Network Working Group   R. Barnes
Internet-Draft   Cisco
Intended status: Informational   J. Millican
Expires: September 16, 2019   Facebook
                                                         E. Omara
                                                           Google
                                                   K. Cohn-Gordon
                                               University of Oxford
                                                        R. Robert
                                                             Wire
                                                   March 15, 2019

The Messaging Layer Security (MLS) Protocol
draft-ietf-mls-protocol-04

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

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

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

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

   For a group of size greater than two, a common strategy is to
   unilaterally broadcast symmetric "sender" keys over existing shared
   symmetric channels, and then for each member to send messages to the
   group encrypted with their own sender key. Unfortunately, while this
   improves efficiency over pairwise broadcast of individual messages
   and provides forward secrecy (with the addition of a hash ratchet),
   it is difficult to achieve post-compromise security with sender keys.
   An adversary who learns a sender key can often indefinitely and
   passively eavesdrop on that member's messages. Generating and
   distributing a new sender key provides a form of post-compromise
   security with regard to that sender. However, it requires

computation and communications resources that scale linearly with the size of the group.

In this document, we describe a protocol based on tree structures that enable asynchronous group keying with forward secrecy and post-compromise security. Based on earlier work on "asynchronous ratcheting trees" [art], the mechanism presented here use a 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-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 (*)

o Added encryption of the Welcome message (*)

draft-02

o Removed ART (*)

o Allowed partial trees to avoid double-joins (*)

o Added explicit key confirmation (*)

draft-01

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

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

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

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

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

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

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

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

o Added an appendix with example code for tree math

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

draft-00

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

2. Terminology

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

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

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 (UserInitKey).

Leaf Key: A secret that represent 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:

o 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.

o 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 8 for further considerations.)

o 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 four major operations in the lifecycle of a group:

o Adding a member, initiated by a current member;

o Updating the leaf secret of a member;

o Removing a member.

Before the initialization of a group, clients publish UserInitKey objects to a directory provided to the Messaging Service.

```
                                                        Group
A                 B                 C          Directory  Channel
|                 |                 |              |         |
| UserInitKeyA    |                 |              |         |
|--------------------------------------------->|         |
|                 |                 |              |         |
|                 | UserInitKeyB    |              |         |
|                 |----------------------------->|         |
|                 |                 |              |         |
|                 |                 | UserInitKeyC |         |
|                 |                 |------------->|         |
|                 |                 |              |         |
```
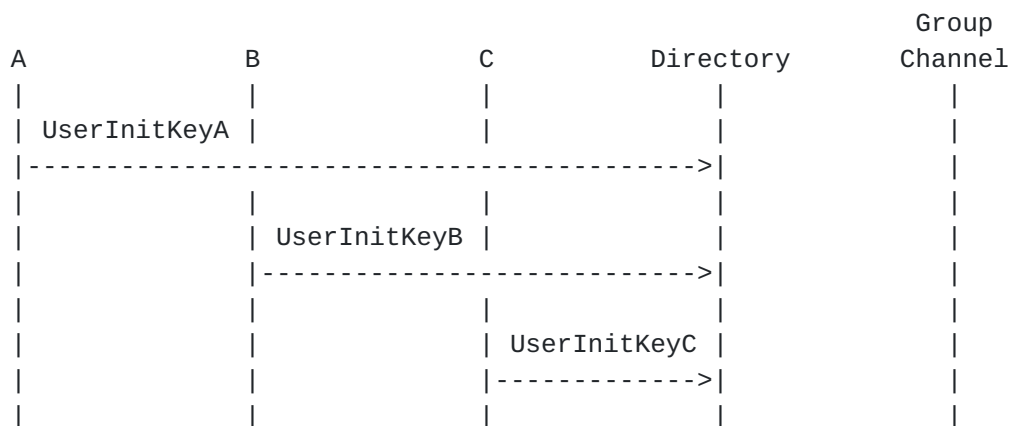
When a client A wants to establish a group with B and C, it first downloads UserInitKeys for B and C. It then initializes a group state containing only itself and uses the UserInitKeys to compute Welcome and Add messages to add B and C, in a sequence chosen by A. The Welcome messages are sent directly 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.

