The Messaging Layer Security (MLS) Architecture
draft-ietf-mls-architecture-02

Abstract

This document describes the architecture and requirements for the Messaging Layer Security (MLS) protocol. MLS provides a security layer for group messaging applications with from two to a large number of clients. It is meant to protect against eavesdropping, tampering, and message forgery.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.
This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ............................................. 3
2. General Setting .......................................... 3
   2.1. Group, Members and Clients ......................... 5
   2.2. Authentication Service ............................ 6
   2.3. Delivery Service ................................... 6
      2.3.1. Key Storage ................................ 7
      2.3.2. Key Retrieval ............................... 7
      2.3.3. Delivery of messages and attachments .......... 8
      2.3.4. Membership knowledge ........................ 8
      2.3.5. Membership and offline members ............... 9
3. System Requirements ..................................... 9
   3.1. Functional Requirements ........................... 9
      3.1.1. Asynchronous Usage ............................ 9
      3.1.2. Recovery After State Loss ..................... 9
      3.1.3. Support for Multiple Devices ................... 9
      3.1.4. Extensibility / Pluggability ................... 10
      3.1.5. Privacy ..................................... 10
      3.1.6. Federation .................................. 10
      3.1.7. Compatibility with future versions of MLS .... 10
   3.2. Security Requirements ............................ 10
      3.2.1. Connections between Clients and Servers (one-to-one) 11
      3.2.2. Message Secrecy and Authentication ............. 11
4. Security Considerations ................................. 13
   4.1. Transport Security Links .......................... 13
   4.2. Delivery Service Compromise ....................... 14
   4.3. Authentication Service Compromise .................. 14
   4.4. Client Compromise ................................ 14
5. IANA Considerations .................................... 15
6. Contributors ............................................ 15
7. Informative References ................................ 15
Authors’ Addresses ........................................ 16
1. Introduction

The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/mlswg/mls-architecture. 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.

End-to-end security is a requirement for instant messaging systems and is commonly deployed in many such systems. In this context, "end-to-end" captures the notion that users of the system enjoy some level of security - with the precise level depending on the system design - even when the messaging service they are using performs unsatisfactorily.

Messaging Layer Security (MLS) specifies an architecture (this document) and an abstract protocol [MLSPROTO] for providing end-to-end security in this setting. MLS is not intended as a full instant messaging protocol but rather is intended to be embedded in a concrete protocol such as XMPP [RFC6120]. In addition, it does not specify a complete wire encoding, but rather a set of abstract data structures which can then be mapped onto a variety of concrete encodings, such as TLS [I-D.ietf-tls-tls13], CBOR [RFC7049], and JSON [RFC7159]. Implementations which adopt compatible encodings will have some degree of interoperability at the message level, though they may have incompatible identity/authentication infrastructures.

This document is intended to describe the overall messaging system architecture which the MLS protocol fits into, and the requirements which it is intended to fulfill.

2. General Setting

A Group using a Messaging Service (MS) comprises a set of participants called Members where each member is typically expected to own multiple devices, called Clients. A group may be as small as two members (the simple case of person to person messaging) or as large as thousands. In order to communicate securely, clients initially use services at their disposal to obtain the necessary secrets and credentials required for security.

The Messaging Service (MS) presents as two abstract services that allow clients to prepare for sending and receiving messages securely:

- An Authentication Service (AS) which is responsible for maintaining user long term identities, issuing credentials which
allow them to authenticate each other, and potentially allowing users to discover each others long-term identity keys.

- A Delivery Service (DS) which is responsible for receiving and redistributing messages between group members. In the case of group messaging, the delivery service may also be responsible for acting as a "broadcaster" where the sender sends a single message to a group which is then forwarded to each recipient in the group by the DS. The DS is also responsible for storing and delivering initial public key material required by clients in order to proceed with the group secret key establishment process.

In many systems, the AS and the DS are actually operated by the same entity and may even be the same server. However, they are logically distinct and, in other systems, may be operated by different entities, hence we show them as being separate here. Other partitions are also possible, such as having a separate directory server.

A typical group messaging scenario might look like this:

1. Alice, Bob and Charlie create accounts with a messaging service and obtain credentials from the AS.

2. Alice, Bob and Charlie authenticate to the DS and store some initial keying material which can be used to send encrypted messages to them for the first time. This keying material is authenticated with their long term credentials.

3. When Alice wants to send a message to Bob and Charlie, she contacts the DS and looks up their initial keying material. She
uses these keys to establish a new set of keys which she can use to send encrypted messages to Bob and Charlie. She then sends the encrypted message(s) to the DS, which forwards them to the recipients.

4. Bob and/or Charlie respond to Alice’s message. Their messages might trigger a new key derivation step which allows the shared group key to be updated to provide post-compromise security (Section 3.2.2.1).

Clients may wish to do the following:

- create a group by inviting a set of other clients;
- add one or more clients to an existing group;
- remove one or more members from an existing group;
- join an existing group;
- leave a group;
- send a message to everyone in the group;
- receive a message from someone in the group.

At the cryptographic level, clients in groups (and by extension Members) are peers. For instance, any client can add another client to a group. This is in contrast to some designs in which there is a single group controller who can modify the group. MLS is compatible with having group administration restricted to certain users, but we assume that those restrictions are enforced by authentication and access control at the application layer. Thus, for instance, while it might be technically possible for any member to send a message adding a new client to a group, the group might have the policy that only certain members are allowed to make changes and thus other members can ignore or reject such a message from an unauthorized user.

2.1. Group, Members and Clients

Informally, a group is a set of users who possibly use multiple endpoint devices to interact with the Messaging Service. These members will typically correspond to end-user devices such as phones, web clients or other devices running MLS, which are called clients.

Each client owns at least one long term identity key pair that uniquely defines its identity to other clients or members in the group.
Group. Because a single user may operate multiple devices simultaneously (e.g., a desktop and a phone) or sequentially (e.g., replacing one phone with another), the formal definition of a group in MLS is the set of clients that has knowledge of the shared group secret established in the group key establishment phase of the protocol. Multiple user devices can be grouped, appearing as one virtual client to the rest of the group.

In some messaging systems, clients belonging to the same user must all share the same identity key pair, but MLS does not assume this. The MLS architecture considers the more general case and allows for important use cases, such as a member adding a new client when all their existing clients are offline.

MLS has been designed to provide similar security guarantees to all clients, for all group sizes, even when it reduces to only two clients.

### 2.2. Authentication Service

The basic function of the Authentication Service (AS) is to provide a trusted mapping from user identities (usernames, phone numbers, etc.), to long-term identity keys, which may either be one per client or may be shared amongst the clients attached to a user. It typically acts as:

- A certification authority, or similar service, which signs some sort of portable credential binding an identity to a key;
- A directory server which provides the key for a given identity (presumably this connection is secured via some form of transport security such as TLS).

By definition, the AS is invested with a large amount of trust. A malicious AS can impersonate - or allow an attacker to impersonate - any user of the system. This risk can be mitigated by publishing the binding between identities and keys in a public log such as Key Transparency (KT) [KeyTransparency]. It is possible to build a functional MLS system without any kind of public key logging, but such a system will necessarily be somewhat vulnerable to attack by a malicious or untrusted AS.

### 2.3. Delivery Service

The Delivery Service (DS) is expected to play multiple roles in the Messaging Service architecture:
o To act as a directory service providing the initial keying material for clients to use. This allows a client to establish a shared key and send encrypted messages to other clients even if the other client is offline.

o To route messages between clients and to act as a message broadcaster, taking in one message and forwarding it to multiple clients (also known as "server side fanout").

