Network Working Group   R. Barnes
Internet-Draft   Cisco
Intended status: Informational   B. Beurdouche
Expires: 4 May 2021   Inria
                                                          J. Millican
                                                           Facebook
                                                            E. Omara
                                                             Google
                                                    K. Cohn-Gordon
                                               University of Oxford
                                                          R. Robert
                                                              Wire
                                                   31 October 2020

The Messaging Layer Security (MLS) Protocol
draft-ietf-mls-protocol-10

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.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF). Note that other groups may also distribute
   working documents as Internet-Drafts. The list of current Internet-
   Drafts is at https://datatracker.ietf.org/drafts/current/.

   Internet-Drafts are draft documents valid for a maximum of six months
   and may be updated, replaced, or obsoleted by other documents at any
   time. It is inappropriate to use Internet-Drafts as reference
   material or to cite them other than as "work in progress."

   This Internet-Draft will expire on 4 May 2021.

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-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 uneeded 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 collection of clients with 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
[I-D.irtf-cfrg-hpke]) public key that can be used to encrypt to
that client.

Initialization Key (InitKey): A key package that is prepublished by
a client, 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.

Terminology specific to tree computations is described in Section 5.

We use the TLS presentation language [RFC8446] to describe the
structure of protocol messages.

3. Basic Assumptions

This protocol is designed to execute in the context of a Service
Provider (SP) as described in [I-D.ietf-mls-architecture]. In
particular, we assume the SP provides the following services:

* A signature key provider which allows clients to authenticate
   protocol messages in a group.

* A broadcast channel, for each group, which will relay a message to
   all members of a group. For the most part, we assume that this
   channel delivers messages in the same order to all participants.
   (See Section 13 for further considerations.)

* A directory to which clients can publish key packages and download
   key packages for other participants.

[4](#). Protocol Overview

   The goal of this protocol is to allow a group of clients to exchange
   confidential and authenticated messages. It does so by deriving a
   sequence of secrets and keys known only to members. Those should be
   secret against an active network adversary and should have both
   forward secrecy and post-compromise security with respect to
   compromise of any members.

   We describe the information stored by each client as _state_, which
   includes both public and private data. An initial state is set up by
   a group creator, which is a group containing only itself. The
   creator then sends _Add_ proposals for each client in the initial set
   of members, followed by a _Commit_ message which incorporates all of
   the _Adds_ into the group state. Finally, the group creator
   generates a _Welcome_ message corresponding to the Commit and sends
   this directly to all the new members, who can use the information it
   contains to set up their own group state and derive a shared secret.
   Members exchange Commit messages for post-compromise security, to add
   new members, and to remove existing members. These messages produce
   new shared secrets which are causally linked to their predecessors,
   forming a logical Directed Acyclic Graph (DAG) of states.

   The protocol algorithms we specify here follow. Each algorithm
   specifies both (i) how a client performs the operation and (ii) how
   other clients update their state based on it.

   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.

   Before the initialization of a group, clients publish InitKeys (as
   KeyPackage objects) to a directory provided by the Service Provider.

```
                                                          Group
    A               B               C         Directory   Channel
    |               |               |               |           |
    | KeyPackageA    |               |               |           |
    |----------------------------------------------------->|           |
    |               |               |               |           |
    |               | KeyPackageB    |               |           |
    |               |-------------------------------->|           |
    |               |               |               |           |
    |               |               | KeyPackageC    |           |
    |               |               |--------------->|           |
    |               |               |               |           |
```
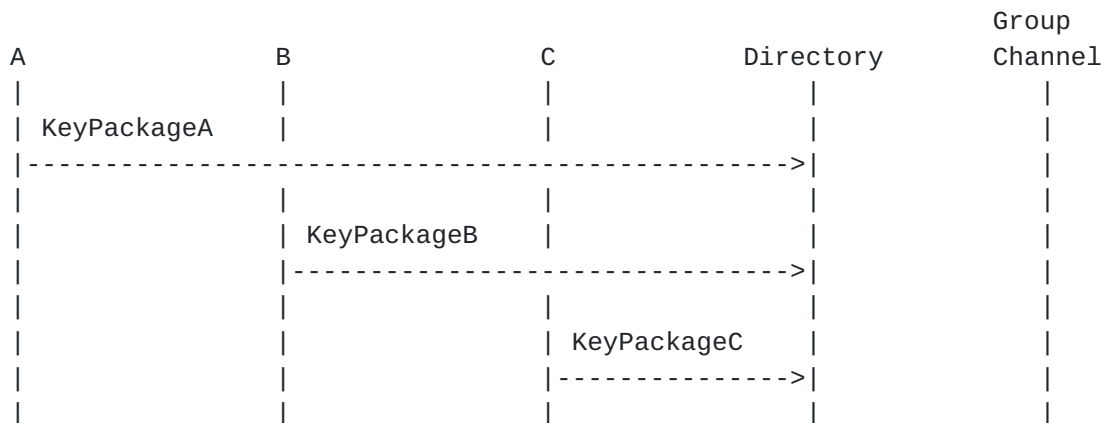
   When a client A wants to establish a group with B and C, it first
   initializes a group state containing only itself and downloads
   KeyPackages for B and C. For each member, A generates an Add and
   Commit message adding that member, and broadcasts them to the group.
   It also generates a Welcome 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 and the corresponding Commit, the
   new member will be able to read and send new messages to the group.
   Messages received before the client has joined the group are ignored.

```
                                                            Group
      A              B              C        Directory     Channel
      |              |              |            |            |
      |          KeyPackageB, KeyPackageC        |            |
      |<-------------------------------------------|            |
      |state.init()  |              |            |            |
      |              |              |            |            |
      |              |              |            | Add(A->AB) |
      |              |              |            | Commit(Add)|
      |------------------------------------------------------->|
      |              |              |            |            |
      |  Welcome(B)  |              |            |            |
      |------------->|state.init()  |            |            |
      |              |              |            |            |
      |              |              |            | Add(A->AB) |
      |              |              |            | Commit(Add)|
      |<-------------------------------------------------------|
      |state.add(B)  |<-----------------------------------------|
      |              |state.join()  |            |            |
      |              |              |            |            |
      |              |              |            | Add(AB->ABC)|
      |              |              |            | Commit(Add)|
      |------------------------------------------------------->|
      |              |              |            |            |
      |              |   Welcome(C) |            |            |
      |--------------------------->|state.init() |            |
      |              |              |            |            |
      |              |              |            | Add(AB->ABC)|
      |              |              |            | Commit(Add)|
      |<-------------------------------------------------------|
      |state.add(C)  |<-----------------------------------------|
      |              |state.add(C)  |<---------------------------|
      |              |              |state.join()  |            |
```
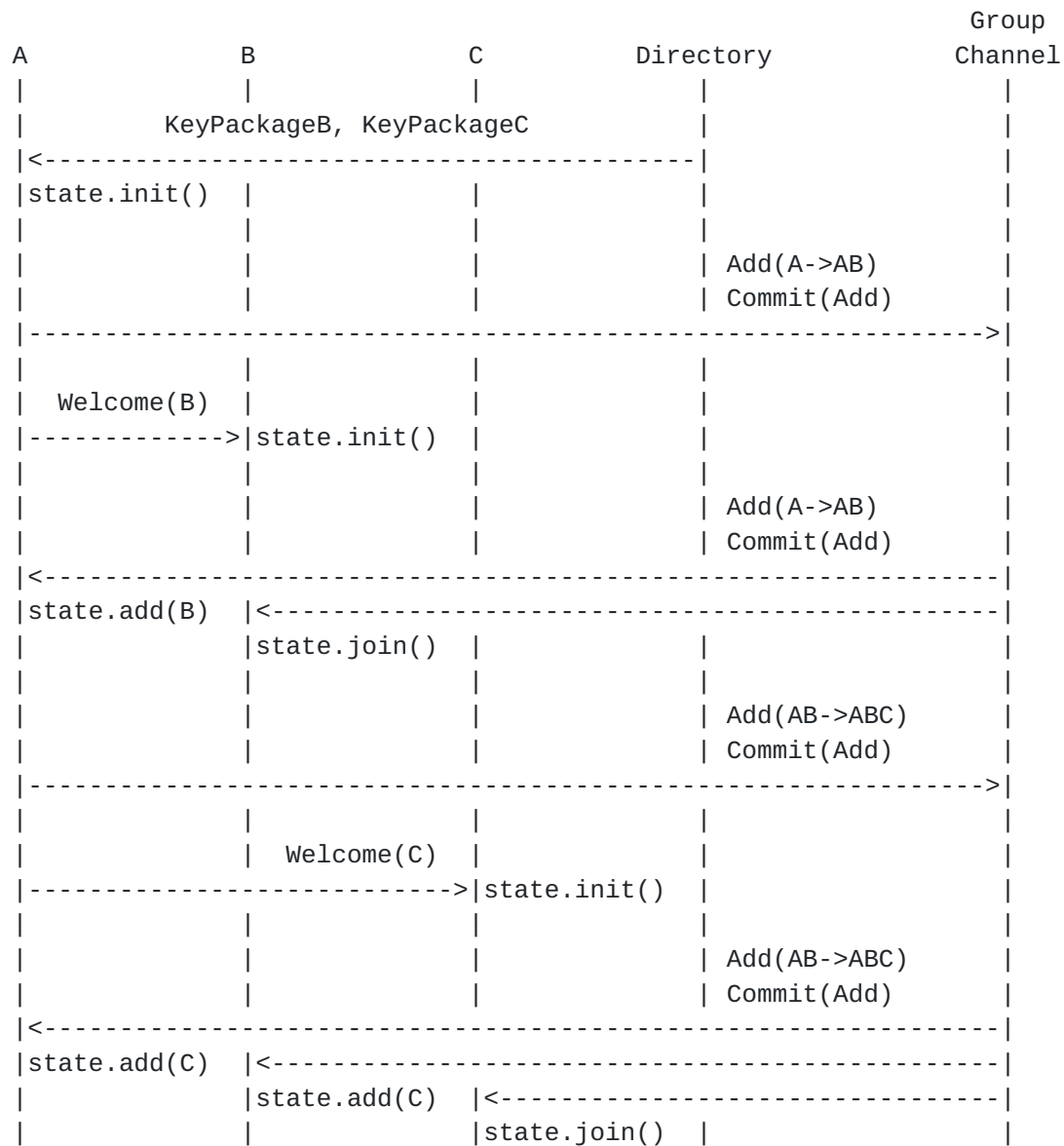
Subsequent additions of group members proceed in the same way. Any
member of the group can download a KeyPackage for a new client and
broadcast an Add message that the current group can use to update
their state, and a Welcome message that the new client can use to
initialize its state.