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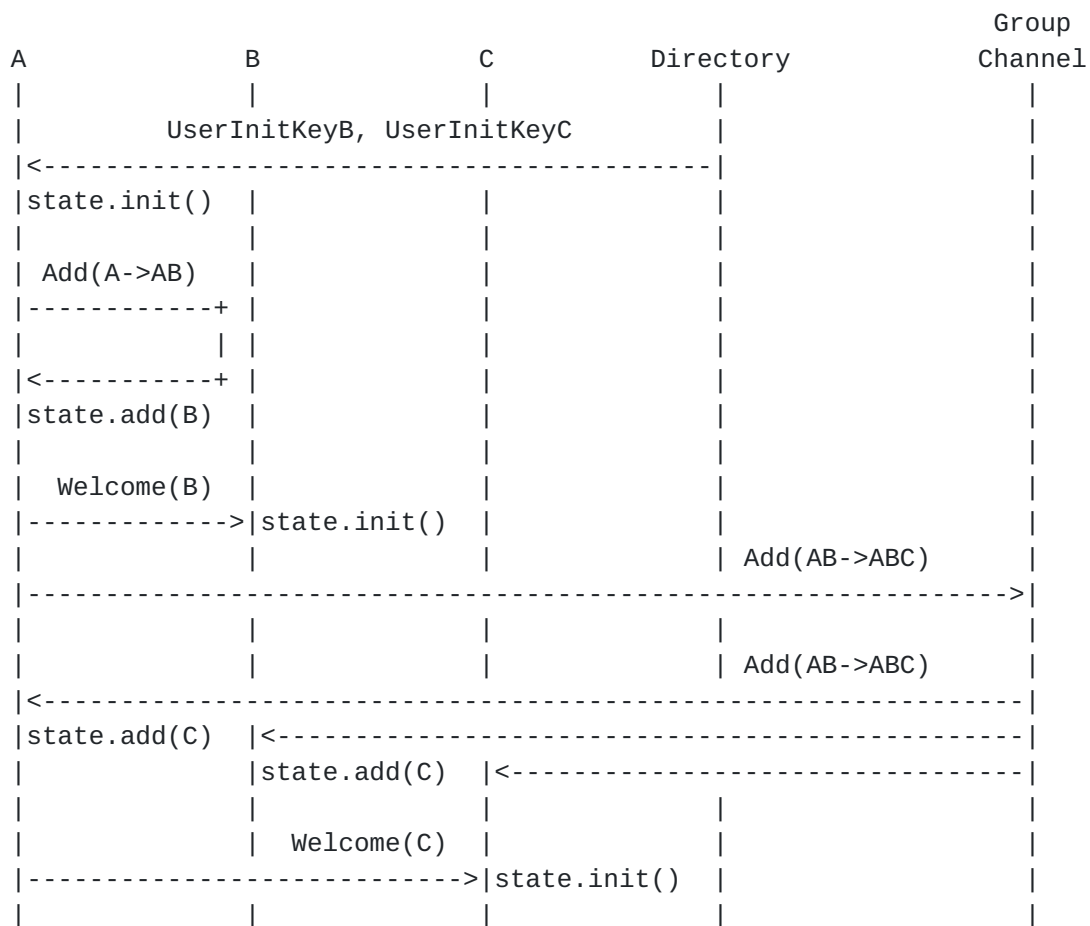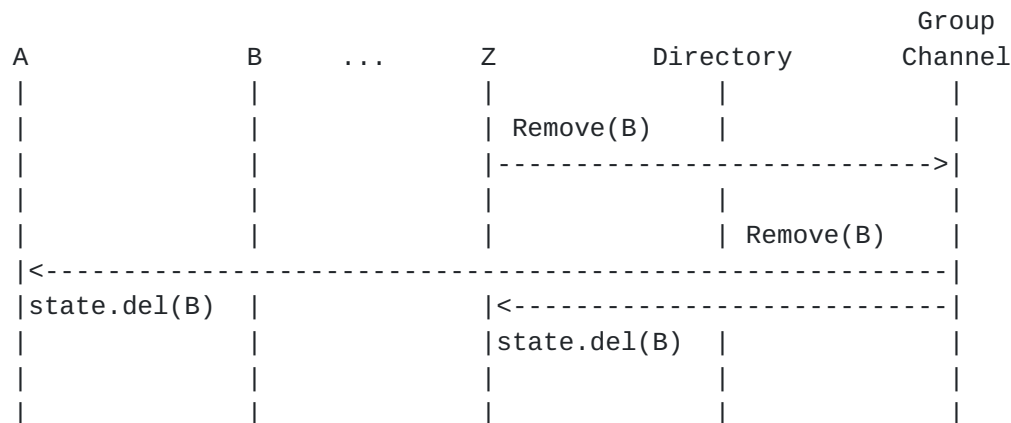
   Subsequent additions of group members proceed in the same way. Any
   member of the group can download an UserInitKey for a new client and
   broadcast an Add message that the current group can use to update
   their state and the new client can use to initialize its state.

