Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 7, 2020

R. Barnes
Cisco
B. Beurdouche
Inria
J. Millican
Facebook
E. Omara
Google
K. Cohn-Gordon
University of Oxford
R. Robert
Wire

March 06, 2020

The Messaging Layer Security (MLS) Protocol
draft-ietf-mls-protocol-09

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

Copyright Notice

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-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 (*)

- o Redesign Welcome to include confirmation and use derived keys (*)
- o Remove ignored proposals (*)
- o Always include an Update with a Commit (*)
- o Add per-message entropy to guard against nonce reuse (*)
- o Use the same hash ratchet construct for both application and handshake keys (*)
- o Add more ciphersuites
- o Use HKDF to derive key pairs (*)
- o Mandate expiration of ClientInitKeys (*)
- o Add extensions to GroupContext and flesh out the extensibility story (*)
- o Rename ClientInitKey to KeyPackage

draft-08

- o Change ClientInitKeys so that they only refer to one ciphersuite (*)
- o Decompose group operations into Proposals and Commits (*)
- o Enable Add and Remove proposals from outside the group (*)
- o Replace Init messages with multi-recipient Welcome message (*)
- o Add extensions to ClientInitKeys for expiration and downgrade resistance (*)
- o Allow multiple Proposals and a single Commit in one MLSPlaintext (*)

draft-07

- o Initial version of the Tree based Application Key Schedule (*)
- o Initial definition of the Init message for group creation (*)
- o Fix issue with the transcript used for newcomers (*)
- o 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 (*)

o Added explicit key confirmation (*)

draft-01

o Initial description of the Message Protection mechanism. (*)

o Initial specification proposal for the Application Key Schedule using the per-participant chaining of the Application Secret design. (*)

o Initial specification proposal for an encryption mechanism to protect Application Messages using an AEAD scheme. (*)

o Initial specification proposal for an authentication mechanism of Application Messages using signatures. (*)

o Initial specification proposal for a padding mechanism to improving protection of Application Messages against traffic analysis. (*)

o Inversion of the Group Init Add and Application Secret derivations in the Handshake Key Schedule to be ease chaining in case we switch design. (*)

o Removal of the UserAdd construct and split of GroupAdd into Add and Welcome messages (*)

o Initial proposal for authenticating handshake messages by signing over group state and including group state in the key schedule (*)

o Added an appendix with example code for tree math

o Changed the ECIES mechanism used by TreeKEM so that it uses nonces generated from the shared secret

draft-00

o 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 clients identity and capabilities, and including an 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.

Identity Key: A long-lived 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 Messaging Service (MS) as described in [I-D.ietf-mls-architecture]. In particular, we assume the MS provides the following services:

- A long-term identity 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 12 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 themself. 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 Messaging Service.

```
                                                          Group
  A                B                C            Directory       Channel
  |                |                |                |              |
  | KeyPackageA    |                |                |              |
  |------------------------------------------------->|              |
  |                |                |                |              |
  |                | KeyPackageB    |                |              |
  |                |-------------------------------->|              |
  |                |                |                |              |
  |                |                | KeyPackageC    |              |
  |                |                |--------------->|              |
  |                |                |                |              |
```

When a client A wants to establish a group with B and C, it first downloads KeyPackages for B and C. It then initializes a group state containing only itself and uses the KeyPackages to compute Welcome and Add messages to add B and C, in a sequence chosen by A. The Welcome messages are sent directly to the new members (there is no need to send them to the group). The Add messages are broadcast to the group, and processed in sequence by B and C. Messages received before a client has joined the group are ignored. Only after A has received its Add messages back from the server does it update its state to reflect their addition.

```
                                                               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 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.

It is left to the application to determine a policy for regularly sending Update messages. This policy can be as strong as requiring an Update+Commit after each application message, or weaker, such as once every hour, day...

```
                                                          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.del(B)  |              |<----------------------------|
|              |              |state.del(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. If a subtree is full and it is not a subset of any other full subtree, then it is _maximal_.

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. In such a left-balanced tree, the "k-th" leaf node refers to the "k-th" leaf node in the tree when counting from the left, starting from 0.

(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.

For example, in the below tree:

o The direct path of C is (CD, ABCD, ABCDEFG)

o 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. 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 based on the following cryptographic primitives, defined by the ciphersuite in use:

- An HPKE ciphersuite, which specifies a Key Encapsulation Mechanism (KEM), an AEAD encryption scheme, and a hash function
- A Derive-Key-Pair function that produces an asymmetric key pair for the specified KEM from a symmetric secret

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.

- o The resolution of a non-blank node comprises the node itself, followed by its list of unmerged leaves, if any

- o The resolution of a blank leaf node is the empty list

- o 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:

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

- o The resolution of node 2 is the empty list []

- o 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

When performing a Commit, the leaf KeyPackage of the committer and its direct path to the root are updated with new secret values. The HPKE leaf public key within the KeyPackage MUST be a freshly generated value to provide post-compromise security.