To enforce the forward secrecy and post-compromise security of
messages, each member periodically updates their leaf secret. Any
member can update this information at any time by generating a fresh
KeyPackage and sending an Update message followed by a Commit
message. Once all members have processed both, 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.

```
                                                          Group
A              B     ...     Z          Directory         Channel
|              |             |              |               |
|              | Update(B)   |              |               |
|              |---------------------------------------------->|
| Commit(Upd)  |             |              |               |
|---------------------------------------------------------->|
|              |             |              |               |
|              |             |              | Update(B)     |
|              |             |              | Commit(Upd)   |
|<----------------------------------------------------------|
|state.upd(B)  |<----------------------------------------------|
|              |state.upd(B)  |<------------------------------|
|              |             |state.upd(B)  |               |
|              |             |              |               |
```
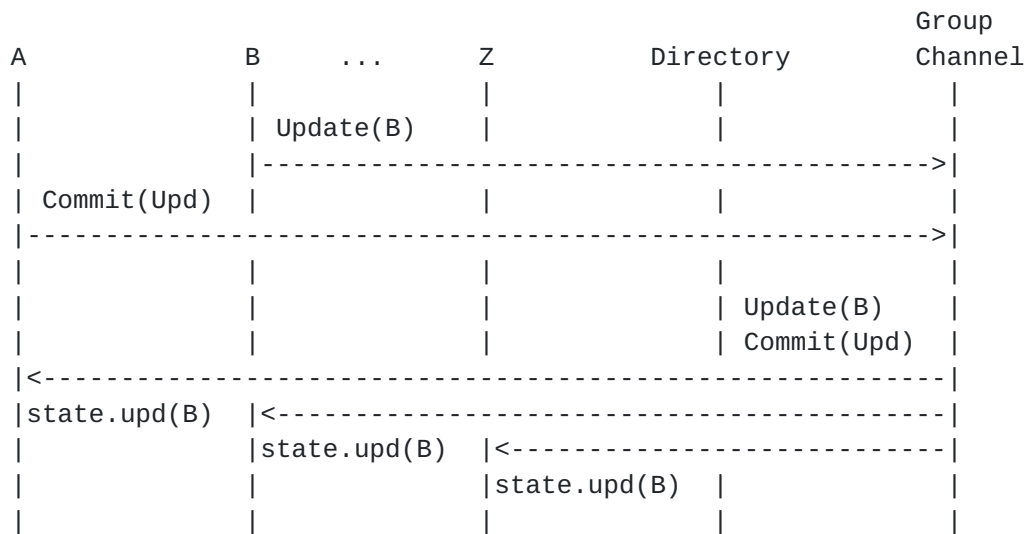
Members are removed from the group in a similar way. Any member of
the group can send a Remove proposal followed by a Commit message,
which adds new entropy to the group state that's known to all except
the removed member. Note that this does not necessarily imply that
any member is actually allowed to evict other members; groups can
enforce access control policies on top of these basic mechanism.

```
                                                          Group
A              B     ...     Z          Directory         Channel
|              |             |              |               |
|              |             | Remove(B)    |               |
|              |             | Commit(Rem)  |               |
|              |             |---------------------------->|
|              |             |              |               |
|              |             |              | Remove(B)     |
|              |             |              | Commit(Rem)   |
|<----------------------------------------------------------|
|state.rem(B)  |             |<----------------------------|
|              |             |state.rem(B)  |               |
|              |             |              |               |
|              |             |              |               |
```
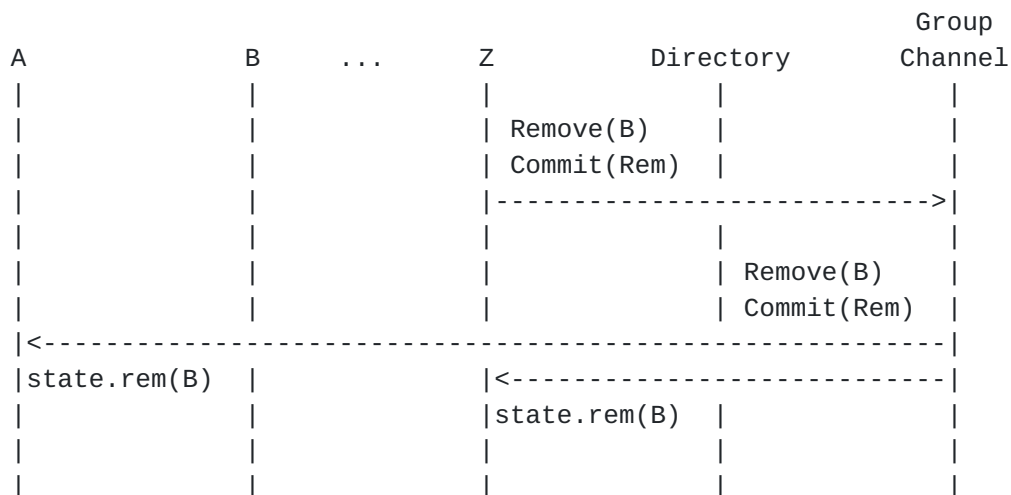
5. Ratchet Trees

The protocol uses "ratchet trees" for deriving shared secrets among a
group of clients.

5.1. Tree Computation 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, its 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. Nodes are _siblings_ if they share the same parent.

   A _subtree_ of a tree is the tree given by the descendants of any
   node, the _head_ of the subtree. 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 (CD, ABCD, ABCDEFG)

   * The copath of C is (D, AB, EFG)

```
                   ABCDEFG
                  /       \
                 /         \
                /           \
            ABCD             EFG
           /    \           /  \
          /      \         /    \
        AB        CD     EF      |
       / \       / \     / \     |
      A   B     C   D   E   F     G


                        1 1 1
      0 1 2 3 4 5 6 7 8 9 0 1 2
```
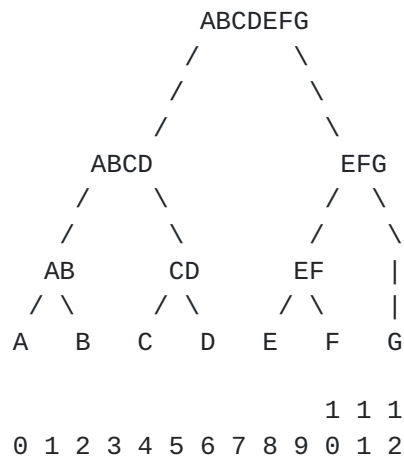
Each node in the tree is assigned a _node index_, starting at zero
and running from left to right. A node is a leaf node if and only if
it has an even index. The node indices for the nodes in the above
tree are as follows:

*   0 = A

*   1 = AB

*   2 = B

*   3 = ABCD

*   4 = C

*   5 = CD

*   6 = D

*   7 = ABCDEFG

*   8 = E

*   9 = EF

*   10 = F

*   11 = EFG

*   12 = G

The leaves of the tree are indexed separately, using a _leaf index_,
since the protocol messages only need to refer to leaves in the tree.
Like nodes, leaves are numbered left to right. The node with leaf

index "k" is also called the "k-th" leaf. Note that given the above
numbering, a node is a leaf node if and only if it has an even node
index, and a leaf node's leaf index is half its node index. The leaf
indices in the above tree are as follows:

* 0 = A

* 1 = B

* 2 = C

* 3 = D

* 4 = E

* 5 = F

* 6 = G

5.2. Ratchet Tree Nodes

A particular instance of a ratchet tree is defined by 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 node in a ratchet tree contains up to five values:

* A private key (only within the member's direct path, see below)

* A public key

* An ordered list of leaf indices for "unmerged" leaves (see
   Section 5.3)

* A credential (only for leaf nodes)

* A hash of the node's parent, as of the last time the node was
   changed.

