Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on May 18, 2020.
Table of Contents

1. Introduction .............................................. 3
   1.1. Change Log ........................................ 4
2. Terminology ................................................ 7
3. Basic Assumptions .......................................... 7
4. Protocol Overview .......................................... 8
5. Ratchet Trees ........................................... 12
   5.1. Tree Computation Terminology ....................... 12
   5.2. Ratchet Tree Nodes .................................. 14
   5.3. Views of a Ratchet Tree .............................. 15
   5.4. Ratchet Tree Updates ................................. 16
   5.5. Synchronizing Views of the Tree ................. 17
6. Cryptographic Objects ..................................... 19
   6.1. Ciphersuites ........................................ 19
   6.1.2. P-256, SHA-256, and AES-128-GCM ................ 20
   6.2. Credentials .......................................... 21
   6.3. Tree Hashes .......................................... 23
   6.4. Group State ......................................... 24
   6.5. Direct Paths ........................................ 25
   6.6. Key Schedule ....................................... 26
   6.7. Encryption Keys .................................... 28
7. Initialization Keys ........................................ 29
   7.1. Supported Versions and Supported Ciphersuites .... 30
   7.2. Expiration ......................................... 31
8. Message Framing ........................................... 31
   8.1. Metadata Encryption ................................. 33
   8.2. Content Signing and Encryption .................... 34
9. Group Creation ........................................... 35
10. Group Evolution .......................................... 37
   10.1. Proposals ......................................... 37
   10.1.1. Add ............................................ 38
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-08

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

draft-07
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**

o  Reorder blanking and update in the Remove operation (*)

o  Rename the GroupState structure to GroupContext (*)

o  Rename UserInitKey to ClientInitKey

o  Resolve the circular dependency that **draft-05** introduced in the confirmation MAC calculation (*)

o  Cover the entire MLSPlaintext in the transcript hash (*)

**draft-05**

o  Common framing for handshake and application messages (*)

o  Handshake message encryption (*)

o  Convert from literal state to a commitment via the "tree hash" (*)

o  Add credentials to the tree and remove the "roster" concept (*)

o  Remove the secret field from tree node values

**draft-04**

o  Updating the language to be similar to the Architecture document

o  ECIES is now renamed in favor of HPKE (*)

o  Using a KDF instead of a Hash in TreeKEM (*)

**draft-03**

o  Added ciphersuites and signature schemes (*)

o  Re-ordered fields in UserInitKey to make parsing easier (*)

o  Fixed inconsistencies between Welcome and GroupState (*)
Added encryption of the Welcome message (*)

draft-02

Removed ART (*)

Allowed partial trees to avoid double-joins (*)

Added explicit key confirmation (*)

draft-01

Initial description of the Message Protection mechanism. (*)

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

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

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

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

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

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

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

Added an appendix with example code for tree math

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

draft-00

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

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

Client: An agent that uses this protocol to establish shared cryptographic state with other clients. A client is defined by the cryptographic keys it holds. An application or user may use one client per device (keeping keys local to each device) or sync keys among a user’s devices so that each user appears as a single client. In the scenario where multiple devices share the cryptographic material the client is referred to as a "virtual" client.

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.

Initialization Key: A short-lived HPKE key pair used to introduce a new client to a group. Initialization keys are published for each client (ClientInitKey).

Leaf Key: A secret that represents a member’s contribution to the group secret (so called because the members’ leaf keys are the leaves in the group’s ratchet tree).

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. These keys MUST be kept for the lifetime of the group as there is no mechanism in the protocol for changing a client’s identity key.
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 11 for further considerations.)

A directory to which clients can publish initialization keys and download initialization keys 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 and post-compromise secrecy with respect to compromise of a participant.

We describe the information stored by each client as a _state_, which includes both public and private data. An initial state, including an initial set of clients, is set up by a group creator using the _Init_ algorithm and based on information pre-published by clients. The creator sends the _Init_ message to the clients, who can then set up their own group state and derive the same shared secret. Clients then exchange messages to produce new shared states which are causally linked to their predecessors, forming a logical Directed Acyclic Graph (DAG) of states. Members can send _Update_ messages for post-compromise secrecy and new clients can be added or existing members removed from the group.