The generator of the Commit starts by using the HPKE secret key "leaf_hpke_secret" associated with the new leaf KeyPackage (see Section 7) to compute "path_secret[0]" and generate a sequence of "path secrets", one for each ancestor of its leaf. That is, path_secret[0] is used for 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.

```
path_secret[0] = HKDF-Expand-Label(leaf_hpke_secret,
                                   "path", "", Hash.Length)
path_secret[n] = HKDF-Expand-Label(path_secret[n-1],
                                   "path", "", Hash.Length)
node_priv[n], node_pub[n] = Derive-Key-Pair(path_secret[n])
```

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

```
          G
         / \
        /   \
       /     \
      E       _
     / \     / \
    A   B   C   D
```

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

```
    path_secret[1] --> node_priv[1], node_pub[1]
         ^
         |
    path_secret[0] --> node_priv[0], node_pub[0]
         ^
         |
  leaf_hpke_secret
```

After the Commit, 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
```

5.5. Synchronizing Views of the Tree

The members of the group need to keep their views of the tree in sync and up to date. When a client commits a change to the tree (e.g., to add or remove a member), it transmits a handshake message containing a set of public values for intermediate nodes in the direct path of a leaf. The other members of the group can use these public values to update their view of the tree, aligning their copy of the tree to the sender's.

To perform an update for a path (a Commit), the sender broadcasts to the group the following information for each node in the direct path of the leaf, including the root:

- o The public key for the node

- o 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])

In this table, the value pk(X) represents the public key derived from the node secret X. The value E(K, S) represents the public-key encryption of the path secret S to the public key K.

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:

- A hash function

- A Diffie-Hellman finite-field group or elliptic curve group

- An AEAD encryption algorithm [RFC5116]

- A signature algorithm

The ciphersuite's Diffie-Hellman group is used to instantiate an HPKE [I-D.irtf-cfrg-hpke] instance for the purpose of public-key encryption. The ciphersuite must specify an algorithm "Derive-Key-Pair" that maps octet strings with length Hash.length to HPKE key pairs.

Ciphersuites are represented with the CipherSuite type. HPKE public keys are opaque values in a format defined by the underlying Diffie-Hellman protocol (see the Ciphersuites 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 15.1.

Depending on the Diffie-Hellman group of the ciphersuite, different rules apply to private key derivation and public key verification. For all ciphersuites defined in this document, the Derive-Key-Pair function begins by deriving a "key pair secret" of appropriate length, then converting it to a private key in the required group. The ciphersuite specifies the required length and the conversion.

```
key_pair_secret = HKDF-Expand-Label(path_secret, "key pair",
                                    "", KeyPairSecretLength)
```

6.1.1. X25519 and X448

For X25519, the key pair secret is 32 octets long. No conversion is required, since any 32-octet string is a valid X25519 private key. The corresponding public key is X25519(SHA-256(X), 9).

For X448, the key pair secret is 56 octets long. No conversion is required, since any 56-octet string is a valid X448 private key. The corresponding public key is X448(SHA-256(X), 5).

Implementations MUST use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementers use an alternative implementation of these elliptic curves, they MUST perform the additional checks specified in Section 7 of [RFC7748]

6.1.1.1. P-256 and P-521

For P-256, the key pair secret is 32 octets long. For P-521, the key pair secret is 66 octets long. In either case, the private key derived from a key pair secret is computed by interpreting the key pair secret as a big-endian integer.

ECDH calculations for these curves (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF

because MLS does not directly use this secret for anything other than for computing other secrets.)

Clients MUST validate remote public values by ensuring that the point is a valid point on the elliptic curve. The appropriate validation procedures are defined in Section 4.3.7 of [X962] and alternatively in Section 5.6.2.3 of [keyagreement]. This process consists of three steps: (1) verify that the value is not the point at infinity (O), (2) verify that for Y = (x, y) both integers are in the correct interval, (3) ensure that (x, y) is a correct solution to the elliptic curve equation. For these curves, implementers do not need to verify membership in the correct subgroup.

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:

- The public key of a signature key pair

- The identity of the holder of the private key

- The signature scheme that the holder will use to sign MLS messages

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

```
enum {
    basic(0),
    x509(1),
    (255)
} CredentialType;

struct {
    opaque identity<0..2^16-1>;
    SignatureScheme algorithm;
    SignaturePublicKey public_key;
} BasicCredential;

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

        case x509:
            opaque cert_data<1..2^24-1>;
    };
} Credential;
```

The SignatureScheme type represents a signature algorithm. Signature public keys are opaque values in a format defined by the signature scheme.