The conditions under which each of these values must or must not be
present are laid out in Section 5.3.

A node in the tree may also be _blank_, indicating that no value is
present at that node. 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 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 tree, where the "_" character
represents a blank node:

```
       _
     /   \
    /     \
   _        CD[C]
  / \      / \
 A   _    C   D

 0 1 2 3 4 5 6
```
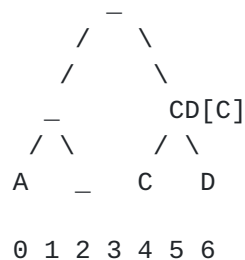
In this tree, we can see all of the above rules in play:

* The resolution of node 5 is the list [CD, C]

* The resolution of node 2 is the empty list []

* The resolution of node 3 is the list [A, CD, 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 7.5.

5.3. 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 that member's leaf is a descendant of
the node.

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 5.4.

5.4. Ratchet Tree Evolution

A member of an MLS group advances the key schedule to provide forward
secrecy and post-compromise security by providing the group with
fresh key material to be added into the group's shared secret. To do
so, one member of the group generates fresh key material, applies it
to their local tree state, and then sends this key material to other
members in the group via an UpdatePath message (see Section 7.7) .
All other group members then apply the key material in the UpdatePath
to their own local tree state to derive the group's now-updated
shared secret.

To begin, the generator of the UpdatePath updates its leaf KeyPackage
and its direct path to the root with new secret values. The HPKE
leaf public key within the KeyPackage MUST be derived from a freshly
generated HPKE secret key to provide post-compromise security.

The generator of the UpdatePath starts by sampling a fresh random
value called "leaf_secret", and uses the leaf_secret to generate
their leaf HPKE key pair (see Section 7) and to seed a sequence of
"path secrets", one for each ancestor of its leaf. In this setting,
path_secret[0] refers to the node directly above the leaf,

path_secret[1] for its parent, and so on. At each step, the path
secret is used to derive a new secret value for the corresponding
node, from which the node's key pair is derived.

```
leaf_node_secret = DeriveSecret(leaf_secret, "node")
path_secret[0] = DeriveSecret(leaf_secret, "path")

path_secret[n] = DeriveSecret(path_secret[n-1], "path")
node_secret[n] = DeriveSecret(path_secret[n], "node")

leaf_priv, leaf_pub = KEM.DeriveKeyPair(leaf_node_secret)
node_priv[n], node_pub[n] = KEM.DeriveKeyPair(node_secret[n])
```

For example, suppose there is a group with four members:

```
     G
    / \
   /   \
  /     \
 E       _
/ \     / \
A  B   C   D
```
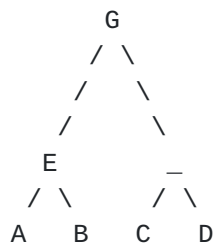
If member B subsequently generates an UpdatePath based on a secret
"leaf_secret", then it would generate the following sequence of path
secrets:

```
path_secret[1] --> node_secret[1] --> node_priv[1], node_pub[1]
     ^
     |
path_secret[0] --> node_secret[0] --> node_priv[0], node_pub[0]
     ^
     |
leaf_secret     --> leaf_node_secret --> leaf_priv, leaf_pub
                                     ~> leaf_key_package
```

After applying the UpdatePath, the tree will have the following
structure, where "np[i]" represents the node_priv values generated as
described above:

```
       np[1]
      /     \
   np[0]     _
   / \      / \
  A   B    C   D
```
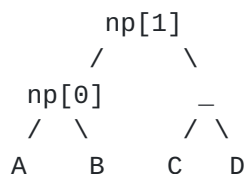
After performing these operations, the generator of the UpdatePath
MUST delete the leaf_secret.

5.5. Synchronizing Views of the Tree

   After generating fresh key material and applying it to ratchet
   forward their local tree state as described in the prior section, 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 message containing a set of public and encrypted
   private values for intermediate nodes in the direct path of a 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.

   To transmit this update, the sender broadcasts to the group the
   following information for each node in the direct path of the 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 leaf node. There is one encrypted path secret
   for each public key in the resolution of the non-updated child.

   The recipient of a path update processes it with the following steps:

   1. Compute the updated path secrets.

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

       * Identify a node in the resolution of the copath node for which
          this node 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.

        * For all updated nodes,

              - Replace the public key for each node with the received
                 public key.

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

              - Store the updated hash of the node's parent (represented as
                 a ParentNode struct), going from root to leaf, so that each
                 hash incorporates all the nodes above it. The root node
                 always has a zero-length hash for this value.

        * For nodes where an updated path secret was computed in step 1,
           compute the corresponding node key pair and replace the values
           stored at the node with the computed values.

   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)                 |
        +===========+================================+
        | pk(ns[1])  | E(pk(C), ps[1]), E(pk(D), ps[1]) |
        +-----------+--------------------------------+
        | pk(ns[0])  | E(pk(A), ps[0])                |
        +-----------+--------------------------------+
```

                              Table 1

   In this table, the value pk(ns[X]) represents the public key derived
   from the node secret X, whereas pk(X) represents the public leaf key
   for user X. The value E(K, S) represents the public-key encryption
   of the path secret S to the public key K.

   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.

[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 signature algorithm

   The HPKE parameters are used to instantiate HPKE [I-D.irtf-cfrg-hpke]
   for the purpose of public-key encryption. The "DeriveKeyPair"
   function associated to the KEM for the ciphersuite maps octet strings
   to HPKE key pairs.

   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<1..2^16-1>;

   The signature algorithm specified in the ciphersuite is the mandatory
   algorithm to be used for signatures in MLSPlaintext and the tree
   signatures. It MUST be the same as the signature algorithm specified
   in the credential field of the KeyPackage objects in the leaves of
   the tree (including the InitKeys used to add new members).

   The ciphersuites are defined in section Section 16.1.

6.2. Credentials

   A member of a group authenticates the identities of other
   participants by means of credentials issued by some authentication
   system, like a PKI. Each type of credential MUST express the
   following data in the context of the group it is used with:

   * The public key of a signature key pair matching the
      SignatureScheme specified by the CipherSuite of the group

   * The identity of the holder of the private key

   Credentials MAY also include information that allows a relying party
   to verify the identity / signing key binding.

   Additionally, Credentials SHOULD specify the signature scheme
   corresponding to each contained public key.