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 ClientInitKey objects to a directory provided to the Messaging Service.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Directory</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientInitKeyA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ClientInitKeyB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ClientInitKeyC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a client A wants to establish a group with B and C, it first downloads ClientInitKeys for B and C. It then initializes a group state containing only itself and uses the ClientInitKeys 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 broadcasted 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.
Subsequent additions of group members proceed in the same way. Any member of the group can download a ClientInitKey for a new client and broadcast an Add message that the current group can use to update their state and the new client can use to initialize its state.

To enforce forward secrecy and post-compromise security of messages, each member periodically updates its leaf secret which represents its contribution to the group secret. Any member of the group can send an Update at any time by generating a fresh leaf secret and sending an Update message that describes how to update the group secret with that new information. Once all members have processed this message, 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 the interval of time between Update messages. This policy could require a change for each message, or it could require sending an update every week or more.

Members are removed from the group in a similar way, as an update is effectively removing the old leaf from the group. Any member of the group can generate a Remove message that adds new entropy to the group state that is known to all members except the removed member. After other participants have processed this message, the group’s secrets will be unknown to the removed participant. Note that this does not necessarily imply that any member is actually allowed to evict other members; groups can layer authentication-based access control policies on top of these basic mechanisms.
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. Note that 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.

The direct path of a root is the empty list, and of any other node is the concatenation of that node with the direct path of its parent. The copath of a node is the list of siblings of nodes in its direct path. The frontier of a tree is the list of heads of the maximal full subtrees of the tree, ordered from left to right.

For example, in the below tree:

- The direct path of C is (C, CD, ABCD)
- The copath of C is (D, AB, EFG)
The frontier of the tree is (ABCD, EF, G)

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

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 Method (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, using the specified hash function.

Each node in a ratchet tree contains up to three values:

- A private key (only within 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)
The conditions under which each of these values must or must not be present are laid out in Section 5.3.

A node in the tree may also be _blank_, indicating that no value is present at that node. The _resolution_ of a node is an ordered list of non-blank nodes that collectively cover all non-blank descendants of the node.

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

For example, consider the following tree, where the "_" character represents a blank node:

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

```
0 1 2 3 4 5 6
```

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

- The resolution of node 5 is the list [CD, C]
- The resolution of node 2 is the empty list []
- The resolution of node 3 is the list [A, CD, C]

Every node, regardless of whether the node is blank or populated, has a corresponding _hash_ that summarizes the contents of the subtree below that node. The rules for computing these hashes are described in Section 6.3.

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 has full knowledge of the secret state of the tree, i.e., private keys associated to the nodes. Instead, each member is assigned to a leaf of the tree, which determines the set of secret state known to the member. 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 or equal to it.

In other words, if a node is not blank, then it holds a key pair, and the private key of that key pair is known only to members holding 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 Updates

Nodes in a tree are always updated along the direct path from a leaf to the root. The generator of the update chooses a random secret value "path_secret[0]", and generates a sequence of "path secrets", one for each node from the leaf to the root. That is, path_secret[0] is used for 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[n] = HKDF-Expand-Label(path_secret[n-1],
                               "path", "", Hash.Length)
node_secret[n] = HKDF-Expand-Label(path_secret[n],
                                 "node", "", Hash.Length)
node_priv[n], node_pub[n] = Derive-Key-Pair(node_secret[n])
```

For example, suppose there is a group with four members:
If the second participant (B) subsequently generates an update based on a secret X, then the sender would generate the following sequence of path secrets and node secrets:

```
  ^
path_secret[1] ---> node_secret[1]
  ^
X = path_secret[0] ---> node_secret[0]
```

After the update, the tree will have the following structure, where "ns[i]" represents the node_secret values generated as described above:

<pre>
ns[2]
 / \
| ns[1] _
| / \ X / \ 
A  ns[0] C  D
</pre>

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 proposes 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 leaf, the sender broadcasts to the group the following information for each node in the direct path of the leaf, as well as the root:

- The public key for the node
- Zero or more encrypted copies of the path secret corresponding to the node
The path secret value for a given node is encrypted for the subtree corresponding to the parent's non-updated child, i.e., 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. In particular, for the leaf node, there are no encrypted secrets, since a leaf node has no children.

The recipient of an 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 node_secret values.

2. Merge the updated path secrets into the tree.

   * Replace the public keys for nodes on the direct path with the received public keys.
   
   * For nodes where an updated path secret was computed in step 1, compute the corresponding node secret and node key pair and replace the values stored at the node with the computed values.
   
   * For all updated nodes, set the list of unmerged leaves to the empty list.

For example, in order to communicate the example update described in the previous section, the sender would transmit the following values:
<table>
<thead>
<tr>
<th>Public Key</th>
<th>Ciphertext(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk(ns[2])</td>
<td>E(pk(C), ps[2]), E(pk(D), ps[2])</td>
</tr>
<tr>
<td>pk(ns[1])</td>
<td>E(pk(A), ps[1])</td>
</tr>
<tr>
<td>pk(ns[0])</td>
<td></td>
</tr>
</tbody>
</table>

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]

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):

```c
enum {
    P256_SHA256_AES128GCM(0x0000),
    X25519_SHA256_AES128GCM(0x0001),
    (0xFFFF)
} CipherSuite;

opaque HPKEPublic<1..2^16-1>;
```
6.1.1. Curve25519, SHA-256, and AES-128-GCM

This ciphersuite uses the following primitives:

- **Hash function**: SHA-256
- **AEAD**: AES-128-GCM

When HPKE is used with this ciphersuite, it uses the following algorithms:

- **KEM**: 0x0002 = DHKEM(Curve25519)
- **KDF**: 0x0001 = HKDF-SHA256
- **AEAD**: 0x0001 = AES-GCM-128

Given an octet string X, the private key produced by the Derive-Key-Pair operation is SHA-256(X). (Recall that any 32-octet string is a valid Curve25519 private key.) The corresponding public key is X25519(SHA-256(X), 9).

Implementations SHOULD 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 SHOULD perform the additional checks specified in Section 7 of [RFC7748].

6.1.2. P-256, SHA-256, and AES-128-GCM

This ciphersuite uses the following primitives:

- **Hash function**: SHA-256
- **AEAD**: AES-128-GCM

When HPKE is used with this ciphersuite, it uses the following algorithms:

- **KEM**: 0x0001 = DHKEM(P-256)
- **KDF**: 0x0001 = HKDF-SHA256
- **AEAD**: 0x0001 = AES-GCM-128
Given an octet string $X$, the private key produced by the Derive-Key-Pair operation is SHA-256($X$), interpreted as a big-endian integer. The corresponding public key is the result of multiplying the standard P-256 base point by this integer.

P-256 ECDH calculations (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, e.g., 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 SHOULD be validated against the Authentication Service.

[[OPEN ISSUE: 1. SHOULD vs MUST. 2. A client that wants to update its identity key can perform the operation UNDER THIS CONDITION by adding a new version of herself using a new credential signed under a new IdentityKey, then performing a remove of the old leaf. This is fine as long as the credential binds to the same identity for the application. If this verification is not met, there is no authentication guarantee at the application layer anyway.]]
6.3. Tree Hashes

To allow group members to verify that they agree on the 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' credentials.

The hash of a tree is the hash of its root node, which we define recursively, starting with the leaves. The hash of a leaf node is the hash of a "LeafNodeHashInput" object:

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

struct {
    HPKEPublicKey public_key;
    Credential credential;
} LeafNodeInfo;

struct {
    uint8 hash_type = 0;
    optional<LeafNodeInfo> info;
} LeafNodeHashInput;
```

The "public_key" and "credential" fields represent the leaf public key and the credential for the member holding that leaf, respectively. The "info" field is equal to the null optional value when the leaf is blank (i.e., no member occupies that leaf).