```
enum {
    ecdsa_secp256r1_sha256(0x0403),
    ed25519(0x0807),
    (0xFFFF)
} SignatureScheme;

opaque SignaturePublicKey<1..2^16-1>;
```

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 identity key can be updated throughout the lifetime of the group by sending a new KeyPackage with a new identity; the new identity 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 14.4). 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 identity 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 {
    mls10(0),
    (255)
} ProtocolVersion;

enum {
    invalid(0),
    supported_versions(1),
    supported_ciphersuites(2),
    expiration(3),
    key_id(4),
    parent_hash(5),
    (65535)
} 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<0..2^16-1>;
    opaque signature<0..2^16-1>;
} KeyPackage;
```

KeyPackage objects MUST contain at least two extensions, one of type "supported_versions" and one of type "supported_ciphersuites". These extensions allow MLS session establishment to be safe from downgrade attacks on these two parameters (as discussed in Section 9), 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. Supported Versions and Supported Ciphersuites

The "supported_versions" extension contains a list of MLS versions that are supported by the client. The "supported_ciphersuites" extension contains a list of MLS ciphersuites that are supported by the client.

```
ProtocolVersion supported_versions<0..255>;
CipherSuite supported_ciphersuites<0..255>;
```

These extensions MUST be always present in a KeyPackage.

7.2. Expiration

The "expiration" extension represents the time at which clients MUST consider this KeyPackage invalid. This time is represented as an absolute time, measured in seconds since the Unix epoch (1970-01-01T00:00:00Z). If a client receives a KeyPackage that contains an expiration extension at a time after its expiration time, then it MUST consider the KeyPackage invalid and not use it for any further processing.

```
uint64 expiration;
```

Applications that rely on "last resort" KeyPackages MAY set the expiration to its maximum value even though this is NOT RECOMMENDED. It is RECOMMENDED to rotate last resort keys at a pace chosen by the application even though they can have much longer lifetimes than other KeyPackages.

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 10.2.1). 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.

[[OPEN ISSUE: This scheme, in which the tree hash covers the parent hash, is designed to allow for more deniable deployments, since a signature by a member covers only its direct path. The other possible scheme, in which the parent hash covers the tree hash, provides better group agreement properties, since a member's signature covers the entire membership of the trees it is in. Further discussion is needed to determine whether the benefits to deniability justify the harm to group agreement properties, or whether there are alternative approaches to deniability that could be compatible with the other approach.]]

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>;

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

struct {
    NodeType node_type;
    select (Node.node_type) {
        case leaf:   optional<KeyPackage> key_package;
        case parent: optional<ParentNode> node;
    };
} Node;

struct {
    HPKEPublicKey public_key;
    uint32_t 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 leaf_index;
    optional<KeyPackage> key_package;
} LeafNodeHashInput;
```

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>;
    Extensions extensions<0..2^16-1>;
} 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 10.1. 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;
} MLSPlaintextCommitContent;

struct {
    opaque confirmation<0..255>;
    opaque signature<0..2^16-1>;
} MLSPlaintextCommitAuthData;

confirmed_transcript_hash_[n] =
    Hash(interim_transcript_hash_[n-1] ||
        MLSPlaintextCommitContent_[n]);

interim_transcript_hash_[n] =
    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 field in the current MLSPlaintext message. The confirmation and signature fields are then included in the transcript for the next epoch. The interim transcript hash is passed to new members in the WelcomeInfo struct, and enables existing members to incorporate a Commit message into the transcript without having to store the whole MLSPlaintextCommitAuthData structure.

When a new group is created, the "interim_transcript_hash" field is set to the zero-length octet string.

7.7. Direct Paths

As described in Section 10.2, each MLS Commit message needs to 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..2^16-1>;
} DirectPathNode;

struct {
    DirectPathNode nodes<0..2^16-1>;
} DirectPath;
```

The number of ciphertexts in the "encrypted_path_secret" vector MUST be equal to the length of the resolution of the corresponding copath node. Each ciphertext in the list is the encryption to the corresponding node in the resolution.

The HPKECiphertext values are computed as

```
kem_output, context = SetupBaseI(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 "SetupBaseI" 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.

7.8. Key Schedule

Group keys are derived using the HKDF-Extract and HKDF-Expand functions as defined in [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, Context, Length) =
    HKDF-Expand(Secret, HKDFLabel, Length)
```

Where HKDFLabel is specified as:

```
struct {
    opaque group_context<0..255> = Hash(GroupContext_[n]);
    uint16 length = Length;
    opaque label<7..255> = "mls10 " + Label;
    opaque context<0..2^32-1> = Context;
} HKDFLabel;

Derive-Secret(Secret, Label) =
    HKDF-Expand-Label(Secret, Label, "", Hash.length)
```

The Hash function used by HKDF is the ciphersuite hash algorithm. Hash.length is its output length in bytes. In the below diagram:

- o HKDF-Extract takes its salt argument from the top and its IKM argument from the left