```
   // See RFC 8446 and the IANA TLS SignatureScheme registry
   uint16 SignatureScheme;

   // See IANA registry for registered values
   uint16 CredentialType;

   struct {
       opaque identity<0..2^16-1>;
       SignatureScheme signature_scheme;
       opaque signature_key<0..2^16-1>;
   } BasicCredential;

   struct {
       opaque cert_data<0..2^16-1>;
   } Certificate;

   struct {
       CredentialType credential_type;
       select (Credential.credential_type) {
           case basic:
               BasicCredential;

           case x509:
               Certificate chain<1..2^32-1>;
       };
   } Credential;
```

   A BasicCredential is a raw, unauthenticated assertion of an identity/
   key binding. The format of the key in the "public_key" field is
   defined by the relevant ciphersuite: the group ciphersuite for a
   credential in a ratchet tree, the KeyPackage ciphersuite for a
   credential in a KeyPackage object.

   For X509Credential, each entry in the chain represents a single DER-
   encoded X509 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 algorithm for the "public_key" in the end-entity
   certificate MUST match the relevant ciphersuite.

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

Note that each new credential that has not already been validated by the application MUST be validated against the Authentication Service.

7. Key Packages

In order to facilitate asynchronous addition of clients to a group, it is possible to pre-publish key packages that provide some public information about a user. KeyPackage structures provide information about a client that any existing member can use to add this client to the group asynchronously.

A KeyPackage object specifies a ciphersuite that the client supports, as well as providing a public key that others can use for key agreement. The client's signature key can be updated throughout the lifetime of the group by sending a new KeyPackage with a new Credential. However, the identity MUST be the same in both Credentials and the new Credential MUST be validated by the authentication service.

When used as InitKeys, KeyPackages are intended to be used only once and SHOULD NOT be reused except in case of last resort. (See Section 15.4). Clients MAY generate and publish multiple InitKeys to support multiple ciphersuites.

KeyPackages contain a public key chosen by the client, which the client MUST ensure uniquely identifies a given KeyPackage object among the set of KeyPackages created by this client.

The value for hpke_init_key MUST be a public key for the asymmetric encryption scheme defined by cipher_suite. The whole structure is signed using the client's signature key. A KeyPackage object with an invalid signature field MUST be considered malformed. The input to the signature computation comprises all of the fields except for the signature field.

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

// See IANA registry for registered values
uint16 ExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey hpke_init_key;
    Credential credential;
    Extension extensions<8..2^32-1>;
    opaque signature<0..2^16-1>;
} KeyPackage;
```

KeyPackage objects MUST contain at least two extensions, one of type
"capabilities", and one of type "lifetime". The "capabilities"
extension allow MLS session establishment to be safe from downgrade
attacks on the parameters described (as discussed in Section 10),
while still only advertising one version / ciphersuite per
KeyPackage.

As the "KeyPackage" is a structure which is stored in the Ratchet
Tree and updated depending on the evolution of this tree, each
modification of its content MUST be reflected by a change of its
signature. This allow other members to control the validity of the
KeyPackage at any time and in particular in the case of a newcomer
joining the group.

7.1. Client Capabilities

The "capabilities" extension indicates what protocol versions,
ciphersuites, and protocol extensions are supported by a client.

```
struct {
    ProtocolVersion versions<0..255>;
    CipherSuite ciphersuites<0..255>;
    ExtensionType extensions<0..255>;
} Capabilities;
```

This extension MUST be always present in a KeyPackage. Extensions
that appear in the "extensions" field of a KeyPackage MUST be
included in the "extensions" field of the "capabilities" extension.

7.2. Lifetime

The "lifetime" extension represents the times between which clients
will consider a KeyPackage valid. This time is represented as an
absolute time, measured in seconds since the Unix epoch
(1970-01-01T00:00:00Z). A client MUST NOT use the data in a
KeyPackage for any processing before the "not_before" date, or after
the "not_after" date.

```
uint64 not_before;
uint64 not_after;
```

Applications MUST define a maximum total lifetime that is acceptable
for a KeyPackage, and reject any KeyPackage where the total lifetime
is longer than this duration.

This extension MUST always be present in a KeyPackage.

7.3. KeyPackage Identifiers

Within MLS, a KeyPackage is identified by its hash (see, e.g.,
Section 11.2.2). The "key_id" extension allows applications to add
an explicit, application-defined identifier to a KeyPackage.

```
opaque key_id<0..2^16-1>;
```

7.4. Parent Hash

The "parent_hash" extension serves to bind a KeyPackage to all the
nodes above it in the group's ratchet tree. This enforces the tree
invariant, meaning that malicious members can't lie about the state
of the ratchet tree when they send Welcome messages to new members.

```
opaque parent_hash<0..255>;
```

This extension MUST be present in all Updates that are sent as part
of a Commit message. If the extension is present, clients MUST
verify that "parent_hash" matches the hash of the leaf's parent node
when represented as a ParentNode struct.

7.5. 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 that represents the contents of the group's
   ratchet tree and the members' KeyPackages.

   The hash of a tree is the hash of its root node, which we define
   recursively, starting with the leaves.

   Elements of the ratchet tree are called "Node" objects and the leaves
   contain an optional "KeyPackage", while the parents contain an
   optional "ParentNode".

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

struct {
    HPKEPublicKey public_key;
    uint32 unmerged_leaves<0..2^32-1>;
    opaque parent_hash<0..255>;
} ParentNode;
```

   When computing the hash of a parent node, the "ParentNodeHashInput"
   structure is used:

```
struct {
    uint32 node_index;
    optional<ParentNode> parent_node;
    opaque left_hash<0..255>;
    opaque right_hash<0..255>;
} ParentNodeHashInput;
```

   The "left_hash" and "right_hash" fields hold the hashes of the node's
   left and right children, respectively. When computing the hash of a
   leaf node, the hash of a "LeafNodeHashInput" object is used:

```
struct {
    uint32 node_index;
    optional<KeyPackage> key_package;
} LeafNodeHashInput;
```

Note that the "node_index" field contains the index of the leaf among
the nodes in the tree, not its index among the leaves; "node_index =
2 * leaf_index".

[7.6](#). Group State

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

```
struct {
    opaque group_id<0..255>;
    uint64 epoch;
    opaque tree_hash<0..255>;
    opaque confirmed_transcript_hash<0..255>;
    Extension extensions<0..2^32-1>;
} 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 key.

* 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 7.5](#).

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

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 11.1](#). 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" is updated with the data for an
   MLSPlaintext message encoding a Commit message in two parts:

```
struct {
    opaque group_id<0..255>;
    uint64 epoch;
    Sender sender;
    ContentType content_type = commit;
    Commit commit;
    opaque signature<0..2^16-1>;
} MLSPlaintextCommitContent;

struct {
    MAC confirmation_tag;
} MLSPlaintextCommitAuthData;

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

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

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

Thus the "confirmed_transcript_hash" field in a GroupContext object
represents a transcript over the whole history of MLSPlaintext Commit
messages, up to the confirmation tag field in the current
MLSPlaintext message. The confirmation tag is then included in the
transcript for the next epoch. The interim transcript hash is passed
to new members in the GroupInfo struct, and enables existing members
to incorporate a Commit message into the transcript without having to
store the whole MLSPlaintextCommitAuthData structure.

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

7.7. Update Paths

As described in Section 11.2, each MLS Commit message may optionally
transmit a KeyPackage leaf and node values along its direct path.
The path contains a public key and encrypted secret value for all
intermediate nodes in the path above the leaf. 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<0..2^16-1>;
    opaque ciphertext<0..2^16-1>;
} HPKECiphertext;

struct {
    HPKEPublicKey public_key;
    HPKECiphertext encrypted_path_secret<0..2^32-1>;
} UpdatePathNode;

struct {
    KeyPackage leaf_key_package;
    UpdatePathNode nodes<0..2^32-1>;
} 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, "")
ciphertext = context.Seal(group_context, 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 [I-D.irtf-cfrg-hpke].

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

8. 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)
```

Where KDFLabel is specified as:

```
struct {
    uint16 length = Length;
    opaque label<7..255> = "mls10 " + Label;
    opaque context<0..2^32-1> = Context;
} KDFLabel;

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

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 IKM
   argument from the left

* DeriveSecret takes its Secret argument from the incoming arrow

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:

```
                      init_secret_[n-1]
                            |
                            V
       commit_secret -> KDF.Extract = joiner_secret
                            |
                            V
                    Derive-Secret(., "member")
                            |
                            V
psk_secret (or 0) -> KDF.Extract = member_secret
                            |
                            +--> Derive-Secret(., "welcome")
                            |      = welcome_secret
                            |
                            V
                    ExpandWithLabel(., "epoch", GroupContext_[n], KDF.Nh)
                            |
                            V
                     epoch_secret
                            |
                            +--> Derive-Secret(., <label>)
                            |      = <secret>
                            |
                            V
                    Derive-Secret(., "init")
                            |
                            V
                    init_secret_[n]
```

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

```
+========================+==================+
| Secret                 | Label            |
+========================+==================+
| sender_data_secret     | "sender data"    |
+------------------------+------------------+
| encryption_secret      | "encryption"     |
+------------------------+------------------+
| exporter_secret        | "exporter"       |
+------------------------+------------------+
| authentication_secret  | "authentication" |
+------------------------+------------------+
| external_secret        | "external"       |
+------------------------+------------------+
| confirmation_key       | "confirm"        |
+------------------------+------------------+
| membership_key         | "membership"     |
+------------------------+------------------+
| resumption_secret      | "resumption"     |
+------------------------+------------------+
```

Table 2

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
"PublicGroupState" struct in order to allow non-members to join the
group using an external commit.

8.1. 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 PublicGroupState and
an external Commit to initialize their copy of the key schedule for
the new epoch.

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

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

8.2. 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.

Each PSK in MLS has a type that designates how it was provisioned.
External PSKs are provided by the application, while recovery and re-
init 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 group state. In particular, in addition to external PSK types,
a PSK derived from within MLS may be used in the following cases:

* Re-Initialization: If during the lifetime of a group, the group
   members decide to switch to a more secure ciphersuite or newer
   protocol version, a PSK can be used to carry entropy from the old
   group forward into a new group with the desired parameters.

* Branching: A PSK may be used to bootstrap a subset of current
   group members into a new group. This applies if a subset of
   current group members wish to branch based on the current group
   state.

The injection of one or more PSKs into the key schedule is signaled
in two ways: 1) as a "PreSharedKey" proposal, and 2) in the
"GroupSecrets" object of a Welcome message sent to new members added
in that epoch.