   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.

```
                                                         Group
   A                B     ...      Z        Directory    Channel
   |                |              |            |           |
   | Update(A)      |              |            |           |
   |---------------------------------------------------------->|
   |                |              |            |           |
   |                |              |            | Update(A) |
   |<---------------------------------------------------------|
   |state.upd(A)    |<-----------------------------------------|
   |                |state.upd(A)  |<----------------------------|
   |                |              |state.upd(A)  |           |
   |                |              |            |           |
```
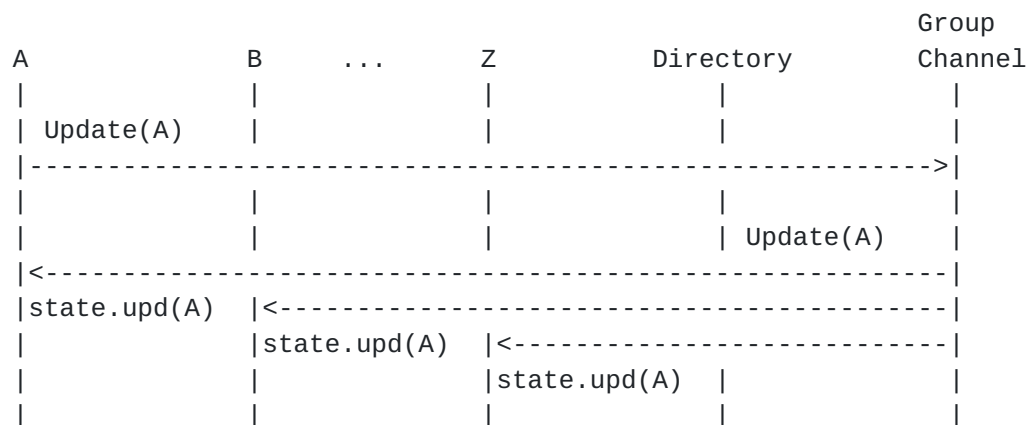
   Members are removed from the group in a similar way, as an update is
   effectively removing the old leaf from the group. Any member of the
   group can generate a Remove message that adds new entropy to the
   group state that is known to all members except the removed member.
   After other participants 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 mechanism.

```
                                                         Group
   A                B     ...      Z        Directory    Channel
   |                |              |            |           |
   |                |              | Remove(B)  |           |
   |                |              |----------------------------->|
   |                |              |            |           |
   |                |              |            | Remove(B) |
   |<---------------------------------------------------------|
   |state.del(B)    |              |<----------------------------|
   |                |              |state.del(B)  |           |
   |                |              |            |           |
   |                |              |            |           |
```

5. Ratchet Trees

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

5.1. Tree Computation Terminology

   Trees consist of _nodes_. A node is a _leaf_ if it has no children,
   and a _parent_ otherwise; note that all parents in our trees have
   precisely two children, a _left_ child and a _right_ child. A node
   is the _root_ of a tree if it has no parents, and _intermediate_ if
   it has both children and parents. The _descendants_ of a node are

that node, its children, and the descendants of its children, and we
say a tree _contains_ a node if that node is a descendant of the root
of the tree. Nodes are _siblings_ if they share the same parent.

A _subtree_ of a tree is the tree given by the descendants of any
node, the _head_ of the subtree. The _size_ of a tree or subtree is
the number of leaf nodes it contains. For a given parent node, its
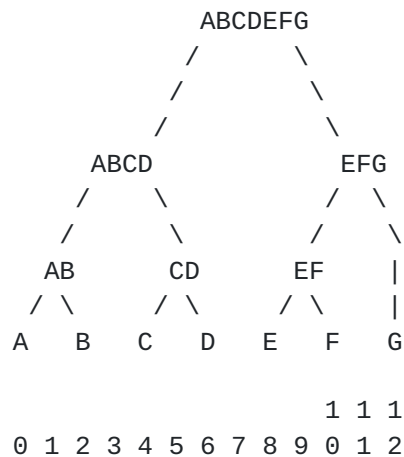_left subtree_ is the subtree with its left child as head
(respectively _right subtree_).

All trees used in this protocol are left-balanced binary trees. A
binary tree is _full_ (and _balanced_) if its size is a power of two
and for any parent node in the tree, its left and right subtrees have
the same size. If a subtree is full and it is not a subset of any
other full subtree, then it is _maximal_.

A binary tree is _left-balanced_ if for every parent, either the
parent is balanced, or the left subtree of that parent is the largest
full subtree that could be constructed from the leaves present in the
parent's own subtree. 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, excluding the root. 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:

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

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

o The frontier of the tree is (ABCD, EF, G)

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

                           1 1 1
     0 1 2 3 4 5 6 7 8 9 0 1 2
```

Each node in the tree is assigned an _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 indices for the nodes in the above tree are
as follows:

o   0 = A

o   1 = AB

o   2 = B

o   3 = ABCD

o   4 = C

o   5 = CD

o   6 = D

o   7 = ABCDEFG

o   8 = E

o   9 = EF

o   10 = F

o   11 = EFG

o   12 = G

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

5.2. Ratchet Tree Nodes and Tree Updates

   Ratchet trees are used for generating shared group secrets. In this
   section, we describe the structure of a ratchet tree. A particular
   instance of a ratchet tree is based on the following cryptographic
   primitives, defined by the ciphersuite in use:

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

   o A Key Derivation Function (KDF)

   o A Derive-Key-Pair function that produces an asymmetric keypair
      from a node secret

   A ratchet tree is a left-balanced binary tree, in which each node
   contains up to three values:

   o A secret octet string (optional)

   o An asymmetric private key (optional)

   o An asymmetric public key

   The contents of the parent are based on the latest-updated child.
   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 participants:

```
        G
       / \
      /   \
     /     \
    /       \
   E         F
  / \       / \
 A   B     C   D
```
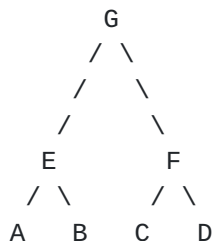
If the first participant subsequently generates an update based on a
secret X, then the sender would generate the following sequence of
path secrets and node secrets:

```
    path_secret[2] ---> node_secret[2]
          ^
          |
    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:

```
        ns[2]
       /     \
    ns[1]      F
    /  \      / \
ns[0]   B    C   D
```

5.3. Blank Nodes and Resolution

A node in the tree may 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 nodes in a resolution are ordered according to
their indices.

