The Messaging Layer Security (MLS) Architecture
draft-ietf-mls-architecture-08

Abstract

The Messaging Layer Security (MLS) protocol [I-D.ietf-mls-protocol] specification has the role of defining a Group Key Agreement protocol, including all the cryptographic operations and serialization/deserialization functions necessary for scalable and secure group messaging. The MLS protocol is meant to protect against eavesdropping, tampering, message forgery, and provide further properties such as Forward Secrecy (FS) and Post-Compromise Security (PCS) in the case of past or future device compromises.

This document describes a general secure group messaging infrastructure and its security goals. It provides guidance on building a group messaging system and discusses security and privacy tradeoffs offered by multiple security mechanisms that are part of the MLS protocol (e.g., frequency of public encryption key rotation).

The document also provides guidance for parts of the infrastructure that are not standardized by the MLS Protocol document and left to the application or the infrastructure architects to design.

While the recommendations of this document are not mandatory to follow in order to interoperate at the protocol level, they affect the overall security guarantees that are achieved by a messaging application. This is especially true in case of active adversaries that are able to compromise clients, the delivery service, or the authentication service.

Discussion Venues
This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the MLS Working Group mailing list (mls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/mls/.

Source for this draft and an issue tracker can be found at https://github.com/mlswg/mls-architecture.

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 18 December 2022.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction ................................................. 4
2. General Setting ............................................. 4
   2.1. Group Members and Clients ............................... 7
3. Authentication Service ....................................... 8
4. Delivery Service ............................................ 9
   4.1. Key Storage ........................................... 10

Beurdouche, et al. Expires 18 December 2022
4.2. Key Retrieval .............................................. 10
4.3. Delivery of Messages ........................................... 11
4.4. Membership knowledge ............................................ 12
4.5. Membership and offline members .................................. 13

5. Functional Requirements ........................................ 13
   5.1. Membership Changes ............................................ 13
   5.2. Parallel Groups ................................................ 14
   5.3. Asynchronous Usage .............................................. 15
   5.4. Access Control .................................................. 15
   5.5. Recovery After State Loss ....................................... 16
   5.6. Support for Multiple Devices .................................... 16
   5.7. Extensibility .................................................... 17
   5.8. Application Data Framing and Negotiation ....................... 17
   5.9. Federation ...................................................... 17
   5.10. Compatibility with Future Versions of MLS .................... 18

6. Operational Requirements ......................................... 18

7. Security and Privacy Considerations .......................... 21
   7.1. Assumptions on Transport Security Links ....................... 22
       7.1.1. Metadata Protection for Unencrypted Group Operations .... 22
       7.1.2. DoS protection ............................................... 23
       7.1.3. Message Suppression and Error Correction .................. 24
   7.2. Intended Security Guarantees .................................... 24
       7.2.1. Message Secrecy and Authentication ....................... 25
       7.2.2. Forward and Post-Compromise Security ....................... 25
       7.2.3. Non-Repudiation vs Deniability ............................. 26
   7.3. Endpoint Compromise ............................................ 26
       7.3.1. Compromise of AEAD key material ............................ 27
       7.3.2. Compromise of the Group Secrets of a single group for one or more group epochs ........................ 28
       7.3.3. Compromise by an active adversary with the ability to sign messages ................................ 29
       7.3.4. Compromise of the authentication with access to a signature key ................................ 29
       7.3.5. Security consideration in the context of a full state compromise ................................ 30
   7.4. Service Node Compromise ....................................... 31
       7.4.1. General considerations ....................................... 31
       7.4.2. Delivery Service Compromise ................................ 32
       7.4.3. Authentication Service Compromise ............................ 34
   7.5. Considerations for attacks outside of the threat model .... 36
   7.6. Cryptographic Analysis of the MLS Protocol .................... 37

8. Informative References ......................................... 37
9. IANA Considerations ............................................. 38
10. References ...................................................... 38
    10.1. Normative References .......................................... 38
    10.2. Informative References ........................................ 38

Contributors ....................................................... 39
1. Introduction

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-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 in the face of malicious actions by the operator of the messaging system.

Messaging Layer Security (MLS) specifies an architecture (this document) and a protocol [I-D.ietf-mls-protocol] 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 concrete protocols, such as XMPP [RFC6120]. Implementations of the MLS protocol will interoperate at the cryptographic level, though they may have incompatibilities in terms of how protected messages are delivered, contents of protected messages, and identity/authentication infrastructures. The MLS protocol has been designed to provide the same security guarantees to all users, for all group sizes, even when it reduces to only two users.

2. General Setting

Informally, a group is a set of users who possibly use multiple endpoint devices to interact with the Service Provider (SP). 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, users initially interact with services at their disposal to establish the necessary values and credentials required for encryption and authentication.

The Service Provider presents two abstract functionalities that allow clients to prepare for sending and receiving messages securely:

* An Authentication Service (AS) functionality which is responsible for attesting to bindings between application-meaningful identifiers and the public key material used for authentication in
the MLS protocol. This functionality must also be able to generate credentials that encode these bindings and validate credentials provided by MLS clients.

* A Delivery Service (DS) functionality which can receive and distribute 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 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 MLS clients in order to proceed with the group secret key establishment that is part of the MLS protocol.

For convenience, this document adopts the representation of these services being standalone servers, however the MLS protocol design is made so that this is not necessarily the case. These services may reside on the same server or different servers; they may be distributed between server and client components; and they may even involve some action by users. For example:

* Several secure messaging services today provide a centralized DS, and rely on manual comparison of clients’ public keys as the AS.

* MLS clients connected to a peer-to-peer network could instantiate a decentralized DS by transmitting MLS messages over that network.

* In an MLS group using a PKI for authentication, the AS would comprise the certificate issuance and validation processes, both of which involve logic inside MLS clients as well as various servers.

It is important to note that the Authentication Service functionality can be completely abstract in the case of a Service Provider which allows MLS clients to generate, redistribute and validate their credentials themselves.

Similarly to the AS, the Delivery Service can be completely abstract if users are able to distribute credentials and messages without relying on a central Delivery Service. Note, though, that the MLS protocol requires group operation messages to be processed in-order by all MLS clients.

In some sense, a set of MLS clients which can achieve the AS and DS functionalities without relying on an external party do not need a Service Provider.
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. Other partitions are also possible, such as having a separate directory functionality or service.

According to this architecture design, a typical group messaging scenario might look like this:

1. Alice, Bob and Charlie create accounts with a service provider 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. In addition, they might choose to update their key material which provides post-compromise security Section 7.2.2. As a consequence of that change, the group secrets are updated.

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;
* update their own key material
* 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 (and by extension members in groups) have equal permissions. For instance, any member can add or remove another client in 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 the MLS protocol allows for any existing member of a group to add a new client, applications which use MLS might enforce additional restrictions for which only a subset of members can qualify, and thus will handle enforcing group policies (such as determining if a user is allowed to add new users to the group) at the application level.

2.1. Group Members and Clients

While informally, a group can be considered to be a set of users possibly using multiple endpoint devices to interact with the Service Provider, this definition is too simplistic.