Likewise, the hash of a parent node (including the root) is the hash of a "ParentNodeHashInput" struct:

```
struct {
    HPKEPublicKey public_key;
    uint32_t unmerged_leaves<0..2^32-1>;
} ParentNodeInfo;

struct {
    uint8 hash_type = 1;
    optional<ParentNodeInfo> info;
    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. The "public_key" field holds the hash of the public key stored at this node, represented as an "optional<HPKEPublicKey>" object, which is null if and only if the node is blank.

6.4. Group State

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

```c
struct {
  opaque group_id<0..255>;
  uint32 epoch;
  opaque tree_hash<0..255>;
  opaque confirmed_transcript_hash<0..255>;
} 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 6.3.
- The "confirmed_transcript_hash" field contains a running hash over the handshake 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:

```c
struct {
    opaque group_id<0..255>
    uint32 epoch;
    uint32 sender;
    ContentType content_type = commit;
    Proposal proposals<0..2^32-1>
    Commit commit;
} MLSPlaintextCommitContent;
```

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

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

```c
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 handshake 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.

### 6.5. Direct Paths

As described in Section 5.4, each MLS message needs to transmit node values along the direct path of a leaf. The path contains a public key for the leaf node, and a public key and encrypted secret value for intermediate nodes in the path. In both cases, the path is ordered from 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 length of the "encrypted_path_secret" vector MUST be zero for the first node in the path. For the remaining elements in the vector, 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

```plaintext
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.

6.6. 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:

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

- The init secret from the previous epoch
- The update secret for the current epoch
- The GroupContext object for current epoch

Given these inputs, the derivation of secrets for an epoch proceeds as shown in the following diagram:
init_secret_[n-1] (or 0)

update_secret -> HKDF-Extract = epoch_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(., "confirm", GroupContext_[n])
        | = confirmation_key
        V
    Derive-Secret(., "init", GroupContext_[n])
    V
init_secret_[n]

6.7. Encryption Keys

As described in Section 8, MLS encrypts three different types of information:

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

Each handshake message is encrypted using a key and a nonce derived from the handshake_secret for a specific sender to prevent two senders to perform in the following way:
handshake_nonce_[sender] =
    HKDF-Expand-Label(handshake_secret, "hs nonce", [sender], nonce_length)

handshake_key_[sender] =
    HKDF-Expand-Label(handshake_secret, "hs key", [sender], key_length)

Here the value [sender] represents the index of the member that will use this key to send, encoded as a uint32. Each sender maintains two "generation" counters, one for application messages and one for handshake messages. These counters are incremented by one each time the sender sends a message.

For application messages, a chain of keys is derived for each sender in a similar fashion. This allows forward secrecy at the level of application messages within and out of an epoch. A step in this chain (the second subscript) is called a "generation". The details of application key derivation are described in the Section 12.1 section below.

For handshake messages (Proposals and Commits), the same key is used for all messages, but the nonce is updated according to the generation of the message:

handshake_nonce_[sender]_[generation] = handshake_nonce_[sender] XOR encode_big_endian(generation)

where "encode_big_endian()" encodes the generation in a big-endian integer of the same size as the base handshake nonce.

7. Initialization Keys

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

A ClientInitKey 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 is intended to be stable throughout the lifetime of the group; there is no mechanism to change it. Init keys are intended to be used only once and SHOULD NOT be reused except in case of last resort. (See Section 13.4). Clients MAY generate and publish multiple ClientInitKey objects to support multiple ciphersuites. ClientInitKeys contain an identifier chosen by the client, which the client MUST ensure uniquely identifies a given ClientInitKey object among the set of ClientInitKeys created by this client.
The value for 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 ClientInitKey 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.

