Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 6, 2018

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

February 02, 2018

The Messaging Layer Security (MLS) Protocol
draft-barnes-mls-protocol-00

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 participants 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.

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/ekr/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.

Groups of agents who want to send each other encrypted messages need 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 groups of size greater than two, a common strategy is to unilaterally broadcast symmetric "sender" keys over existing shared symmetric channels, and then for each agent to send messages to the group encrypted with their own sender key. Unfortunately, while this improves efficiency over pairwise broadcast of individual messages and (with the addition of a hash ratchet) provides forward secrecy, 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 sender'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 as 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. The use of "asynchronous ratcheting trees" [art] allows the members of the group to derive and update shared

keys with costs that scale as the log of the group size. The use of Merkle trees to store identity information allows strong authentication of group membership, again with logarithmic cost.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

[TODO: The architecture document uses "Client" instead of "Participant". Harmonize terminology.]

Participant: An agent that uses this protocol to establish shared cryptographic state with other participants. A participant is defined by the cryptographic keys it holds. An application may use one participant per device (keeping keys local to each device) or sync keys among a user's devices so that each user appears as a single participant.

Group: A collection of participants with shared cryptographic state.

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

Initialization Key: A short-lived Diffie-Hellman key pair used to introduce a new member to a group. Initialization keys can be published for both individual participants (UserInitKey) and groups (GroupInitKey).

Leaf Key: A short-lived Diffie-Hellman key pair that represents a group member's contribution to the group secret, so called because the participants 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 [I-D.ietf-tls-tls13] 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.rescorla-mls-architecture]. In particular, we assume the MS provides the following services:

- A long-term identity key provider which allows participants 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 participant'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 9 for further considerations.)

- A directory to which participants can publish initialization keys, and from which participant can download initialization keys for other participants.

4. Protocol Overview

The goal of this protocol is to allow a group of participants to exchange confidential and authenticated messages. It does so by deriving a sequence of keys known only to group members. Keys 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 participant as a _state_, which includes both public and private data. An initial state, including an initial set of participants, is set up by a group creator using the _Init_ algorithm and based on information pre-published by the initial members. The creator sends the _GroupInit_ message to the participants, who can then set up their own group state and derive the same shared key. Participants then exchange messages to produce new shared states which are causally linked to their predecessors, forming a logical Directed Acyclic Graph (DAG) of states. Participants can send _Update_ messages for post-compromise secrecy and new participants can be added or existing participants removed from the group.

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

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

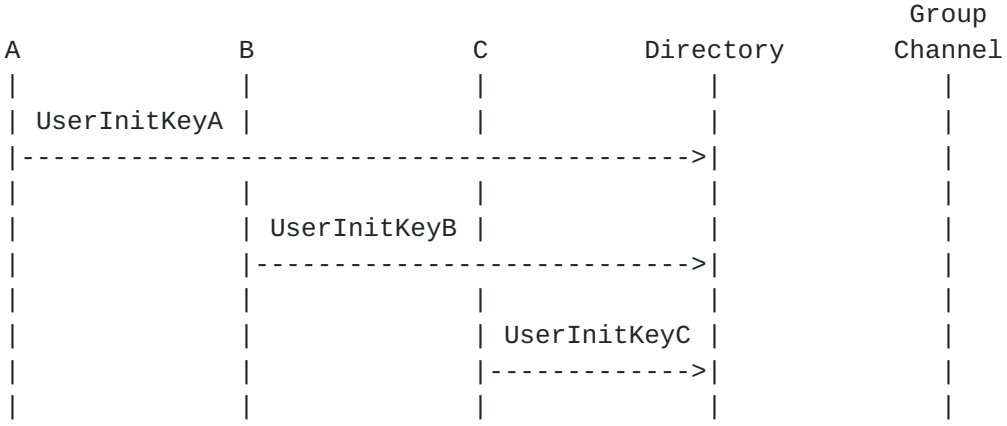
- Adding a member, initiated by a current member