```
   enum {
     reserved(0),
     external(1),
     reinit(2),
     branch(3),
     (255)
   } PSKType;

   struct {
     PSKType psktype;
     select (PreSharedKeyID.psktype) {
       case external:
         opaque psk_id<0..255>;

       case reinit:
         opaque psk_group_id<0..255>;
         uint64 psk_epoch;

       case branch:
         opaque psk_group_id<0..255>;
         uint64 psk_epoch;
     }
     opaque psk_nonce<0..255>;
   } PreSharedKeyID;

   struct {
       PreSharedKeyID psks<0..2^16-1>;
   } 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_secret" 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_input_[i] = KDF.Extract(0, psk_[i])
psk_secret_[i] = ExpandWithLabel(psk_input_[i], "derived psk", PSKLabel,
KDF.Nh)
psk_secret    = psk_secret_[0] || ... || psk_secret_[n-1]
```

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.

8.3. 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. Nodes are also assigned
an index according to their position in the array representation of
the tree (described in Appendix A). If N is a node index in the
Secret Tree then left(N) and right(N) denote the children of N (if
they exist).

The secret of any other node in the tree is derived from its parent's
secret using a call to DeriveTreeSecret:

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

Where TreeContext is specified as:

```
struct {
    uint32 node = Node;
    uint32 generation = Generation;
} TreeContext;
```

If N is a node index in the Secret Tree then the secrets of the
children of N are defined to be:

```
tree_node_[N]_secret
        |
        |
        +--> DeriveTreeSecret(., "tree", left(N), 0, KDF.Nh)
        |     = tree_node_[left(N)]_secret
        |
        +--> DeriveTreeSecret(., "tree", right(N), 0, KDF.Nh)
              = tree_node_[right(N)]_secret
```

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 8.4. The root of each
ratchet is computed as:

```
tree_node_[N]_secret
         |
         |
         +--> DeriveTreeSecret(., "handshake", N, 0, KDF.Nh)
         |     = handshake_ratchet_secret_[N]_[0]
         |
         +--> DeriveTreeSecret(., "application", N, 0, KDF.Nh)
               = application_ratchet_secret_[N]_[0]
```

8.4. Encryption Keys

As described in Section 9, 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 8.3. 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 index
of the sender's leaf in the ratchet tree and the current position in
the ratchet. In particular, the node index of the sender's leaf in
the ratchet tree is the same as the node index of the leaf in the
Secret Tree used to initialize the sender's ratchet.

```
ratchet_secret_[N]_[j]
      |
      +--> DeriveTreeSecret(., "nonce", N, j, AEAD.Nn)
      |      = ratchet_nonce_[N]_[j]
      |
      +--> DeriveTreeSecret(., "key", N, j, AEAD.Nk)
      |      = ratchet_key_[N]_[j]
      |
      V
DeriveTreeSecret(., "secret", N, j, KDF.Nh)
= ratchet_secret_[N]_[j+1]
```

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

8.5. 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 node index N 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", "member_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 index N,

  * the first j secrets in the application data ratchet of node index
     N and

  * "application_ratchet_nonce_[N]_[j]" and
     "application_ratchet_key_[N]_[j]".

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

```
        G
      /   \
     /       \
    E         F
   / \       / \
  A   B     C   D
                 / \
              HR0  AR0 -+- K0
                    |    |
                    |    +- N0
                    |
                  AR1 -+- K1
                    |    |
                    |    +- N1
                    |
                  AR2
```
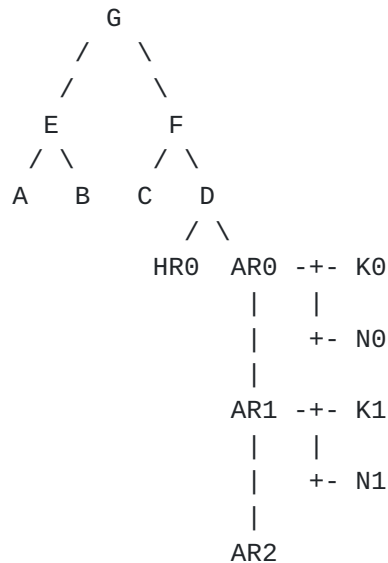
  Then if a client uses key K1 and nonce N1 during epoch n then it must
  consume (at least) values G, F, D, AR0, AR1, K1, N1 as well as the
  key schedule secrets used to derive G (the "encryption_secret"),
  namely "init_secret" of epoch n-1 and "commit_secret",
  "joiner_secret", "member_secret", "epoch_secret" 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.

8.6. 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.

8.7. Resumption Secret

The main MLS key schedule provides a "resumption_secret" which can
provide extra security in some cross-group operations.

The application SHOULD specify an upper limit on the number of past
epochs for which the "resumption_secret" may be stored.

There are two ways in which a "resumption_secret" can be used: to re-
initialize the group with different parameters, or to create a sub-
group of an existing group as detailed in Section 8.2.

Resumption keys are distinguished from exporter keys in that they
have specific use inside the MLS protocol, whereas the use of
exporter secrets may be decided by an external application. They are
thus derived separately to avoid key material reuse.

8.8. State Authentication Keys

The main MLS key schedule provides a per-epoch
"authentication_secret". If one of the parties is being actively
impersonated by an attacker, their "authentication_secret" will
differ from that of the other group members. Thus, members of a
group MAY use their "authentication_secrets" within an out-of-band
authentication protocol to ensure that they share the same view of
the group.

9. 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 two main structures involved are MLSPlaintext and MLSCiphertext.
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),
    application(1),
    proposal(2),
    commit(3),
    (255)
} ContentType;

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

struct {
    SenderType sender_type;
    uint32 sender;
} Sender;

struct {
    opaque mac_value<0..255>;
} MAC;

struct {
    opaque group_id<0..255>;
    uint64 epoch;
    Sender sender;
    opaque authenticated_data<0..2^32-1>;

    ContentType content_type;
    select (MLSPlaintext.content_type) {
        case application:
          opaque application_data<0..2^32-1>;

        case proposal:
          Proposal proposal;
```

```
        case commit:
          Commit commit;
    }

    opaque signature<0..2^16-1>;
    optional<MAC> confirmation_tag;
    optional<MAC> membership_tag;
} MLSPlaintext;

struct {
    opaque group_id<0..255>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<0..2^32-1>;
    opaque encrypted_sender_data<0..255>;
    opaque ciphertext<0..2^32-1>;
} MLSCiphertext;
```

The field "confirmation_tag" MUST be present if "content_type" equals
commit. Otherwise, it MUST NOT be present.

External sender types are sent as MLSPlaintext, see [Section 11.1.7](#)
for their use.

The remainder of this section describes how to compute the signature
of an MLSPlaintext object and how to convert it to an MLSCiphertext
object for "member" sender types. The steps are:

*   Set group_id, epoch, content_type and authenticated_data fields
    from the MLSPlaintext object directly

*   Identify the key and key generation depending on the content type

*   Encrypt an MLSCiphertextContent for the ciphertext field using the
    key identified and MLSPlaintext object

*   Encrypt the sender data using a key and nonce derived from the
    "sender_data_secret" for the epoch and a sample of the encrypted
    MLSCiphertextContent.

Decryption is done by decrypting the sender data, then the message,
and then verifying the content signature.

The following sections describe the encryption and signing processes
in detail.