o The resolution of a non-blank node is a one element list
   containing the node itself

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

o The resolution of a blank intermediate node is the result of
   concatinating 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
    / \      / \
   A   _    C   D

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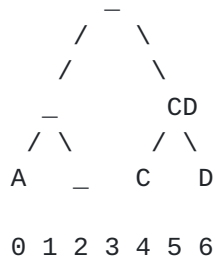
In this tree, we can see all three of the above rules in play:

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

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

o   The resolution of node 3 is the list [A, CD]

5.4. Ratchet Tree Updates

In order to update the state of the group such as adding and removing
clients, MLS messages are used to make changes to the group's ratchet
tree. The member proposing an update to the tree transmits a set of
values for intermediate nodes in the direct path of a leaf. Other
members in the group can use these nodes 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 transmits the following
information for each node in the direct path of the leaf:

o   The public key for the node

o   Zero or more encrypted copies of the node's parent secret value

The secret value 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 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 secret values * 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 secret
     value for the parent of the copath node using the private key

from the resolution node * Derive secret values for ancestors of
that node using the KDF keyed with the decrypted secret

2. Merge the updated secrets into the tree * Replace the public keys
    for nodes on the direct path with the received public keys * For
    nodes where an updated secret was computed in step 1, replace the
    secret value for the node with the updated value

For example, suppose we had the following tree:

```
      G
    /   \
   /     \
  E       _
 / \     / \
A   B   C   D
```
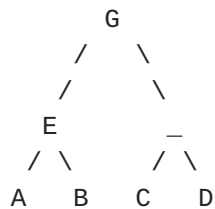
If an update is made along the direct path B-E-G, then the following
values will be transmitted (using pk(X) to represent the public key
corresponding to the secret value X and E(K, S) to represent public-
key encryption to the public key K of the secret value S):

```
+------------+--------------------------+
| Public Key | Ciphertext(s)            |
+------------+--------------------------+
| pk(G)      | E(pk(C), G), E(pk(D), G) |
|            |                          |
| pk(E)      | E(pk(A), E)              |
|            |                          |
| pk(B)      |                          |
+------------+--------------------------+
```

5.5. Cryptographic Objects

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

o A hash function

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

o An AEAD encryption algorithm [RFC5116]

The ciphersuite must also specify an algorithm "Derive-Key-Pair" that
maps octet strings with the same length as the output of the hash
function to key pairs for the asymmetric encryption scheme.

Public keys used in the protocol are opaque values in a format
defined by the ciphersuite, using the following types:

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

Cryptographic algorithms are indicated using the following types:

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

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

5.5.1. Curve25519, SHA-256, and AES-128-GCM

This ciphersuite uses the following primitives:

o   Hash function: SHA-256

o   Diffie-Hellman group: Curve25519 [RFC7748]

o   AEAD: AES-128-GCM

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]

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

This ciphersuite uses the following primitives:

o   Hash function: SHA-256

o Diffie-Hellman group: secp256r1 (NIST P-256)

o AEAD: AES-128-GCM

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.

5.6. 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:

o The public key of a signature key pair

o The identity of the holder of the private key

o 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_type) {
        case basic:
            BasicCredential;

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

5.7. Group State

Each member of the group maintains a representation of the state of
the group:

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

struct {
  opaque group_id<0..255>;
  uint32 epoch;
  optional<Credential> roster<1..2^32-1>;
  optional<HPKEPublicKey> tree<1..2^32-1>;
  opaque transcript_hash<0..255>;
} GroupState;
```

The fields in this state have the following semantics:

o The "group_id" field is an application-defined identifier for the
   group.

o The "epoch" field represents the current version of the group key.

o The "roster" field contains credentials for the occupied slots in
   the tree, including the identity and signature public key for the
   holder of the slot.

o The "tree" field contains the public keys corresponding to the
   nodes of the ratchet tree for this group. The length of this
   vector MUST be "2*size - 1", where "size" is the length of the
   roster, since this is the number of nodes in a tree with "size"
   leaves, according to the structure described in Section 5.

o The "transcript_hash" field contains the list of "GroupOperation"
   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 GroupState.

Different group operations will have different effects on the group
state. These effects are described in their respective subsections
of Section 7. The following rules apply to all operations:

o The "group_id" field is constant

o The "epoch" field increments by one for each GroupOperation that
   is processed

o The "transcript_hash" is updated by a GroupOperation message
   "operation" in the following way:

transcript_hash_[n] = Hash(transcript_hash_[n-1] || operation)

When a new one-member group is created (which requires no
GroupOperation), the "transcript_hash" field is set to an all-zero
vector of length Hash.length, where the Hash algorithm is defined by
the ciphersuite.

5.8. 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 {
    HPKEPublicKey ephemeral_key;
    opaque ciphertext<0..2^16-1>;
} HPKECiphertext;

struct {
    HPKEPublicKey public_key;
    HPKECiphertext node_secrets<0..2^16-1>;
} RatchetNode;

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

The length of the "node_secrets" vector MUST be zero for the first
node in the path. For the remaining elements in the vector, the
number of ciphertexts in the "node_secrets" 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 according to the Encrypt
function defined in [I-D.barnes-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.

5.9. Key Schedule

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

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

Where HkdfLabel is specified as:

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

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

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

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

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

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

o The init secret from the previous epoch

o The update secret for the current epoch

o The GroupState object for current epoch

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

