generate the provisional ratchet tree and GroupContext by applying the proposals referenced in the initial Commit object, as described in Section 13.1. Update proposals are applied first, followed by Remove proposals, and then finally Add proposals. Add proposals are applied in the order listed in the proposals vector, and always to the leftmost unoccupied leaf in the tree, or the right edge of the tree if all leaves are occupied.

  - Note that the order in which different types of proposals are applied should be updated by the implementation to include any new proposals added by negotiated group extensions.

* Verify that the path value is populated if the proposals vector contains any Update or Remove proposals, or if it's empty. Otherwise, the path value MAY be omitted.

* If the path value is populated: Process the path value using the provisional ratchet tree and GroupContext, to generate the new ratchet tree and the commit_secret:

  - Validate the LeafNode as specified in Section 8.3. The leaf_node_source field MUST be set to commit.

  - Verify that the encryption_key value in the LeafNode is different from the committer's current leaf node.

  - Apply the UpdatePath to the tree, as described in Section 8.6, and store leaf_node at the committer's leaf.

  - Verify that the LeafNode has a parent_hash field and that its value matches the new parent of the sender's leaf node.

  - Define commit_secret as the value path_secret[n+1] derived from the path_secret[n] value assigned to the root node.

* If the path value is not populated: Define commit_secret as the all-zero vector of length KDF.Nh (the same length as a path_secret value would be).

* Update the confirmed and interim transcript hashes using the new Commit, and generate the new GroupContext.

* Derive the psk_secret as specified in Section 9.4, where the order of PSKs in the derivation corresponds to the order of PreSharedKey proposals in the proposals vector.

*Use the init_secret from the previous epoch, the commit_secret and the psk_secret as defined in the previous steps, and the new GroupContext to compute the new joiner_secret, welcome_secret, epoch_secret, and derived secrets for the new epoch.

*Use the confirmation_key for the new epoch to compute the confirmation tag for this message, as described below, and verify that it is the same as the confirmation_tag field in the MLSMessageAuth object.

*If the above checks are successful, consider the new GroupContext object as the current state of the group.

*If the Commit included a ReInit proposal, the client MUST NOT use the group to send messages anymore. Instead, it MUST wait for a Welcome message from the committer meeting the requirements of Section 12.2.

13.2.3. Adding Members to the Group

New members can join the group in two ways. Either by being added by a group member, or by adding themselves through an external Commit. In both cases, the new members need information to bootstrap their local group state.

```
struct {
    CipherSuite cipher_suite;
    opaque group_id<V>;
    uint64 epoch;
    opaque tree_hash<V>;
    opaque confirmed_transcript_hash<V>;
    Extension group_context_extensions<V>;
    Extension other_extensions<V>;
    MAC confirmation_tag;
    LeafNodeRef signer;
    // SignWithLabel(., "GroupInfoTBS", GroupInfoTBS)
    opaque signature<V>;
} GroupInfo;
```

New members MUST verify the signature using the public key taken from the credential in the leaf node of the member with LeafNodeRef signer. The signature covers the following structure, comprising all the fields in the GroupInfo above signature:

```
struct {
    CipherSuite cipher_suite;
    opaque group_id<V>;
    uint64 epoch;
    opaque tree_hash<V>;
    opaque confirmed_transcript_hash<V>;
    Extension group_context_extensions<V>;
    Extension other_extensions<V>;
    MAC confirmation_tag;
    LeafNodeRef signer;
} GroupInfoTBS;
```

13.2.3.1. Joining via External Commits

External Commits are a mechanism for new members (external parties that want to become members of the group) to add themselves to a group, without requiring that an existing member has to come online to issue a Commit that references an Add Proposal.

Whether existing members of the group will accept or reject an External Commit follows the same rules that are applied to other handshake messages.

New members can create and issue an External Commit if they have access to the following information for the group's current epoch:

- group ID
- epoch ID
- ciphersuite
- public tree hash
- confirmed transcript hash
- confirmation tag of the most recent Commit
- group extensions
- external public key

In other words, to join a group via an External Commit, a new member needs a GroupInfo with an ExternalPub extension present in the other_extensions.

```
struct {
    HPKEPublicKey external_pub;
} ExternalPub;
```

Thus, a member of the group can enable new clients to join by making a GroupInfo object available to them. Note that because a GroupInfo object is specific to an epoch, it will need to be updated as the group advances. In particular, each GroupInfo object can be used for one external join, since that external join will cause the epoch to change.

Note that the tree_hash field is used the same way as in the Welcome message. The full tree can be included via the ratchet_tree extension Section 13.3.

The information in a GroupInfo is not generally public information, but applications can choose to make it available to new members in order to allow External Commits.

In principle, External Commits work like regular Commits. However, their content has to meet a specific set of requirements:

* External Commits MUST contain a path field (and is therefore a "full" Commit). The joiner is added at the leftmost free leaf node (just as if they were added with an Add proposal), and the path is calculated relative to that leaf node.

* The Commit MUST NOT include any proposals by reference, since an external joiner cannot determine the validity of proposals sent within the group

* The proposals included by value in an External Commit MUST meet the following conditions:

  - There MUST be a single ExternalInit proposal.

  - There MAY be a single Remove proposal, where the LeafNode in the path field MUST meet the same criteria as the LeafNode in an Update for the removed leaf (see Section 13.1.2). In particular, the credential in the LeafNode MUST present a set of identifiers that is acceptable to the application for the removed participant.

  - There MAY be one or more PreSharedKey proposals.

  - There MUST NOT be any other proposals.

* External Commits MUST be signed by the new member. In particular, the signature on the enclosing MLSPlaintext MUST verify using the public key for the credential in the leaf_node of the path field.

* When processing a Commit, both existing and new members MUST use the external init secret as described in Section 9.3.

* The sender type for the MLSPlaintext encapsulating the External Commit MUST be new_member

External Commits come in two "flavors" -- a "join" commit that adds the sender to the group or a "resync" commit that replaces a member's prior appearance with a new one.

Note that the "resync" operation allows an attacker that has compromised a member's signature private key to introduce themselves into the group and remove the prior, legitimate member in a single Commit. Without resync, this can still be done, but requires two operations, the external Commit to join and a second Commit to remove the old appearance. Applications for whom this distinction is salient can choose to disallow external commits that contain a Remove, or to allow such resync commits only if they contain a "reinit" PSK proposal that demonstrates the joining member's presence in a prior epoch of the group. With the latter approach, the attacker would need to compromise the PSK as well as the signing key, but the application will need to ensure that continuing, non-resynchronizing members have the required PSK.