Formally, a client is a set of cryptographic objects composed of public values such as a name (an identity), a public encryption key and a public signature key. Ownership of a client by a user is determined by the fact that the user has knowledge of the associated secret values. When a client is part of a Group, it is called a Member. In some messaging systems, clients belonging to the same user must all share the same signature key pair, but MLS does not assume this.

Users will often use multiple devices, e.g., a phone as well as a laptop. Different devices may be represented as different clients, with independent cryptographic state, or they may share cryptographic state, relying on some application-provided mechanism to sync across devices.
The formal definition of a Group in MLS is the set of clients that have knowledge of the shared group secret established in the group key establishment phase of the protocol and have contributed to it. Until a Member has been added to the group and contributed to the group secret in a manner verifiable by other members of the group, other members cannot assume that the Member is a member of the group.

3. Authentication Service

The Authentication Service (AS) has to provide two functionalities:

1. Issue credentials to clients that attest to bindings between identities and signature key pairs

2. Enable a group member to verify that a credential presented by another member is valid

A member with a valid credential authenticates its MLS messages by signing them with the private key corresponding to the public key in its credential.

The AS is considered an abstract layer by the MLS specification, part of this service could be, for instance, running on the members’ devices, while another part is a separate entity entirely. The following examples illustrate the breadth of this concept:

* A PKI could be used as an AS [RFC5280]. The issuance function would be provided by the certificate authorities in the PKI, and the verification function would correspond to certificate verification by clients.

* Several current messaging applications rely on users verifying each others’ key fingerprints for authentication. In this scenario, the issuance function is simply the generation of a key pair (i.e., credential is just an identifier and public key, with no information to assist in verification). The verification function is the application functionality that enables users to verify keys.

* In a system based on Key Transparency (KT) [KeyTransparency], the issuance function would correspond to the insertion of a key in a KT log under a user’s identity. The verification function would correspond to verifying a key’s inclusion in the log for a claimed identity, together with the KT log’s mechanisms for a user to monitor and control which keys are associated to their identity.
By the nature of its roles in MLS authentication, the AS is invested with a large amount of trust and the compromise of one of its functionalities could allow an adversary to, among other things, impersonate group members. We discuss security considerations regarding the compromise of the different AS functionalities in detail in Section 7.4.3.

The association between members' identities and signature keys is fairly flexible in MLS. As noted above, there is no requirement that all clients belonging to a given user use the same key pair (in fact, such key reuse is forbidden to ensure clients have independent cryptographic state). A member can also rotate the signature key they use within a group. These mechanisms allow clients to use different signature keys in different contexts and at different points in time, providing unlinkability and post-compromise security benefits. Some security trade-offs related to this flexibility are discussed in the security considerations.

In many applications, there are multiple MLS clients that represent a single entity, for example a human user with a mobile and desktop version of an application. Often the same set of clients is represented in exactly the same list of groups. In applications where this is the intended situation, other clients can check that a user is consistently represented by the same set of clients. This would make it more difficult for a malicious AS to issue fake credentials for a particular user because clients would expect the credential to appear in all groups of which the user is a member. If a client credential does not appear in all groups after some relatively short period of time, clients have an indication that the credential might have been created without the user's knowledge. Due to the asynchronous nature of MLS, however, there may be transient inconsistencies in a user's client set, so correlating users' clients across groups is more of a detection mechanism than a prevention mechanism.

4. Delivery Service

The Delivery Service (DS) is expected to play multiple roles in the Service Provider architecture:

* Acting 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.

* Routing MLS messages among clients.
Depending on the level of trust given by the group to the Delivery Service, the functional and privacy guarantees provided by MLS may differ but the authentication and confidentiality guarantees remain the same.

Unlike the Authentication Service which is trusted for authentication and secrecy, the Delivery Service is completely untrusted regarding this property. While privacy of group membership might be a problem in the case of a Delivery Service server fanout, the Delivery Service can be considered as an active, adaptive network attacker from the point of view of the security analysis.

4.1. Key Storage

Upon joining the system, each client stores its initial cryptographic key material with the Delivery Service. This key material, called a KeyPackage, advertises the functional abilities of the client such as supported protocol versions, supported extensions, and the following cryptographic information:

* A credential from the Authentication Service attesting to the binding between the identity and the client’s signature key.

* The client’s asymmetric encryption public key material.

All the parameters in the KeyPackage are signed with the signature private key corresponding to the credential.

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 key material, and for ensuring that the identifier for these keys are unique across all keys stored on the Delivery Service.

4.2. Key Retrieval

When a client wishes to establish a group, it first contacts the Delivery Service to request a KeyPackage for each other client, authenticates the KeyPackages using the signature keys, and then can use those to form the group.
4.3. Delivery of Messages

The main responsibility of the Delivery Service is to ensure delivery of messages. Some MLS messages need only be delivered to some members of a group (e.g., the message initializing a new member's state), while others need to be delivered to all members. The Delivery Service may enable these delivery patterns via unicast channels (sometimes known as "client fanout"), broadcast channels ("server fanout"), or a mix of both.

For the most part, MLS does not require the Delivery Service to deliver messages in any particular order. The one requirement is that because an MLS group has a linear history, the members of the group must agree on the order in which changes are applied. Concretely, the group must agree on which MLS Commit messages to apply. There are a variety of ways to achieve this agreement, but most of them rely on some help from the Delivery Service. For example, if a Delivery Service provides delivery in the same order to all group members, then the members can simply apply Commits in the order in which they appear.

Each Commit is premised on a given state or "epoch" of the group. The Delivery Service must transmit to the group exactly one Commit message per epoch.

Much like the Authentication Service, the Delivery Service can be split between server and client components. Achieving the required uniqueness property will typically require a combination of client and server behaviors. For example, all of the following examples provide a unique Commit per epoch:

* A "filtering server" Delivery Service where a server rejects all but the first Commit for an epoch and clients apply each Commit they receive.

* An "ordering server" Delivery Service where a server forwards all messages but assures that all clients see Commits in the same order, and clients.

* A "passive server" Delivery Service where a server forwards all messages without ordering or reliability guarantees, and clients execute some secondary consensus protocol to choose among the Commits received in a window.

The MLS protocol provides three important pieces of information within an MLSCiphertext message in order to provide ordering:
* The Group Identifier (group ID) to allow for distinguishing the group for which the message has been sent;

* The Epoch number, which represents the number of changes (version) of the group associated with a specific group ID, and allows for lexicographical ordering of messages from different epochs within the same group;

* The Content Type of the message, which allows the Delivery Service to determine the ordering requirement on the message, in particular distinguishing Commit messages from other messages.

The MLS protocol itself can verify these properties. For instance, if the Delivery Service reorders messages from a client or provides different clients with inconsistent orderings, then clients can put messages back in their proper order. The asynchronous nature of MLS means that within an epoch, messages are only ordered per-sender, not globally.

Note that some forms of Delivery Service misbehavior are still possible and difficult to detect. For instance, a Delivery Service 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.

4.4. Membership knowledge

Group membership is itself sensitive information and MLS is designed to limit the amount of persistent metadata. However, large groups often require an infrastructure which provides server fanout. In the case of client fanout, the destination of a message is known by all clients, hence the server usually does not need this information. However, they may learn this information through traffic analysis. Unfortunately, in a server-side fanout model, the Delivery Service can learn 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 Delivery Service.