- o Derive-Secret takes its Secret argument from the incoming arrow

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

- o The init secret from the previous epoch

- o The commit secret for the current epoch

- o 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] (or 0)
                     |
                     V
    PSK (or 0) -> HKDF-Extract = early_secret
                     |
               Derive-Secret(., "derived", "")
                     |
                     V
commit_secret -> HKDF-Extract = epoch_secret
                     |
                     +--> HKDF-Expand(., "mls 1.0 welcome", Hash.length)
                     |    = welcome_secret
                     |
                     +--> Derive-Secret(., "sender data", GroupContext_[n])
                     |    = sender_data_secret
                     |
                     +--> Derive-Secret(., "handshake", GroupContext_[n])
                     |    = handshake_secret
                     |
                     +--> Derive-Secret(., "app", GroupContext_[n])
                     |    = application_secret
                     |
                     +--> Derive-Secret(., "exporter", GroupContext_[n])
                     |    = exporter_secret
                     |
                     +--> Derive-Secret(., "confirm", GroupContext_[n])
                     |    = confirmation_key
                     |
                     V
               Derive-Secret(., "init", GroupContext_[n])
                     |
                     V
               init_secret_[n]
```

7.9. Pre-Shared Keys

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

At any epoch, including the initial state, an application can decide to synchronize the injection of a PSK into the MLS key schedule.

This mechanism can be used to improve security in the cases where having a full run of updates across members is too expensive or in the case where the external group key establishment mechanism provides stronger security against classical or quantum adversaries.

The security level associated with the PSK injected in the key schedule SHOULD match at least the security level of the ciphersuite in use in the group.

Note that, as a PSK may have a different lifetime than an update, it does not necessarily provide the same FS or PCS guarantees than a Commit message.

[[OPEN ISSUE: We have to decide if we want an external coordination via the application of a Handshake proposal.]]

7.10. Encryption Keys

As described in Section 8, 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 the content encryption is encrypted under AEAD using a random nonce and the "sender_data_key" which is derived from the "sender_data_secret" as follows:

```
sender_data_key =
    HKDF-Expand-Label(sender_data_secret, "sd key", "", key_length)
```

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. For application keys, the base secret is derived as described in Section 13.1. For handshake keys, base secrets are derived directly from the "handshake_secret".

```
application_secret_[sender]_[0] = astree_node_[N]_secret

handshake_secret_[sender]_[0] =
    HKDF-Expand-Label(handshake_secret, "hs", [sender], nonce_length)
```

The base secret of for each sender is used to initiate 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. In

particular, each key/nonce pair MUST NOT be used to encrypt more than one message.

Keys, nonces and secrets of ratchets are derived using Derive-App-Secret. 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 index of the sender's leaf in the ratchet tree is the same as the index of the leaf in the AS Tree used to initialize the sender's ratchet.

```
ratchet_secret_[N]_[j]
      |
      +--> Derive-App-Secret(., "nonce", N, j, AEAD.nonce_length)
      |    = ratchet_nonce_[N]_[j]
      |
      +--> Derive-App-Secret(., "key", N, j, AEAD.key_length)
      |    = ratchet_key_[N]_[j]
      |
      V
Derive-App-Secret(., "secret", N, j, Hash.length)
= ratchet_secret_[N]_[j+1]
```

Here, AEAD.nonce_length and AEAD.key_length denote the lengths in bytes of the nonce and key for the AEAD scheme defined by the ciphersuite. "ratchet" should be understood to mean "handshake" or "application" depending on the context.

7.11. 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) =
       HKDF-Expand-Label(Derive-Secret(exporter_secret, Label),
                         "exporter", Hash(Context), key_length)
```

The context used for the derivation of the "exported_value" MAY be empty while each application SHOULD provide a unique label as an input of the HKDF-Expand-Label for each 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 group operation is processed.

8. 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 SHOULD use MLSCiphertext to encode both application and 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 {
    invalid(0),
    application(1),
    proposal(2),
    commit(3),
    (255)
} ContentType;

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

struct {
    SenderType sender_type;
    uint32 sender;
} Sender;

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 confirmation<0..255>;
      }

      opaque signature<0..2^16-1>;
  } MLSPlaintext;

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

External sender types are sent as MLSPlaintext, see Section 10.1.4 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

- Randomly generate the sender_data_nonce field

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

- Encrypt an MLSSenderData object for the encrypted_sender_data field from MLSPlaintext and the key generation

- Generate and sign an MLSPlaintextTBS object from the MLSPlaintext object

- Encrypt an MLSCiphertextContent for the ciphertext field using the key identified, the signature, and MLSPlaintext object

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

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

8.1. Metadata Encryption

The "sender data" used to look up the key for the content encryption is encrypted under AEAD using the MLSCiphertext sender_data_nonce and the sender_data_key from the keyschedule. It 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 8.2.

The Additional Authenticated Data (AAD) for the SenderData ciphertext computation is its prefix in the MLSCiphertext, namely:

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

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.