13.2.3.2. Joining via Welcome Message

The sender of a Commit message is responsible for sending a single Welcome message to all the new members added via Add proposals. The Welcome message provides the new members with the current state of the group, after the application of the Commit message. The new members will not be able to decrypt or verify the Commit message, but will have the secrets they need to participate in the epoch initiated by the Commit message.

In order to allow the same Welcome message to be sent to all new members, information describing the group is encrypted with a symmetric key and nonce derived from the joiner_secret for the new epoch. The joiner_secret is then encrypted to each new member using HPKE. In the same encrypted package, the committer transmits the path secret for the lowest (closest to the leaf) node which is contained in the direct paths of both the committer and the new member. This allows the new member to compute private keys for nodes in its direct path that are being reset by the corresponding Commit.

If the sender of the Welcome message wants the receiving member to include a PSK in the derivation of the epoch_secret, they can populate the psks field indicating which PSK to use.

```
struct {
  opaque path_secret<V>;
} PathSecret;

struct {
  opaque joiner_secret<V>;
  optional<PathSecret> path_secret;
  PreSharedKeys psks;
} GroupSecrets;

struct {
  KeyPackageRef new_member;
  HPKECiphertext encrypted_group_secrets;
} EncryptedGroupSecrets;

struct {
  CipherSuite cipher_suite;
  EncryptedGroupSecrets secrets<V>;
  opaque encrypted_group_info<V>;
} Welcome;
```

The client processing a Welcome message will need to have a copy of the group's ratchet tree. The tree can be provided in the Welcome message, in an extension of type ratchet_tree. If it is sent otherwise (e.g., provided by a caching service on the Delivery Service), then the client MUST download the tree before processing the Welcome.

On receiving a Welcome message, a client processes it using the following steps:

* Identify an entry in the secrets array where the new_member value corresponds to one of this client's KeyPackages, using the hash indicated by the cipher_suite field. If no such field exists, or if the ciphersuite indicated in the KeyPackage does not match the one in the Welcome message, return an error.

* Decrypt the encrypted_group_secrets using HPKE with the algorithms indicated by the ciphersuite and the HPKE private key corresponding to the GroupSecrets. If a PreSharedKeyID is part of the GroupSecrets and the client is not in possession of the corresponding PSK, return an error.

* From the joiner_secret in the decrypted GroupSecrets object and the PSKs specified in the GroupSecrets, derive the welcome_secret and using that the welcome_key and welcome_nonce. Use the key and nonce to decrypt the encrypted_group_info field.

```
welcome_nonce = KDF.Expand(welcome_secret, "nonce", AEAD.Nn)
welcome_key = KDF.Expand(welcome_secret, "key", AEAD.Nk)
```

* Verify the signature on the GroupInfo object. The signature input comprises all of the fields in the GroupInfo object except the signature field. The public key and algorithm are taken from the credential in the leaf node of the member with LeafNodeRef signer. If there is no matching leaf node, or if signature verification fails, return an error.

* Verify the integrity of the ratchet tree.

  - Verify that the tree hash of the ratchet tree matches the tree_hash field in the GroupInfo.

  - For each non-empty parent node, verify that exactly one of the node's children are non-empty and have the hash of this node set as their parent_hash value (if the child is another parent) or has a parent_hash field in the LeafNode containing the same value (if the child is a leaf). If either of the node's children is empty, and in particular does not have a parent hash, then its respective children's parent_hash values have to be considered instead.

  - For each non-empty leaf node, validate the LeafNode as described in Section 8.3.

* Identify a leaf in the tree array (any even-numbered node) whose LeafNode is identical to the one in the KeyPackage. If no such field exists, return an error. Let my_leaf represent this leaf in the tree.

* Construct a new group state using the information in the GroupInfo object.

  - The GroupContext contains the group_id, epoch, tree_hash, confirmed_transcript_hash, and group_context_extensions fields from the GroupInfo object.

  - The new member's position in the tree is at the leaf my_leaf, as defined above.

- Update the leaf my_leaf with the private key corresponding to the public key in the node.

- If the path_secret value is set in the GroupSecrets object: Identify the lowest common ancestor of the leaf node my_leaf and of the node of the member with LeafNodeRef GroupInfo.signer. Set the private key for this node to the private key derived from the path_secret.

- For each parent of the common ancestor, up to the root of the tree, derive a new path secret and set the private key for the node to the private key derived from the path secret. The private key MUST be the private key that corresponds to the public key in the node.

* Use the joiner_secret from the GroupSecrets object to generate the epoch secret and other derived secrets for the current epoch.

* Set the confirmed transcript hash in the new state to the value of the confirmed_transcript_hash in the GroupInfo.

* Verify the confirmation tag in the GroupInfo using the derived confirmation key and the confirmed_transcript_hash from the GroupInfo.

* Use the confirmed transcript hash and confirmation tag to compute the interim transcript hash in the new state.

13.3. Ratchet Tree Extension

By default, a GroupInfo message only provides the joiner with a commitment to the group's ratchet tree. In order to process or generate handshake messages, the joiner will need to get a copy of the ratchet tree from some other source. (For example, the DS might provide a cached copy.) The inclusion of the tree hash in the GroupInfo message means that the source of the ratchet tree need not be trusted to maintain the integrity of tree.

In cases where the application does not wish to provide such an external source, the whole public state of the ratchet tree can be provided in an extension of type ratchet_tree, containing a ratchet_tree object of the following form:

```
enum {
    reserved(0),
    leaf(1),
    parent(2),
    (255)
} NodeType;

struct {
    NodeType node_type;
    select (Node.node_type) {
        case leaf:   LeafNode leaf_node;
        case parent: ParentNode parent_node;
    };
} Node;

optional<Node> ratchet_tree<V>;
```

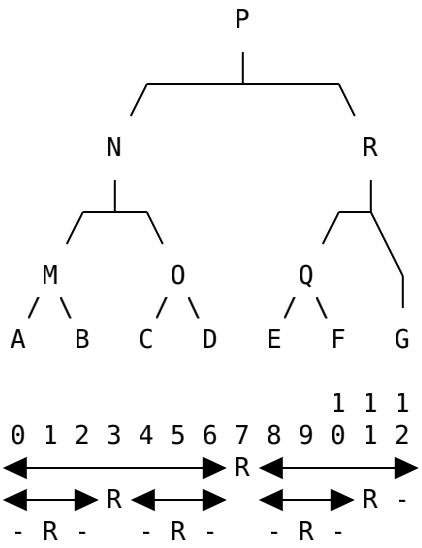
The nodes are listed in the order specified by a left-to-right in-order traversal of the rachet tree. Each node is listed between its left subtree and its right subtree. (This is the same ordering as specified for the array-based trees outlined in Appendix B.)