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

enum {
    invalid(0),
    supported_versions(1),
    supported_ciphersuites(2),
    expiration(3),
    (65535)
} ExtensionType;

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

struct {
    ProtocolVersion supported_version;
    opaque client_init_key_id<0..255>;
    CipherSuite cipher_suite;
    HPKEPublicKey init_key;
    Credential credential;
    Extension extensions<0..2^16-1>;
    opaque signature<0..2^16-1>;
} ClientInitKey;

ClientInitKey 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 ClientInitKey.

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>;

7.2.  Expiration

The "expiration" extension represents the time at which clients MUST consider this ClientInitKey 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 ClientInitKey that contains an expiration extension at a time after its expiration time, then it MUST consider the ClientInitKey invalid and not use it for any further processing.

uint64 expiration;

Note that as an extension, it is not required that any given ClientInitKey have an expiration time. In particular, applications that rely on "last resort" ClientInitKeys to ensure continued reachability may choose to omit the expiration extension from these keys, or give them much longer lifetimes than other ClientInitKeys.

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;

struct {
    opaque group_id<0..255>;
    uint32 epoch;
    uint32 sender;
    ContentType content_type;
    opaque authenticated_data<0..2^32-1>;

    select (MLSPlaintext.content_type) {
        case application:
            opaque application_data<0..2^32-1>;
    
        case proposal:
            Proposal proposals<1..2^32-1>;
    
        case commit:
            Proposal proposals<1..2^32-1>;
            Commit commit;
            opaque confirmation<0..255>;
    
    }

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

struct {
    opaque group_id<0..255>;
    uint32 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;

The remainder of this section describes how to compute the signature of an MLSPlaintext object and how to convert it to an MLSCiphertext object. The overall process is as follows:

  o Gather the required metadata:

    * Group ID
* Epoch
* Content Type
* Nonce
* Sender index
* Key generation

- Sign the plaintext metadata - the group ID, epoch, sender index, and content type - as well as the authenticated data and message content.
- Randomly generate sender_data_nonce and encrypt the sender information using it and the key derived from the sender_data_secret.
- Encrypt the content using a content encryption key identified by the metadata.

The group identifier, epoch, content_type and authenticated data fields are copied from the MLSPlaintext object directly. The content encryption process populates the ciphertext field of the MLSCiphertext object. The metadata encryption step populates the encrypted_sender_data field.

Decryption follows the same step in reverse: Decrypt the metadata, then the message and verify the content signature.

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:

```c
struct {
    uint32 sender;
    uint32 generation;
} MLSSenderData;
```

The Additional Authenticated Data (AAD) for the SenderData ciphertext computation is its prefix in the MLSCiphertext, namely:
struct {
    opaque group_id<0..255>;
    uint32 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, i.e., all fields 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>;
    uint32 epoch;
    uint32 sender;
    ContentType content_type;
    opaque authenticated_data<0..2^32-1>;

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

        case proposal:
            Proposal proposals<1..2^32-1>;

        case commit:
            Proposal proposals<1..2^32-1>;
            Commit commit;
            opaque confirmation<0..255>;
    }
} MLSPlaintextSignatureInput;

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 handshake:
            GroupOperation operation;
            opaque confirmation<0..255>;

        case application:
            opaque application_data<0..2^32-1>;
    }

    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.

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>;
    uint32 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:

- Fetch ClientInitKeys 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 ClientInitKeys to verify that the chosen version and ciphersuite is the best option supported by all members.

- Initialize a one-member group with the following initial values (where "0" represents an all-zero vector of size 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: 0x00000000
  * Tree hash: The root hash of the above ratchet tree
  * Confirmed transcript hash: 0
  * Interim transcript hash: 0
  * Init secret: 0

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

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

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

- Transmit the Welcome message to the other new members

The recipient of a Welcome message processes it as described in Section 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 HPKE
key and credential, 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 four-octet epoch values. When a new group is 
initialized, its initial state epoch 0x00000000. 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. ]]

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 ClientInitKey be added to the group.

struct {
    ClientInitKey init_key;
} 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.