8.2. Content Signing and Encryption

The signature field in an MLSPlaintext object is computed using the signing private key corresponding to the credential at the leaf in 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" field. The signature also covers the GroupContext for the current epoch, so that signatures are specific to a given group and epoch.

```
struct {
    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;
          opaque confirmation<0..255>;
    }
} MLSPlaintextTBS;
```

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 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 confirmation<0..255>;
    }

    opaque signature<0..2^16-1>;
    opaque padding<0..2^16-1>;
} MLSCiphertextContent;
```

The key and nonce used for the encryption of the message depend on the content type of the message. The sender chooses the handshake key for a handshake message or an unused generation from its (per-sender) application key chain for the current epoch, according to the type of message being encrypted.

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>;
    opaque sender_data_nonce<0..255>;
    opaque encrypted_sender_data<0..255>;
} MLSCiphertextContentAAD;
```

The ciphertext field of the MLSCiphertext object is produced by supplying these inputs to the AEAD function specified by the ciphersuite in use.

9. 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 10.1.1, Section 10.2, and Section 10.2.1.

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

- o Fetch KeyPackages for the members to be added, and selects a version and ciphersuite according to the capabilities of the members. To protect against downgrade attacks, the creator MUST use the "supported_versions" and "supported_ciphersuites" fields in these KeyPackages to verify that the chosen version and ciphersuite is the best option supported by all members.

- o Initialize a one-member group with the following initial values (where "0" represents an all-zero vector of size Hash.length):

  * 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: 0

- o For each member, construct an Add proposal from the KeyPackage for that member (see Section 10.1.1)

- o Construct a Commit message that commits all of the Add proposals, in any order chosen by the creator (see Section 10.2)

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

- o Transmit the Welcome message to the other new members

The recipient of a Welcome message processes it as described in Section 10.2.1.

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

A new member receiving a Welcome message can recognize group creation if the number of entries in the "members" array is equal to the number of leaves in the tree minus one. A client receiving a Welcome message SHOULD verify whether it is a newly created group, and if so, SHOULD verify that the above process was followed by reconstructing the Add and Commit messages and verifying that the resulting transcript hashes and epoch secret match those found in the Welcome message.

10. 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 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 0x0000000000000000. Each time a state transition occurs, the epoch number is incremented by one.

[[OPEN ISSUE: It would be better to have non-linear epochs, in order to tolerate forks in the history. There is a need to discuss whether we want to keep lexicographical ordering for the public value we serialize in the common framing, as it influence the ability of the DS to order messages.]]

10.1. Proposals

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

```
enum {
    invalid(0),
    add(1),
    update(2),
    remove(3),
    (255)
} ProposalType;

struct {
    ProposalType msg_type;
    select (Proposal.msg_type) {
        case add:    Add;
        case update: Update;
        case remove: Remove;
    };
} 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 using a ProposalID in a later Commit message.

10.1.1. 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 adds 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.

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

- o 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.

- o 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

10.1.2. 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

10.1.3. 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

10.1.4. 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 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 an "preconfigured" or "new_member" SenderType in MLSPlaintext.

The "new_member" SenderType is used for clients proposing that they themselves be added. For this ID type the sender value MUST be zero. Proposals with types other than Add MUST NOT be sent with this sender type. In such cases, 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. 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.

[[TODO: Should recognized external signers be added to some object that the group explicitly agrees on, e.g., as an extension to the GroupContext?]]

10.2. 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 identified by a ProposalID value, which contains the hash of the MLSPlaintext in which the Proposal was sent, using the hash function from the group's ciphersuite.

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

struct {
    ProposalID updates<0..2^16-1>;
    ProposalID removes<0..2^16-1>;
    ProposalID adds<0..2^16-1>;

    KeyPackage key_package;
    DirectPath 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.

[[OPEN ISSUE: This structure loses the welcome_info_hash, because new participants are no longer expected to have access to the Commit message adding them to the group. It might be we need to re-introduce this assumption, though it seems like the information confirmed by the welcome_info_hash is confirmed at the next epoch change anyway.]]

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

o Construct an initial Commit object with "updates", "removes", and "adds" fields populated from Proposals received during the current epoch, and empty "key_package" and "path" fields.

o Generate a provisional GroupContext object by applying the proposals referenced in the initial Commit object in the order provided, as described in Section 10.1. Add proposals are applied left to right: Each Add proposal is applied at the leftmost unoccupied leaf, or appended to the right edge of the tree if all leaves are occupied.

o Create a DirectPath using the new tree (which includes any new members). The GroupContext for this operation uses the "group_id", "epoch", "tree", and "prior_confirmed_transcript_hash" values in the initial GroupInfo object.

  * Assign this DirectPath to the "path" fields in the Commit and GroupInfo objects.

  * Apply the DirectPath 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.

o 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.

- o Construct an MLSPlaintext object containing the Commit object. Use the "commit_secret" to advance the key schedule and compute the "confirmation" value in the MLSPlaintext. Sign the MLSPlaintext using the current epoch's GroupContext as context.

- o Update the tree in the provisional state by applying the direct path

- o Construct a GroupInfo reflecting the new state:

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

  * The confirmation 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 "epoch_secret" for the new epoch (see Section 10.2.1)

- o 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

  * 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 for the common ancestor.

- o Construct a Welcome message from the encrypted GroupInfo object and the encrypted group secrets.

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

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

- o 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.

- o Generate a provisional GroupContext object by applying the proposals referenced in the commit object in the order provided,

as described in Section 10.1. Add proposals are applied left to right: Each Add proposal is applied at the leftmost unoccupied leaf, or appended to the right edge of the tree if all leaves are occupied.

- Process the "path" value using the ratchet tree the provisional GroupContext, to update the ratchet tree and generate the "commit_secret":

  * Apply the DirectPath 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.

- Update the new GroupContexts confirmed and interim transcript hashes using the new Commit.

- Use the "commit_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 MAC for this message, as described below, and verify that it is the same as the "confirmation" field in the MLSPlaintext object.

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

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