Depending on the level of trust given by the group to the Delivery Service, the functional and security guarantees provided by MLS may differ.

2.3.1. Key Storage

Upon joining the system, each client stores its initial cryptographic key material with the DS. This key material represents the initial contribution that will be used in the establishment of the shared group secret. This initial keying material is authenticated using the client’s identity key. Thus, the client stores:

o A credential from the Authentication service attesting to the binding between the user and the client’s identity key.

o The client’s initial keying material signed with the client’s identity key.

As noted above, users may own multiple clients, each with their own keying material, and thus there may be multiple entries stored by each user.

The Delivery Service is also responsible for allowing users to add, remove or update their initial keying material and to ensure that the identifier for these keys are unique across all keys stored on the DS.

2.3.2. Key Retrieval

When a client wishes to establish a group and send an initial message to that group, it contacts the DS and retrieves the initial key material for each other client, verifies it using the identity key, and from those forms the group secret, which it can use for the encryption of messages.
2.3.3. Delivery of messages and attachments

The DS’s main responsibility is to ensure delivery of messages. Specifically, we assume that DSs provide:

- **Reliable delivery**: when a message is provided to the DS, it is eventually delivered to all clients.

- **In-order delivery**: messages are delivered to the group in the order they are received from a given client and in approximately the order in which they are sent by clients. The latter is an approximate guarantee because multiple clients may send messages at the same time and so the DS needs some latitude in enforcing ordering across clients.

- **Consistent ordering**: the DS must ensure that all clients have the same view of message ordering for cryptographically relevant operations. This means that the DS MUST enforce global consistency of the ordering of these messages while MLS provides causal consistency of the application messages for each sender.

Note that the DS may provide ordering guarantees by ensuring in-order delivery or by providing messages with some kind of sequence information and allowing clients to reorder on receipt.

The MLS protocol itself can verify these properties. For instance, if the DS reorders messages from a client or provides different clients with inconsistent orderings, then clients can detect this misconduct. However, MLS need not provide mechanisms to recover from a misbehaving DS.

Note that some forms of DS misbehavior are still possible and difficult to detect. For instance, a DS can simply refuse to relay messages to and from a given client. Without some sort of side information, other clients cannot generally distinguish this form of Denial of Service (DoS) attack.

2.3.4. Membership knowledge

Group membership is itself sensitive information and MLS is designed so that neither the DS nor the AS need have static knowledge of which clients are in which group. However, they may learn this information through traffic analysis. For instance, in a server side fanout model, the DS learns that a given client is sending the same message to a set of other clients. In addition, there may be applications of MLS in which the group membership list is stored on some server associated with the MS.
2.3.5. Membership and offline members

Because Forward Secrecy (FS) and Post-Compromise Security (PCS) rely on the deletion and replacement of keying material, any client which is persistently offline may still be holding old keying material and thus be a threat to both FS and PCS if it is later compromised. MLS does not inherently defend against this problem, but MLS-using systems can enforce some mechanism for doing so. Typically this will consist of evicting clients which are idle for too long, thus containing the threat of compromise. The precise details of such mechanisms are a matter of local policy and beyond the scope of this document.

3. System Requirements

3.1. Functional Requirements

MLS is designed as a large scale group messaging protocol and hence aims to provide performance and safety to its users. Messaging systems that implement MLS provide support for conversations involving two or more members, and aim to scale to approximately 50,000 members, typically including many users using multiple devices.

3.1.1. Asynchronous Usage

No operation in MLS requires two distinct users or clients to be online simultaneously. In particular, clients participating in conversations protected using MLS can update shared keys, add or remove new members, and send messages and attachments without waiting for another user’s reply.

Messaging systems that implement MLS provide a transport layer for delivering messages asynchronously and reliably.

3.1.2. Recovery After State Loss

Conversation participants whose local MLS state is lost or corrupted can reinitialize their state and continue participating in the conversation. This may entail some level of message loss, but does not result in permanent exclusion from the group.

3.1.3. Support for Multiple Devices

It is typically expected for users within Group to own different devices.
A new device can be added to a group by sharing of an existing client secrets or be considered as a new client by the protocol. This client will not gain access to the history even if it is owned by someone who owns another member of the Group. Restoring history is typically not allowed at the protocol level but applications can elect to provide such a mechanism outside of MLS.

3.1.4. Extensibility / Pluggability

Messages that do not affect the group state can carry an arbitrary payload with the purpose of sharing that payload between group members. No assumptions are made about the format of the payload.

3.1.5. Privacy

The protocol is designed in a way that limits the server-side (AS and DS) metadata footprint. The DS only persists data required for the delivery of messages and avoid Personally Identifiable Information (PII) or other sensitive metadata wherever possible. A Messaging Service provider that has control over both the AS and the DS, will not be able to correlate encrypted messages forwarded by the DS, with the initial public keys signed by the AS.

3.1.6. Federation

The protocol aims to be compatible with federated environments. While this document does not specify all necessary mechanisms required for federation, multiple MLS implementations can interoperate to form federated systems if they use compatible wire encodings.

3.1.7. Compatibility with future versions of MLS

It is important that multiple versions of MLS be able to coexist in the future. Thus, MLS offers a version negotiation mechanism; this mechanism prevents version downgrade attacks where an attacker would actively rewrite messages messages with a lower protocol version than the ones originally offered by the endpoints. When multiple versions of MLS are available, the negotiation protocol guarantees that the version agreed upon will be the highest version supported in common by the group.

3.2. Security Requirements
3.2.1. Connections between Clients and Servers (one-to-one)

We assume that all transport connections are secured via some transport layer security mechanism such as TLS [I-D.ietf-tls-tls13]. However, as noted above, the security of MLS will generally survive compromise of the transport layer, so long as identity keys provided by the AS are authenticated at a minimum.

3.2.2. Message Secrecy and Authentication

The trust establishment step of the MLS protocol is followed by a conversation protection step where encryption is used by clients to transmit authenticated messages to other clients through the DS. This ensures that the DS does not have access to the group’s private content.

MLS aims to provide secrecy, integrity and authentication for all messages.

Message Secrecy in the context of MLS means that only intended recipients (current group members), can read any message sent to the group, even in the context of an active adversary as described in the threat model.

Message Integrity and Authentication mean that an honest client can only accept a message if it was sent by a group member and that a client cannot send a message which other clients would accept as being from a different client.

A corollary to this statement is that the AS and the DS cannot read the content of messages sent between members as they are not members of the group. MLS optionally provides additional protections regarding traffic analysis so as to reduce the ability of adversaries, to deduce the content of the messages depending on (for example) their size. One of these protections includes padding messages in order to produce ciphertexts of standard length. While this protection is highly recommended it is not mandatory as it can be costly in terms of performance for clients and the MS.

Message content can be deniable if the signature keys are exchanged over a deniable channel prior to signing messages.

3.2.2.1. Forward and Post-Compromise Security

MLS provides additional protection regarding secrecy of past messages and future messages. These cryptographic security properties are Forward Secrecy (FS) and Post-Compromise Security (PCS).
FS means that access to all encrypted traffic history combined with an access to all current keying material on clients will not defeat the secrecy properties of messages older than the oldest key of the compromised client. Note that this means that clients have the extremely important role of deleting appropriate keys as soon as they have been used with the expected message, otherwise the secrecy of the messages and the security for MLS is considerably weakened.