While this knowledge is not a breach of the protocol’s authentication or confidentiality guarantees, it is a serious issue for privacy. In the case where metadata has to be persisted for functionality, it SHOULD be stored encrypted at rest. Applications should also consider anonymous systems for server fanout such as Loopix [Loopix].
4.5. Membership and offline members

Because Forward Secrecy (FS) and Post-Compromise Security (PCS) rely on the active 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 cannot inherently defend against this problem, especially in the case where the client has not processed messages, but MLS-using systems can enforce some mechanism to try to retain these properties. Typically this will consist of evicting clients which are idle for too long, or mandating a key update from clients that are not otherwise sending messages. The precise details of such mechanisms are a matter of local policy and beyond the scope of this document.

5. Functional Requirements

MLS is designed as a large-scale group messaging protocol and hence aims to provide both 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 groups with tens of thousands of members, typically including many users using multiple devices.

5.1. 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. Such policies can be implemented at the application layer, on top of MLS. Regardless, MLS does not allow for or support addition or removal of group members without informing all other members.

Membership of an MLS group is managed at the level of individual clients. In most cases, a client corresponds to a specific device used by a user. If a user has multiple devices, the user will be represented in a group by multiple clients. If an application wishes to implement operations at the level of users, it is up to the application to track which clients belong to a given user and ensure that they are added / removed consistently.
MLS provides two mechanisms for changing the membership of a group. The primary mechanism is for an authorized member of the group to send a Commit that adds or removes other members. The second mechanism is an "external join": A member of the group publishes certain information about the group, which a new member can use to construct an "external" Commit message that adds the new member to the group. (There is no similarly unilateral way for a member to leave the group; they must be removed by a remaining member.)

With both mechanisms, changes to the membership are initiated from inside the group. When members perform changes directly, this is clearly the case. External joins are authorized indirectly, in the sense that a member publishing a GroupInfo object authorizes anyone to join who has access to the GroupInfo object. External joins do not allow for more granular authorization checks to be done before the new member is added to the group, so if an application wishes to both allow external joins and enforce such checks, then the application will need to do such checks when a member joins, and remove them if checks fail.

Application setup may also determine other criteria for membership validity. For example, per-device signature keys can be signed by an identity key recognized by other participants. If a certificate chain is used to sign off on device signature keys, then revocation by the owner adds an alternative flag to prompt membership removal.

An MLS group’s secrets change on every change of membership, so each client only has access to the secrets used by the group while they are a member. Messages sent before a client joins or after they are removed are protected with keys that are not accessible to the client. Compromise of a member removed from a group does not affect the security of messages sent after their removal. Messages sent during the client’s membership are also secure as long as the client has properly implemented the MLS deletion schedule.

5.2. Parallel Groups

Any user or client may have membership in several groups simultaneously. The set of members of any group may or may not form a subset of the members of another group. MLS guarantees that the FS and PCS goals within a given group are maintained and not weakened by user membership in multiple groups. However, actions in other groups likewise do not strengthen the FS and PCS guarantees within a given group, e.g. key updates within a given group following a device compromise does not provide PCS healing in other groups; each group must be updated separately to achieve internal goals. This also applies to future groups that a member has yet to join, that are likewise unaffected by updates performed in current groups.
Applications may strengthen connectivity among parallel groups by requiring periodic key updates from a user across all groups in which they have membership.

Applications may use the PSK mechanism to link healing properties among parallel groups. For example, suppose a common member M of two groups A and B has performed a key update in group A but not in group B. The key update provides PCS with regard to M in group A. If a PSK is exported from group A and injected into group B, then some of these PCS properties carry over to group B, since the PSK and secrets derived from it are only known to the new, updated version of M, not to the old, possibly compromised version of M.

5.3. Asynchronous Usage

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

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

5.4. Access Control

The MLS protocol allows each member of the messaging group to perform operations equally. This is because all clients within a group (members) have access to the shared cryptographic material. However every service/infrastructure has control over policies applied to its own clients. Applications managing MLS clients can be configured to allow for specific group operations. On the one hand, an application could decide that a group administrator will be the only member to perform add and remove operations. On the other hand, in many settings such as open discussion forums, joining can be allowed for anyone.

The MLS protocol can, in certain modes, exchange unencrypted group operation messages. This flexibility is to allow services to perform access control tasks on behalf of the group.

While the Application messages will always be encrypted, having the handshake messages in plaintext has inconveniences in terms of privacy as someone could collect the signatures on the handshake messages and use them for tracking.

*RECOMMENDATION:* Prefer using encrypted group operation messages to avoid privacy issues related to non-encrypted signatures.
Note that in the default case of encrypted handshake messages, any access control policies will be applied at the client, so the application must ensure that the access control policies are consistent across all clients to make sure that they remain in sync. If two different policies were applied, the clients might not accept or reject a group operation and end-up in different cryptographic states, breaking their ability to communicate.

*RECOMMENDATION:* Avoid using inconsistent access control policies in the case of encrypted group operations.

MLS allows actors outside the group to influence the group in two ways: External signers can submit proposals for changes to the group, and new joiners can use an external join to add themselves to the group. The `external_senders` extension ensures that all members agree on which signers are allowed to send proposals, but any other policies must be assured to be consistent as above.

**RECOMMENDATION:** Have an explicit group policy setting the conditions under which external joins are allowed.

5.5. Recovery After State Loss

Group members whose local MLS state is lost or corrupted can reinitialize their state by re-joining the group as a new member and removing the member representing their earlier state. An application can require that a client performing such a reinitialization prove its prior membership with a PSK.

There are a few practical challenges to this approach. For example, the application will need to ensure that all members have the required PSK, including any new members that have joined the group since the epoch in which the PSK was issued.

Reinitializing in this way does not provide the member with access to group messages from during the state loss window, but enables proof of prior membership in the group. Applications may choose various configurations for providing lost messages to valid group members that are able to prove prior membership.

5.6. Support for Multiple Devices

It is typically expected for users within a group to own various devices. A new device can be added to a group and 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. Such mechanisms, if used, may reduce the FS and PCS guarantees provided by MLS.

5.7. Extensibility

The MLS protocol provides several extension points where additional information can be provided. Extensions to KeyPackages allow clients to disclose additional information about their capabilities. Groups can also have extension data associated with them, and the group agreement properties of MLS will confirm that all members of the group agree on the content of these extensions.

5.8. Application Data Framing and Negotiation

Application messages carried by MLS are opaque to the protocol; they can contain arbitrary data. Each application which uses MLS needs to define the format of its application_data and any mechanism necessary to negotiate the format of that content over the lifetime of an MLS group. In many applications this means managing format migrations for groups with multiple members who may each be offline at unpredictable times.

*RECOMMENDATION:* Use the default content mechanism defined in [I-D.mahy-mls-content-neg], unless the specific application defines another mechanism which more appropriately addresses the same requirements for that application of MLS.

The MLS framing for application messages also provides a field where clients can send information that is authenticated but not encrypted. Such information can be used by servers that handle the message, but group members are assured that it has not been tampered with.