```
MLSPlaintext.confirmation =
    HMAC(confirmation_key, GroupContext.confirmed_transcript_hash)
```

HMAC [RFC2104] uses the Hash algorithm for the ciphersuite in use.

[[OPEN ISSUE: It is not possible for the recipient of a handshake message to verify that ratchet tree information in the message is accurate, because each node can only compute the secret and private key for nodes in its direct path. This creates the possibility that a malicious participant could cause a denial of service by sending a

handshake message with invalid values for public keys in the ratchet tree.]]

10.2.1. 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.

```
struct {
  opaque group_id<0..255>;
  uint64 epoch;
  optional<Node> tree<1..2^32-1>;
  opaque confirmed_transcript_hash<0..255>;
  opaque interim_transcript_hash<0..255>;
  Extensions extensions<0..2^16-1>;

  opaque confirmation<0..255>
  uint32 signer_index;
  opaque signature<0..2^16-1>;
} GroupInfo;

struct {
  opaque epoch_secret<1..255>;
  opaque path_secret<1..255>;
} 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;
```

In the description of the tree as a list of nodes, the "key_package" field for a node MUST be populated if and only if that node is a leaf in the tree.

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.

- From the "epoch_secret" in the decrypted GroupSecrets object, derive the "welcome_secret", "welcome_key", and "welcome_nonce". Use the key and nonce to decrypt the "encrypted_group_info" field.

```
welcome_secret = HKDF-Expand(epoch_secret, "mls 1.0 welcome", Hash.length)
welcome_nonce = HKDF-Expand(welcome_secret, "nonce", nonce_length)
welcome_key = HKDF-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.

  * 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.

  * 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" in the KeyPackage object.

  * 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 "epoch_secret" from the KeyPackage 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 MAC in the GroupInfo using the derived confirmation key and the "confirmed_transcript_hash" from the GroupInfo.

11. 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.

- A client joining a group MUST populate the GroupContext extensions with exactly the contents of the extensions field in the Welcome message. If any extension is unrecognized (i.e., not contained in the corresponding KeyPackage), then the client MUST reject the Welcome message and not join the group.

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.

[[OPEN ISSUE: Should we put bounds on what an extension can change? For example, should we make an explicit guarantee that as long as you're speaking MLS 1.0, the format of the KeyPackage will remain the same? (Analogous to the TLS invariant with regard to ClientHello.) If we are explicit that effectively arbitrary changes can be made to protocol behavior with the consent of the members, we will need to note that some such changes can undermine the security of the protocol.]]

12. Sequencing of State Changes

[[OPEN ISSUE: This section has an initial set of considerations regarding sequencing. It would be good to have some more detailed discussion, and hopefully have a mechanism to deal with this issue.]]

Each Commit message is premised on a given starting state, indicated in its "prior_epoch" field. 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:

- o Have the delivery service enforce a total order

- o 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.

12.1. 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.

12.2. 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.

13. 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.

13.1. Tree of Application Secrets

The application key schedule begins with the application secrets which are arranged in an "Application Secret Tree" or AS Tree for short; a left balanced binary tree with the same set of nodes and edges as the epoch's ratchet tree. Each leaf in the AS 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 AS Tree then left(N) and right(N) denote the children of N (if they exist).

Each node in the tree is assigned a secret. The root's secret is simply the application_secret of that epoch. (See Section 7.8 for the definition of application_secret.)

```
astree_node_[root]_secret = application_secret
```

The secret of any other node in the tree is derived from its parent's secret using a call to Derive-App-Secret.

```
Derive-App-Secret(Secret, Label, Node, Generation, Length) =
    HKDF-Expand-Label(Secret, Label, ApplicationContext, Length)
```

Where ApplicationContext is specified as:

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

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

```
astree_node_[N]_secret
        |
        |
        +--> Derive-App-Secret(., "tree", left(N), 0, Hash.length)
        |    = astree_node_[left(N)]_secret
        |
        +--> Derive-App-Secret(., "tree", right(N), 0, Hash.length)
             = astree_node_[right(N)]_secret
```

Note that fixing concrete values for GroupContext_[n] and application_secret completely defines all secrets in the AS Tree.