The leaves of the tree are stored in even-numbered entries in the array (the leaf with index L in array position 2*L). The root node of the tree is at position 2^k - 1 of the array, where k is the largest number such that 2^k is smaller than the length of the array. Intermediate parent nodes can be identified by performing the same calculation to the subarrays to the left and right of the root, following something like the following algorithm:

```
# Assuming a class Node that has left and right members
def subtree_root(nodes):
    # If there is only one node in the array return it
    if len(nodes) == 1:
        return Node(nodes[0])

    # Otherwise, the length of the array MUST be odd
    if len(nodes) % 2 == 0:
        raise Exception("Malformed node array {}", len(nodes))

    # Identify the root of the subtree
    k = 0
    while (2**(k+1)) < len(nodes):
       k += 1
    R = 2**k - 1
    root = Node(nodes[R])
    root.left = subtree_root(nodes[:R])
    root.right = subtree_root(nodes[(R+1):])
    return root
```

(Note that this is the same ordering of nodes as in the array-based tree representation described in Appendix B. The algorithms in that section may be used to simplify decoding this extension into other representations.)

The example tree in Section 5.1 would be represented as an array of nodes in the following form, where R represents the "subtree root" for a given subarray of the node array:



The presence of a ratchet_tree extension in a GroupInfo message does not result in any changes to the GroupContext extensions for the group. The ratchet tree provided is simply stored by the client and used for MLS operations.

If this extension is not provided in a Welcome message, then the client will need to fetch the ratchet tree over some other channel before it can generate or process Commit messages. Applications should ensure that this out-of-band channel is provided with security protections equivalent to the protections that are afforded to Proposal and Commit messages. For example, an application that encrypts Proposal and Commit messages might distribute ratchet trees encrypted using a key exchanged over the MLS channel.

Regardless of how the client obtains the tree, the client MUST verify that the root hash of the ratchet tree matches the tree_hash of the GroupContext before using the tree for MLS operations.

14. Extensibility

The base MLS protocol can be extended in a few ways. New ciphersuites can be added to enable the use of new cryptographic algorithms. New types of proposals can be used to perform new actions within an epoch. Extension fields can be used to add additional information to the protocol. In this section, we discuss some constraints on these extensibility mechanisms that are necessary to ensure broad interoperability.

14.1. Ciphersuites

As discussed in Section 6.1, MLS allows the participants in a group to negotiate the cryptographic algorithms used within the group. This extensibility is important for maintaining the security of the protocol over time [RFC7696]. It also creates a risk of interoperability failure due to clients not supporting a common ciphersuite.

The ciphersuite registry defined in Section 18.1 attempts to strike a balance on this point. On the one hand, the base policy for the registry is Specification Required, a fairly low bar designed to avoid the need for standards work in cases where different ciphers are needed for niche applications. There is a higher bar (Standards Action) for ciphers to set the Recommended field in the registry. This higher bar is there in part to ensure that the interoperability implications of new ciphersuites are considered.

MLS ciphersuites are defined independent of MLS versions, so that in principle the same ciphersuite can be used across versions. Standards work defining new versions of MLS should consider whether it is desirable for the new version to be compatible with existing

ciphersuites, or whether the new version should rule out some ciphersuites. For example, a new version could follow the example of HTTP/2, which restricted the set of allowed TLS ciphers (see Section 9.2.2 of [RFC7540].

14.2. Proposals

Commit messages do not have an extension field because the set of protocols is extensible. As discussed in Section 13.2, Proposals with a non-default proposal type MUST NOT be included in a commit unless the proposal type is supported by all the members of the group that will process the Commit.

14.3. Credential Extensibility

In order to ensure that MLS provides meaningful authentication it is important that each member is able to authenticate some identity information for each other member. Identity information is encoded in Credentials, so this property is assured by ensuring that members use compatible credential types.

The types of credential that may be used in a group is restricted to what all members of the group support, as specified by the capabilities field of each LeafNode in the RatchetTree. An application can introduce new credential types by choosing an unallocated identifier from the registry in Section 18.4 and indicating support for the credential type in published LeafNodes, whether in Update proposals to existing groups or KeyPackages that are added to new groups. Once all members in a group indicate support for the credential type, members can start using LeafNodes with the new credential. Application may enforce that certain credential types always remain supported by adding a required_capabilities extension to the group's GroupContext, which would prevent any member from being added to the group that doesn't support them.

In future extensions to MLS, it may be useful to allow a member to present more than one credential. For example, such credentials might present different attributes attested by different authorities. To be consistent with the general principle stated at the beginning of this section, such an extension would need to ensure that each member can authenticate some identity for each other member. For each pair of members (Alice, Bob), Alice would need to present at least one credential of a type that Bob supports. ## Extensions

This protocol includes a mechanism for negotiating extension parameters similar to the one in TLS [RFC8446]. In TLS, extension negotiation is one-to-one: The client offers extensions in its ClientHello message, and the server expresses its choices for the

session with extensions in its ServerHello and EncryptedExtensions messages. In MLS, extensions appear in the following places:

- In KeyPackages, to describe additional information related to the client
- In LeafNodes, to describe additional information about the client or its participation in the group (once in the ratchet tree)
- In the GroupInfo, to tell new members of a group what parameters are being used by the group, and to provide any additional details required to join the group
- In the GroupContext object, to ensure that all members of the group have the same view of the parameters in use

In other words, an application can use GroupContext extensions to ensure that all members of the group agree on a set of parameters. Clients indicate their support for parameters in the capabilities field of their LeafNode. New members of a group are informed of the group's GroupContext extensions via the group_context_extensions field in the GroupInfo object. The other_extensions field in a GroupInfo object can be used to provide additional parameters to new joiners that are used to join the group.

This extension mechanism is designed to allow for secure and forward-compatible negotiation of extensions. For this to work, implementations MUST correctly handle extensible fields:

- A client that posts a KeyPackage MUST support all parameters advertised in it. Otherwise, another client might fail to interoperate by selecting one of those parameters.
- A client initiating a group MUST ignore all unrecognized ciphersuites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients.
- A client adding a new member to a group MUST verify that the LeafNode for the new member is compatible with the group's extensions. The capabilities field MUST indicate support for each extension in the GroupContext.
- If any extension in a GroupInfo message is unrecognized (i.e., not contained in the capabilities of the corresponding KeyPackage), then the client MUST reject the Welcome message and not join the group.
- The extensions populated into a GroupContext object are drawn from those in the GroupInfo object, according to the definitions of those extensions.

- *Any field containing a list of extensions MUST NOT have more than one extension of any given type.