- Adding a member, initiated by the new member

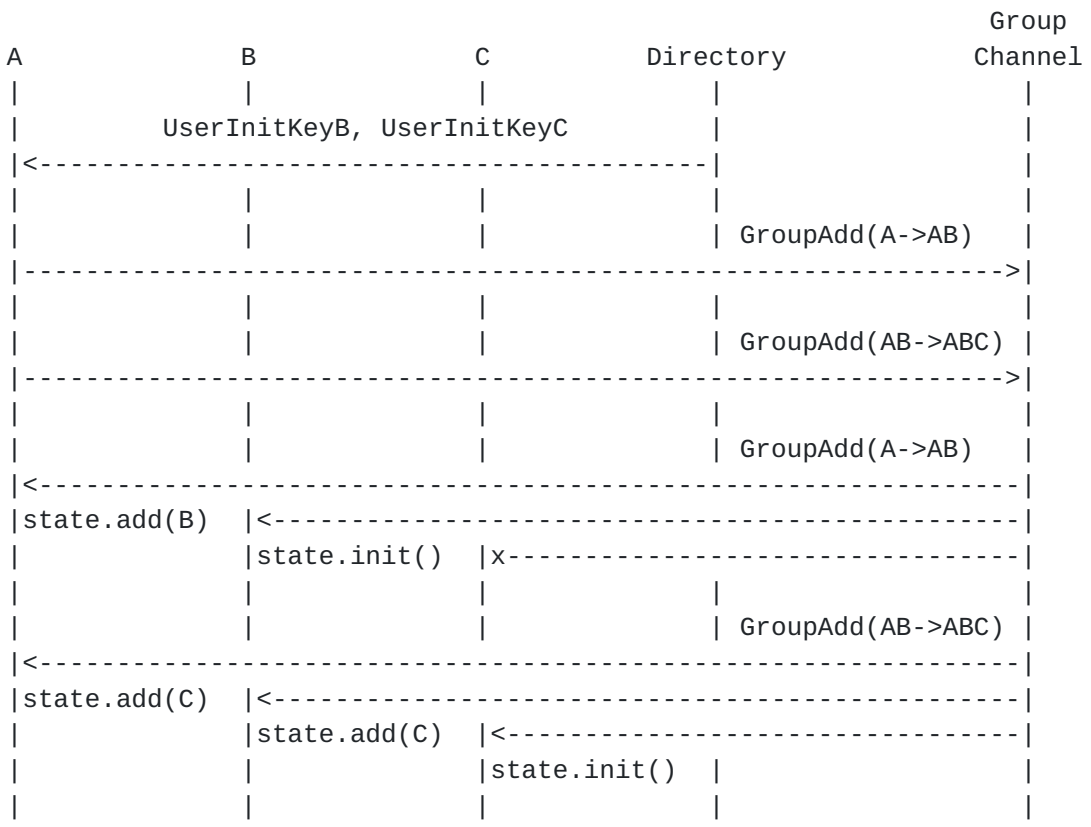
- Key update

o Removal of a member

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



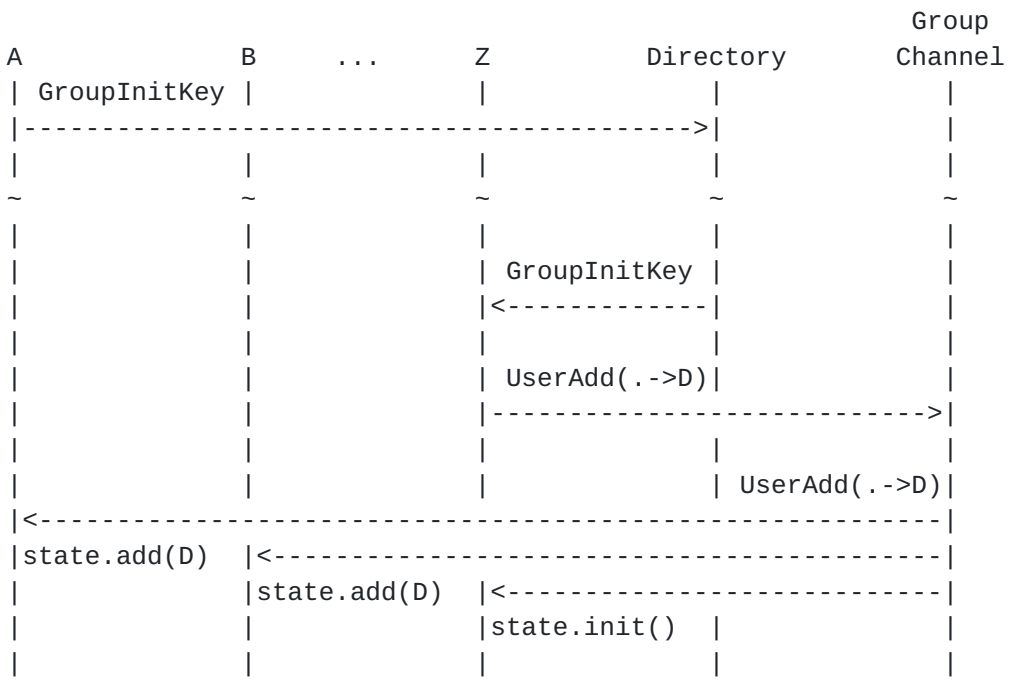
When a participant A wants to establish a group with B and C, it first downloads InitKeys for B and C. It then initializes a group state containing only itself and uses the InitKeys to compute GroupAdd messages to add B and C, in a sequence chosen by A. These messages are broadcasted to the Group, and processed in sequence by B and C. Messages received before a participant has joined the group are ignored. Only after A has received its GroupAdd 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 an InitKey for a new participant and broadcast a GroupAdd which the current group can use to update their state and the new participant can use to initialize its state.