5.9. 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 authentication mechanisms, ciphersuites, and infrastructure functionalities.
5.10. 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 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.

In MLS 1.0, the creator of the group is responsible for selecting the best ciphersuite supported across clients. Each client is able to verify availability of protocol version, ciphersuites and extensions at all times once he has at least received the first group operation message.

Each member of an MLS group advertises the protocol functionality they support. These capability advertisements can be updated over time, e.g., if client software is updated while the client is a member of a group. Thus, in addition to preventing downgrade attacks, the members of a group can also observe when it is safe to upgrade to a new ciphersuite or protocol version.

6. Operational Requirements

MLS is a security layer that needs to be integrated with an application. A fully-functional deployment of MLS will have to make a number of decisions about how MLS is configured and operated. Deployments that wish to interoperate will need to make compatible decisions. This section lists all of the dependencies of an MLS deployment that are external to the protocol specification, but would still need to be aligned within a given MLS deployment, or for two deployments to potentially interoperate.

The protocol has a built-in ability to negotiate protocol versions, ciphersuites, extensions, credential types, and additional proposal types. For two deployments to interoperate, they must have overlapping support in each of these categories. A required_capabilities extension can help maintain interoperability with a wider set of clients by ensuring that certain functionality continues to be supported by a group, even if the clients in the group aren’t currently relying on it.

MLS relies on the following network services. These network services would need to be compatible in order for two different deployments based on them to interoperate.
* An *Authentication Service*, described fully in Section 3, defines the types of credentials which may be used in a deployment and provides methods for:

1. Issuing new credentials,
2. Validating a credential against a reference identifier, and
3. Validating whether or not two credentials represent the same user.

* A *Delivery Service*, described fully in Section 4, provides methods for:

1. Delivering messages sent to a group to all members in the group.
2. Delivering Welcome messages to new members of a group.
3. Downloading KeyPackages for specific clients, and uploading new KeyPackages for a user’s own clients.

* Additional services may or may not be required depending on the application design:

- If assisted joining is desired (meaning that the ratchet tree is not provided in Welcome messages), there must be a method to download the ratchet tree corresponding to a group.

- If assisted joining is desired and the Delivery Service is not able to compute the ratchet tree itself (because some proposals or commits are sent encrypted), there must be a method for group members to publish the updated ratchet tree after each commit.

- If external joiners are allowed, there must be a method to publish a serialized GroupInfo object (with an external_pub extension) that corresponds to a specific group and epoch, and keep that object in sync with the state of the group.

- If an application chooses not to allow assisted or external joining, it may instead provide a method for external users to solicit group members (or a designated service) to add them to a group.

- If the application uses external PSKs, or uses resumption PSKs that all members of a group may not have access to, there must be a method for distributing these PSKs to group members.
- If an application wishes to detect and possibly discipline members that send malformed commits with the intention of corrupting a group’s state, there must be a method for reporting and validating malformed commits.

MLS requires the following parameters to be defined, which must be the same for two implementations to interoperate:

* The maximum total lifetime that is acceptable for a KeyPackage.

* How long to store the resumption secret for past epochs of a group.

* The degree of tolerance that’s allowed for out-of-order message delivery:
  - How long to keep unused nonce and key pairs for a sender
  - A maximum number of unused key pairs to keep.
  - A maximum number of steps that clients will move a secret tree ratchet forward in response to a single message before rejecting it.

MLS provides the following locations where an application may store arbitrary data. The format and intention of any data in these locations must align for two deployments to interoperate:

* Application data, sent as the payload of an encrypted message.

* Additional authenticated data, sent unencrypted in an otherwise encrypted message.

* Group IDs, as decided by group creators and used to uniquely identify a group.

* The application_id extension of a LeafNode.

MLS requires the following policies to be defined, which restrict the set of acceptable behavior in a group. These policies must be consistent between deployments for them to interoperate:

* A policy on when to send proposals and commits in plaintext instead of encrypted.

* A policy for which proposals are valid to have in a commit, including but not limited to:
- When a member is allowed to add or remove other members of the group.

- When, and under what circumstances, a reinitialization proposal is allowed.

- When proposals from external senders are allowed.

- When external joiners are allowed.

* A policy for when two credentials represent the same client. Note that many credentials may be issued authenticating the same identity but for different signature keys, because each credential corresponds to a different device (client) owned by the same application user. However, one device may control many signature keys but should still only be considered a single client.

* A policy on how long to allow a member to stay in a group without updating its leaf keys before removing them.

Finally, there are some additional application-defined behaviors that are partially an individual application’s decision but may overlap with interoperability:

* If there’s any policy on how or when to pad messages.

* If there is any policy for when to send a reinitialization proposal.

* How often clients should update their leaf keys.

* Whether to prefer sending full commits or partial/empty commits.

* Whether there should be a required_capabilities extension in groups.

7. Security and Privacy 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 the face of such attackers.

* The attacker can monitor the entire network.

* The attacker can read unprotected messages.
* The attacker can generate, inject and delete any message in the unprotected transport layer.

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.

Generally, MLS is designed under the assumption that the transport layer is present to protect metadata and privacy in general, while the MLS protocol is providing stronger guarantees such as confidentiality, integrity and authentication guarantees. Stronger properties such as deniability can also be achieved in specific architecture designs.

7.1. Assumptions on Transport Security Links

Any secure channel can be used as a transport layer to protect MLS messages such as QUIC, TLS, WireGuard or TOR. However, the MLS protocol is designed to consider the following threat-model:

* The attacker can read, write, and delete arbitrary messages inside the secure transport channel.

This departs from most threat models where we consider that the secure channel used for transport always provides secrecy. The reason for this consideration is that in the group setting, active malicious insiders or adversarial services are to be considered.

7.1.1. Metadata Protection for Unencrypted Group Operations

The main use of the secure transport layer for MLS is to protect the already limited amount of metadata. Very little information is contained in the unencrypted header of the MLS protocol message format for group operation messages, and application messages are always encrypted in MLS.

MLS avoids needing to send the full list of recipients to the server for dispatching messages because that list is potentially extremely large in MLS. Therefore, the metadata typically consists of a pseudo-random Group Identifier (GID), a numerical value to determine the epoch of the group (the number of changes that have been made to the group), and another numerical value referring to the specific key needed to decrypt the ciphertext content.

The MLS protocol provides an authenticated "Additional Authenticated Data" field for applications to make data available outside the MLSCiphertext.
*RECOMMENDATION:* Use the "Additional Authenticated Data" field of the MLSCiphertext message instead of using other unauthenticated means of sending metadata throughout the infrastructure. If the data is private, the infrastructure should use encrypted Application messages instead.

Even though some of this metadata information does not consist of secret payloads, in correlation with other data a network observer might be able to reconstruct sensitive information. Using a secure channel to transfer this information will prevent a network attacker from accessing this MLS protocol metadata if it cannot compromise the secure channel.

More importantly, there is one specific case where having no secure channel to exchange the MLS messages can have a serious impact on privacy. In the case of unencrypted group operation messages, observing the signatures of the group operation messages may lead an adversary to extract information about the group memberships.