PCS means that if a group member is compromised at some time T but subsequently performs an update at some time T’, then all MLS guarantees apply to messages sent after time T’. For example, if an adversary learns all secrets known to Alice at time T, including both Alice’s secrets and all shared group secrets, but Alice performs a key update at time T’, which is not under the control of the adversary, then the adversary is unable to violate any of the MLS security properties after time T’.

Both of these properties are satisfied even against compromised DSs and ASs.

3.2.2.2. Membership Changes

MLS aims to provide agreement on group membership, meaning that all group members have agreed on the list of current group members.

Some applications may wish to enforce ACLs to limit addition or removal of group members, to privileged clients or users. Others may wish to require authorization from the current group members or a subset thereof. Regardless, MLS does not allow addition or removal of group members without informing all other members.

Once a client is part of a group, the set of devices controlled by the user can only be altered by an authorized member of the group. This authorization could depend on the application: some applications might want to allow certain other members of the group to add or remove devices on behalf of another member, while other applications might want a more strict policy and allow only the owner of the devices to add or remove them at the potential cost of weaker PCS guarantees.

Members who are removed from a group do not enjoy special privileges: compromise of a removed group member does not affect the security of messages sent after their removal but might affect previous messages if the group secrets have not been deleted properly.
3.2.2.3. Security of Attachments

The security properties expected for attachments in the MLS protocol are very similar to the ones expected from messages. The distinction between messages and attachments stems from the fact that the typical average time between the download of a message and the one from the attachments may be different. For many reasons (a typical reason being the lack of high bandwidth network connectivity), the lifetime of the cryptographic keys for attachments is usually higher than for messages, hence slightly weakening the PCS guarantees for attachments.

3.2.2.4. Denial of Service

In general we do not consider Denial of Service (DoS) resistance to be the responsibility of the protocol. However, it should not be possible for anyone aside from the DS to perform a trivial DoS attack from which it is hard to recover.

3.2.2.5. Non-Repudiation vs Deniability

As described in Section 4.4, MLS provides strong authentication within a group, such that a group member cannot send a message that appears to be from another group member. Additionally, some services require that a recipient be able to prove to the messaging service that a message was sent by a given client, in order to report abuse. MLS supports both of these use cases. In some deployments, these services are provided by mechanisms which allow the receiver to prove a message’s origin to a third party (this if often called "non-repudiation"), but it should also be possible to operate MLS in a "deniable" mode where such proof is not possible. [[OPEN ISSUE: Exactly how to supply this is still a protocol question.]]

4. Security Considerations

MLS adopts the Internet threat model [RFC3552] and therefore assumes that the attacker has complete control of the network. It is intended to provide the security services described in in the face of such attackers. In addition, these guarantees are intended to degrade gracefully in the presence of compromise of the transport security links as well as of both clients and elements of the messaging system, as described in the remainder of this section.

4.1. Transport Security Links

[TODO: Mostly DoS, message suppression, and leakage of group membership.]
4.2. Delivery Service Compromise

MLS is intended to provide strong guarantees in the face of compromise of the DS. Even a totally compromised DS should not be able to read messages or inject messages that will be acceptable to legitimate clients. It should also not be able to undetectably remove, reorder or replay messages.

However, a DS can mount a variety of DoS attacks on the system, including total DoS attacks (where it simply refuses to forward any messages) and partial DoS attacks (where it refuses to forward messages to and from specific clients). As noted in Section 2.3.3, these attacks are only partially detectable by clients without an out-of-band channel. Ultimately, failure of the DS to provide reasonable service must be dealt with as a customer service matter, not via technology.

Because the DS is responsible for providing the initial keying material to clients, it can provide stale keys. This does not inherently lead to compromise of the message stream, but does allow it to attack forward security to a limited extent. This threat can be mitigated by having initial keys expire.

4.3. Authentication Service Compromise

A compromised AS is a serious matter, as the AS can provide incorrect or adversarial identities to clients. As noted in Section 2.2, mitigating this form of attack requires some sort of transparency/logging mechanism. Without such a mechanism, MLS will only provide limited security against a compromised AS.

4.4. Client Compromise

In general, MLS only provides limited protection against compromised clients. When the client secrets are compromised, then the attacker will obviously be able to decrypt any messages for groups in which the client is a member. It will also be able to send messages impersonating the compromised client until the client updates its keying material (see Section 3.2.2.1). MLS attempts to provide some security in the face of client compromise.

In addition, a client cannot send a message to a group which appears to be from another client with a different identity. Note that if devices from the same user share keying material, then one will be able to impersonate another.

Finally, clients should not be able to perform DoS attacks Section 3.2.2.4.
5. IANA Considerations

This document makes no requests of IANA.

6. Contributors

- Katriel Cohn-Gordon
  University of Oxford
  me@katriel.co.uk

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

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

- Jon Millican
  Facebook
  jmillican@fb.com

- Raphael Robert
  Wire
  raphael@wire.com

7. Informative References

[I-D.ietf-tls-tls13]

[KeyTransparency]

[MLSPROTO]


Authors’ Addresses

Emad Omara
Google
Email: emadomara@google.com

Benjamin Beurdouche
INRIA
Email: benjamin.beurdouche@inria.fr

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Srinivas Inguva
Twitter
Email: singuva@twitter.com

Albert Kwon
MIT
Email: kwonal@mit.edu

Alan Duric
Wire
Email: alan@wire.com

The Messaging Layer Security (MLS) Protocol

draft-ietf-mls-protocol-07

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 2020.
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-07
- Initial version of the Tree based Application Key Schedule (*)
- Initial definition of the Init message for group creation (*)
- Fix issue with the transcript used for newcomers (*)
- Clarifications on message framing and HPKE contexts (*)

draft-06
- Reorder blanking and update in the Remove operation (*)
- Rename the GroupState structure to GroupContext (*)
- Rename UserInitKey to ClientInitKey
- Resolve the circular dependency that draft-05 introduced in the confirmation MAC calculation (*)
- Cover the entire MLSPlaintext in the transcript hash (*)

draft-05
- Common framing for handshake and application messages (*)
- Handshake message encryption (*)
- Convert from literal state to a commitment via the "tree hash" (*)
- Add credentials to the tree and remove the "roster" concept (*)
- Remove the secret field from tree node values

**draft-04**
- Updating the language to be similar to the Architecture document
- ECIES is now renamed in favor of HPKE (*)
- Using a KDF instead of a Hash in TreeKEM (*)

**draft-03**
- Added ciphersuites and signature schemes (*)
- Re-ordered fields in UserInitKey to make parsing easier (*)
- Fixed inconsistencies between Welcome and GroupState (*)
- Added encryption of the Welcome message (*)

**draft-02**
- Removed ART (*)
- Allowed partial trees to avoid double-joins (*)
- Added explicit key confirmation (*)

**draft-01**
- Initial description of the Message Protection mechanism. (*)
- Initial specification proposal for the Application Key Schedule using the per-participant chaining of the Application Secret design. (*)
- Initial specification proposal for an encryption mechanism to protect Application Messages using an AEAD scheme. (*)
- Initial specification proposal for an authentication mechanism of Application Messages using signatures. (*)
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. In the scenario where multiple devices share the cryptographic material the client is referred to as a "virtual" client.

Group: A collection of clients with shared cryptographic state.

Member: A client that is included in the shared state of a group, hence has access to the group’s secrets.

Initialization Key: A short-lived HPKE key pair used to introduce a new client to a group. Initialization keys are published for each client (ClientInitKey).
Leaf Key: A secret that represents a member’s contribution to the group secret (so called because the members’ leaf keys are the leaves in the group’s ratchet tree).

Identity Key: A long-lived signing key pair used to authenticate the sender of a message.