Note that the latter two requirements mean that all MLS extensions are mandatory, in the sense that an extension in use by the group MUST be supported by all members of the group.

This document does not define any way for the parameters of the group to change once it has been created; such a behavior could be implemented as an extension.

15. Sequencing of State Changes

Each Commit message is premised on a given starting state, indicated by the epoch field of the enclosing MLSMessageContent. If the changes implied by a Commit messages are made starting from a different state, the results will be incorrect.

This need for sequencing is not a problem as long as each time a group member sends a Commit message, it is based on the most current state of the group. In practice, however, there is a risk that two members will generate Commit messages simultaneously, based on the same state.

When this happens, there is a need for the members of the group to deconflict the simultaneous Commit messages. There are two general approaches:

- *Have the Delivery Service enforce a total order

- *Have a signal in the message that clients can use to break ties

As long as Commit messages cannot be merged, there is a risk of starvation. In a sufficiently busy group, a given member may never be able to send a Commit message, because he always loses to other members. The degree to which this is a practical problem will depend on the dynamics of the application.

It might be possible, because of the non-contributivity of intermediate nodes, that Commit messages could be applied one after the other without the Delivery Service having to reject any Commit message, which would make MLS more resilient regarding the concurrency of Commit messages. The Messaging system can decide to choose the order for applying the state changes. Note that there are certain cases (if no total ordering is applied by the Delivery Service) where the ordering is important for security, ie. all updates must be executed before removes.

Regardless of how messages are kept in sequence, implementations MUST only update their cryptographic state when valid Commit messages are

received. Generation of Commit messages MUST NOT modify a client's state, since the endpoint doesn't know at that time whether the changes implied by the Commit message will succeed or not.

15.1. Server-Enforced Ordering

With this approach, the Delivery Service ensures that incoming messages are added to an ordered queue and outgoing messages are dispatched in the same order. The server is trusted to break ties when two members send a Commit message at the same time.

Messages should have a counter field sent in clear-text that can be checked by the server and used for tie-breaking. The counter starts at 0 and is incremented for every new incoming message. If two group members send a message with the same counter, the first message to arrive will be accepted by the server and the second one will be rejected. The rejected message needs to be sent again with the correct counter number.

To prevent counter manipulation by the server, the counter's integrity can be ensured by including the counter in a signed message envelope.

This applies to all messages, not only state changing messages.

15.2. Client-Enforced Ordering

Order enforcement can be implemented on the client as well, one way to achieve it is to use a two step update protocol: the first client sends a proposal to update and the proposal is accepted when it gets 50%+ approval from the rest of the group, then it sends the approved update. Clients which didn't get their proposal accepted, will wait for the winner to send their update before retrying new proposals.

While this seems safer as it doesn't rely on the server, it is more complex and harder to implement. It also could cause starvation for some clients if they keep failing to get their proposal accepted.

16. Application Messages

The primary purpose of the Handshake protocol is to provide an authenticated group key exchange to clients. In order to protect Application messages sent among the members of a group, the Application secret provided by the Handshake key schedule is used to derive nonces and encryption keys for the Message Protection Layer according to the Application Key Schedule. That is, each epoch is equipped with a fresh Application Key Schedule which consist of a tree of Application Secrets as well as one symmetric ratchet per group member.

Each client maintains their own local copy of the Application Key Schedule for each epoch during which they are a group member. They derive new keys, nonces and secrets as needed while deleting old ones as soon as they have been used.

Application messages MUST be protected with the Authenticated-Encryption with Associated-Data (AEAD) encryption scheme associated with the MLS ciphersuite using the common framing mechanism. Note that "Authenticated" in this context does not mean messages are known to be sent by a specific client but only from a legitimate member of the group. To authenticate a message from a particular member, signatures are required. Handshake messages MUST use asymmetric signatures to strongly authenticate the sender of a message.

16.1. Message Encryption and Decryption

The group members MUST use the AEAD algorithm associated with the negotiated MLS ciphersuite to AEAD encrypt and decrypt their Application messages according to the Message Framing section.

The group identifier and epoch allow a recipient to know which group secrets should be used and from which Epoch secret to start computing other secrets and keys. The sender identifier is used to identify the member's symmetric ratchet from the initial group Application secret. The application generation field is used to determine how far into the ratchet to iterate in order to reproduce the required AEAD keys and nonce for performing decryption.

Application messages SHOULD be padded to provide some resistance against traffic analysis techniques over encrypted traffic. [CLINIC] [HCJ16] While MLS might deliver the same payload less frequently across a lot of ciphertexts than traditional web servers, it might still provide the attacker enough information to mount an attack. If Alice asks Bob: "When are we going to the movie ?" the answer "Wednesday" might be leaked to an adversary by the ciphertext length. An attacker expecting Alice to answer Bob with a day of the week might find out the plaintext by correlation between the question and the length.

The content and length of the padding field in MLSCiphertextContent can be chosen at the time of message encryption by the sender. It is recommended that padding data is comprised of zero-valued bytes and follows an established deterministic padding scheme.

16.2. Restrictions

During each epoch senders MUST NOT encrypt more data than permitted by the security bounds of the AEAD scheme used.

Note that each change to the Group through a Handshake message will also set a new encryption_secret. Hence this change MUST be applied before encrypting any new application message. This is required both to ensure that any users removed from the group can no longer receive messages and to (potentially) recover confidentiality and authenticity for future messages despite a past state compromise.

16.3. Delayed and Reordered Application messages

Since each Application message contains the group identifier, the epoch and a message counter, a client can receive messages out of order. If they are able to retrieve or recompute the correct AEAD decryption key from currently stored cryptographic material clients can decrypt these messages.

For usability, MLS clients might be required to keep the AEAD key and nonce for a certain amount of time to retain the ability to decrypt delayed or out of order messages, possibly still in transit while a decryption is being done.

17. Security Considerations

The security goals of MLS are described in [I-D.ietf-mls-architecture]. We describe here how the protocol achieves its goals at a high level, though a complete security analysis is outside of the scope of this document.

17.1. Confidentiality of the Group Secrets

Group secrets are partly derived from the output of a ratchet tree. Ratchet trees work by assigning each member of the group to a leaf in the tree and maintaining the following property: the private key of a node in the tree is known only to members of the group that are assigned a leaf in the node's subtree. This is called the *ratchet tree invariant* and it makes it possible to encrypt to all group members except one, with a number of ciphertexts that's logarithmic in the number of group members.

The ability to efficiently encrypt to all members except one allows members to be securely removed from a group. It also allows a member to rotate their keypair such that the old private key can no longer be used to decrypt new messages.