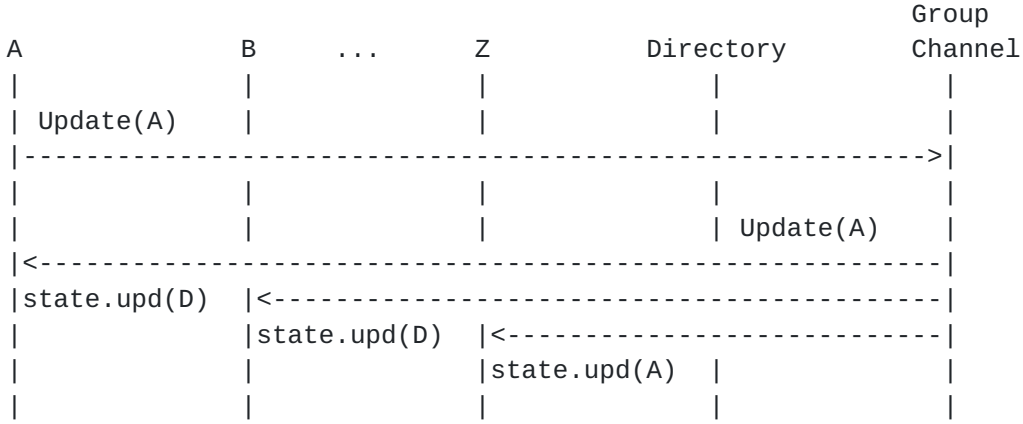
It is sometimes necessary for a new participant to join without an explicit invitation from a current member. For example, if a user that is authorized to be in the group logs in on a new device, that device will need to join the group as a new participant, but will not have been invited.

In these "user-initiated join" cases, the "InitKey + Add message" flow is reversed. We assume that at some previous point, a group member has published a GroupInitKey reflecting the current state of the group (A, B, C). The new participant Z downloads that GroupInitKey from the directory, generates a UserAdd message, and broadcasts it to the group. Once current members process this message, they will have a shared state that also includes Z.