Terminology specific to tree computations is described in Section 5.

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

3. Basic Assumptions

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

- A long-term identity key provider which allows clients to authenticate protocol messages in a group. These keys MUST be kept for the lifetime of the group as there is no mechanism in the protocol for changing a client’s identity key.

- A broadcast channel, for each group, which will relay a message to all members of a group. For the most part, we assume that this channel delivers messages in the same order to all participants. (See Section 10 for further considerations.)

- A directory to which clients can publish initialization keys and download initialization keys for other participants.

4. Protocol Overview

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

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

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

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

- Adding a member, initiated by a current member;
- Updating the leaf secret of a member;
- Removing a member.

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

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

When a client A wants to establish a group with B and C, it first downloads ClientInitKeys for B and C. It then initializes a group state containing only itself and uses the ClientInitKeys to compute Welcome and Add messages to add B and C, in a sequence chosen by A. The Welcome messages are sent directly to the new members (there is no need to send them to the group). The Add messages are broadcasted to the group, and processed in sequence by B and C. Messages received before a client has joined the group are ignored. Only after A has received its Add messages back from the server does it update its state to reflect their addition.
Subsequent additions of group members proceed in the same way. Any member of the group can download an ClientInitKey for a new client and broadcast an Add message that the current group can use to update their state and the new client can use to initialize its state.

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

It is left to the application to determine the interval of time between Update messages. This policy could require a change for each message, or it could require sending an update every week or more.
Members are removed from the group in a similar way, as an update is effectively removing the old leaf from the group. Any member of the group can generate a Remove message that adds new entropy to the group state that is known to all members except the removed member. After other participants have processed this message, the group’s secrets will be unknown to the removed participant. Note that this does not necessarily imply that any member is actually allowed to evict other members; groups can layer authentication-based access control policies on top of these basic mechanism.

5. Ratchet Trees

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

5.1. Tree Computation Terminology

Trees consist of _nodes_. A node is a _leaf_ if it has no children, and a _parent_ otherwise; note that all parents in our trees have precisely two children, a _left_ child and a _right_ child. A node is the _root_ of a tree if it has no parents, and _intermediate_ if it has both children and parents. The _descendants_ of a node are
that node, its children, and the descendants of its children, and we say a tree _contains_ a node if that node is a descendant of the root of the tree. Nodes are _siblings_ if they share the same parent.

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

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

A binary tree is _left-balanced_ if for every parent, either the parent is balanced, or the left subtree of that parent is the largest full subtree that could be constructed from the leaves present in the parent’s own subtree. Note that given a list of "n" items, there is a unique left-balanced binary tree structure with these elements as leaves. In such a left-balanced tree, the "k-th" leaf node refers to the "k-th" leaf node in the tree when counting from the left, starting from 0.

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

For example, in the below tree:

- The direct path of C is (C, CD, ABCD)
- The copath of C is (D, AB, EFG)
- The frontier of the tree is (ABCD, EF, G)
Each node in the tree is assigned an _node index_, starting at zero and running from left to right. A node is a leaf node if and only if it has an even index. The node indices for the nodes in the above tree are as follows:

- 0 = A
- 1 = AB
- 2 = B
- 3 = ABCD
- 4 = C
- 5 = CD
- 6 = D
- 7 = ABCDEFG
- 8 = E
- 9 = EF
- 10 = F
- 11 = EFG
- 12 = G

(Note that left-balanced binary trees are the same structure that is used for the Merkle trees in the Certificate Transparency protocol [I-D.ietf-trans-rfc6962-bis].)
The leaves of the tree are indexed separately, using a _leaf index_, since the protocol messages only need to refer to leaves in the tree. Like nodes, leaves are numbered left to right. Note that given the above numbering, a node is a leaf node if and only if it has an even node index, and a leaf node's leaf index is half its node index. The leaf indices in the above tree are as follows:

- 0 = A
- 1 = B
- 2 = C
- 3 = D
- 4 = E
- 5 = F
- 6 = G

### 5.2. Ratchet Tree Nodes

A particular instance of a ratchet tree is based on the following cryptographic primitives, defined by the ciphersuite in use:

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

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

- A private key (only within direct path, see below)
- A public key
- A credential (only for leaf nodes)

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

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

- The resolution of a non-blank node is a one element list containing the node itself
- The resolution of a blank leaf node is the empty list
- The resolution of a blank intermediate node is the result of concatenating the resolution of its left child with the resolution of its right child, in that order

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

```
    _
   / \
  /   \
_/     \
A   _   C   D
0 1 2 3 4 5 6
```

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

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

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

5.3. Views of a Ratchet Tree

We generally assume that each participant maintains a complete and up-to-date view of the public state of the group’s ratchet tree, including the public keys for all nodes and the credentials associated with the leaf nodes.

No participant in an MLS group has full knowledge of the secret state of the tree, i.e., private keys associated to the nodes. Instead, each member is assigned to a leaf of the tree, which determines the
set of secret state known to the member. The credential stored at
that leaf is one provided by the member.

In particular, MLS maintains the members’ views of the tree in such a
way as to maintain the _tree invariant_:

The private key for a node in the tree is known to a member of
the group if and only if that member’s leaf is a descendant of
the node or equal to it.

In other words, each member holds the private keys for nodes in its
direct path, and no others.

5.4. Ratchet Tree Updates

Nodes in a tree are always updated along the direct path from a leaf
to the root. The generator of the update chooses a random secret
value "path_secret[0]", and generates a sequence of "path secrets",
one for each node from the leaf to the root. That is, path_secret[0]
is used for the leaf, path_secret[1] for its parent, and so on. At
each step, the path secret is used to derive a new secret value for
the corresponding node, from which the node’s key pair is derived.

\[
\begin{align*}
\text{path_secret}[n] &= \text{HKDF-Expand-Label}(\text{path_secret}[n-1], \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{"path", "", Hash.Length}) \\
\text{node_secret}[n] &= \text{HKDF-Expand-Label}(\text{path_secret}[n], \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{"node", "", Hash.Length}) \\
\text{node_priv}[n], \text{node_pub}[n] &= \text{Derive-Key-Pair}(\text{node_secret}[n])
\end{align*}
\]

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

\[
\begin{align*}
G \\
/ \ \\
/ \ \\
E \quad F \\
/ \ \\
A \quad B \quad C \quad D
\end{align*}
\]