The secret in the leaf of the AS tree is used to initiate a symmetric hash ratchet, from which a sequence of single-use keys and nonces are derived, as described in Section 7.10.

13.2. Deletion Schedule

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

- S was used to encrypt or (successfully) decrypt a message, or if

- a key, nonce, or secret derived from S has been consumed. (This goes for values derived via Derive-Secret as well as HKDF-Expand-Label.)

Here, S may be the "init_secret", "commit_secret", "epoch_secret", "application_secret" as well as any secret in the AS 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 a message using the j-th key and nonce in the i-th ratchet. Then, for that member, at least the following values have been consumed and MUST be deleted:

- the "init_secret", "commit_secret", "epoch_secret", "application_secret" of that epoch,

- all node secrets in the AS Tree on the path from the root to the leaf with index i,

- the first j secrets in the i-th ratchet and

- "application_[i]_[j]_key" and "application_[i]_[j]_nonce".

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

```
       G
     /   \
    /     \
   E       F
  / \     / \
A0  B0  C0  D0 -+- KD0
            |   |
            |   +- ND0
            |
            D1 -+- KD1
            |   |
            |   +- ND1
            |
            D2
```

Then if a client uses key KD1 and nonce ND1 during epoch n then it must consume (at least) values G, F, D0, D1, KD1, ND1 as well as the "commit_secret" and init_secret used to derive G (the application_secret). The client MAY retain (not consume) the values KD0 and ND0 to allow for out-of-order delivery, and SHOULD retain D2 to allow for processing future messages.

13.3. Further 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 application_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.

[[OPEN ISSUE: At the moment there is no contributivity of Application secrets chained from the initial one to the next generation of Epoch secret. While this seems safe because cryptographic operations using the application secrets can't affect the group init_secret, it remains to be proven correct.]]

13.4. 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.

[[TODO: A preliminary formal security analysis has yet to be performed on this authentication scheme.]]

[[OPEN ISSUE: Should the padding be required for handshake messages ? Can an adversary get more than the position of a participant in the tree without padding ? Should the base ciphertext block length be negotiated or is is reasonable to allow to leak a range for the length of the plaintext by allowing to send a variable number of ciphertext blocks ?]]

13.5. 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.

[[TODO: Describe here or in the Architecture spec the details. Depending on which Secret or key is kept alive, the security guarantees will vary.]]

14. 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.

14.1. Confidentiality of the Group Secrets

Group secrets are derived from (i) previous group secrets, and (ii) the root key of a ratcheting tree. Only group members know their leaf private key in the group, therefore, the root key of the group's ratcheting tree is secret and thus so are all values derived from it.

Initial leaf keys are known only by their owner and the group creator, because they are derived from an authenticated key exchange protocol. Subsequent leaf keys are known only by their owner. [[TODO: or by someone who replaced them.]]

Note that the long-term identity keys used by the protocol MUST be distributed by an "honest" authentication service for clients to authenticate their legitimate peers.

14.2. Authentication

There are two forms of authentication we consider. The first form considers authentication with respect to the group. That is, the group members can verify that a message originated from one of the members of the group. This is implicitly guaranteed by the secrecy of the shared key derived from the ratcheting trees: if all members of the group are honest, then the shared group key is only known to the group members. By using AEAD or appropriate MAC with this shared key, we can guarantee that a member in the group (who knows the shared secret key) has sent a message.

The second form considers authentication with respect to the sender, meaning the group members can verify that a message originated from a particular member of the group. This property is provided by digital signatures on the messages under identity keys.

[[OPEN ISSUE: Signatures under the identity keys, while simple, have the side-effect of precluding deniability. We may wish to allow other options, such as (ii) a key chained off of the identity key, or (iii) some other key obtained through a different manner, such as a pairwise channel that provides deniability for the message contents.]]

14.3. Forward and post-compromise security

Message encryption keys are derived via a hash ratchet, which provides a form of forward secrecy: learning a message key does not reveal previous message or root keys. Post-compromise security is

provided by Commit operations, in which a new root key is generated from the latest ratcheting tree. If the adversary cannot derive the updated root key after an Commit operation, it cannot compute any derived secrets.

In the case where the client could have been compromised (device loss...), the client SHOULD signal the delivery service to expire all the previous KeyPackages and publish fresh ones for PCS.

14.4. 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.

15. IANA Considerations

This document requests the creation of the following new IANA registries: MLS Ciphersuites (Section 15.1). All of these registries should be under a heading of "Message Layer Security", and assignments are made via the Specification Required policy [RFC8126]. See Section 15.2 for additional information about the MLS Designated Experts (DEs).