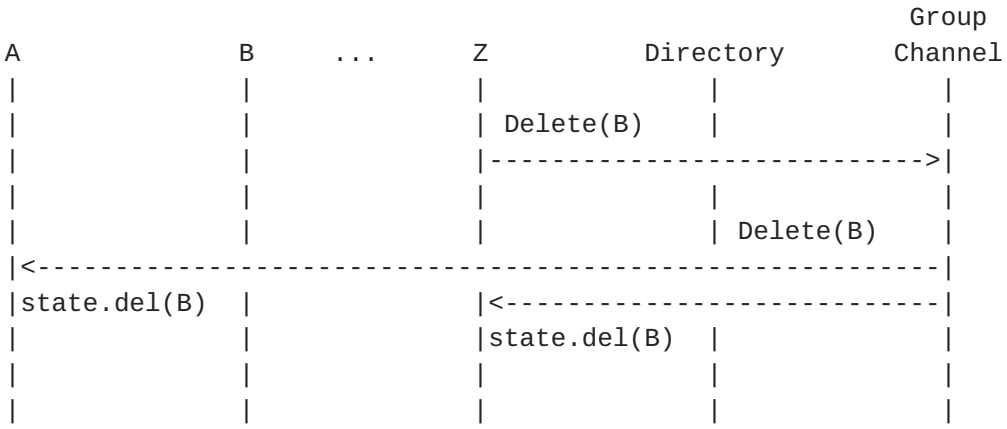
To enforce forward secrecy and post-compromise security of messages, each participant periodically updates its leaf key, the DH key pair that represents its contribution to the group key. Any member of the group can send an Update at any time by generating a fresh leaf key pair and sending an Update message that describes how to update the group key with that new key pair. Once all participants have processed this message, the group's secrets will be unknown to an attacker that had compromised the sender's prior DH leaf private key.

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.



Users are deleted from the group in a similar way, as a key update is effectively removing the old leaf from the group. Any member of the

group can generate a Delete message that adds new entropy to the group state that is known to all members except the deleted member. After other participants have processed this message, the group's secrets will be unknown to the deleted 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.



5. Binary Trees

The protocol uses two types of binary tree structures:

- Merkle trees for efficiently committing to a set of group participants.
- Asynchronous ratcheting trees for deriving shared secrets among this group of participants.

The two trees in the protocol share a common structure, allowing us to maintain a direct mapping between their nodes when manipulating group membership. The "nth" leaf in each tree is owned by the "nth" group participant.

5.1. Terminology

We use a common set of terminology to refer to both types of binary tree.

Trees consist of various different types of _nodes_. A node is a _leaf_ if it has no children, and a _parent_ otherwise; note that all parents in our Merkle or asynchronous ratcheting 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_).

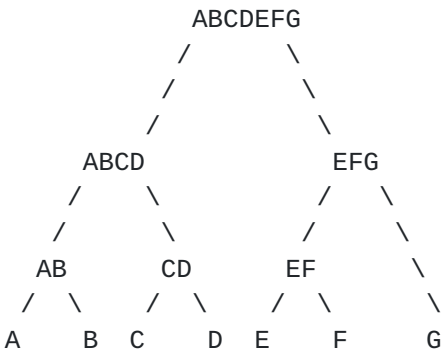
All trees used in this protocol are left-balanced binary trees. A binary tree is _full_ (and _balanced_) if it its size is a power of two and for any parent node in the tree, its left and right subtrees have the same size. If a subtree is full and it is not a subset of any other full subtree, then it 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, which has no sibling. 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)



We extend both types of tree to include a concept of "blank" nodes; which are used to replace group members who have been removed. We expand on how these are used and implemented in the sections below.

(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].)