If the second participant (B) subsequently generates an update based
on a secret \(X\), then the sender would generate the following sequence
of path secrets and node secrets:
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
  / \  /  \  
A ns[0] C  D
```

5.5.  Synchronizing Views of the Tree

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

To perform an update for a leaf, the sender broadcasts to the group the following information for each node in the direct path of the leaf, as well as the root:

- The public key for the node
- Zero or more encrypted copies of the path secret corresponding to the node

The path secret value for a given node is encrypted for the subtree corresponding to the parent’s non-updated child, i.e., the child on the copath of the leaf node. There is one encrypted path secret for each public key in the resolution of the non-updated child. In particular, for the leaf node, there are no encrypted secrets, since a leaf node has no children.

The recipient of an update processes it with the following steps:

1.  Compute the updated path secrets
2.  Identify a node in the direct path for which the local member is in the subtree of the non-
updated child * Identify a node in the resolution of the copath node for which this node has a private key * Decrypt the path secret for the parent of the copath node using the private key from the resolution node * Derive path secrets for ancestors of that node using the algorithm described above * The recipient SHOULD verify that the received public keys agree with the public keys derived from the new node_secret values

2. Merge the updated path secrets into the tree * Replace the public keys for nodes on the direct path with the received public keys * For nodes where an updated path secret was computed in step 1, compute the corresponding node secret and node key pair and replace the values stored at the node with the computed values.

For example, in order to communicate the example update described in the previous section, the sender would transmit the following values:

<table>
<thead>
<tr>
<th>Public Key</th>
<th>Ciphertext(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pk(ns[2])</td>
<td>E(pk(C), ps[2]), E(pk(D), ps[2])</td>
</tr>
<tr>
<td>pk(ns[1])</td>
<td>E(pk(A), ps[1])</td>
</tr>
<tr>
<td>pk(ns[0])</td>
<td></td>
</tr>
</tbody>
</table>

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

6. Cryptographic Objects

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
- 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;

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

This ciphersuite uses the following primitives:

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

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

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

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

Implementations SHOULD use the approach specified in [RFC7748] to calculate the Diffie-Hellman shared secret. Implementations MUST check whether the computed Diffie-Hellman shared secret is the all-zero value and abort if so, as described in Section 6 of [RFC7748]. If implementers use an alternative implementation of these elliptic
6.1.1. P-256, SHA-256, and AES-128-GCM

This ciphersuite uses the following primitives:

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

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

- KEM: 0x0001 = DHKEM(P-256)
- KDF: 0x0001 = HKDF-SHA256
- AEAD: 0x0001 = AES-GCM-128

Given an octet string X, the private key produced by the Derive-Key-Pair operation is SHA-256(X), interpreted as a big-endian integer. The corresponding public key is the result of multiplying the standard P-256 base point by this integer.

P-256 ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because MLS does not directly use this secret for anything other than for computing other secrets.)

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

6.2. Credentials

A member of a group authenticates the identities of other participants by means of credentials issued by some authentication system, e.g., a PKI. Each type of credential MUST express the following data:

- The public key of a signature key pair
- The identity of the holder of the private key
- The signature scheme that the holder will use to sign MLS messages

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

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

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

```c
struct {
    CredentialType credential_type;
    select (credential_type) {
        case basic:
            BasicCredential;
        case x509:
            opaque cert_data<1..2^24-1>;
    }
} Credential;
```

6.3. Tree Hashes

To allow group members to verify that they agree on the cryptographic state of the group, this section defines a scheme for generating a hash value that represents the contents of the group’s ratchet tree and the members’ credentials.
The hash of a tree is the hash of its root node, which we define recursively, starting with the leaves. The hash of a leaf node is the hash of a "LeafNodeHashInput" object:

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

struct {
    HPKEPublicKey public_key;
    Credential credential;
} LeafNodeInfo;

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

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

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

```c
struct {
    uint8 hash_type = 1;
    optional<HPKEPublicKey> public_key;
    opaque left_hash<0..255>;
    opaque right_hash<0..255>;
} ParentNodeHashInput
```

The "left_hash" and "right_hash" fields hold the hashes of the node’s left and right children, respectively. The "public_key" field holds the hash of the public key stored at this node, represented as an "optional<HPKEPublicKey>" object, which is null if and only if the node is blank.

6.4. Group State

Each member of the group maintains a GroupContext object that summarizes the state of the group:
struct {
    opaque group_id<0..255>;
    uint32 epoch;
    opaque tree_hash<0..255>;
    opaque confirmed_transcript_hash<0..255>;
} GroupContext;

The fields in this state have the following semantics:

- The "group_id" field is an application-defined identifier for the group.
- The "epoch" field represents the current version of the group key.
- The "tree_hash" field contains a commitment to the contents of the group’s rachet tree and the credentials for the members of the group, as described in Section 6.3.
- The "confirmed_transcript_hash" field contains a running hash over the handshake messages that led to this state.

When a new member is added to the group, an existing member of the group provides the new member with a Welcome message. The Welcome message provides the information the new member needs to initialize its GroupContext.

Different group operations will have different effects on the group state. These effects are described in their respective subsections of Section 9. The following rules apply to all operations:

- The "group_id" field is constant
- The "epoch" field increments by one for each GroupOperation that is processed
- The "tree_hash" is updated to represent the current tree and credentials
- The "confirmed_transcript_hash" is updated with the data for an MLSPlaintext message encoding a group operation in two parts:
struct {
    opaque group_id<0..255>
    uint32 epoch
    uint32 sender
    ContentType content_type = handshake
    GroupOperation operation
} MLSPlaintextOpContent;

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

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

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

This structure incorporates everything in an MLSPlaintext up to the
confirmation field in the transcript that is included in that
confirmation field (via the GroupContext). The confirmation and
signature fields are then included in the transcript for the next
operation. The interim transcript hash is passed to new members in
the WelcomeInfo struct, and enables existing members to incorporate a
handshake message into the transcript without having to store the
whole MLSPlaintextOpAuthData structure.

When a new one-member group is created (which requires no
GroupOperation), the "interim_transcript_hash" field is set to the
zero-length octet string.

6.5. Direct Paths

As described in Section 5.4, each MLS message needs to transmit node
values along the direct path of a leaf. The path contains a public
key for the leaf node, and a public key and encrypted secret value
for intermediate nodes in the path. In both cases, the path is
ordered from the leaf to the root; each node MUST be the parent of
its predecessor.
struct {
    HPKEPublicKey ephemeral_key;
    opaque ciphertext<0..2^16-1>;
} HPKECiphertext;

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

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

The length of the "encrypted_path_secret" vector MUST be zero for the first node in the path. For the remaining elements in the vector, the number of ciphertexts in the "encrypted_path_secret" vector MUST be equal to the length of the resolution of the corresponding copath node. Each ciphertext in the list is the encryption to the corresponding node in the resolution.

The HPKECiphertext values are computed as:

ephemeral_key, context = SetupBaseI(node_public_key, "")
ciphertext = context.Seal("", path_secret)

where "node_public_key" is the public key of the node that the path secret is being encrypted for, and the functions "SetupBaseI" and "Seal" are defined according to [I-D.irtf-cfrg-hpke].

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

6.6. Key Schedule

Group keys are derived using the HKDF-Extract and HKDF-Expand functions as defined in [RFC5869], as well as the functions defined below:
HKDF-Expand-Label(Secret, Label, Context, Length) = HKDF-Expand(Secret, HkdfLabel, Length)

Where HkdfLabel is specified as:

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

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

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

- HKDF-Extract takes its salt argument from the top and its IKM argument from the left
- Derive-Secret takes its Secret argument from the incoming arrow

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

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

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

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

- Metadata (sender information)
- Handshake messages
- Application messages

The sender information used to look up the key for the content encryption is encrypted under AEAD using a random nonce and the sender_data_key which is derived from the sender_data_secret as follows:

\[
\text{sender_data_key} = \text{HKDF-Expand-Label(sender_data_secret, "sd key", ",", key_length)}
\]

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

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

Here the value [sender] represents the index of the member that will use this key to send, encoded as a uint32.

For application messages, a chain of keys is derived for each sender in a similar fashion. This allows forward secrecy at the level of application messages within and out of an epoch. A step in this chain (the second subscript) is called a "generation". The details of application key derivation are described in the Section 11.1 section below.

7. Initialization Keys

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

A ClientInitKey object specifies 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 12.4). ClientInitKeys also contain an identifier chosen by the client, which the client MUST assure uniquely identifies a given ClientInitKey object among the set of ClientInitKeys 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 ClientInitKey object with an invalid signature field MUST be considered malformed. The input to the signature computation comprises all of the fields except for the signature field.
uint8 ProtocolVersion;

struct {
    opaque client_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>;
} ClientInitKey;

8. Message Framing

Handshake and application messages use a common framing structure. This framing provides encryption to assure confidentiality within the group, as well as signing to authenticate the sender within the group.

The two main structures involved are MLSPlaintext and MLSCiphertext. MLSCiphertext represents a signed and encrypted message, with protections for both the content of the message and related metadata. MLSPlaintext represents a message that is only signed, and not encrypted. Applications SHOULD use MLSCiphertext to encode both application and handshake messages, but MAY transmit handshake messages encoded as MLSPlaintext objects in cases where it is necessary for the delivery service to examine such messages.
enum {
    invalid(0),
    handshake(1),
    application(2),
    (255)
} ContentType;

struct {
    opaque group_id<0..255>;
    uint32 epoch;
    uint32 sender;
    ContentType content_type;

    select (MLSPlaintext.content_type) {
        case handshake:
            GroupOperation operation;
            opaque confirmation<0..255>;

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

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

struct {
    opaque group_id<0..255>;
    uint32 epoch;
    ContentType content_type;
    opaque sender_data_nonce<0..255>;
    opaque encrypted_sender_data<0..255>;
    opaque ciphertext<0..2^32-1>;
} MLSCiphertext;

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

- Gather the required metadata:
  - Group ID
  - Epoch
  - Content Type
  - Nonce
* Sender index

* Key generation

- Sign the plaintext metadata — the group ID, epoch, sender index, and content type — as well as the message content

- Randomly generate sender_data_nonce and encrypt the sender information using it and the key derived from the sender_data_secret

- Encrypt the content using a content encryption key identified by the metadata

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

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

8.1. Metadata Encryption

The "sender data" used to look up the key for the content encryption is encrypted under AEAD using the MLSCiphertext sender_data_nonce and the sender_data_key from the keyschedule. It is encoded as an object of the following form:

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

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