- If necessary, extend the tree to the right until it has at least index + 1 leaves
For each intermediate node along the path from the leaf at position "index" to the root, add "index" to the "unmerged_leaves" list for the node.

Blank the path from the leaf at position "index" to the root

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

10.1.2. Update

An Update proposal requests that the sender’s leaf node in the tree be updated with a new HPKE public key.

```c
struct {
    HPKEPublicKey leaf_key;
} Update;
```

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

- Update the sender’s leaf node by replacing the HPKE public key with the public key 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.

```c
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 having a "sender" value in the range 0xFFFFFF00 - 0xFFFFFFFF.

The specific value 0xFFFFFFFF is reserved for clients proposing that they themselves be added. Proposals with types other than Add MUST NOT be sent with this sender index. In such cases, the MLSPlaintext MUST be signed with the private key corresponding to the ClientInitKey in the Add message. Recipients MUST verify that the MLSPlaintext carrying the Proposal message is validly signed with this key.

The remaining values 0xFFFFFF00 - 0xFFFFFFFFFE are reserved for signer 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 sender indices in this range 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.

A group member that has observed one or more Proposal messages 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 information is transmitted.
The sender of a Commit message MUST include in it all valid Proposals that the sender has received during the current epoch. Invalid Proposals include, for example, Proposals with an invalid signature or Proposals that are semantically inconsistent, such as a Remove proposal for an unoccupied leaf. The Commit MUST NOT combine Proposals sent within different epochs. Despite these requirements, it is still possible for a valid Proposal not to be covered by a Commit, e.g., because the sender of the Commit did not receive the Proposal. In such cases, the sender of the proposal can retransmit the Proposal in the new epoch.

Each proposal covered by the Commit is identified by a ProposalID structure. The "sender" field in this structure indicates the member of the group that sent the proposal (according to their index in the ratchet tree). The "hash" field contains the hash of the MLSPlaintext in which the Proposal was sent, using the hash function for the group's ciphersuite.