*RECOMMENDATION:* Never use the unencrypted mode for group operations without using a secure channel for the transport layer.

7.1.2. DoS protection

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 Delivery Service to perform a trivial DoS attack from which it is hard to recover. This can be achieved through the secure transport layer.

In the centralized setting, DoS protection can typically be performed by using tickets or cookies which identify users to a service for a certain number of connections. Such a system helps in preventing anonymous clients from sending arbitrary numbers of group operation messages to the Delivery Service or the MLS clients.

*RECOMMENDATION:* Anonymous credentials can be used in order to help DoS attacks prevention, in a privacy preserving manner. Note that the privacy of these mechanisms has to be adjusted in accordance with the privacy expected from the secure transport links. (See more discussion further down.)
7.1.3. Message Suppression and Error Correction

As noted above, MLS is designed to provide some robustness in the face of tampering within the secure transport, i.e., tampering by the Delivery Service. The confidentiality and authenticity properties of MLS prevent the DS reading or writing messages. MLS also provides a few tools for detecting message suppression, with the caveat that message suppression cannot always be distinguished from transport failure.

Each encrypted MLS message carries a "generation" number which is a per-sender incrementing counter. If a group member observes a gap in the generation sequence for a sender, then they know that they have missed a message from that sender. MLS also provides a facility for group members to send authenticated acknowledgements of application messages received within a group.

As discussed in Section 4, the Delivery Service is trusted to select the single Commit message that is applied in each epoch from among the ones sent by group members. Since only one Commit per epoch is meaningful, it’s not useful for the DS to transmit multiple Commits to clients. The risk remains that the DS will use the ability maliciously.

While it is difficult or impossible to prevent a network adversary from suppressing payloads in transit, in certain infrastructures such as banks or governments settings, unidirectional transports can be used and be enforced via electronic or physical devices such as diodes. This can lead to payload corruption which does not affect the security or privacy properties of the MLS protocol but does affect the reliability of the service. In that case specific measures can be taken to ensure the appropriate level of redundancy and quality of service for MLS.

*RECOMMENDATION:* If unidirectional transport is used for the secure transport channel, prefer using a protocol which provides Forward Error Correction.

7.2. Intended Security Guarantees

MLS aims to provide a number of security guarantees, covering authentication, as well as confidentiality guarantees to different degrees in different scenarios.
7.2.1. Message Secrecy and Authentication

MLS enforces the encryption of application messages and thus generally guarantees authentication and confidentiality of application messages sent in a group.

In particular, this means that only other members of a given group can decrypt the payload of a given application message, which includes information about the sender of the message.

Similarly, group members receiving a message from another group member can authenticate that group member as the sender of the message and verify the message’s integrity.

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

Depending on the group settings, handshake messages can be encrypted as well. If that is the case, the same security guarantees apply.

MLS optionally allows the addition of padding to messages, mitigating the amount of information leaked about the length of the plaintext to an observer on the network.

7.2.2. 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’s state is compromised at some time t but the group member subsequently performs an update at some time t’, then all MLS guarantees apply to messages sent by the member after time t’, and by other members after they have processed the update. For example, if an attacker learns all secrets known to Alice at time t, including both Alice’s long-term secret keys and all shared group keys, but Alice performs a key update at time t’, then the attacker is unable to violate any of the MLS security properties after the updates have been processed.
Both of these properties are satisfied even against compromised DSs and ASs.

7.2.3. Non-Repudiation vs Deniability

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 service provider 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 is often called "non-repudiation".

Roughly speaking, "deniability" is the opposite of "non-repudiation", i.e., the property that it is impossible to prove to a third party that a message was sent by a given sender. MLS does not make any claims with regard to deniability. It may be possible to operate MLS in ways that provide certain deniability properties, but defining the specific requirements and resulting notions of deniability requires further analysis.

7.3. Endpoint Compromise

The MLS protocol adopts a threat model which includes multiple forms of endpoint/client compromise. While adversaries are in a very strong position if they have compromised an MLS client, there are still situations where security guarantees can be recovered thanks to the PCS properties achieved by the MLS protocol.

In this section we will explore the consequences and recommendations regarding the following compromise scenarios:

* The attacker has access to a specific symmetric encryption key
* The attacker has access to the group secrets for one group
* The attacker has access to a signature oracle for any group
* The attacker has access to the signature key for one group
* The attacker has access to all secrets of a user for all groups (full state compromise)
Recall that the MLS protocol provides chains of AEAD keys, per sender that are generated from Group Secrets. These keys are used to protect MLS Plaintext messages which can be Group Operation or Application messages. The Group Operation messages offer an additional protection as the secret exchanged within the TreeKEM group key agreement are public-key encrypted to subgroups with HPKE.

7.3.1. Compromise of AEAD key material

In some circumstances, adversaries may have access to specific AEAD keys and nonces which protect an Application or a Group Operation message. While this is a very weak kind of compromise, it can be realistic in cases of implementation vulnerabilities where only part of the memory leaks to the adversary.

When an AEAD key is compromised, the adversary has access to a set of AEAD keys for the same chain and the same epoch, hence can decrypt messages sent using keys of this chain. An adversary cannot send a message to a group which appears to be from any valid client since they cannot forge the signature.

The MLS protocol will ensure that an adversary cannot compute any previous AEAD keys for the same epoch, or any other epochs. Because of its Forward Secrecy guarantees, MLS will also retain secrecy of all other AEAD keys generated for _other_ MLS clients, outside this dedicated chain of AEAD keys and nonces, even within the epoch of the compromise. However the MLS protocol does not provide Post Compromise Secrecy for AEAD encryption within an epoch. This means that if the AEAD key of a chain is compromised, the adversary can compute an arbitrary number of subsequent AEAD keys for that chain.

These guarantees are ensured by the structure of the MLS key schedule which provides Forward Secrecy for these AEAD encryptions, across the messages within the epoch and also across previous epochs. Those chains are completely disjoint and compromising keys across the chains would mean that some Group Secrets have been compromised, which is not the case in this attack scenario (we explore stronger compromise scenarios as part of the following sections).

MLS provides Post-Compromise Secrecy against an active adaptive attacker across epochs for AEAD encryption, which means that as soon as the epoch is changed, if the attacker does not have access to more secret material they won’t be able to access any protected messages from future epochs.

In the case of an Application message, an AEAD key compromise means that the encrypted application message will be leaked as well as the signature over that message. This means that the compromise has both
confidentiality and privacy implications on the future AEAD encryptions of that chain. In the case of a Group Operation message, only the privacy is affected, as the signature is revealed, because the secrets themselves are protected by HPKE encryption.

Note that under that compromise scenario, authentication is not affected in neither of these cases. As every member of the group can compute the AEAD keys for all the chains (they have access to the Group Secrets) in order to send and receive messages, the authentication provided by the AEAD encryption layer of the common framing mechanism is very weak. Successful decryption of an AEAD encrypted message only guarantees that a member of the group sent the message.