```c
struct {
    opaque group_id<0..255>;
    uint32 epoch;
    ContentType content_type;
    opaque sender_data_nonce<0..255>;
} MLSCiphertextSenderDataAAD;
```

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

The signature field in an MLSPlaintext object is computed using the signing private key corresponding to the credential at the leaf in the tree indicated by the sender field. The signature covers the plaintext metadata and message content, i.e., all fields of MLSPlaintext except for the "signature" field.

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

```c
struct {
    opaque content[length_of_content];
    uint8 signature[MLSCiphertextContent.sig_len];
    uint16 sig_len;
    uint8  marker = 1;
    uint8  zero_padding[length_of_padding];
} MLSCiphertextContent;
```

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

The Additional Authenticated Data (AAD) input to the encryption contains an object of the following form, with the values used to identify the key and nonce:

```c
struct {
    opaque group_id<0..255>;
    uint32 epoch;
    ContentType content_type;
    opaque sender_data_nonce<0..255>;
    opaque encrypted_sender_data<0..255>;
} MLSCiphertextContentAAD;
```

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

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

- Group initialization
- A member adding a new client
- A member updating its leaf key
- A member deleting another 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 `GroupOperation` message that accomplishes a change to the group state. It is carried in an `MLSPacket` message that provides a signature by the sender of the message. Applications may choose to send handshake messages in encrypted form, as MLSCiphertext messages.

```c
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;
```

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

1. If the handshake message is encrypted (i.e., encoded as an MLSCiphertext object), decrypt it following the procedures described in Section 8.
2. Verify that the "epoch" field of enclosing MLSPlaintext message is equal the "epoch" field of the current GroupContext object.

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

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

5. Use the "confirmation_key" for the new epoch to compute the confirmation MAC for this message, as described below, and verify that it is the same as the "confirmation" field in the MLSPlaintext object.

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

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

MLSPaintext.confirmation = 
HMAC(confirmation_key, GroupContext.transcript_hash)

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

[[ 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. ]]]

9.1. Init

A group can always be created by initializing a one-member group and using adding members individually. For cases where the initial list of members is known, the Init message allows a group to be created more efficiently.
struct {
    opaque group_id<0..255>;
    ProtocolVersion version;
    CipherSuite cipher_suite;
    ClientInitKey members<0..2^32-1>;
    DirectPath path;
} Init;

The creator of the group constructs an Init message as follows:

- Fetch a UserInitKey for each member (including the creator)
- Identify a protocol version and cipher suite that is supported by all proposed members.
- Construct a ratchet tree with its leaves populated with the public keys and credentials from the UserInitKeys of the members, and all other nodes blank.
- Generate a fresh leaf key pair for the first leaf
- Compute its direct path in this ratchet tree

Each member of the newly-created group initializes its state from the Init message as follows:

- Note the group ID, protocol version, and cipher suite in use
- Construct a ratchet tree as above
- Update the cached ratchet tree by replacing nodes in the direct path from the first leaf using the direct path
- Update the cached ratchet tree by replacing nodes in the direct path from the first leaf using the information contained in the "path" attribute

The update secret for this interaction, used with an all-zero init secret to generate the first epoch secret, is the "path_secret[i+1]" derived from the "path_secret[i]" associated to the root node. The members learn the relevant path secrets by decrypting one of the encrypted path secrets in the DirectPath and working back to the root (as in normal DirectPath processing).

[[ OPEN ISSUE: This approach leaks the initial contents of the tree to the Delivery Service, unlike the sequential-Add case. ]]
OPEN ISSUE: It might be desirable for the group creator to be able to "pre-warm" the tree, by providing values for some nodes not on its direct path. This would violate the tree invariant, so we would need to figure out what mitigations would be necessary.

9.2. 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 GroupContext 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 ClientInitKey, corresponding to the indicated ciphersuite. The "add_key_nonce" field contains the key and nonce used to encrypt the corresponding Add message; if it is not encrypted, then this field MUST be set to the null optional value.
struct {
    HPKEPublicKey public_key;
    optional<Credential> credential;
} RatchetNode;

struct {
    opaque key<0..255>;
    opaque nonce<0..255>;
} KeyAndNonce;

struct {
    ProtocolVersion version;
    opaque group_id<0..255>;
    uint32 epoch;
    optional<RatchetNode> tree<1..2^32-1>;
    opaque interim_transcript_hash<0..255>;
    opaque init_secret<0..255>;
    optional<KeyAndNonce> add_key_nonce;
} WelcomeInfo;

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

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

Note that the "init_secret" in the Welcome message is the "init_secret" at the output of the key schedule diagram in Section 6.6. 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 GroupContext 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 GroupContext with information about the new member:

```
struct {
    uint32 index;
    ClientInitKey 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 ClientInitKey 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:

- Prepare a new GroupContext object based on the Welcome message
- 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:

- If the "index" value is equal to the size of the group, increment the size of the group, and extend the tree accordingly
- Verify the signature on the included ClientInitKey; if the signature verification fails, abort
- 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
- Update the ratchet tree by setting to blank all nodes in the direct path of the new node
Set the leaf node in the tree at position "index" to a new node containing the public key from the ClientInitKey in the Add corresponding to the ciphersuite in use, as well as the credential under which the ClientInitKey was signed.

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.

9.3. 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:

- Generate a fresh leaf key pair
- 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:

- 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 "path_secret[i+1]" derived from the "path_secret[i]" associated to the root node.