17.2. Authentication

The first form of authentication we provide is that group members can verify a message originated from one of the members of the group. For encrypted messages, this is guaranteed because messages are encrypted with an AEAD under a key derived from the group secrets. For plaintext messages, this is guaranteed by the use of a membership_tag

which constitutes a MAC over the message, under a key derived from the group secrets.

The second form of authentication is that group members can verify a message originated from a particular member of the group. This is guaranteed by a digital signature on each message from the sender's signature key.

The signature keys held by group members are critical to the security of MLS against active attacks. If a member's signature key is compromised, then an attacker can create LeafNodes and KeyPackages impersonating the member; depending on the application, this can then allow the attacker to join the group with the compromised member's identity. For example, if a group has enabled external parties to join via external commits, then an attacker that has compromised a member's signature key could use an external commit to insert themselves into the group -- even using a "resync"-style external commit to replace the compromised member in the group.

Applications can mitigate the risks of signature key compromise using pre-shared keys. If a group requires joiners to know a PSK in addition to authenticating with a credential, then in order to mount an impersonation attack, the attacker would need to compromise the relevant PSK as well as the victim's signature key. The cost of this mitigation is that the application needs some external arrangement that ensures that the legitimate members of the group to have the required PSKs.

17.3. Forward Secrecy and Post-Compromise Security

Post-compromise security is provided between epochs by members regularly updating their leaf key in the ratchet tree. Updating their leaf key prevents group secrets from continuing to be encrypted to previously compromised public keys.

Forward-secrecy between epochs is provided by deleting private keys from past version of the ratchet tree, as this prevents old group secrets from being re-derived. Forward secrecy *within* an epoch is provided by deleting message encryption keys once they've been used to encrypt or decrypt a message.

Post-compromise security is also provided for new groups by members regularly generating new KeyPackages and uploading them to the Delivery Service, such that compromised key material won't be used when the member is added to a new group.

17.4. KeyPackage Reuse

KeyPackages are intended to be used only once. That is, once a KeyPackage has been used to introduce the corresponding client to a

group, it SHOULD be deleted from the KeyPackage publication system. Reuse of KeyPackages can lead to replay attacks.

An application MAY allow for reuse of a "last resort" KeyPackage in order to prevent denial of service attacks. Since a KeyPackage is needed to add a client to a new group, an attacker could prevent a client being added to new groups by exhausting all available KeyPackages. To prevent such a denial of service attack, the KeyPackage publication system SHOULD rate limit KeyPackage requests, especially if not authenticated.

17.5. Group Fragmentation by Malicious Insiders

It is possible for a malicious member of a group to "fragment" the group by crafting an invalid UpdatePath. Recall that an UpdatePath encrypts a sequence of path secrets to different subtrees of the group's ratchet trees. These path secrets should be derived in a sequence as described in Section 8.4, but the UpdatePath syntax allows the sender to encrypt arbitrary, unrelated secrets. The syntax also does not guarantee that the encrypted path secret encrypted for a given node corresponds to the public key provided for that node.

Both of these types of corruption will cause processing of a Commit to fail for some members of the group. If the public key for a node does not match the path secret, then the members that decrypt that path secret will reject the commit based on this mismatch. If the path secret sequence is incorrect at some point, then members that can decrypt nodes before that point will compute a different public key for the mismatched node than the one in the UpdatePath, which also causes the Commit to fail. Applications SHOULD provide mechanisms for failed commits to be reported, so that group members who were not able to recognize the error themselves can reject the commit and roll back to a previous state if necessary.

Even with such an error reporting mechanism in place, however, it is still possible for members to get locked out of the group by a malformed commit. Since malformed Commits can only be recognized by certain members of the group, in an asynchronous application, it may be the case that all members that could detect a fault in a Commit are offline. In such a case, the Commit will be accepted by the group, and the resulting state possibly used as the basis for further Commits. When the affected members come back online, they will reject the first commit, and thus be unable to catch up with the group.

Applications can address this risk by requiring certain members of the group to acknowledge successful processing of a Commit before the group regards the Commit as accepted. The minimum set of acknowledgements necessary to verify that a Commit is well-formed comprises an acknowledgement from one member per node in the

UpdatePath, that is, one member from each subtree rooted in the copath node corresponding to the node in the UpdatePath.

18. IANA Considerations

This document requests the creation of the following new IANA registries:

- *MLS Ciphersuites (Section 18.1)
- *MLS Extension Types (Section 18.2)
- *MLS Proposal Types (Section 18.3)
- *MLS Credential Types (Section 18.4)

All of these registries should be under a heading of "Messaging Layer Security", and assignments are made via the Specification Required policy [RFC8126]. See Section 18.5 for additional information about the MLS Designated Experts (DEs).

RFC EDITOR: Please replace XXXX throughout with the RFC number assigned to this document

18.1. MLS Ciphersuites

A ciphersuite is a combination of a protocol version and the set of cryptographic algorithms that should be used.

Ciphersuite names follow the naming convention:

```
CipherSuite MLS_LVL_KEM_AEAD_HASH_SIG = VALUE;
```

Where VALUE is represented as a sixteen-bit integer:

```
uint16 CipherSuite;
```

Component	Contents
LVL	The security level
KEM	The KEM algorithm used for HPKE in ratchet tree operations
AEAD	The AEAD algorithm used for HPKE and message protection
HASH	The hash algorithm used for HPKE and the MLS transcript hash
SIG	The Signature algorithm used for message authentication

Table 4

The columns in the registry are as follows:

- *Value: The numeric value of the ciphersuite

- *Name: The name of the ciphersuite

- *Recommended: Whether support for this ciphersuite is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

- *Reference: The document where this ciphersuite is defined

Initial contents:

Value	Name	R	Ref
0x0000	RESERVED	-	RFC XXXX
0x0001	MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519	Y	RFC XXXX
0x0002	MLS_128_DHKEMP256_AES128GCM_SHA256_P256	Y	RFC XXXX
0x0003	MLS_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519	Y	RFC XXXX
0x0004	MLS_256_DHKEMX448_AES256GCM_SHA512_Ed448	Y	RFC XXXX
0x0005	MLS_256_DHKEMP521_AES256GCM_SHA512_P521	Y	RFC XXXX
0x0006	MLS_256_DHKEMX448_CHACHA20POLY1305_SHA512_Ed448	Y	RFC XXXX
0x0007	MLS_256_DHKEMP384_AES256GCM_SHA384_P384.	Y	RFC XXXX
0xff00 - 0xffff	Reserved for Private Use	-	RFC XXXX

Table 5

All of these ciphersuites use HMAC [RFC2104] as their MAC function, with different hashes per ciphersuite. The mapping of ciphersuites to HPKE primitives, HMAC hash functions, and TLS signature schemes is as follows [RFC9180] [RFC8446]:

Value	KEM	KDF	AEAD	Hash	Signature
0x0001	0x0020	0x0001	0x0001	SHA256	ed25519
0x0002	0x0010	0x0001	0x0001	SHA256	ecdsa_secp256r1_sha256
0x0003	0x0020	0x0001	0x0003	SHA256	ed25519
0x0004	0x0021	0x0003	0x0002	SHA512	ed448

Value	KEM	KDF	AEAD	Hash	Signature
0x0005	0x0012	0x0003	0x0002	SHA512	ecdsa_secp521r1_sha512
0x0006	0x0021	0x0003	0x0003	SHA512	ed448
0x0007	0x0011	0x0002	0x0002	SHA384	ecdsa_secp384r1_sha384

Table 6

The hash used for the MLS transcript hash is the one referenced in the ciphersuite name. In the ciphersuites defined above, "SHA256", "SHA384", and "SHA512" refer to the SHA-256, SHA-384, and SHA-512 functions defined in [SHS].

It is advisable to keep the number of ciphersuites low to increase the chances clients can interoperate in a federated environment, therefore the ciphersuites only include modern, yet well-established algorithms. Depending on their requirements, clients can choose between two security levels (roughly 128-bit and 256-bit). Within the security levels clients can choose between faster X25519/X448 curves and FIPS 140-2 compliant curves for Diffie-Hellman key negotiations. Additionally clients that run predominantly on mobile processors can choose ChaCha20Poly1305 over AES-GCM for performance reasons. Since ChaCha20Poly1305 is not listed by FIPS 140-2 it is not paired with FIPS 140-2 compliant curves. The security level of symmetric encryption algorithms and hash functions is paired with the security level of the curves.

The mandatory-to-implement ciphersuite for MLS 1.0 is MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519 which uses Curve25519 for key exchange, AES-128-GCM for HPKE, HKDF over SHA2-256, and Ed25519 for signatures.

Values with the first byte 255 (decimal) are reserved for Private Use.

New ciphersuite values are assigned by IANA as described in Section 18.

18.2. MLS Extension Types

This registry lists identifiers for extensions to the MLS protocol. The extension type field is two bytes wide, so valid extension type values are in the range 0x0000 to 0xffff.

Template:

- *Value: The numeric value of the extension type

- *Name: The name of the extension type

*Message(s): The messages in which the extension may appear, drawn from the following list:

  -KP: KeyPackage objects

  -LN: LeafNode objects

  -GC: GroupContext objects (and the group_context_extensions field of GroupInfo objects)

  -GI: The other_extensions field of GroupInfo objects

*Recommended: Whether support for this extension is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

*Reference: The document where this extension is defined

Initial contents:

Value	Name	Message(s)	Recommended	Reference
0x0000	RESERVED	N/A	N/A	RFC XXXX
0x0001	external_key_id	KP	Y	RFC XXXX
0x0002	ratchet_tree	GI	Y	RFC XXXX
0x0003	required_capabilities	GC	Y	RFC XXXX
0x0004	external_pub	GI	Y	RFC XXXX
0xff00 - 0xffff	Reserved for Private Use	N/A	N/A	RFC XXXX

Table 7

18.3. MLS Proposal Types

This registry lists identifiers for types of proposals that can be made for changes to an MLS group. The extension type field is two bytes wide, so valid extension type values are in the range 0x0000 to 0xffff.

Template:

*Value: The numeric value of the proposal type

*Name: The name of the proposal type

*Recommended: Whether support for this extension is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires

Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

*Path Required: Whether a Commit covering a proposal of this type is required to have its path field populated (see Section 13.2).

*Reference: The document where this extension is defined

Initial contents:

Value	Name	Recommended	Path Required	Reference
0x0000	RESERVED	N/A	N/A	RFC XXXX
0x0001	add	Y	N	RFC XXXX
0x0002	update	Y	Y	RFC XXXX
0x0003	remove	Y	Y	RFC XXXX
0x0004	psk	Y	N	RFC XXXX
0x0005	reinit	Y	N	RFC XXXX
0x0006	external_init	Y	Y	RFC XXXX
0x0007	app_ack	Y	N	RFC XXXX
0x0008	group_context_extensions	Y	Y	RFC XXXX
0xff00 - 0xffff	Reserved for Private Use	N/A	N/A	RFC XXXX

Table 8

18.4. MLS Credential Types

This registry lists identifiers for types of credentials that can be used for authentication in the MLS protocol. The credential type field is two bytes wide, so valid credential type values are in the range 0x0000 to 0xffff.

Template:

*Value: The numeric value of the credential type

*Name: The name of the credential type

*Recommended: Whether support for this credential is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

*Reference: The document where this credential is defined

Initial contents:

Value	Name	Recommended	Reference
0x0000	RESERVED	N/A	RFC XXXX
0x0001	basic	Y	RFC XXXX
0x0002	x509	Y	RFC XXXX
0xff00 - 0xffff	Reserved for Private Use	N/A	RFC XXXX

Table 9

18.5. MLS Designated Expert Pool

Specification Required [RFC8126] registry requests are registered after a three-week review period on the MLS DEs' mailing list: mls-reg-review@ietf.org, on the advice of one or more of the MLS DEs. However, to allow for the allocation of values prior to publication, the MLS DEs may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the MLS DEs mailing list for review SHOULD use an appropriate subject (e.g., "Request to register value in MLS Bar registry").

Within the review period, the MLS DEs will either approve or deny the registration request, communicating this decision to the MLS DEs mailing list and IANA. Denials SHOULD include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG's attention for resolution using the iesg@ietf.org mailing list.

Criteria that SHOULD be applied by the MLS DEs includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or useful only for a single application, and whether the registration description is clear. For example, the MLS DEs will apply the ciphersuite-related advisory found in Section 6.1.

IANA MUST only accept registry updates from the MLS DEs and SHOULD direct all requests for registration to the MLS DEs' mailing list.

It is suggested that multiple MLS DEs be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular MLS DE, that MLS DE SHOULD defer to the judgment of the other MLS DEs.

18.6. The "message/mls" MIME Type

This document registers the "message/mls" MIME media type in order to allow other protocols (ex: HTTP [RFC7540]) to convey MLS messages.

Media type name:
message

Media subtype name: mls

Required parameters: none

Optional parameters:

version

version: The MLS protocol version expressed as a string <major>.<minor>. If omitted the version is "1.0", which corresponds to MLS ProtocolVersion mls10. If for some reason the version number in the MIME type parameter differs from the ProtocolVersion embedded in the protocol, the protocol takes precedence.