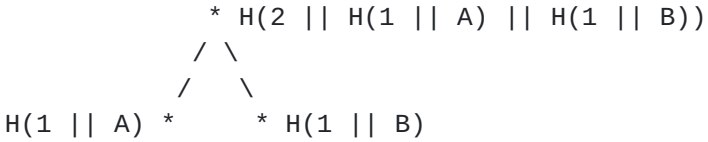
5.2. Merkle Trees

Merkle trees are used to efficiently commit to a collection of group members. We require a hash function, denoted H, to construct this tree.

Each node in a Merkle tree is the output of the hash function, computed as follows:

- Leaf nodes: "H(0x01 || leaf-value)"
- Parent nodes: "H(0x02 || left-value || right-value)"
- Blank leaf nodes: "H(0x00)"

The below tree provides an example of a size 2 tree, containing identity keys "A" and "B".

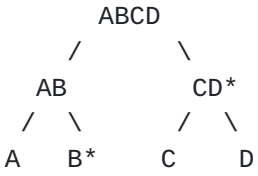


In Merkle trees, blank nodes appear only at the leaves. In computation of intermediate nodes, they are treated in the same way as other nodes.

5.2.1. Merkle Proofs

A proof of a given leaf being a member of the Merkle tree consists of the value of the leaf node, as well as the values of each node in its copath. From these values, its path to the root can be verified; proving the inclusion of the leaf in the Merkle tree.

In the below tree, we denote with a star the Merkle proof of membership for leaf node "A". For brevity, we notate "Hash(0x02 || A || B)" as "AB".



5.3. Ratchet Trees

Ratchet trees are used for generating shared group secrets. These are constructed as a series of Diffie-Hellman keys in a binary tree arrangement, with each user knowing their direct path, and thus being able to compute the shared root secret.

To construct these trees, we require:

- a Diffie-Hellman finite-field group or elliptic curve;
- a Derive-Key-Pair function that produces a key pair from an octet string, such as the output of a DH computation

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

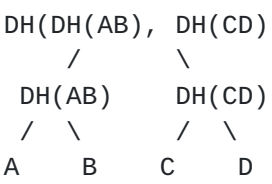
- A secret octet string (optional)
- A DH private key (optional)
- A DH public key

To compute the private values (secret and private key) for a given node, one must first know the private key from one of its children, and the public key from the other child. Then the value of the parent is computed as follows:

- secret = DH(L, R)
- private, public = Derive-Key-Pair(secret)

Ratchet trees are constructed as left-balanced trees, defined such that each parent node's key pair is derived from the Diffie-Hellman shared secret of its two child nodes. To compute the root secret and private key, a participant must know the public keys of nodes in its copath, as well as its own leaf private key.

For example, the ratchet tree consisting of the private keys (A, B, C, D) is constructed as follows:

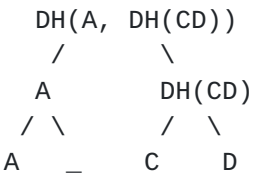


Ratchet trees constructed this way provide the property that one must hold at least one private key from the tree to compute the secret root key. With all participants holding one leaf private key; this allows any individual to update their own key and change the shared root key, such that only group members can compute the new key.

5.3.1. Blank Ratchet Tree Nodes

Nodes in a ratchet tree can have a special value "_", used to indicate that the node should be ignored during path computations. Such nodes are used to replace leaves when participants are deleted from the group.

If any node in the copath of a leaf is _, it should be ignored during the computation of the path. For example, the tree consisting of the private keys (A, _, C, D) is constructed as follows:



If two sibling nodes are both _, their parent value also becomes _.

Blank nodes effectively result in an unbalanced tree, but allow the tree management to behave as for a balanced tree for programming simplicity.