9.1. Content Authentication

   The "signature" field in an MLSPlaintext object is computed using the
   signing private key corresponding to the credential at the leaf of
   the tree indicated by the sender field. The signature covers the
   plaintext metadata and message content, which is all of MLSPlaintext
   except for the "signature", the "confirmation_tag" and
   "membership_tag" fields. If the sender is a member of the group, the
   signature also covers the GroupContext for the current epoch, so that
   signatures are specific to a given group and epoch.

```
struct {
    select (MLSPlaintextTBS.sender.sender_type) {
        case member:
            GroupContext context;
    }

    opaque group_id<0..255>;
    uint64 epoch;
    Sender sender;
    opaque authenticated_data<0..2^32-1>;

    ContentType content_type;
    select (MLSPlaintextTBS.content_type) {
        case application:
          opaque application_data<0..2^32-1>;

        case proposal:
          Proposal proposal;

        case commit:
          Commit commit;
    }
} MLSPlaintextTBS;
```

   The "membership_tag" field in the MLSPlaintext object authenticates
   the sender's membership in the group. For an MLSPlaintext with a
   sender type other than "member", this field MUST be omitted. For
   messages sent by members, it MUST be present and set to the following
   value:

```
struct {
  MLSPlaintextTBS tbs;
  opaque signature<0..2^16-1>;
  optional<MAC> confirmation_tag;
} MLSPlaintextTBM;
```

   membership_tag = MAC(membership_key, MLSPlaintextTBM);

Note that the "membership_tag" only needs to be computed for
MLSPlaintext messages that will be sent over the wire, and isn't
needed for those that will be encrypted and transmitted as
MLSCiphertext messages.

9.2. Content Encryption

The "ciphertext" field of the MLSCiphertext object is produced by
supplying the inputs described below to the AEAD function specified
by the ciphersuite in use. The plaintext input contains the content
and signature of the MLSPlaintext, plus optional padding. These
values are encoded in the following form:

```
struct {
    select (MLSCiphertext.content_type) {
        case application:
          opaque application_data<0..2^32-1>;

        case proposal:
          Proposal proposal;

        case commit:
          Commit commit;
    }

    opaque signature<0..2^16-1>;
    optional<MAC> confirmation_tag;
    opaque padding<0..2^16-1>;
} 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.

```
+-+-+-+-+----------..----+
|   Key Schedule Nonce  |
+-+-+-+-+----------..----+
            XOR
+-+-+-+-+----------..----+
| Guard |       0       |
+-+-+-+-+----------..----+
            ===
+-+-+-+-+----------..----+
| Encrypt/Decrypt Nonce |
+-+-+-+-+----------..----+
```

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<0..255>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<0..2^32-1>;
} MLSCiphertextContentAAD;
```

9.3. 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 {
    uint32 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 9.2](#).

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 all the fields of MLSCiphertext excluding "encrypted_sender_data":

```
struct {
    opaque group_id<0..255>;
    uint64 epoch;
    ContentType content_type;
} MLSSenderDataAAD;
```

When parsing a SenderData struct as part of message decryption, the
recipient MUST verify that the sender field represents an occupied
leaf in the ratchet tree. In particular, the sender index value MUST
be less than the number of leaves in the tree.

[10]. 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 11.1.1], [Section 11.2], and [Section 11.2.2].

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" extensions 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
      (where "0" represents an all-zero vector of size KDF.Nh):

      - 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: 0

      - Interim transcript hash: 0

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

   * For each member, construct an Add proposal from the KeyPackage for
      that member (see [Section 11.1.1](#))

   * Construct a Commit message that commits all of the Add proposals,
      in any order chosen by the creator (see [Section 11.2](#))

   * 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 11.2.2](#).

   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, the leaf secret from which the Commit is built, and the
   intermediate key pairs along the direct path to the root.

[10.1](#). Linking a New Group to an Existing Group

   A new group may be tied to an already existing group for the purpose
   of re-initializing the existing group, or to branch into a sub-group.
   Re-initializing an existing group may be used, for example, to
   restart the group with a different ciphersuite or protocol version.
   Branching may be used to bootstrap a new group consisting of a subset
   of current group members, based on the current group state.

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

10.1.1. Sub-group Branching

If a client wants to create a subgroup of an existing group, they MAY
choose to include a "PreSharedKeyID" in the "GroupSecrets" object of
the Welcome message choosing the "psktype" "branch", the "group_id"
of the group from which a subgroup is to be branched, as well as an
epoch within the number of epochs for which a "resumption_secret" is
kept.

11. 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

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.

11.1. Proposals

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

```
enum {
    reserved(0),
    add(1),
    update(2),
    remove(3),
    psk(4),
    reinit(5),
    external_init(6),
    (255)
} 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;
    };
} Proposal;
```

On receiving an MLSPlaintext containing a Proposal, a client MUST
verify the signature on the enclosing MLSPlaintext. 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.

11.1.1. Add

An Add proposal requests that a client with a specified KeyPackage be
added to the group.

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

The proposer of the Add does not control where in the group's ratchet
tree the new member is added. Instead, the sender of the Commit
message chooses a location for each added member and states it in the
Commit message.

An Add is applied after being included in a Commit message. The
position of the Add in the list of proposals determines the leaf
index "index" where the new member will be added. For the first Add
in the Commit, "index" is the leftmost empty leaf in the tree, for
the second Add, the next empty leaf to the right, etc.

   * If necessary, extend the tree to the right until it has at least
      index + 1 leaves

   * For each non-blank intermediate node along the path from the leaf
      at position "index" to the root, add "index" to the
      "unmerged_leaves" list for the node.

   * Set the leaf node in the tree at position "index" to a new node
      containing the public key from the KeyPackage in the Add, as well
      as the credential under which the KeyPackage was signed

11.1.2. Update

   An Update proposal is a similar mechanism to Add with the distinction
   that it is the sender's leaf KeyPackage in the tree which would be
   updated with a new KeyPackage.

```
struct {
    KeyPackage key_package;
} Update;
```

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

   * Replace the sender's leaf KeyPackage with the one contained in the
      Update proposal

   * Blank the intermediate nodes along the path from the sender's leaf
      to the root

11.1.3. Remove

   A Remove proposal requests that the client at a specified index in
   the tree be removed from the group.

```
struct {
    uint32 removed;
} Remove;
```

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

   * Replace the leaf node at position "removed" with a blank node

   * Blank the intermediate nodes along the path from the removed leaf
      to the root

11.1.4. 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 8.2, where the order of the PSKs corresponds to
the order of the "PreSharedKey" proposals in the Commit.

11.1.5. ReInit

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

```
struct {
    opaque group_id<0..255>;
    ProtocolVersion version;
    CipherSuite cipher_suite;
    Extension extensions<0..2^32-1>;
} ReInit;
```

A member of the group applies a ReInit proposal by waiting for the
committer to send the Welcome message and by checking that the
"group_id" and the parameters of the new group corresponds to the
ones specified in the proposal. The Welcome message MUST specify
exactly one pre-shared key with "psktype = reinit", and with
"psk_group_id" and "psk_epoch" equal to the "group_id" and "epoch" of
the existing group after the Commit containing the "reinit" Proposal
was processed. The Welcome message may specify the inclusion of
other pre-shared keys with a "psktype" different from "reinit".

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.

11.1.6. ExternalInit

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

```
struct {
  opaque kem_output<0..2^16-1>;
} ExternalInit;
```

A member of the group applies an ExternalInit message by initializing
the next epoch using an init secret computed as described in
Section 8.1. The "kem_output" field contains the required KEM
output.

11.1.7. 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 "preconfigured" or "new_member" SenderType
in MLSPlaintext.

ReInit proposals can also be sent to the group by a "preconfigured"
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 "preconfigured" SenderType is reserved for signers that are pre-
provisioned to the clients within a group. If proposals with these
sender IDs are to be accepted within a group, the members of the
group MUST be provisioned by the application with a mapping between
these IDs and authorized signing keys. Recipients MUST verify that
the MLSPlaintext carrying the Proposal message is validly signed with
the corresponding key. To ensure consistent handling of external
proposals, the application MUST ensure that the members of a group
have the same mapping and apply the same policies to external
proposals.

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

11.2. 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. Proposals supplied by value are included directly in the
Commit object. Proposals supplied by reference are specified by
including the hash of the MLSPlaintext in which the Proposal was
sent, using the hash function from the group's ciphersuite. For
proposals supplied by value, the sender of the proposal is the same
as the sender of the Commit. Conversely, proposals sent by people
other than the committer MUST be included by reference.