Encoding scheme: MLS messages are represented using the TLS presentation language [RFC8446]. Therefore MLS messages need to be treated as binary data.

Security considerations: MLS is an encrypted messaging layer designed to be transmitted over arbitrary lower layer protocols. The security considerations in this document (RFC XXXX) also apply.

19. Contributors

*Joel Alwen
Wickr
joel.alwen@wickr.com

*Karthikeyan Bhargavan
INRIA
karthikeyan.bhargavan@inria.fr

*Cas Cremers
University of Oxford
cremers@cispa.de

*Alan Duric
Wire
alan@wire.com

*Britta Hale
Naval Postgraduate School
britta.hale@nps.edu

*Srinivas Inguva
Twitter
singuva@twitter.com

*Konrad Kohbrok
Aalto University
konrad.kohbrok@datashrine.de

*Albert Kwon
MIT
kwonal@mit.edu

*Brendan McMillion
Cloudflare
brendan@cloudflare.com

*Eric Rescorla
Mozilla
ekr@rtfm.com

*Michael Rosenberg
Trail of Bits
michael.rosenberg@trailofbits.com

*Thyla van der Merwe
Royal Holloway, University of London
thyla.van.der@merwe.tech

20. References

20.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/rfc/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/rfc/rfc7540>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26,

RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

20.2. Informative References

[art] Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., and K. Milner, "On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees", 18 January 2018, <https://eprint.iacr.org/2017/666.pdf>.

[CLINIC] Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know Why You Went to the Clinic: Risks and Realization of HTTPS Traffic Analysis", Privacy Enhancing Technologies pp. 143-163, DOI 10.1007/978-3-319-08506-7_8, 2014, <https://doi.org/10.1007/978-3-319-08506-7_8>.

[doubleratchet] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and D. Stebila, "A Formal Security Analysis of the Signal Messaging Protocol", 2017 IEEE European Symposium on Security and Privacy (EuroS&P), DOI 10.1109/eurosp.2017.27, April 2017, <https://doi.org/10.1109/eurosp.2017.27>.

[HCJ16] Husák, M., Čermák, M., Jirsík, T., and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016, <https://doi.org/10.1186/s13635-016-0030-7>.

[I-D.ietf-mls-architecture] Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., Kwon, A., and A. Duric, "The Messaging Layer Security (MLS) Architecture", Work in Progress, Internet-Draft, draft-ietf-mls-architecture-07, 4 October 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-architecture-07>.

[I-D.ietf-trans-rfc6962-bis] Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", Work in Progress, Internet-Draft, draft-ietf-trans-rfc6962-bis-42, 31 August 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-trans-rfc6962-bis-42>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/rfc/rfc6125>.

[RFC7696] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, DOI 10.17487/RFC7696, November 2015, <https://www.rfc-editor.org/rfc/rfc7696>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <https://secg.org/sec1-v2.pdf>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[signal] Perrin(ed), T. and M. Marlinspike, "The Double Ratchet Algorithm", 20 November 2016, <https://www.signal.org/docs/specifications/doubleratchet/>.

Appendix A. Protocol Origins of Example Trees

Protocol operations in MLS give rise to specific forms of ratchet tree, typically affecting a whole direct path at once. In this section, we describe the protocol operations that could have given rise to the various example trees in this document.

To construct the tree in Figure 9:

* A creates a group with B, ..., G
* Each member sends an empty Commit setting its direct path

To construct the tree in Figure 10:

* A creates a group with B, C, D, as well as some members outside this subtree
* D removes C, setting Z and the top node (as well as any further nodes in the direct path)
* A member outside this subtree removes B, blanking B's direct path
* A adds a new member at C with a partial Commit, adding it as unmerged at Z

To construct the tree in Figure 11:

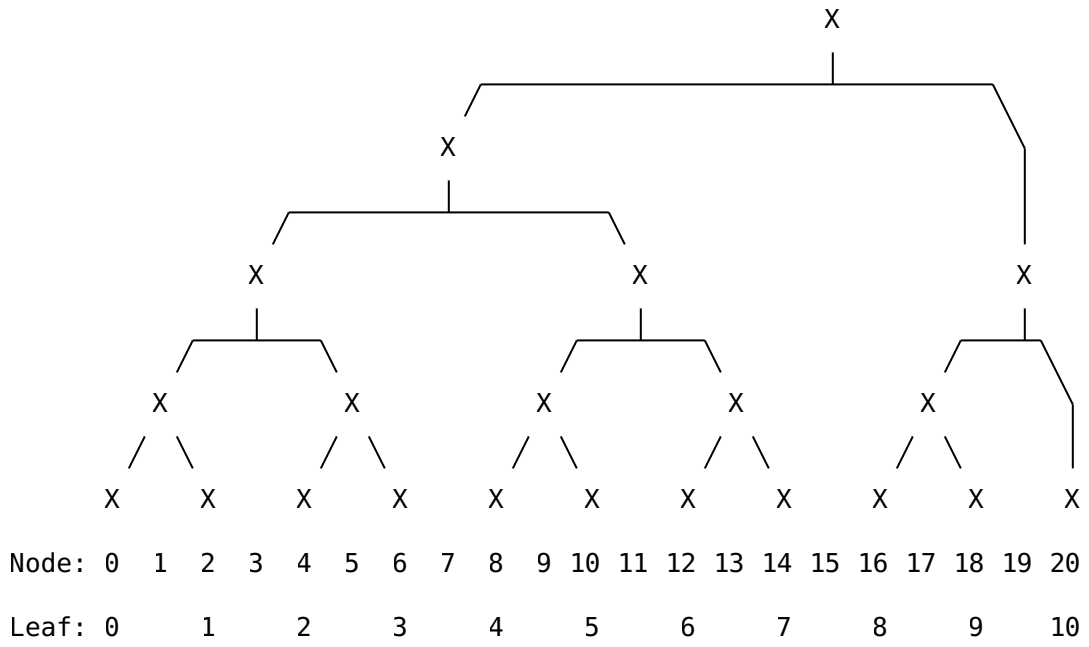
* A creates a group with B, C, D
* B sends a full Commit, setting X and Y
* D removes C, setting Z and Y
* B adds a new member at C with a full Commit
    - The Add proposal adds C as unmerged at Z and Y
    - The path in the Commit resets X and Y, clearing Y's unmerged leaves

To construct the tree in Figure 12:

* A creates a group with B, ..., G
* A removes F in a full Commit, setting T, U, and W
* E sends an empty Commit, setting Y and W
* A adds a new member at F in a partial Commit, adding F as unmerged at Y and W