7.3.2. Compromise of the Group Secrets of a single group for one or more group epochs

The attack scenario considering an adversary gaining access to a set of Group secrets is significantly stronger. This can typically be the case when a member of the group is compromised. For this scenario, we consider that the signature keys are not compromised. This can be the case for instance if the adversary has access to part of the memory containing the group secrets but not to the signature keys which might be stored in a secure enclave.

In this scenario, the adversary gains the ability to compute any number of AEAD encryption keys for any AEAD chains and can encrypt and decrypt all messages for the compromised epochs.

If the adversary is passive, it is expected from the PCS properties of the MLS protocol that, as soon as an honest Commit message is sent by the compromised party, the next epochs will provide message secrecy.

If the adversary is active, the adversary can follow the protocol and perform updates on behalf of the compromised party with no ability to an honest group to recover message secrecy. However, MLS provides PCS against active adaptive attackers through its Remove group operation. This means that, as long as other members of the group are honest, the protocol will guarantee message secrecy for all messages exchanged in the epochs after the compromised party has been removed.
7.3.3. Compromise by an active adversary with the ability to sign messages

Under such a scenario, where an active adversary has compromised an MLS client, two different settings emerge. In the strongest compromise scenario, the attacker has access to the signing key and can forge authenticated messages. In a weaker, yet realistic scenario, the attacker has compromised a client but the client signature keys are protected with dedicated hardware features which do not allow direct access to the value of the private key and instead provide a signature API.

When considering an active adaptive attacker with access to a signature oracle, the compromise scenario implies a significant impact on both the secrecy and authentication guarantees of the protocol, especially if the attacker also has access to the group secrets. In that case both secrecy and authentication are broken. The attacker can generate any message, for the current and future epochs until an honest update from the compromised client happens.

Note that under this compromise scenario, the attacker can perform all operations which are available to an legitimate client even without access to the actual value of the signature key.

Without access to the group secrets, the adversary will not have the ability to generate messages which look valid to other members of the group and to the infrastructure as they need to have access to group secrets to compute the encryption keys or the membership tag.

7.3.4. Compromise of the authentication with access to a signature key

The difference between having access to the value of the signature key and only having access to a signing oracle is not about the ability of an active adaptive network attacker to perform different operations during the time of the compromise, the attacker can perform every operation available to a legitimate client in both cases.

There is a significant difference, however in terms of recovery after a compromise.
Because of the PCS guarantees provided by the MLS protocol, when a previously compromised client performs an honest Commit which is not under the control of the adversary, both secrecy and authentication of messages can be recovered in the case where the attacker didn’t get access to the key. Because the adversary doesn’t have the key and has lost the ability to sign messages, they cannot authenticate messages on behalf of the compromised party, even if they still have control over some group keys by colluding with other members of the group.

This is in contrast with the case where the signature key is leaked. In that case PCS of the MLS protocol will eventually allow recovery of the authentication of messages for future epochs but only after compromised parties refresh their credentials securely.

Beware that in both oracle and private key access, an active adaptive attacker, can follow the protocol and request to update its own credential. This in turn induces a signature key rotation which could provide the attacker with part or the full value of the private key depending on the architecture of the service provider.

*RECOMMENDATION:* Signature private keys should be compartmentalized from other secrets and preferably protected by an HSM or dedicated hardware features to allow recovery of the authentication for future messages after a compromise.

7.3.5. Security consideration in the context of a full state compromise

In real-world compromise scenarios, it is often the case that adversaries target specific devices to obtain parts of the memory or even the ability to execute arbitrary code in the targeted device.

Also, recall that in this setting, the application will often retain the unencrypted messages. If so, the adversary does not have to break encryption at all to access sent and received messages. Messages may also be sent by using the application to instruct the protocol implementation.

*RECOMMENDATION:* If messages are stored on the device, they should be protected using encryption at rest, and the keys used should be stored securely using dedicated mechanisms on the device.

*RECOMMENDATION:* If the threat model of the system is against an adversary which can access the messages on the device without even needing to attack MLS, the application should delete plaintext messages and ciphertexts immediately after encryption or decryption.
Even though, from the strict point of view of the security formalization, a ciphertext is always public and will forever be, there is no loss in trying to erase ciphertexts as much as possible.

Note that this document makes a clear distinction between the way signature keys and other group shared secrets must be handled. In particular, a large set of group secrets cannot necessarily be assumed to be protected by an HSM or secure enclave features. This is especially true because these keys are extremely frequently used and changed with each message received by a client.

However, the signature private keys are mostly used by clients to send a message. They also are providing the strong authentication guarantees to other clients, hence we consider that their protection by additional security mechanism should be a priority.

Overall there is no way to detect or prevent these compromise, as discussed in the previous sections, performing separation of the application secret states can help recovery after compromise, this is the case for signature keys but similar concern exists for the encryption private key used in the TreeKEM Group Key Agreement.

*RECOMMENDATION:* The secret keys used for public key encryption should be stored similarly to the way the signature keys are stored, as keys can be used to decrypt the group operation messages and contain the secret material used to compute all the group secrets.

Even if secure enclaves are not perfectly secure, or even completely broken, adopting additional protections for these keys can ease recovery of the secrecy and authentication guarantees after a compromise where, for instance, an attacker can sign messages without having access to the key. In certain contexts, the rotation of credentials might only be triggered by the AS through ACLs, hence be outside of the capabilities of the attacker.

7.4. Service Node Compromise

7.4.1. General considerations

7.4.1.1. Privacy of the network connections

There are many scenarios leading to communication between the application on a device and the Delivery Service or the Authentication Service. In particular when:

* The application connects to the Authentication Service to generate or validate a new credential before distributing it.
* The application fetches credentials at the Delivery Service prior to creating a messaging group (one-to-one or more than two clients).

* The application fetches service provider information or messages on the Delivery Service.

* The application sends service provider information or messages to the Delivery Service.

In all these cases, the application will often connect to the device via a secure transport which leaks information about the origin of the request such as the IP address and depending on the protocol the MAC address of the device.

Similar concerns exist in the peer-to-peer use cases of MLS.

*RECOMMENDATION:* In the case where privacy or anonymity is important, using adequate protection such as TOR or a VPN can improve metadata protection.

More generally, using anonymous credentials in an MLS based architecture might not be enough to provide strong privacy or anonymity properties.

7.4.2. 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 4.3, 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.
7.4.2.1. Privacy of delivery and push notifications

An important mechanism that is often ignored from the privacy considerations are the push-tokens. In many modern messaging architectures, applications are using push notification mechanisms typically provided by OS vendors. This is to make sure that when messages are available at the Delivery Service (or by other mechanisms if the DS is not a central server), the recipient application on a device knows about it. Sometimes the push notification can contain the application message itself which saves a round trip with the DS.

To "push" this information to the device, the service provider and the OS infrastructures use unique per-device, per-application identifiers called push-tokens. This means that the push notification provider and the service provider have information on which devices receive information and at which point in time.

Even though they can’t necessarily access the content, which is typically encrypted MLS messages, the service provider and the push notification provider have to be trusted to avoid making correlation on which devices are recipients of the same message.

For secure messaging systems, push notification are often sent real-time as it is not acceptable to create artificial delays for message retrieval.