6. Group State

The state of an MLS group at a given time comprises:

- A group identifier (GID)
- A ciphersuite used for cryptographic computations
- A Merkle tree over the participants' identity keys
- A ratchet tree over the participants' leaf key pairs
- A message master secret (known only to participants)
- An add key pair (private key known only to participants)
- An init secret (known only to participants)

Since a group can evolve over time, a session logically comprises a sequence of states. The time in which each individual state is used is called an "epoch", and each state is assigned an epoch number that increments when the state changes.

MLS handshake messages provide each node with enough information about the trees to authenticate messages within the group and compute the group secrets.

Thus, each participant will need to store the following information about each state of the group:

1. The participant's index in the identity/ratchet trees
2. The private key associated with the participant's leaf public key
3. The private key associated with the participant's identity public key
4. The current epoch number
5. The group identifier (GID)
6. A subset of the identity tree comprising at least the copath for the participant's leaf
7. A subset of the ratchet tree comprising at least the copath for the participant's leaf

8. The current message encryption shared secret, called the master secret

9. The current add key pair

10. The current init secret

6.1. Cryptographic Objects

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

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 Diffie-Hellman group.

Public keys and Merkle tree nodes used in the protocol are opaque values in a format defined by the ciphersuite, using the following four types:

```
uint16 CipherSuite;
opaque DHPublicKey<1..2^16-1>;
opaque SignaturePublicKey<1..2^16-1>;
opaque MerkleNode<1..255>
```

[[OPEN ISSUE: In some cases we will want to include a raw key when we sign and in others we may want to include an identity or a certificate containing the key. This type needs to be extended to accommodate that.]]

6.1.1. Curve25519 with SHA-256

This ciphersuite uses the following primitives:

- Hash function: SHA-256
- Diffie-Hellman group: Curve25519 [RFC7748]

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 with SHA-256

This ciphersuite uses the following primitives:

- Hash function: SHA-256
- Diffie-Hellman group: secp256r1 (NIST P-256)

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. 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:

```
Derive-Secret(Secret, Label, ID, Epoch, Msg) =
     HKDF-Expand(Secret, HkdfLabel, Length)
```

Where HkdfLabel is specified as:

```
struct {
    uint16 length = Length;
    opaque label<7..255> = "mls10 " + Label;
    opaque group_id<0..2^16-1> = ID;
    uint32 epoch = Epoch;
    opaque message<1..2^16-1> = Msg
} HkdfLabel;
```

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 form 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 participant combines the following information to derive new epoch secrets:

- The init secret from the previous epoch
- The update secret for the current epoch
- The handshake message that caused the epoch change
- The current group identifier (GID) and epoch

The derivation of the update secret depends on the change being made, as described below.

For UserAdd or GroupAdd, the new user does not know the prior epoch init secret. Instead, entropy from the prior epoch is added via the update secret, and an all-zero vector with the same length as a hash output is used in the place of the init secret.

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

7. Initialization Keys

In order to facilitate asynchronous addition of participants to a group, it is possible to pre-publish initialization keys that provide some public information about a user or group. UserInitKey messages provide information about a potential group member, that a group member can use to add this user to a group without asynchronously. GroupInitKey messages provide information about a group that a new user can use to join the group without any of the existing members of the group being online.

7.1. UserInitKey

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 11.4).

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 DH group defined by the corresponding entry in the cipher_suites array.

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.

```
struct {
    CipherSuite cipher_suites<0..255>;
    DHPublicKey init_keys<1..2^16-1>;
    SignaturePublicKey identity_key;
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} UserInitKey;
```

7.2. GroupInitKey

A GroupInitKey object specifies the aspects of a group's state that a new member needs to initialize its state (together with an identity key and a fresh leaf key pair).