```
                  init_secret_[n-1] (or 0)
                          |
                          V
    update_secret -> HKDF-Extract = epoch_secret
                          |
                          +--> Derive-Secret(., "app", GroupState_[n])
                          |      = application_secret
                          |
                          +--> Derive-Secret(., "confirm", GroupState_[n])
                          |      = confirmation_key
                          |
                          V
                  Derive-Secret(., "init", GroupState_[n])
                          |
                          V
                  init_secret_[n]
```

6. 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. UserInitKey messages provide
   information about a client that any existing member can use to add
   this client to the group asynchronously.

   A UserInitKey object specifies what ciphersuites a client supports,
   as well as providing public keys that the client can use for key
   derivation and signing. 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 a very limited number
   of times, potentially once. (see Section 10.4). UserInitKeys also
   contain an identifier chosen by the client, which the client MUST
   assure uniquely identifies a given UserInitKey object among the set
   of UserInitKeys created by this client.

   The init_keys array MUST have the same length as the cipher_suites
   array, and each entry in the init_keys array MUST be a public key for
   the asymmetric encryption scheme defined in the cipher_suites array
   and used in the HPKE construction for TreeKEM.

   The whole structure is signed using the client's identity key. A
   UserInitKey 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.

```
uint8 ProtocolVersion;

struct {
    opaque user_init_key_id<0..255>;
    ProtocolVersion supported_versions<0..255>;
    CipherSuite cipher_suites<0..255>;
    HPKEPublicKey init_keys<1..2^16-1>;
    Credential credential;
    opaque signature<0..2^16-1>;
} UserInitKey;
```

7. Handshake Messages

Over the lifetime of a group, its state will change for:

o Group initialization

o A current member adding a new client

o A current member updating its leaf key

o A current member deleting another current member

In MLS, these changes are accomplished by broadcasting "handshake"
messages to the group. Note that unlike TLS and DTLS, there is not a
consolidated handshake phase to the protocol. Rather, handshake
messages are exchanged throughout the lifetime of a group, whenever a
change is made to the group state. This means an unbounded number of
interleaved application and handshake messages.

An MLS handshake message encapsulates a specific "key exchange"
message that accomplishes a change to the group state. It also
includes a signature by the sender of the message over the GroupState
object representing the state of the group after the change has been
made.

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

struct {
    GroupOperationType msg_type;
    select (GroupOperation.msg_type) {
        case init:      Init;
        case add:       Add;
        case update:    Update;
        case remove:    Remove;
    };
} GroupOperation;

struct {
    uint32 prior_epoch;
    GroupOperation operation;

    uint32 signer_index;
    opaque signature<1..2^16-1>;
    opaque confirmation<1..2^8-1>;
} Handshake;
```

The high-level flow for processing a Handshake message is as follows:

1. Verify that the "prior_epoch" field of the Handshake message is
    equal the "epoch" field of the current GroupState object.

2. Use the "operation" message to produce an updated, provisional
    GroupState object incorporating the proposed changes.

3. Look up the public key for slot index "signer_index" from the
    roster in the current GroupState object (before the update).

4. Use that public key to verify the "signature" field in the
    Handshake message, with the updated GroupState object as input.

5. If the signature fails to verify, discard the updated GroupState
    object and consider the Handshake message invalid.

6. Use the "confirmation_key" for the new group state to compute the
    "confirmation" MAC for this message, as described below, and
    verify that it is the same as the "confirmation" field.

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

The "signature" and "confirmation" values are computed over the
transcript of group operations, using the transcript hash from the
provisional GroupState object:

```
signature_data = GroupState.transcript_hash
Handshake.signature = Sign(identity_key,
                           signature_data)

confirmation_data = GroupState.transcript_hash ||
                    Handshake.signature
Handshake.confirmation = HMAC(confirmation_key,
                              confirmation_data)
```

[[OPEN ISSUE: The confirmation data and signature data should
probably cover the same data as the one we cover with the GroupState.
]]

HMAC [RFC2104] uses the Hash algorithm for the ciphersuite in use.
Sign uses the signature algorithm indicated by the signer's
credential in the roster.

[[OPEN ISSUE: The Add and Remove operations create a "double-join"
situation, where a member's leaf key is also known to another client.
When a member A is double-joined to another B, deleting A will not
remove them from the conversation, since they will still hold the
leaf key for B. These situations are resolved by updates, but since
operations are asynchronous and members may be offline for a long
time, the group will need to be able to maintain security in the
presence of double-joins.]]

[[OPEN ISSUE: It is not possible for the recipient of an 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 in the ratchet tree.]]

7.1. Init

[[OPEN ISSUE: Direct initialization is currently undefined. A
client can create a group by initializing its own state to reflect a
group including only itself, then adding the initial members. This
has computation and communication complexity O(N log N) instead of
the O(N) complexity of direct initialization.]]