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

struct {
  ProposalOrRefType type;
  select (ProposalOrRef.type) {
    case proposal:  Proposal proposal;
    case reference: opaque hash<0..255>;
  }
} ProposalOrRef;

struct {
    ProposalOrRef proposals<0..2^32-1>;
    optional<UpdatePath> path;
} Commit;
```

A group member that has observed one or more 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. If there are
multiple proposals that apply to the same leaf, the committer chooses
one and includes only that one in the Commit, considering the rest
invalid. 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 for the same client, the committer again
chooses one to include and considers the rest invalid.

The Commit MUST NOT combine proposals sent within different epochs.
In the event that a valid proposal is omitted from the next Commit,
the sender of the proposal SHOULD retransmit it in the new 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.

The "path" field of a Commit message MUST be populated if the Commit
covers at least one Update or Remove proposal. The "path" field MUST
also be populated if the Commit covers no proposals at all (i.e., if
the proposals vector is empty). The "path" field MAY be omitted if
the Commit covers only Add proposals. In pseudocode, the logic for
validating a Commit is as follows:

```
hasUpdates = false
hasRemoves = false

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

    hasUpdates = hasUpdates || proposal.msg_type == update
    hasRemoves = hasRemoves || proposal.msg_type == remove

if len(commit.proposals) == 0 || hasUpdates || hasRemoves:
  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. An "add-only" Commit that references only Add proposals, in which
    the path is optional. Such a commit provides PCS with regard to
    the committer only if the path field is present.

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.

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
   empty "key_package" and "path" fields.

* Generate a provisional GroupContext object by applying the
   proposals referenced in the initial Commit object, as described in
   [Section 11.1](#). 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 a UpdatePath using the new
   tree. Any new member (from an add proposal) MUST be exluded from
   the resolution during the computation of the UpdatePath. The
   GroupContext for this operation uses the "group_id", "epoch",
   "tree_hash", and "confirmed_transcript_hash" values in the initial
   GroupContext object.

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

   - Apply the UpdatePath to the tree, as described in Section 5.5.
     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 the same length as a "path_secret" value would be.

* Generate a new KeyPackage for the Committer's own leaf, with a
   "parent_hash" extension. Store it in the ratchet tree and assign
   it to the "key_package" field in the Commit object.

* If one or more PreSharedKey proposals are part of the commit,
   derive the "psk_secret" as specified in Section 8.2, where the
   order of PSKs in the derivation corresponds to the order of
   PreSharedKey proposals in the "proposals" vector. Otherwise, set
   "psk_secret" to a zero-length octet string.

* Construct an MLSPlaintext object containing the Commit object.
   Sign the MLSPlaintext using the current epoch's GroupContext as
   context. Use the signature, the "commit_secret" and the
   "psk_secret" to advance the key schedule and compute the
   "confirmation_tag" value in the MLSPlaintext.

* Update the tree in the provisional state by applying the direct
   path

* Construct a GroupInfo reflecting the new state:

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

   - The confirmation_tag from the MLSPlaintext object

   - 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 11.2.2)

* 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 "psktype" "reinit" and with "psk_group_id" and "psk_epoch" corresponding to the current group and the epoch after the commit was processed.

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

* Verify that the "epoch" field of the enclosing MLSPlaintext message is equal to the "epoch" field of the current GroupContext object

* Verify that the signature on the MLSPlaintext message verifies using the public key from the credential stored at the leaf in the tree indicated by the "sender" field.

* Verify that all PSKs specified in any PreSharedKey proposals in the "proposals" vector are available.

* Generate a provisional GroupContext object by applying the proposals referenced in the initial Commit object, as described in [Section 11.1](#). 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 ratchet tree the provisional GroupContext, to update the
  ratchet tree and generate the "commit_secret":

  - Apply the UpdatePath to the tree, as described in Section 5.5,
    and store "key_package" at the Committer's leaf.

  - Verify that the KeyPackage has a "parent_hash" extension 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 the same length as a "path_secret" value
  would be.

* Update the new GroupContext's confirmed and interim transcript
  hashes using the new Commit.

* If the "proposals" vector contains any PreSharedKey proposals,
  derive the "psk_secret" as specified in Section 8.2, where the
  order of PSKs in the derivation corresponds to the order of
  PreSharedKey proposals in the "proposals" vector. Otherwise, set
  "psk_secret" to 0.

* Use the "commit_secret", the "psk_secret", the provisional
  GroupContext, and the init secret from the previous epoch to
  compute the 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
  MLSPlaintext object.

* If the above checks are successful, consider the updated
  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 and check that

  - The "version", "cipher_suite" and "extensions" fields of the
    new group corresponds to the ones in the "ReInit" proposal, and
    that the "version" is greater than or equal to that of the
    original group.

- The "psks" field in the Welcome message includes a
  "PreSharedKeyID" with "psktype" = "reinit", and "psk_epoch" and
  "psk_group_id" equal to the epoch and group ID of the original
  group after processing the Commit.

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

MLSPlaintext.confirmation_tag =
    MAC(confirmation_key, GroupContext.confirmed_transcript_hash)

11.2.1. 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

* interim transcript hash

* group extensions

* external public key

This information is aggregated in a "PublicGroupState" object as
follows:

"struct { CipherSuite cipher_suite; opaque group_id<0..255>; uint64
epoch; opaque tree_hash<0..255>; opaque
interim_transcript_hash<0..255>; Extension extensions<0..2^32-1>;
HPKEPublicKey external_pub; } PublicGroupState;"

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 11.3.

The information above are not deemed public data in general, but
applications can choose to make them available to new members in
order to allow External Commits.

External Commits work like regular Commits, with a few differences:

* External Commits MUST reference an Add Proposal that adds the
   issuing new member to the group

* External Commits MUST contain a "path" field (and is therefore a
   "full" Commit)

* 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_key_package" of the
   "path" field.

* An external commit MUST reference no more than one ExternalInit
   proposal, and the ExternalInit proposal MUST be supplied by value,
   not by reference. When processing a Commit, both existing and new
   members MUST use the external init secret as described in
   Section 8.1.

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

* If the Add Proposal is also issued by the new member, its member
   SenderType MUST be "new_member"

11.2.2. Welcoming New Members

The sender of a Commit message is responsible for sending a Welcome
message to any 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 randomly chosen by the sender. This key and
nonce are then encrypted to each new member using HPKE. In the same
encrypted package, the committer transmits the path secret for the

lowest node 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 group_id<0..255>;
  uint64 epoch;
  opaque tree_hash<0..255>;
  opaque confirmed_transcript_hash<0..255>;
  Extension extensions<0..2^32-1>;
  MAC confirmation_tag;
  uint32 signer_index;
  opaque signature<0..2^16-1>;
} GroupInfo;

struct {
  opaque path_secret<1..255>;
} PathSecret;

struct {
  opaque joiner_secret<1..255>;
  optional<PathSecret> path_secret;
  optional<PreSharedKeys> psks;
} GroupSecrets;

struct {
  opaque key_package_hash<1..255>;
  HPKECiphertext encrypted_group_secrets;
} EncryptedGroupSecrets;

struct {
  ProtocolVersion version = mls10;
  CipherSuite cipher_suite;
  EncryptedGroupSecrets secrets<0..2^32-1>;
  opaque encrypted_group_info<1..2^32-1>;
} 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
    "key_package_hash" 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
    "member_secret" and using that the "welcome_secret",
    "welcome_key", and "welcome_nonce". Use the key and nonce to
    decrypt the "encrypted_group_info" field.

```
welcome_secret = Derive-Secret(member_secret, "welcome")
welcome_nonce = KDF.Expand(welcome_secret, "nonce", nonce_length)
welcome_key = KDF.Expand(welcome_secret, "key", key_length)
```

*   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 at position "signer_index". If this
    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" extension in the KeyPackage
        containing the same value (if the child is a leaf).

    -   For each non-empty leaf node, verify the signature on the
        KeyPackage.

*   Identify a leaf in the "tree" array (any even-numbered node) whose
    "key_package" field is identical to the the KeyPackage. If no
    such field exists, return an error. Let "index" represent the
    index of this node among the leaves in the tree, namely the index
    of the node in the "tree" array divided by two.

*   Construct a new group state using the information in the GroupInfo
    object. The new member's position in the tree is "index", as
    defined above. In particular, the confirmed transcript hash for
    the new state is the "prior_confirmed_transcript_hash" in the
    GroupInfo object.

    -   Update the leaf at index "index" 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 leaves at "index"
        and at "GroupInfo.signer_index". 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.