- The current epoch number
- The number of participants currently in the group
- The group ID
- The cipher suite used by the group
- The public key of the current update key pair for the group
- The frontier of the identity tree, as a sequence of hash values
- The frontier of the ratchet tree, as a sequence of public keys

GroupInitKey messages are not themselves signed. A GroupInitKey should not be published "bare"; instead, it should be published by constructing a handshake message with type "none", which will include a signature by a member of the group and a proof of membership in the group.

```
struct {
    uint32 epoch;
    uint32 group_size;
    opaque group_id<0..2^16-1>;
    CipherSuite cipher_suite;
    DHPublicKey add_key;
    MerkleNode identity_frontier<0..2^16-1>;
    DHPublicKey ratchet_frontier<0..2^16-1>;
} GroupInitKey;
```

8. Handshake Messages

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

- Group initialization
- A current member adding a new participant
- A new participant adding themselves
- A current participant updating its leaf key
- 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.

An MLS handshake message encapsulates a specific message that accomplishes a change to the group state. It also includes two other important features:

- A GroupInitKey so that a new participant can observe the latest state of the handshake and initialize itself
- A signature by a member of the group, together with a Merkle inclusion proof that demonstrates that the signer is a legitimate member of the group.

Before considering a handshake message valid, the recipient MUST verify both that the signature is valid, the Merkle inclusion proof is valid, and the sender is authorized to make the change according to group policy. The input to the signature computations comprises the entire handshake message except for the signature field.

The Merkle tree head to be used for validating the inclusion proof MUST be one that the recipient trusts to represent the current list of participant identity keys.

```
enum {
    none(0),
    init(1),
    user_add(2),
    group_add(3),
    update(4),
    delete(5),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 inner_length;
    select (Handshake.msg_type) {
        case none:      struct{};
        case init:      Init;
        case user_add:  UserAdd;
        case group_add: GroupAdd;
        case update:    Update;
        case delete:    Delete;
    };

    uint32 prior_epoch;
    GroupInitKey init_key;

    uint32 signer_index;
    MerkleNode identity_proof<1..2^16-1>;
    SignaturePublicKey identity_key;

    SignatureScheme algorithm;
    opaque signature<1..2^16-1>;
} Handshake;
```

[[OPEN ISSUE: There will be a need to integrate credentials from an authentication service that associate identities to the identity keys used to sign messages. This integration will enable meaningful authentication (of identities, rather than keys), and will need to be done in such a way as to prevent unknown key share attacks.]]

[[OPEN ISSUE: The GroupAdd and Delete operations create a "double-join" situation, where a participants leaf key is also known to another participant. When a participant 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 participants 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 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.]]

8.1. Init

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

8.2. GroupAdd

A GroupAdd message is sent by a group member to add a new participant to the group. The content of the message is only the UserInitKey for the user being added.

```
struct {
    UserInitKey init_key;
} GroupAdd;
```

A group member generates such a message by requesting from the directory a UserInitKey for the user to be added. The new participant processes the message together with the private key corresponding to the UserInitKey to initialize his state as follows:

- Compute the participant's leaf key pair by combining the init key in the UserInitKey with the prior epoch's add key pair
- Use the frontiers in the GroupInitKey of the Handshake message to add its keys to the trees

An existing participant receiving a GroupAdd message first verifies the signature on the message, then verifies its identity proof against the identity tree held by the participant. The participant then updates its state as follows:

- Compute the new participant's leaf key pair by combining the leaf key in the UserInitKey with the prior epoch add key pair
- Update the group's identity tree and ratchet tree with the new participant's information

The update secret resulting from this change is the output of a DH computation between the private key for the root of the ratchet tree and the add public key from the previous epoch.

[[ALTERNATIVE: The sender could also generate the new participant's leaf using a fresh key pair, as opposed to a key pair derived from the prior epoch's secret. This would reduce the "double-join" problem, at the cost of the GroupAdd having to include a new ratchet frontier.]]

8.3. UserAdd

A UserAdd message is sent by a new group participant to add themselves to the group, based on having already had access to a GroupInitKey for the group.

```
struct {
    DHPublicKey add_path<1..2^16-1>;
} UserAdd;
```

A new participant generates this message using the following steps:

- Fetch a GroupInitKey for the group
- Use the frontiers in the GroupInitKey to add its keys to the trees
- Compute the direct path from the new participant's leaf in the new ratchet tree (the add_path).