7.2. Add

   In order to add a new member to the group, an existing member of the
   group must take two actions:

   1. Send a Welcome message to the new member

   2. Send an Add message to the group (including the new member)

   The Welcome message contains the information that the new member
   needs to initialize a GroupState object that can be updated to the
   current state using the Add message. This information is encrypted
   for the new member using HPKE. The recipient key pair for the HPKE
   encryption is the one included in the indicated UserInitKey,
   corresponding to the indicated ciphersuite.

   ```
   struct {
     ProtocolVersion version;
     opaque group_id<0..255>;
     uint32 epoch;
     optional<Credential> roster<1..2^32-1>;
     optional<HPKEPublicKey> tree<1..2^32-1>;
     opaque transcript_hash<0..255>;
     opaque init_secret<0..255>;
   } WelcomeInfo;

   struct {
     opaque user_init_key_id<0..255>;
     CipherSuite cipher_suite;
     HPKECiphertext encrypted_welcome_info;
   } Welcome;
   ```

   Note that the "init_secret" in the Welcome message is the
   "init_secret" at the output of the key schedule diagram in
   Section 5.9. That is, if the "epoch" value in the Welcome message is
   "n", then the "init_secret" value is "init_secret_[n]". The new
   member can combine this init secret with the update secret
   transmitted in the corresponding Add message to get the epoch secret
   for the epoch in which it is added. No secrets from prior epochs are
   revealed to the new member.

   Since the new member is expected to process the Add message for
   itself, the Welcome message should reflect the state of the group
   before the new user is added. The sender of the Welcome message can
   simply copy all fields from their GroupState object.

   [[OPEN ISSUE: The Welcome message needs to be synchronized in the
   same way as the Add. That is, the Welcome should be sent only if the

Add succeeds, and is not in conflict with another, simultaneous Add.
]]

An Add message provides existing group members with the information
they need to update their GroupState with information about the new
member:

```
struct {
    uint32 index;
    UserInitKey init_key;
    opaque welcome_info_hash<0..255>;
} Add;
```

The "index" field indicates where in the tree the new member should
be added. The new member can be added at an existing, blank leaf
node, or at the right edge of the tree. In any case, the "index"
value MUST satisfy "0 <= index <= n", where "n" is the size of the
group. The case "index = n" indicates an add at the right edge of
the tree). If "index < n" and the leaf node at position "index" is
not blank, then the recipient MUST reject the Add as malformed.

The "welcome_info_hash" field contains a hash of the WelcomeInfo
object sent in a Welcome message to the new member.

A group member generates this message by requesting a UserInitKey
from the directory for the user to be added, and encoding it into an
Add message.

The client joining the group processes Welcome and Add messages
together as follows:

o Prepare a new GroupState object based on the Welcome message

o Process the Add message as an existing member would

An existing member receiving a Add message first verifies the
signature on the message, then updates its state as follows:

o If the "index" value is equal to the size of the group, increment
   the size of the group, and extend the tree and roster accordingly

o Verify the signature on the included UserInitKey; if the signature
   verification fails, abort

o Generate a WelcomeInfo object describing the state prior to the
   add, and verify that its hash is the same as the value of the
   "welcome_info_hash" field

   o Set the roster entry at position "index" to the credential in the
      included UserInitKey

   o Update the ratchet tree by setting to blank all nodes in the
      direct path of the new node

   o Set the leaf node in the tree at position "index" to a new node
      containing the public key from the UserInitKey in the Add
      corresponding to the ciphersuite in use

   The update secret resulting from this change is an all-zero octet
   string of length Hash.length.

   After processing an Add message, the new member SHOULD send an Update
   immediately to update its key. This will help to limit the tree
   structure degrading into subtrees, and thus maintain the protocol's
   efficiency.

7.3. Update

   An Update message is sent by a group member to update its leaf secret
   and key pair. This operation provides post-compromise security with
   regard to the member's prior leaf private key.

```
struct {
    DirectPath path;
} Update;
```

   The sender of an Update message creates it in the following way:

   o Generate a fresh leaf key pair

   o Compute its direct path in the current ratchet tree

   A member receiving a Update message first verifies the signature on
   the message, then updates its state as follows:

   o Update the cached ratchet tree by replacing nodes in the direct
      path from the updated leaf using the information contained in the
      Update message

   The update secret resulting from this change is the secret for the
   root node of the ratchet tree.

7.4. Remove

A Remove message is sent by a group member to remove one or more
members from the group.

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

The sender of a Remove message generates it as as follows:

o Generate a fresh leaf key pair

o Compute its direct path in the current ratchet tree, starting from
   the removed leaf

A member receiving a Remove message first verifies the signature on
the message. The member then updates its state as follows:

o Update the roster by setting the credential in the removed slot to
   the null optional value

o Update the ratchet tree by replacing nodes in the direct path from
   the removed leaf using the information in the Remove message

o Reduce the size of the roster and the tree until the rightmost
   element roster element and leaf node are non-null

o Update the ratchet tree by setting to blank all nodes in the
   direct path of the removed leaf

The update secret resulting from this change is the secret for the
root node of the ratchet tree after the second step (after the third
step, the root is blank).

8. 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:

o Have the delivery service enforce a total order

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

As long as 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, ie. all
updates must be executed before removes.

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

8.1. 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.

8.2. 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.