*RECOMMENDATION:* If real time notifications are not necessary and that specific steps must be taken to improve privacy, one can delay notifications randomly across recipient devices using a mixnet or other techniques.

Note that it is quite easy for legal requests to ask the service provider for the push-token associated to an identifier and perform a second request to the company operating the push-notification system to get information about the device, which is often linked with a real identity via a cloud account, a credit card or other information.

*RECOMMENDATION:* If stronger privacy guarantees are needed vis-a-vis the push notification provider, the client can choose to periodically connect to the Delivery Service without the need of a dedicated push notification infrastructure.
7.4.3. Authentication Service Compromise

The Authentication Service design is left to the infrastructure designers. In most designs, a compromised AS is a serious matter, as the AS can serve incorrect or attacker-provided identities to clients.

* The attacker can link an identity to a credential
* The attacker can generate new credentials
* The attacker can sign new credentials
* The attacker can publish or distribute credentials

Infrastructures that provide cryptographic material or credentials in place of the MLS client (which is under the control of the user) have often the ability to use the associated secrets to perform operations on behalf of the user, which is unacceptable in many situations. Other mechanisms can be used to prevent this issue, such as the service blessing cryptographic material used by an MLS client.

*RECOMMENDATION:* Make clients submit signature public keys to the AS, this is usually better than the AS generating public key pairs because the AS cannot sign on behalf of the client. This is a benefit of a Public Key Infrastructure in the style of the Internet PKI.

An attacker that can generate or sign new credentials may or may not have access to the underlying cryptographic material necessary to perform such operations. In that last case, it results in windows of time for which all emitted credentials might be compromised.

*RECOMMENDATION:* Using HSMs to store the root signature keys to limit the ability of an adversary with no physical access to extract the top-level signature key.

7.4.3.1. Authentication compromise: Ghost users and impersonations

One thing for which the MLS Protocol is designed for is to make sure that all clients know who is in the group at all times. This means that - if all Members of the group and the Authentication Service are honest - no other parties than the members of the current group can read and write messages protected by the protocol for that Group.

Beware though, the link between the cryptographic identity of the Client and the real identity of the User is important. With some Authentication Service designs, a private or centralized authority
can be trusted to generate or validate signature keypairs used in the MLS protocol. This is typically the case in some of the biggest messaging infrastructures.

While this service is often very well protected from external attackers, it might be the case that this service is compromised. In such infrastructure, the AS could generate or validate a signature keypair for an identity which is not the expected one. Because a user can have many MLS clients running the MLS protocol, it possibly has many signature keypairs for multiple devices.

In the case where an adversarial keypair is generated for a specific identity, an infrastructure without any transparency mechanism or out-of-band authentication mechanism could inject a malicious client into a group by impersonating a user. This is especially the case in large groups where the UI might not reflect all the changes back to the users.

*RECOMMENDATION:* Make sure that MLS clients reflect all the membership changes to the users as they happen. If a choice has to be made because the number of notifications is too high, a public log should be maintained in the state of the device so that the user can examine it.

While the ways to handle MLS credentials are not defined by the protocol or the architecture documents, the MLS protocol has been designed with a mechanism that can be used to provide out-of-band authentication to users. The "authentication_secret" generated for each user at each epoch of the group is a one-time, per client, authentication secret which can be exchanged between users to prove their identity to each other. This can be done for instance using a QR code that can be scanned by the other parties.

Another way to improve the security for the users is to provide a transparency mechanism which allows each user to check if credentials used in groups have been published in the transparency log. Another benefit of this mechanism is for revocation. The users of a group could check for revoked keys (in case of compromise detection) using a mechanism such as CRLite or some more advanced privacy preserving technology.

*RECOMMENDATION:* Provide a Key Transparency and Out-of-Band authentication mechanisms to limit the impact of an Authentication Service compromise.
We note, again, that as described prior to that section, the Authentication Service is facultative to design a working infrastructure and can be replaced by many mechanisms such as establishing prior one-to-one deniable channels, gossiping, or using TOFU for credentials used by the MLS Protocol.

Another important consideration is the ease of redistributing new keys on client compromise, which helps recovering security faster in various cases.

7.4.3.2. Privacy of the Group Membership

Often, expectation from users is that the infrastructure will not retain the ability to constantly map the user identity to signature public keys of the MLS protocol. Some infrastructures will keep a mapping between signature public keys of clients and user identities. This can benefit an adversary that has compromised the AS (or required access according to regulation) the ability of monitoring unencrypted traffic and correlating the messages exchanged within the same group.

*RECOMMENDATION:* Always use encrypted group operation messages to reduce issues related to privacy.

In certain cases, the adversary can access specific bindings between public keys and identities. If the signature keys are reused across groups, the adversary can get more information about the targeted user.

*RECOMMENDATION:* Do not use the same signature keypair across groups.

*RECOMMENDATION:* Separate the service binding the identities and the public keys from the service which generates or validates the credentials or cryptographic material of the Clients.

7.5. Considerations for attacks outside of the threat model

Physical attacks on devices storing and executing MLS principals are not considered in depth in the threat model of the MLS protocol. While non-permanent, non-invasive attacks can sometimes be equivalent to software attacks, physical attacks are considered outside of the MLS threat model.

Compromise scenarios typically consist in a software adversary, which can maintain active adaptive compromise and arbitrarily change the behavior of the client or service.
On the other hand, security goals consider that honest clients will always run the protocol according to its specification. This relies on implementations of the protocol to securely implement the specification, which remains non-trivial.

*RECOMMENDATION:* Additional steps should be taken to protect the device and the MLS clients from physical compromise. In such settings, HSMs and secure enclaves can be used to protect signature keys.

7.6. Cryptographic Analysis of the MLS Protocol

Various academic works have analyzed MLS and the different security guarantees it aims to provide. The security of large parts of the protocol has been analyzed by [BBN19] (draft 7), [ACDT21] (draft 11) and [AJM20] (draft 12).

Individual components of various drafts of the MLS protocol have been analyzed in isolation and with differing adversarial models, for example, [BBR18], [ACDT19], [ACCKKMPPWY19], [AJM20] and [ACJM20] analyze the ratcheting tree as the sub-protocol of MLS that facilitates key agreement, while [BCK21] analyzes the key derivation paths in the ratchet tree and key schedule. Finally, [CHK19] analyzes the authentication and cross-group healing guarantees provided by MLS.

8. Informative References

* ACDT19: https://eprint.iacr.org/2019/1189
* ACCKKMPPWY19: https://eprint.iacr.org/2019/1489
* ACJM20: https://eprint.iacr.org/2020/752
* AJM20: https://eprint.iacr.org/2020/1327
* ACDT21: https://eprint.iacr.org/2021/1083
* AHKM21: https://eprint.iacr.org/2021/1456
* CHK19: https://eprint.iacr.org/2021/137
* BCK21: https://eprint.iacr.org/2021/137
* BBR18: https://hal.inria.fr/hal-02425247
* BBN19: https://hal.laas.fr/INRIA/hal-02425229
9. IANA Considerations

This document makes no requests of IANA.