9.4. Remove

A Remove message is sent by a group member to remove one or more other members from the group. A member MUST NOT use a Remove message to remove themselves from the group. If a member of a group receives a Remove message where the removed index is equal to the signer index, the recipient MUST reject the message as malformed.
struct {
    uint32 removed;
    DirectPath path;
} Remove;

The sender of a Remove message generates it as follows:

- Blank the path from the removed leaf to the root node for the time of the computation
- Truncate the tree such that the rightmost non-blank leaf is the last node of the tree, for the time of the computation
- Generate a fresh leaf key pair
- Compute its direct path in the current ratchet tree, starting from the sender’s leaf

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

- Update the ratchet tree by setting to blank all nodes in the direct path of the removed leaf, and also setting the root node to blank
- Truncate the tree such that the rightmost non-blank leaf is the last node of the tree
- Update the ratchet tree by replacing nodes in the direct path from the sender’s leaf using the information in the Remove message

Note that there must be at least one non-null element in the tree, since any valid GroupContext must have the current member in the tree and self-removal is prohibited.

The "update_secret" resulting from this change is the "path_secret[i+1]" derived from the "path_secret[i]" associated to the root node.

10. Sequencing of State Changes

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

Each handshake message is premised on a given starting state, indicated in its "prior_epoch" field. If the changes implied by a
handshake messages are made starting from a different state, the results will be incorrect.

This need for sequencing is not a problem as long as each time a group member sends a handshake message, it is based on the most current state of the group. In practice, however, there is a risk that two members will generate handshake messages simultaneously, based on the same state.

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

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

As long as handshake messages cannot be merged, there is a risk of starvation. In a sufficiently busy group, a given member may never be able to send a handshake message, because he always loses to other members. The degree to which this is a practical problem will depend on the dynamics of the application.

It might be possible, because of the non-contributivity of intermediate nodes, that update messages could be applied one after the other without the Delivery Service having to reject any handshake message, which would make MLS more resilient regarding the concurrency of handshake messages. The Messaging system can decide to choose the order for applying the state changes. Note that there are certain cases (if no total ordering is applied by the Delivery Service) where the ordering is important for security, i.e. all updates must be executed before removes.

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

10.1. 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.

10.2. 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.

10.3. 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:
Now suppose B and C simultaneously decide to update to X and Y, respectively. They will send out updates of the following form:

\[
\begin{array}{c}
\text{Update from B} \\
\hline
\text{Update from C} \\
\end{array}
\]

\[
\begin{array}{c}
KDF(KDF(X)) \\
\hline
KDF(KDF(Y)) \\
\end{array}
\]

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:

\[
\begin{array}{c}
KDF(KDF(Y)) \\
\hline
KDF(X) \\
\end{array}
\]

11. Application Messages

The primary purpose of the Handshake protocol is to provide an authenticated group key exchange to clients. In order to protect Application messages sent among the members of a group, the Application secret provided by the Handshake key schedule is used to derive nonces and encryption keys for the Message Protection Layer according to the Application Key Schedule. That is, each epoch is equipped with a fresh Application Key Schedule which consist of a tree of Application Secrets as well as one symmetric ratchet per group member.

Each client maintains their own local copy of the Application Key Schedule for each epoch during which they are a group member. They derive new keys, nonces and secrets as needed while deleting old ones as soon as they have been used.

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

11.1. Tree of Application Secrets

The application key schedule begins with the application secrets which are arranged in an "Application Secret Tree" or AS Tree for short; a left balanced binary tree with the same set of nodes and edges as the epoch’s ratchet tree. Each leaf in the AS Tree is associated with the same group member as the corresponding leaf in the ratchet tree. Nodes are also assigned an index according to their position in the array representation of the tree (described in Appendix A). If N is a node index in the AS Tree then left(N) and right(N) denote the children of N (if they exist).

Each node in the tree is assigned a secret. The root’s secret is simply the application_secret of that epoch. (See Section 6.6 for the definition of application_secret.)

\[
\text{astree_node\_\{root\}\_secret} = \text{application\_secret}
\]

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

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

Where ApplicationContext is specified as:

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

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

\[
\text{astree_node\_[N]\_secret} \\
\text{+--> Derive-App-Secret(., "tree", left(N), 0, Hash.length) = astree_node\_[left(N)]\_secret} \\
\text{+--> Derive-App-Secret(., "tree", right(N), 0, Hash.length) = astree_node\_[right(N)]\_secret}
\]
Note that fixing concrete values for GroupContext\_\[n\] and application\_secret completely defines all secrets in the AS Tree.

### 11.2. Sender Ratchets

The secret of a leaf in the AS Tree is used to initiate a symmetric hash ratchet which generates a sequence of keys and nonces. The group member assigned to that leaf uses the j-th key/nonce pair in the sequence to encrypt (using the AEAD) the j-th message they send during that epoch. In particular, each key/nonce pair MUST NOT be used to encrypt more than one message.

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

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

Keys, nonces and secrets of ratchets are derived using Derive-App-Secret. The context in a given call consists of the index of the sender’s leaf in the ratchet tree and the current position in the ratchet. In particular, the index of the sender’s leaf in the ratchet tree is the same as the index of the leaf in the AS Tree used to initialize the sender’s ratchet.

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

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

### 11.3. Deletion Schedule

It is important to delete all security sensitive values S as soon as they, or another value derived from them, is used for encryption or decryption.
More precisely, the values application\_[i][j].key and application\_[i][j].nonce are said to be "consumed" if they were used either to:

- encrypt or (successfully) decrypt a message or
- if a key, nonce or secret derived from S has been consumed.

(This goes both for values derived via Derive-Secret and HKDF-Expand-Label.)

Here, S may be the init_secret, update_secret, epoch_secret, application_secret as well as any secret in the AS Tree or one of the ratchets.

As soon as a group member consumes a value they MUST immediately delete (all representations of) that value. This is crucial to ensuring Forward Secrecy for past messages. Members MAY keep unconsumed values around for some reasonable amount of time even if their generating secret was already consumed (e.g. due to out of order message delivery).

For example, suppose a group member encrypts or (successfully) decrypts a message using the j-th key and nonce in the i-th ratchet. Then, for that member, at least the following values have been consumed and MUST be deleted:

- the init_secret, update_secret, epoch_secret, application_secret of that epoch,
- all node secrets in the AS Tree on the path from the root to the leaf with index i,
- the first j secrets in the i-th ratchet and
- application\_[i][j].key and application\_[i][j].nonce.

Concretely, suppose we have the following AS Tree and ratchet for participant D:
Then if a client uses key KD1 and nonce ND1 during epoch n then it must consume (at least) values G, F, D0, D1, KD1, ND1 as well as the update_secret and init_secret used to derive G (i.e. the application_secret). The client MAY retain (i.e., not consume) the values KD0 and ND0 to allow for out-of-order delivery, and SHOULD retain D2 to allow for processing future messages.

11.4. Further Restrictions

During each epoch senders MUST NOT encrypt more data than permitted by the security bounds of the AEAD scheme used.

Note that each change to the Group through a Handshake message will also set a new application_secret. Hence this change MUST be applied before encrypting any new Application message. This is required both to ensure that any users removed from the group can no longer receive messages and to (potentially) recover confidentiality and authenticity for future messages despite a past state compromise.

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

11.5. Message Encryption and Decryption

The group members MUST use the AEAD algorithm associated with the negotiated MLS ciphersuite to AEAD encrypt and decrypt their Application messages according to the Message Framing section.

The group identifier and epoch allow a recipient to know which group secrets should be used and from which Epoch secret to start computing
other secrets and keys. The sender identifier is used to identify
the member’s symmetric ratchet from the initial group Application
secret. The application generation field is used to determine how
far into the ratchet to iterate in order to reproduce the required
AEAD keys and nonce for performing decryption.