```
struct {
    uint32 sender;
    opaque hash<0..255>;
} ProposalID;
```

```
struct {
    ProposalID updates<0..2^16-1>;
    ProposalID removes<0..2^16-1>;
    ProposalID adds<0..2^16-1>;
    ProposalID ignored<0..2^16-1>;
    DirectPath path;
} Commit;
```

The sender of a Commit message MUST include in it all proposals that it has received during the current epoch. Proposals that recipients should implement are placed in the "updates", "removes", and "adds" vector, according to their type. Proposals that should not be implemented are placed in the "ignored" vector. For example, if two Update proposals are issued for the same leaf, then one of them (presumably the earlier one) should be ignored and the other (presumably the later) should be added to the "updates" vector.

[[ 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 reintroduce 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 applies a Commit message by taking the following steps:

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

2. 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.

3. 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.

4. Process the "path" value to update the ratchet tree referenced by the provisional GroupContext and generate the update secret:
   * Update the ratchet tree by replacing nodes in the direct path of the sender with the corresponding nodes in the path (see Section 6.5).
   * The update secret is the value "path_secret[n+1]" derived from the "path_secret[n]" value associated to the root node.

5. Use the update secret, the provisional GroupContext, and the init secret from the previous epoch to compute the epoch secret and derived secrets for the new epoch.

6. 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.

7. 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:

\[
\text{ MLSPlaintext.confirmation } = \text{ HMAC(confirmation_key, GroupContext.confirmed_transcript_hash) }
\]

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 {
    HPKEPublicKey public_key;
    uint32_t unmerged_leaves<0..2^32-1>;
    optional<Credential> credential;
} RatchetNode;

struct {
    // GroupContext inputs
    opaque group_id<0..255>;
    uint32 epoch;
    optional<RatchetNode> tree<1..2^32-1>;
    opaque confirmed_transcript_hash<0..255>;

    // Inputs to the next round of the key schedule
    opaque interim_transcript_hash<0..255>;
    opaque epoch_secret<0..255>;
    uint32 signer_index;
    opaque signature<0..255>;
} GroupInfo;

struct {
    opaque group_info_key<1..255>;
    opaque group_info_nonce<1..255>;
    opaque path_secret<1..255>;
} KeyPackage;

struct {
    opaque client_init_key_hash<1..255>;
    HPKECiphertext encrypted_key_package;
} EncryptedKeyPackage;

struct {
    ProtocolVersion version = mls10;
    CipherSuite cipher_suite;
    EncryptedKeyPackage key_packages<1..V>;
    opaque encrypted_group_info;
} Welcome;

In the description of the tree as a list of nodes, the "credential" field for a node MUST be populated if and only if that node is a leaf in the tree (i.e., a node with an even index).

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

- Identify an entry in the "key_packages" array where the "client_init_key_hash" value corresponds to one of this client’s
ClientInitKeys, using the hash indicated by the "cipher_suite" field. If no such field exists, or if the ciphersuite indicated in the ClientInitKey does not match the one in the Welcome message, return an error.

- Decrypt the "encrypted_key_package" using HPKE with the algorithms indicated by the ciphersuite and the HPKE public key in the ClientInitKey.

- Decrypt the "encrypted_group_info" field using the key and nonce in the decrypted KeyPackage object.

- 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.

- Identify a leaf in the "tree" array (i.e., an even-numbered node) whose "public_key" and "credential" fields are identical to the corresponding fields in the ClientInitKey. 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.

- Identify the lowest node at which the direct paths from "index" and "signer_index" overlap. Set private keys for that node and its parents up to the root of the tree, using the "path_secret" from the KeyPackage and following the algorithm in Section 5.4 to move up the tree.

11. 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 handshake message is premised on a given starting state, indicated in its "prior_epoch" field. If the changes implied by a handshake 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 handshake message, it is based on the most current state of the group. In practice, however, there is a risk that two members will generate handshake messages simultaneously, based on the same state.

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

- Have the delivery service enforce a total order
- Have a signal in the message that clients can use to break ties

As long as handshake 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 handshake 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 update messages could be applied one after the other without the Delivery Service having to reject any handshake message, which would make MLS more resilient regarding the concurrency of handshake 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, i.e. all updates must be executed before removes.

Regardless of how messages are kept in sequence, implementations MUST only update their cryptographic state when valid handshake messages are received. Generation of handshake messages MUST be stateless, since the endpoint cannot know at that time whether the change implied by the handshake message will succeed or not.

### 11.1. 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 resolve conflicts during race-conditions (when two members send a message at the same time), as the server doesn’t have any additional knowledge thanks to the confidentiality of the messages.

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.

11.2. 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.

12. 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.

12.1. 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 6.6 for the definition of application_secret.)

\[
\text{astree_node}_{[\text{root}]}_{\text{secret}} = \text{application_secret}