8.3. Merging Updates

It is possible in principle to partly address the problem of
concurrent changes by having the recipients of the changes merge
them, rather than having the senders retry. Because the value of
intermediate node is determined by its last updated child, updates
can be merged by recipients as long as the recipients agree on an
order - the only question is which node was last updated.

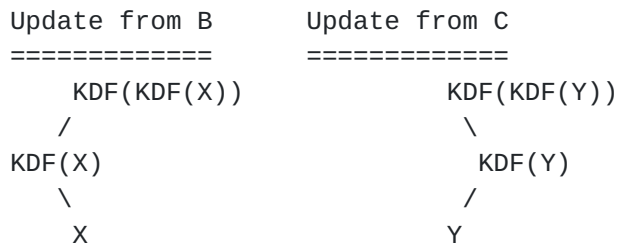Recall that the processing of an update proceeds in two steps:

1. Compute updated secret values by hashing up the tree

2. Update the tree with the new secret and public values

To merge an ordered list of updates, a recipient simply performs
these updates in the specified order.

For example, suppose we have a tree in the following configuration:

```
     KDF(KDF(D))
      /       \
  KDF(B)     KDF(D)
   /  \       /  \
  A    B    C    D
```

Now suppose B and C simultaneously decide to update to X and Y,
respectively. They will send out updates of the following form:

```
  Update from B        Update from C
  =============        =============
     KDF(KDF(X))          KDF(KDF(Y))
      /                        \
  KDF(X)                      KDF(Y)
      \                        /
       X                      Y
```

Assuming that the ordering agreed by the group says that B's update
should be processed before C's, the other members in the group will
overwrite the root value for B with the root value from C, and all
arrive at the following state:

```
     KDF(KDF(Y))
      /       \
  KDF(X)     KDF(Y)
   /  \       /  \
  A    X    Y    D
```

9. Message Protection

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 those members of a group, the
Application secret provided by the Handshake key schedule is used to
derive encryption keys for the Message Protection Layer.

Application messages MUST be protected with the Authenticated-
Encryption with Associated-Data (AEAD) encryption scheme associated
with the MLS ciphersuite. 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.

Each member maintains their own chain of Application secrets, where
the first one is derived based on a secret chained from the Epoch
secret. As shown in Section 5.9, the initial Application secret is

bound to the identity of each client to avoid collisions and allow
support for decryption of reordered messages.

Subsequent Application secrets MUST be rotated for each message sent
in order to provide stronger cryptographic security guarantees. The
Application Key Schedule use this rotation to generate fresh AEAD
encryption keys and nonces used to encrypt and decrypt future
Application messages. In all cases, a participant MUST NOT encrypt
more than expected by the security bounds of the AEAD scheme used.

Note that each change to the Group through a Handshake message will
cause a change of the group Secret. Hence this change MUST be
applied before encrypting any new Application message. This is
required for confidentiality reasons in order for members to avoid
receiving messages from the group after leaving, being added to, or
excluded from the group.

9.1. Application Key Schedule

After computing the initial group Application Secret, which is
derived from the main key schedule, each member creates an initial
sender Application Secret to be used for its own sending chain:

```
        application_secret
                |
                V
        HKDF-Expand-Label(., "app sender", [sender], Hash.length)
                |
                V
        application_secret_[sender]_[0]
```

Note that [sender] represents the index of the member in the roster.

Updating the Application secret and deriving the associated AEAD key
and nonce can be summarized as the following Application key schedule
where each member's Application secret chain looks as follows after
the initial derivation:

```
          application_secret_[sender]_[N-1]
                    |
                    +--> HKDF-Expand-Label(.,"nonce", "", nonce_length)
                    |     = write_nonce_[sender]_[N-1]
                    |
                    +--> HKDF-Expand-Label(.,"key", "", key_length)
                    |     = write_key_[sender]_[N-1]
                    V
          HKDF-Expand-Label(., "app sender", [sender], Hash.length)
                    |
                    V
          application_secret_[sender]_[N]
```

The Application context provided together with the previous
Application secret is used to bind the Application messages with the
next key and add some freshness.

[[OPEN ISSUE: The HKDF context field is left empty for now. A proper
security study is needed to make sure that we do not need more
information in the context to achieve the security goals.]]

[[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.]]

9.1.1. Updating the Application Secret

The following rules apply to an Application Secret:

o Senders MUST only use the Application Secret once and
   monotonically increment the generation of their secret. This is
   important to provide Forward Secrecy at the level of Application
   messages. An attacker getting hold of a member specific
   Application Secret at generation [N+1] will not be able to derive
   the member's Application Secret [N] nor the associated AEAD key
   and nonce.

o Receivers MUST delete an Application Secret once it has been used
   to derive the corresponding AEAD key and nonce as well as the next
   Application Secret. Receivers MAY keep the AEAD key and nonce
   around for some reasonable period.

o Receivers MUST delete AEAD keys and nonces once they have been
   used to successfully decrypt a message.

9.1.2. Application AEAD Key Calculation

   The Application AEAD keying material is generated from the following
   input values:

   o The Application Secret value;

   o A purpose value indicating the specific value being generated;

   o The length of the key being generated.

   Note, that because the identity of the participant using the keys to
   send data is included in the initial Application Secret, all
   successive updates to the Application secret will implicitly inherit
   this ownership.