11.3. 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:   KeyPackage key_package;
        case parent: ParentNode node;
    };
} Node;

optional<Node> ratchet_tree<1..2^32-1>;
```

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.

12. Extensibility

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 client capabilities and aspects of
    their participation in the group (once in the ratchet tree)

*   In the Welcome message, to tell new members of a group what
    parameters are being used by 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, clients advertise their capabilities in KeyPackage
extensions, the creator of the group expresses its choices for the
group in Welcome extensions, and the GroupContext confirms that all
members of the group have the same view of the group's extensions.

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
    KeyPackage for the new member contains extensions that are
    consistent with the group's extensions. For each extension in the
    GroupContext, the KeyPackage MUST have an extension of the same
    type, and the contents of the extension MUST be consistent with
    the value of the extension in the GroupContext, according to the
    semantics of the specific extension.

*   If any extension in a GroupInfo message is unrecognized (i.e., not
    contained in 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.

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.

13. Sequencing of State Changes

   Each Commit message is premised on a given starting state, indicated
   by the "epoch" field of the enclosing MLSPlaintext message. 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.

[13.1](#). 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.

[13.2](#). 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.

[14](#). 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.

14.1. 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.

Similarly to TLS 1.3, if padding is used, the MLS messages MUST be
padded with zero-valued bytes before AEAD encryption. Upon AEAD
decryption, the length field of the plaintext is used to compute the
number of bytes to be removed from the plaintext to get the correct
data. As the padding mechanism is used to improve protection against
traffic analysis, removal of the padding SHOULD be implemented in a
"constant-time" manner at the MLS layer and above layers to prevent
timing side-channels that would provide attackers with information on
the size of the plaintext. The padding length length_of_padding can

be chosen at the time of the message encryption by the sender.
Recipients can calculate the padding size from knowing the total size
of the ApplicationPlaintext and the length of the content.

14.2. 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.

14.3. 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.

15. 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.

15.1. 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.

15.2. 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
identity key.

15.3. 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 InitKeys 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.

15.4. InitKey Reuse

InitKeys are intended to be used only once. That is, once an InitKey
has been used to introduce the corresponding client to a group, it
SHOULD be deleted from the InitKey publication system. Reuse of
InitKeys can lead to replay attacks.

An application MAY allow for reuse of a "last resort" InitKey in
order to prevent denial of service attacks. Since an InitKey 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
InitKeys.

16. IANA Considerations

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

* MLS Ciphersuites (Section 16.1)

* MLS Extension Types (Section 16.2)

* MLS Credential Types (Section 16.3)

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 16.4 for additional information about
the MLS Designated Experts (DEs).

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

16.1. 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                              |
+===========+=======================================+
| MLS       | The string "MLS" followed by the major |
|           | and minor version, e.g.  "MLS10"      |
+-----------+---------------------------------------+
| LVL       | The security level                    |
+-----------+---------------------------------------+
| KEM       | The KEM algorithm used for HPKE in    |
|           | TreeKEM group 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 3

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 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 ciphersuite is defined

Initial contents:

Value	Name	Recommended	Reference
0x0000	RESERVED	N/A	RFC XXXX
0x0001	MLS10_128_DHKEMX25519_AES128GCM_SHA256_Ed25519	Y	RFC XXXX
0x0002	MLS10_128_DHKEMP256_AES128GCM_SHA256_P256	Y	RFC XXXX
0x0003	MLS10_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519	Y	RFC XXXX
0x0004	MLS10_256_DHKEMX448_AES256GCM_SHA512_Ed448	Y	RFC XXXX
0x0005	MLS10_256_DHKEMP521_AES256GCM_SHA512_P521	Y	RFC XXXX
0x0006	MLS10_256_DHKEMX448_CHACHA20POLY1305_SHA512_Ed448	Y	RFC XXXX
0xff00 - 0xffff	Reserved for Private Use	N/A	RFC XXXX

Table 4

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 [I-D.irtf-cfrg-hpke] [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                 |
+------+--------+--------+--------+--------+-----------------------+
|0x0005| 0x0012 | 0x0003 | 0x0002 | SHA512 | ecdsa_secp521r1_sha512 |
+------+--------+--------+--------+--------+-----------------------+
|0x0006| 0x0021 | 0x0003 | 0x0003 | SHA512 | ed448                 |
+------+--------+--------+--------+--------+-----------------------+
```

                         Table 5

The hash used for the MLS transcript hash is the one referenced in
the ciphersuite name. In the ciphersuites defined above, "SHA256"
and "SHA512" refer to the SHA-256 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 inlcude 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
"MLS10_128_HPKE25519_AES128GCM_SHA256_Ed25519" which uses
Curve25519 for key exchange, AES-128-GCM for HPKE, HKDF over
SHA2-256, AES for metadata masking, 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 16.

16.2. 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 messages

   - GI: 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   | capabilities | KP         | Y           | RFC XXXX  |
+----------+--------------+------------+-------------+-----------+
| 0x0002   | lifetime     | KP         | Y           | RFC XXXX  |
+----------+--------------+------------+-------------+-----------+
| 0x0003   | key_id       | KP         | Y           | RFC XXXX  |
+----------+--------------+------------+-------------+-----------+
| 0x0004   | parent_hash  | KP         | Y           | RFC XXXX  |
+----------+--------------+------------+-------------+-----------+
| 0x0005   | ratchet_tree | GI         | Y           | RFC XXXX  |
+----------+--------------+------------+-------------+-----------+
| 0xff00 - | Reserved for | N/A        | N/A         | RFC XXXX  |
| 0xffff   | Private Use  |            |             |           |
+----------+--------------+------------+-------------+-----------+
```

                                Table 6

16.3. MLS Credential Types

This registry lists identifiers for types of credentials that can be
used for authentication in 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 credential type

* Name: The name of the credential 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.

* Reference: The document where this extension 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 | N/A         | RFC XXXX  |
|                 | Private Use  |             |           |
+-----------------+--------------+-------------+-----------+
```

Table 7

16.4. 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 (mailto: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 (mailto: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.

[17](#). Contributors

* Joel Alwen

  Wickr

  joel.alwen@wickr.com

* Karthikeyan Bhargavan

  INRIA

  karthikeyan.bhargavan@inria.fr

* Cas Cremers

  University of Oxford

  cas.cremers@cs.ox.ac.uk

* 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

18. References

18.1. Normative References

   [I-D.irtf-cfrg-hpke]
              Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid
              Public Key Encryption", Work in Progress, Internet-Draft,
              draft-irtf-cfrg-hpke-06, 23 October 2020,
              <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hpke-
              06.txt>.

   [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/info/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/info/rfc2119>.

   [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/info/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/info/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/info/rfc8446>.

18.2. 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]   Husak, M., Cermak, M., Jirsik, T., and P. Celeda, "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]
            Omara, E., Beurdouche, B., Rescorla, E., Inguva, S., Kwon,
            A., and A. Duric, "The Messaging Layer Security (MLS)
            Architecture", Work in Progress, Internet-Draft, draft-
            ietf-mls-architecture-05, 26 July 2020,
            <http://www.ietf.org/internet-drafts/draft-ietf-mls-
            architecture-05.txt>.

[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-34,
            4 November 2019, <http://www.ietf.org/internet-drafts/
            draft-ietf-trans-rfc6962-bis-34.txt>.

[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/info/rfc8032>.

[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,
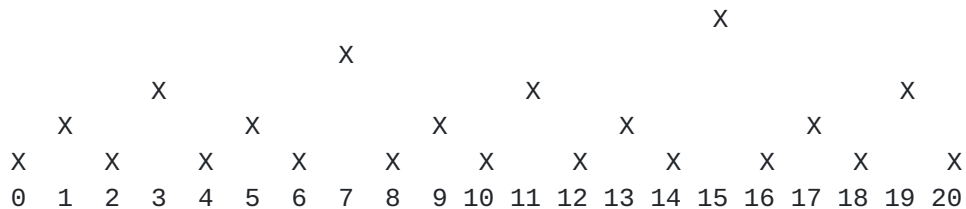   DOI 10.6028/nist.fips.180-4, July 2015,
            <https://doi.org/10.6028/nist.fips.180-4>.

[signal]   Perrin(ed), T. and M. Marlinspike, "The Double Ratchet
            Algorithm", 20 November 2016,
            <https://www.signal.org/docs/specifications/
            doubleratchet/>.

Appendix A. Tree Math

   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, an 11-element tree has
   the following structure:

```
                                                    X
                             X
             X                             X                             X
        X             X             X             X             X
     X      X      X      X      X      X      X      X      X      X      X
     0   1  2   3  4   5  6   7  8   9 10 11 12 13 14 15 16 17 18 19 20
```

   This allows us to compute relationships between tree nodes simply by
   manipulating indices, rather than having to maintain complicated
   structures in memory, even for partial trees. The basic rule is that
   the high-order bits of parent and child nodes have the following
   relation (where "x" is an arbitrary bit string):

   parent=01x => left=00x, right=10x

   The following python code demonstrates the tree computations
   necessary for MLS. Test vectors can be derived from the diagram
   above.

```python
# 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
```

Authors' Addresses

Richard Barnes
Cisco

Email: rlb@ipv.sx

Benjamin Beurdouche
Inria

Email: benjamin.beurdouche@inria.fr

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

Raphael Robert
Wire

Email: raphael@wire.com