Appendix B. Array-Based Trees

One benefit of using left-balanced trees is that they admit a simple flat array representation. In this representation, leaf nodes are even-numbered nodes, with the n-th leaf at 2*n. Intermediate nodes are held in odd-numbered nodes. For example, tree with 11 leaves has the following structure:



This allows us to compute relationships between tree nodes simply by manipulating indices, rather than having to maintain complicated structures in memory. The basic rule is that the high-order bits of parent and child nodes indices have the following relation (where x is an arbitrary bit string):

```
parent=01x => left=00x, right=10x
```

Since node relationships are implicit, the algorithms for adding and removing nodes at the right edge of the tree are quite simple:

- Add: Append a blank parent node to the array of nodes, then append the new leaf node

- Remove: Remove the rightmost two nodes from the array of nodes

The following python code demonstrates the tree computations necessary to use an array-based tree for MLS.

```
# The exponent of the largest power of 2 less than x. Equivalent to:
#   int(math.floor(math.log(x, 2)))
def log2(x):
    if x == 0:
        return 0

    k = 0
    while (x >> k) > 0:
        k += 1
    return k-1

# The level of a node in the tree. Leaves are level 0, their parents
# are level 1, etc. If a node's children are at different levels,
# then its level is the max level of its children plus one.
def level(x):
    if x & 0x01 == 0:
        return 0

    k = 0
    while ((x >> k) & 0x01) == 1:
        k += 1
    return k

# The number of nodes needed to represent a tree with n leaves.
def node_width(n):
    if n == 0:
        return 0
    else:
        return 2*(n - 1) + 1

# The index of the root node of a tree with n leaves.
def root(n):
    w = node_width(n)
    return (1 << log2(w)) - 1

# The left child of an intermediate node. Note that because the tree
# is left-balanced, there is no dependency on the size of the tree.
def left(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')

    return x ^ (0x01 << (k - 1))

# The right child of an intermediate node. Depends on the number of
# leaves because the straightforward calculation can take you beyond
# the edge of the tree.
def right(x, n):
    k = level(x)
    if k == 0:
```

```
        raise Exception('leaf node has no children')

    r = x ^ (0x03 << (k - 1))
    while r >= node_width(n):
        r = left(r)
    return r

# The immediate parent of a node. May be beyond the right edge of the
# tree.
def parent_step(x):
    k = level(x)
    b = (x >> (k + 1)) & 0x01
    return (x | (1 << k)) ^ (b << (k + 1))

# The parent of a node. As with the right child calculation, we have
# to walk back until the parent is within the range of the tree.
def parent(x, n):
    if x == root(n):
        raise Exception('root node has no parent')

    p = parent_step(x)
    while p >= node_width(n):
        p = parent_step(p)
    return p

# The other child of the node's parent.
def sibling(x, n):
    p = parent(x, n)
    if x < p:
        return right(p, n)
    else:
        return left(p)

# The direct path of a node, ordered from leaf to root.
def direct_path(x, n):
    r = root(n)
    if x == r:
        return []

    d = []
    while x != r:
        x = parent(x, n)
        d.append(x)
    return d

# The copath of a node, ordered from leaf to root.
def copath(x, n):
    if x == root(n):
        return []
```

```
    d = direct_path(x, n)
    d.insert(0, x)
    d.pop()
    return [sibling(y, n) for y in d]

# The common ancestor of two nodes is the lowest node that is in the
# direct paths of both leaves.
def common_ancestor_semantic(x, y, n):
    dx = set([x]) | set(direct_path(x, n))
    dy = set([y]) | set(direct_path(y, n))
    dxy = dx & dy
    if len(dxy) == 0:
        raise Exception('failed to find common ancestor')

    return min(dxy, key=level)

# The common ancestor of two nodes is the lowest node that is in the
# direct paths of both leaves.
def common_ancestor_direct(x, y, _):
    # Handle cases where one is an ancestor of the other
    lx, ly = level(x)+1, level(y)+1
    if (lx <= ly) and (x>>ly == y>>ly):
      return y
    elif (ly <= lx) and (x>>lx == y>>lx):
      return x

    # Handle other cases
    xn, yn = x, y
    k = 0
    while xn != yn:
       xn, yn = xn >> 1, yn >> 1
       k += 1
    return (xn << k) + (1 << (k-1)) - 1
```

Appendix C. Link-Based Trees

An implementation may choose to store ratchet trees in a "link-based" representation, where each node stores references to its parents and/or children. (As opposed to the array-based representation suggested above, where these relationships are computed from relationships between nodes' indices in the array.) Such an implementation needs to update these links to maintain the left-balanced structure of the tree as the tree is extended to add new members, or truncated when memebers are removed.

The following code snippet shows how these algorithms could be implemented in Python.

```
class Node:
    def __init__(self, value, parent=None, left=None, right=None):
        self.value = value    # Value of the node
        self.parent = parent  # Parent node
        self.left = left      # Left child node
        self.right = right    # Right child node

    def leaf(self):
        return self.left == None and self.right == None

    def span(self):
        if self.leaf():
            return 1
        return self.left.span() + self.right.span()

    def full(self):
        span = self.span()
        while span % 2 == 0:
            span >>= 1
        return span == 1

    def rightmost_leaf(self):
        X = self
        while X.right != None:
            X = X.right
        return X

class Tree:
    def __init__(self):
        self.root = None  # Root node of the tree, initially empty

    def extend(self, N):
        if self.root == None:
            self.root = N
            return

        # Identify the proper point to insert the new parent node
        X = self.root.rightmost_leaf()
        while X.full() and X != self.root:
            X = X.parent

        # If X is not full, insert the new parent under X
        P = Node("_", right=N)
        N.parent = P
        if not X.full():
            P.parent = X
            P.left = X.right
            X.right.parent = P
            X.right = P
            return
```

```
        # If X is full, then X is the root, so P replaces the root
        P.left = self.root
        self.root.parent = P
        self.root = P
        return

    def truncate(self):
        X = self.root.rightmost_leaf()
        if X == self.root:
            self.root = None
            return

        # If X's parent is the root, then shift the root to the left
        if X.parent == self.root:
            self.root = self.root.left
            self.root.parent = None
            return

        # Otherwise, reassign the right child of the parent's parent
        Q = X.parent.parent
        Q.right = X.parent.left
        Q.right.parent = Q
        return
```

Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Benjamin Beurdouche
Inria & Mozilla

Email: ietf@beurdouche.com

Raphael Robert

Email: ietf@raphaelrobert.com

Jon Millican
Facebook

Email: jmillican@fb.com

Emad Omara
Google

Email: emadomara@google.com

Katriel Cohn-Gordon
University of Oxford

Email: me@katriel.co.uk