Application messages SHOULD be padded to provide some resistance
against traffic analysis techniques over encrypted traffic. [CLINIC]
[HCJ16] While MLS might deliver the same payload less frequently
across a lot of ciphertexts than traditional web servers, it might
still provide the attacker enough information to mount an attack. If
Alice asks Bob: "When are we going to the movie ?" the answer
"Wednesday" might be leaked to an adversary by the ciphertext length.
An attacker expecting Alice to answer Bob with a day of the week
might find out the plaintext by correlation between the question and
the length.

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

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

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

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

Since each Application message contains the group identifier, the epoch and a message counter, a client can receive messages out of order. If they are able to retrieve or recompute the correct AEAD decryption key from currently stored cryptographic material clients can decrypt these messages.

For usability, MLS clients might be required to keep the AEAD key and nonce for a certain amount of time to retain the ability to decrypt delayed or out of order messages, possibly still in transit while a decryption is being done.

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

12. 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.

12.1. 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.

12.2. 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.]]

12.3. 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.

12.4. 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.

13. IANA Considerations

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

14. Contributors

- Joel Alwen
  Wickr
  joel.alwen@wickr.com

- Karthikeyan Bhargavan
  INRIA
  karthikeyan.bhargavan@inria.fr

- Cas Cremers
  University of Oxford
15. References

15.1. Normative References

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

[IEEE1363]

[RFC2104]

15.2. Informative References


Appendix A. Tree Math

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

```
   X
   X
 X   X   X   X
X   X   X   X   X   X
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
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

def copath(x, n):
    d = dirpath(x, n)
    if x != sibling(x, n):
        d.append(x)
    return [sibling(y, n) for y in d]

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

def leaves(n):
    return [2*i for i in range(n)]

def resolve(tree, x, n):
    if tree[x] != None:
        return [x]
    if level(x) == 0:
        return []
L = resolve(tree, left(x), n)
R = resolve(tree, right(x, n), n)
return L + R

Authors’ Addresses

Richard Barnes
Cisco
Email: rlb@ipv.sx

Benjamin Beurdouche
Inria
Email: benjamin.beurdouche@inria.fr

Jon Millican
Facebook
Email: jmillican@fb.com

Emad Omara
Google
Email: emadomara@google.com

Katriel Cohn-Gordon
University of Oxford
Email: me@katriel.co.uk

Raphael Robert
Wire
Email: raphael@wire.com
The Messaging Layer Security (MLS) Federation
draft-omara-mls-federation-00

Abstract

This document describes how the Messaging Layer Security (MLS) can be used in a federated environment where different MLS implementations can interoperate by defining the message format for user key retrieval. The document also describes some use cases where federation could be useful.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 2
2. Terminology .................................................... 3
3. Use cases ..................................................... 4
   3.1. Different Delivery Servers ............................... 4
   3.2. Different client applications ........................... 4
4. Functional Requirements ....................................... 4
   4.1. Delivery service ......................................... 4
      4.1.1. Client fanout ....................................... 5
      4.1.2. Server fanout ....................................... 5
   4.2. Authentication service .................................. 6
5. Message format ................................................ 6
6. Security Considerations ....................................... 7
   6.1. Version negotiation ..................................... 7
7. IANA Considerations .......................................... 7
8. References .................................................... 8
   8.1. Normative References ................................... 8
   8.2. Informative References ................................ 8
Authors’ Addresses ............................................... 8

1. Introduction

MLS Architecture draft [MLSARCH] describes the overall MLS system architecture assuming the client and servers (Delivery Service and Authentication Service) are operated by the same entity. This document describes the minimum changes needed to allow different MLS clients operated by the same or different entities to communicate with each and explaining The use cases where federation could be useful.

The focus of this document will be the interaction between the client and the Delivery Service, specifically how the client retrieves the identityKey and InitKeys for another client. There is no changes needed for the Authentication Service.

Discovering which Delivery service the client communicates with is out of the scope of this document.

The below diagram shows an MLS group where all clients are operated under the same deliver service:
one possible environment is to have different client implementations operated by the same delivery service, which will look like the diagram above, another environment is to have different or same clients operated by different delivery services:

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
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.

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

**Identity Key:** A long-lived signing key pair used to authenticate the sender of a message.

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

### 3. Use cases

#### 3.1. Different Delivery Servers

Different applications operated by different entities can use MLS to exchange E2EE messages. For example in email applications, clients of email1.com can encrypt and decrypt E2EE email messages from email2.com.

#### 3.2. Different client applications

Different client applications operated by the same server can use MLS to exchange E2EE handshake and application messages. For example different browsers can implement the MLS protocol, and web developers write web applications that use the MLS implementation in the browser to encrypt and decrypt the messages. This will require a new standard Web API to allow the client applications to set the address of the delivery service in the browser. A more concrete example is using MLS in the browser to exchange SRTP keys for multi-party conference call.

### 4. Functional Requirements

#### 4.1. Delivery service

In MLS environment the messages can either be delivered using client fanout or server fanout, each will have different requirements.
In a federated environment the client may communicate with one or more delivery services. Discovering the delivery service and syncing between different delivery services are out of scope of this document.

4.1.1. Client fanout

In this mode, the client SHOULD support communicating with multiple delivery services. Discovering the delivery service is out of scope of this document.

```
+-----------------+            +---------+
| Deliver Service B | +------> + Client B1 |
+-----------------+            +---------+
                       + +------> + Client B1 +
                           + +-----------------+

+------> + Deliver Service A + +------> + Client A2 +
+-----------------+            +---------+
                  + +-----------------+

+-----------------+            +---------+
| Deliver Service C | +------> + Client C1 |
+-----------------+            +---------+
```

In this mode, the delivery service SHULD be stateless, and it the client's responsibility to maintain the group state. OPEN QUESTION: How ordering could be enforced in this mode?

4.1.2. Server fanout

Multiple delivery services can be avoided, with server side fan out, and all keys requests can be proxied through a single delivery service. The protocol between different delivery services is out of the scope of this document.
OPEN QUESTION: How server assist could be used with multiple servers? how the server state is shared and synced?

4.2. Authentication service

There is no change needed for the authentication service, however the authentication in a federated environment becomes more important. The ideal solution would be using a shared transparency log like [KeyTransparency].

5. Message format

The encrypted message payload is defined in the MLS protocol document [MLSPROTO], in order to get federation between different systems, the identity key and user init key retrieval MUST be defined as well. The identity key can always be included in the user init key response.
enum {
    P256_SHA256_AES128GCM(0x0000),
    X25519_SHA256_AES128GCM(0x0001),
    (0xFFFF)
} CipherSuite;

struct {
    opaque identity<0..2^16-1>;
    CipherSuite supported_suites<0..255>;
} GetUserInitKeyRequest;

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

struct {
    opaque identity<0..2^16-1>;
    UserInitKey user_init_key;
} UserInitKeyBundle;

The delivery service will return one or more user init key bundles, one for each member.

struct {
    UserInitKeyBundle user_init_keys<0..2^32-1>;
} GetUserInitKeyResponse;

OPEN QUESTION: What if different clients have different cipher suites?

6. Security Considerations

6.1. Version negotiation

In a federated environment, version negotiation is more critical, to avoid forcing a downgrade attack by malicious 3rd party delivery services. The negotiation could either be done in the UserInitKeyBundle or in a separate handshake message.

7. IANA Considerations

This document makes no requests of IANA.
8. References

8.1. Normative References


8.2. Informative References


Authors’ Addresses

Emad Omara
Google

Email: emadomara@google.com

Raphael Robert
Wire

Email: raphael@wire.com