An existing participant receiving a UserAdd first verifies the signature on the message, then verifies its identity inclusion proof against the updated identity tree expressed in the GroupInitKey of the Handshake message (since the signer is not included in the prior group state held by the existing participant). The participant then updates its state as follows:

- Update trees with the descriptions in the new GroupInitKey
- Update the local ratchet tree with the add path in the UserAdd message, replacing any common nodes with the values in the add path

The update secret resulting from this change is the output of a DH computation between the private key for the root of the ratchet tree and the add public key from the previous epoch.

8.4. Update

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

```
struct {
    DHPublicKey ratchetPath<1..2^16-1>;
} Update;
```

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

- Generate a fresh leaf key pair
- Compute its direct path in the current ratchet tree

An existing participant receiving a Update message first verifies the signature on the message, then verifies its identity proof against the identity tree held by the participant. The participant then updates its state as follows:

- Update the cached ratchet tree by replacing nodes in the direct path from the updated leaf with the corresponding nodes in the Update message

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

8.5. Delete

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

```
struct {
    uint32 deleted;
    DHPublicKey path<1..2^16-1>;
} Delete;
```

The sender of a Delete message must know the deleted node's copath. Based on this knowledge, it computes a Delete message as follows:

- Generate a fresh leaf key pair
- Compute the direct path from the deleted node's index with the fresh leaf key pair in the current ratchet tree

An existing participant receiving a Update message first verifies the signature on the message, then verifies its identity proof against

the identity tree held by the participant. The participant then updates its state as follows:

- Update the cached ratchet tree by replacing nodes in the direct path from the deleted leaf with the corresponding nodes in the Update message

- Update the cached ratchet tree and identity tree by replacing the deleted node's leaves with blank nodes

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

9. 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

In either case, 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.

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.

9.1. Server-side 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.

9.2. Client-side 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.

[[OPEN ISSUE: Another possibility here is batching + deterministic selection.]]

10. Message Protection

[[OPEN ISSUE: This section has initial considerations about message protection. This issue clearly needs more specific recommendations, possibly a protocol specification in this document or a separate one.]]

The primary purpose of this protocol is to enable an authenticated group key exchange among participants. In order to protect messages sent among those participants, an application will need to specify how messages are protected.

For every epoch, the root key of the ratcheting tree can be used to derive key material for symmetric operations such as encryption/AEAD and MAC; AEAD or MAC MUST be used to ensure that the message originated from a member of the group.

In addition, asymmetric signatures SHOULD be used to authenticate the sender of a message.

In combination with server-side enforced ordering, data from previous messages is used (as a salt when hashing) to:

- add freshness to derived symmetric keys
- cryptographically bind the transcript of all previous messages with the current group shared secret

Possible candidates for that are:

- the key used for the previous message (hash ratcheting)
- the counter of the previous message (needs to be known to new members of the group)
- the hash of the previous message (proof that other participants saw the same history)

The requirement for this is that all participants know these values. If additional clear-text fields are attached to messages (like the counter), those fields MUST be protected by a signed message envelope.

Alternatively, the hash of the previous message can also be included as an additional field rather than change the encryption key. This allows for a more flexible approach, because the receiving party can choose to ignore it (if the value is not known, or if transcript security is not required).

11. Security Considerations

The security goals of MLS are described in [[the architecture doc]]. 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.

11.1. 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 parties to authenticate their legitimate peers.

11.2. 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 participant 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.]]

11.3. 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.

11.4. 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.

12. IANA Considerations

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

13. Contributors

- Benjamin Beurdouche
  INRIA
  benjamin.beurdouche@ens.fr

- 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

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

14. References

14.1. Normative References

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-23 (work in progress), January 2018.

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

[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>.

[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>.

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

14.2. 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>.

[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.

[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-27 (work in progress), October 2017.

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

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