   All the traffic keying material is recomputed whenever the underlying
   Application Secret changes.

9.2. 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 and sign them as follows:

   struct {
       opaque content<0..2^32-1>;
       opaque signature<0..2^16-1>;
       uint8 zeros[length_of_padding];
   } ApplicationMessageContent;

   struct {
       uint8 group[32];
       uint32 epoch;
       uint32 generation;
       uint32 sender;
       opaque encrypted_content<0..2^32-1>;
   } ApplicationMessage;

   The group identifier and epoch allow a device 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 derive the
   member's Application secret chain from the initial group Application
   secret. The application generation field is used to determine which
   Application secret should be used from the chain to compute the
   correct AEAD keys before performing decryption.

   The signature field allows strong authentication of messages:

```
struct {
    uint8  group[32];
    uint32 epoch;
    uint32 generation;
    uint32 sender;
    opaque content<0..2^32-1>;
} SignatureContent;
```

The signature used in the ApplicationMessageContent is computed over the SignatureContent which covers the metadata information about the current state of the group (group identifier, epoch, generation and sender's Leaf index) to prevent group members from impersonating other clients. It is also necessary in order to prevent cross-group attacks.

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

9.2.1. 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.]]

10. 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.

10.1. Confidentiality of the Group Secrets

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

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

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

10.2. **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.]]

10.3. **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.

10.4. **Init Key Reuse**

   Initialization keys are intended to be used only once and then
   deleted. Reuse of init keys is not believed to be inherently
   insecure [dhreuse], although it can complicate protocol analyses.

11. **IANA Considerations**

   TODO: Registries for protocol parameters, e.g., ciphersuites

12. Contributors

o Benjamin Beurdouche
   INRIA
   benjamin.beurdouche@ens.fr

o Karthikeyan Bhargavan
   INRIA
   karthikeyan.bhargavan@inria.fr

o Cas Cremers
   University of Oxford
   cas.cremers@cs.ox.ac.uk

o Alan Duric
   Wire
   alan@wire.com

o Srinivas Inguva
   Twitter
   singuva@twitter.com

o Albert Kwon
   MIT
   kwonal@mit.edu

o Eric Rescorla
   Mozilla
   ekr@rtfm.com

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

13. References

13.1. Normative References

[I-D.barnes-cfrg-hpke]
          Barnes, R. and K. Bhargavan, "Hybrid Public Key
          Encryption", draft-barnes-cfrg-hpke-01 (work in progress),
          March 2019.

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

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

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

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

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

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

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

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

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

13.2. Informative References

[art]   Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J.,
            and K. Milner, "On Ends-to-Ends Encryption: Asynchronous
            Group Messaging with Strong Security Guarantees", January
            2018, <https://eprint.iacr.org/2017/666.pdf>.

[CLINIC]   Miller, B., Huang, L., Joseph, A., and J. Tygar, "I Know
            Why You Went to the Clinic: Risks and Realization of HTTPS
            Traffic Analysis", Privacy Enhancing Technologies pp.
            143-163, DOI 10.1007/978-3-319-08506-7_8, 2014.

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

   [doubleratchet]
               Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L.,
               and D. Stebila, "A Formal Security Analysis of the Signal
               Messaging Protocol", 2017 IEEE European Symposium on
               Security and Privacy (EuroS&P),
               DOI 10.1109/eurosp.2017.27, April 2017.

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

   [I-D.ietf-trans-rfc6962-bis]
               Laurie, B., Langley, A., Kasper, E., Messeri, E., and R.
               Stradling, "Certificate Transparency Version 2.0", draft-
               ietf-trans-rfc6962-bis-31 (work in progress), February
               2019.

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

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

Appendix A. Tree Math

   One benefit of using left-balanced trees is that they admit a simple
   flat array representation. In this representation, leaf nodes are
   even-numbered nodes, with the n-th leaf at 2*n. Intermediate nodes
   are held in odd-numbered nodes. For example, a 11-element tree has
   the following structure:

```
                                              X
                      X
            X                       X                           X
      X           X           X           X           X
   X     X     X     X     X     X     X     X     X     X     X
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```
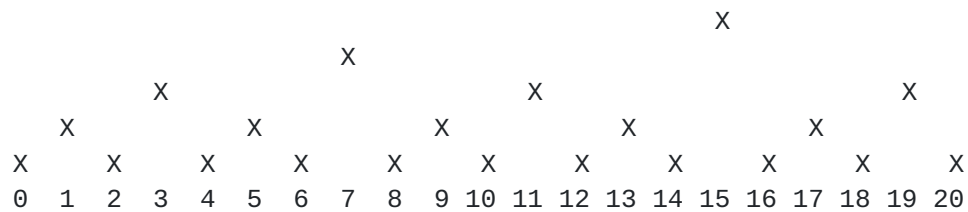
This allows us to compute relationships between tree nodes simply by
manipulating indices, rather than having to maintain complicated
structures in memory, even for partial trees. The basic rule is that
the high-order bits of parent and child nodes have the following
relation (where "x" is an arbitrary bit string):

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

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

```python
# 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 is the list of full subtrees, from left to right.  A
    # balance 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

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