\]

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

\[
\text{Derive-App-Secret}(\text{Secret}, \text{Label}, \text{Node}, \text{Generation}, \text{Length}) = \text{HKDF-Expand-Label}(\text{Secret}, \text{Label}, \text{ApplicationContext}, \text{Length})
\]

Where ApplicationContext is specified as:

\[
\text{struct} \{
  \text{uint32 node} = \text{Node};
  \text{uint32 generation} = \text{Generation};
\} \text{ApplicationContext;
}\]

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

\[
\text{astree_node}_{[N]}_{\text{secret}}
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that fixing concrete values for GroupContext[N] and application_secret completely defines all secrets in the AS Tree.
12.2. Sender Ratchets

The secret of a leaf in the AS Tree is used to initiate a symmetric hash ratchet which generates a sequence of keys and nonces. The group member assigned to that leaf 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.

More precisely, the initial secret of the ratchet for the group member assigned to the leaf with node index \( N \) is simply the secret of that leaf.

\[
\text{application}_{[N]}_{[0]}\_\text{secret} = \text{astree}\_\text{node}_{[N]}\_\text{secret}
\]

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.

\[
\begin{align*}
\text{application}_{[N]}_{[j]}\_\text{secret} & \quad \rightarrow \quad \text{Derive-App-Secret}(., \text{"app-nonce"}, N, j, \text{AEAD.nonce_length}) \\
& \quad = \text{application}_{[N]}_{[j]}\_\text{nonce} \\
\rightarrow + \quad \text{Derive-App-Secret}(., \text{"app-key"}, N, j, \text{AEAD.key_length}) \\
& \quad = \text{application}_{[N]}_{[j]}\_\text{key} \\
\quad + \quad \text{Derive-App-Secret}(., \text{"app-secret"}, N, j, \text{Hash.length}) \\
& \quad = \text{application}_{[N]}_{[j+1]}\_\text{secret}
\end{align*}
\]

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.

12.3. 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", "update_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 even if their generating secret was already consumed (e.g. due to 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", "update_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 update_secret and init_secret used to derive G (i.e. the application_secret). The client MAY retain (i.e., not consume) the
values KD0 and ND0 to allow for out-of-order delivery, and SHOULD retain D2 to allow for processing future messages.

12.4. 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. ]]

12.5. 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: Currently, the group identifier, epoch and generation are contained as meta-data of the Signature. A different solution could be to include the GroupContext instead, if more information is required to achieve the security goals regarding cross-group attacks. ]]

[[ 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 it reasonable to allow to leak a range for the length of the plaintext by allowing to send a variable number of ciphertext blocks? ]]

12.6. 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.]]
13. 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.


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.

13.2. 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 preclude 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.]]
13.3. 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 Update 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 Update 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 ClientInitKeys and publish fresh ones for PCS.

13.4. Init Key Reuse

Initialization keys are intended to be used only once and then deleted. Reuse of init keys can lead to replay attacks.

14. IANA Considerations

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

- MLS Ciphersuites

All of these registries should be under a heading of "Message Layer Security", and administered under a Specification Required policy [RFC8126].

14.1. MLS Ciphersuites

The "MLS Ciphersuites" registry lists identifiers for suites of cryptographic algorithms defined for use with MLS. These are two-byte values, so the maximum possible value is 0xFFFF = 65535. Values in the range 0xF000 - 0xFFFF are reserved for vendor-internal usage.

Template:

- Value: The two-byte identifier for the ciphersuite
- Name: The name of the ciphersuite
- Reference: Where this algorithm is defined

The initial contents for this registry are as follows:
<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>P256_SHA256_AES128GCM</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0001</td>
<td>X25519_SHA256_AES128GCM</td>
<td>RFC XXXX</td>
</tr>
</tbody>
</table>

[[ Note to RFC Editor: Please replace "XXXX" above with the number assigned to this RFC. ]]

15. 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

- 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
16. References

16.1. Normative References

[I-D.irtf-cfrg-hpke]
Barnes, R. and K. Bhargavan, "Hybrid Public Key Encryption", draft-irtf-cfrg-hpke-00 (work in progress), July 2019.

[IEEE1363]


16.2. Informative References


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
```

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]

# Frontier is the list of full subtrees, from left to right. A # balanced binary tree with n leaves has a full subtree for every # power of two where n has a bit set, with the largest subtrees # furthest to the left. For example, a tree with 11 leaves has full # subtrees of size 8, 2, and 1.
def frontier(n):
    st = [1 << k for k in range(log2(n) + 1) if n & (1 << k) != 0]
    st = reversed(st)

    base = 0
    f = []
    for size in st:
        f.append(root(size) + base)
        base += 2*size
    return f

# Leaves are in even-numbered nodes
def leaves(n):
    return [2*i for i in range(n)]

# The resolution of a node is the collection of non-blank
# descendants of this node. Here the tree is represented by a list
# of nodes, where blank nodes are represented by None
def resolve(tree, x, n):
    if tree[x] != None:
        return [x]

    if level(x) == 0:
        return []

    L = resolve(tree, left(x, n))
    R = resolve(tree, right(x, n), n)
    return L + R

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