15.1. 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 two 8bit octets:

```
uint8 CipherSuite[2];
```

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 KDF
SIG	The Signature algorithm used for message authentication

This specification defines the following ciphersuites for use with MLS 1.0.

Description	Value
MLS10_128_DHKEMX25519_AES128GCM_SHA256_Ed25519	{ 0x00,0x01 }
MLS10_128_DHKEMP256_AES128GCM_SHA256_P256	{ 0x00,0x02 }
MLS10_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519	{ 0x00,0x03 }
MLS10_256_DHKEMX448_AES256GCM_SHA512_Ed448	{ 0x00,0x04 }
MLS10_256_DHKEMP521_AES256GCM_SHA512_P521	{ 0x00,0x05 }
MLS10_256_DHKEMX448_CHACHA20POLY1305_SHA512_Ed448	{ 0x00,0x06 }

The KEM/DEM constructions used for HPKE are defined by [I-D.irtf-cfrg-hpke]. The corresponding AEAD algorithms AEAD_AES_128_GCM and AEAD_AES_256_GCM, are defined in [RFC5116]. AEAD_CHACHA20_POLY1305 is defined in [RFC7539]. The corresponding hash algorithms are 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, HKDF over SHA2-256 and AES-128-GCM for HPKE, and AES-128-GCM with Ed25519 for symmetric encryption and signatures.

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

New ciphersuite values are assigned by IANA as described in Section 15.

15.2. MLS Designated Expert Pool

[[OPEN ISSUE: pick DE mailing address. Maybe mls-des@ or mls-de-pool.]]

Specification Required [RFC8126] registry requests are registered after a three-week review period on the MLS DEs' mailing list: TBD@ietf.org [1], 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 [2] 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.

16. 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

- Srinivas Inguva
  Twitter
  singuva@twitter.com

- 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

17. References

17.1. Normative References

[I-D.irtf-cfrg-hpke]
Barnes, R. and K. Bhargavan, "Hybrid Public Key Encryption", draft-irtf-cfrg-hpke-02 (work in progress), November 2019.

[IEEE1363]
"IEEE Standard Specifications for Password-Based Public-Key Cryptographic Techniques", IEEE standard, DOI 10.1109/ieeestd.2009.4773330, n.d..

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/info/rfc5116>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <https://www.rfc-editor.org/info/rfc7539>.

[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>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.180-4, July 2015.

[X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

17.2. 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", 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.

[dhreuse] Menezes, A. and B. Ustaoglu, "On reusing ephemeral keys in Diffie-Hellman key agreement protocols", International Journal of Applied Cryptography Vol. 2, pp. 154, DOI 10.1504/ijact.2010.038308, 2010.

[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.

[HCJ16] Husak, M., Čermak, M., Jirsik, T., and P. Čeleda, "HTTPS traffic analysis and client identification using passive SSL/TLS fingerprinting", EURASIP Journal on Information Security Vol. 2016, DOI 10.1186/s13635-016-0030-7, February 2016.

[I-D.ietf-trans-rfc6962-bis] Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", draft-ietf-trans-rfc6962-bis-34 (work in progress), November 2019.

[keyagreement]
Barker, E., Chen, L., Roginsky, A., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar2, May 2013.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

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

17.3. URIs

[1] mailto:TBD@ietf.org

[2] mailto:iesg@ietf.org

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, a 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.

```
# The largest power of 2 less than n.  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
# level, 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):
    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.  The child of a leaf node is itself.
def left(x):
    k = level(x)
    if k == 0:
        return x

    return x ^ (0x01 << (k - 1))

# The right child of an intermediate node.  Depends on the size of
# the tree because the straightforward calculation can take you
# beyond the edge of the tree.  The child of a leaf node is itself.
def right(x, n):
    k = level(x)
    if k == 0:
```

```
            return x

        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, have
    # to walk back until the parent is within the range of the tree.
    def parent(x, n):
        if x == root(n):
            return x

        p = parent_step(x)
        while p >= node_width(n):
            p = parent_step(p)
        return p

    # The other child of the node's parent.  Root's sibling is itself.
    def sibling(x, n):
        p = parent(x, n)
        if x < p:
            return right(p, n)
        elif x > p:
            return left(p)

        return p

    # The direct path of a node, ordered from the root
    # down, not including the root or the terminal node
    def direct_path(x, n):
        d = []
        p = parent(x, n)
        r = root(n)
        while p != r:
            d.append(p)
            p = parent(p, n)
        return d

    # The copath of the node is the siblings of the nodes on its direct
    # path (including the node itself)
```

```
def copath(x, n):
    d = dirpath(x, n)
    if x != sibling(x, n):
        d.append(x)

    return [sibling(y, n) for y in d]

# The common ancestor of two leaves is the lowest node that is in the
# lowest-level node that is in the direct paths of both leaves.
def common_ancestor(x, y):
    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