10. References

10.1. Normative References

[I-D.ietf-mls-protocol]
Barnes, R., Beurdouche, B., Robert, R., Millican, J.,
Omara, E., and K. Cohn-Gordon, "The Messaging Layer
Security (MLS) Protocol", Work in Progress, Internet-
Draft, draft-ietf-mls-protocol-14, 3 May 2022,
<https://datatracker.ietf.org/doc/html/draft-ietf-mls-
protocol-14>.

10.2. Informative References

[I-D.mahy-mls-content-neg]
Mahy, R., "Content Negotiation for Message Layer Security
(MLS)", Work in Progress, Internet-Draft, draft-mahy-mls-
content-neg-00, 31 March 2022,
<https://datatracker.ietf.org/doc/html/draft-mahy-mls-
content-neg-00>.

[KeyTransparency]
Google, "Key Transparency", 2017,
<https://KeyTransparency.org>.

[Loopix]
Piotrowska, A. M., Hayes, J., Elahi, T., Meiser, S., and

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC
Text on Security Considerations", BCP 72, RFC 3552,
DOI 10.17487/RFC3552, July 2003,

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
Housley, R., and W. Polk, "Internet X.509 Public Key
Infrastructure Certificate and Certificate Revocation List

Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120,
Contributors

Richard Barnes
Cisco
Email: rlb@ipv.sx

Katriel Cohn-Gordon
Meta Platforms
Email: me@katriel.co.uk

Cas Cremers
CISPA Helmholtz Center for Information Security
Email: cremers@cispa.de

Britta Hale
Naval Postgraduate School
Email: britta.hale@nps.edu

Konrad Kohbrok
Email: konrad.kohbrok@datashrine.de

Brendan McMillion
Email: brendanmcmillion@gmail.com

Thyla van der Merwe
Email: tjjvdmerwe@gmail.com

Jon Millican
Meta Platforms
Email: jmillican@fb.com

Raphael Robert
Email: ietf@raphaelrobert.com

Authors’ Addresses

Benjamin Beurdouche
Inria & Mozilla
Email: ietf@beurdouche.com
Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Emad Omara
Google
Email: emadomara@google.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-16

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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/mlswg/mls-protocol (https://github.com/mlswg/mls-protocol).

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 12 January 2023.

Copyright Notice

Copyright (c) 2022 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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction .................................................. 4
   1.1. Change Log ............................................. 5
2. Terminology .................................................... 13
   2.1. Presentation Language ................................. 14
       2.1.1. Optional Value .................................. 14
       2.1.2. Variable-size Vector Headers .................... 14
3. Operating Context ............................................... 16
4. Protocol Overview .............................................. 17
   4.1. Cryptographic State and Evolution ...................... 18
   4.2. Example Protocol Execution .............................. 19
   4.3. Relationships Between Epochs ......................... 23
5. Ratchet Tree Concepts .......................................... 24
   5.1. Ratchet Tree Terminology ............................... 25
       5.1.1. Ratchet Tree Nodes ............................... 26
       5.1.2. Paths through a Ratchet Tree .................... 27
   5.2. Views of a Ratchet Tree ................................ 28
6. Cryptographic Objects .......................................... 30
   6.1. Ciphersuites ............................................ 30
   6.2. Hash-Based Identifiers ................................ 31
   6.3. Credentials ............................................. 32
       6.3.1. Credential Validation ............................ 33
       6.3.2. Uniquely Identifying Clients ................... 34
7. Message Framing ................................................. 35
   7.1. Content Authentication ................................. 38
   7.2. Encoding and Decoding a Plaintext ...................... 40
7.3. Encoding and Decoding a Ciphertext ................................ 41
  7.3.1. Content Encryption ........................................ 41
  7.3.2. Sender Data Encryption ..................................... 43
8. Ratchet Tree Operations .............................................. 44
  8.1. Parent Node Contents ......................................... 44
  8.2. Leaf Node Contents ........................................... 44
  8.3. Leaf Node Validation .......................................... 47
  8.4. Ratchet Tree Evolution ....................................... 49
  8.5. Synchronizing Views of the Tree .............................. 51
  8.6. Update Paths .................................................. 54
  8.7. Adding and Removing Leaves ................................... 55
  8.8. Tree Hashes .................................................... 56
  8.9. Parent Hashes .................................................. 57
  8.9.1. Using Parent Hashes ....................................... 60
  8.9.2. Verifying Parent Hashes ................................... 60
9. Key Schedule .......................................................... 61
  9.1. Group Context .................................................. 64
  9.2. Transcript Hashes ............................................. 66
  9.3. External Initialization ....................................... 67
  9.4. Pre-Shared Keys ............................................... 67
  9.5. Exporters ....................................................... 70
  9.6. Resumption PSK ............................................... 71
  9.7. Epoch Authenticators ......................................... 71
10. Secret Tree ........................................................... 72
  10.1. Encryption Keys ............................................... 72
  10.2. Deletion Schedule ............................................ 74
11. Key Packages ........................................................ 75
  11.1. KeyPackage Validation ...................................... 77
12. Group Creation ........................................................ 77
  12.1. Required Capabilities ....................................... 79
  12.2. Reinitialization ............................................. 79
  12.3. Subgroup Branching ......................................... 80
13. Group Evolution ......................................................... 81
  13.1. Proposals ...................................................... 81
  13.1.1. Add .......................................................... 82
  13.1.2. Update ..................................................... 83
  13.1.3. Remove ..................................................... 83
  13.1.4. PreSharedKey .............................................. 84
  13.1.5. ReInit ..................................................... 84
  13.1.6. ExternalInit .............................................. 85
  13.1.7. GroupContextExtensions .................................. 85
  13.1.8. External Proposals ..................................... 86
  13.2. Proposal List Validation ................................... 87
  13.3. Applying a Proposal List .................................... 88
  13.4. Commit .......................................................... 89
  13.4.1. Creating a Commit ....................................... 92
  13.4.2. Processing a Commit ..................................... 95
  13.4.3. Adding Members to the Group ............................. 98
1. Introduction

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

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

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

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

In this document, we describe a protocol based on tree structures that enable asynchronous group keying with forward secrecy and post-compromise security. Based on earlier work on "asynchronous ratcheting trees" [art], the protocol presented here uses an asynchronous key-encapsulation mechanism for tree structures. This mechanism allows the members of the group to derive and update shared keys with costs that scale as the log of the group size.

1.1. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-15

* Include ciphersuite in group context (*)
* Add new new_proposal_member SenderType (*)
* Always use a full tree (*)
* Change KeyPackage identifier extension to be LeafNode identifier (*)
* Use new tree for context in path secret encryption (*)
* Use a hash function for hash identifiers (*)
* Add a marker byte to tree hash input structs (*)
* Recommend that group ids are generated randomly (*)
* Update external senders extension to have SignaturePublicKey and Credential (*)
* Replace LeafNodeRef with leaf index (*)
* Remove AppAck proposal (*)
* Make padding arbitrary-size and all-zero (*)
* Require that unmerged_leaves be ordered
* Derive the commit secret from the end of the UpdatePath, not the root
* Specify the precise points in the protocol where credential validation must be done
* Make PSK provisions more uniform, e.g., always generating a fresh random nonce
* Improve parent hash guarantees with stricter checks on tree correctness
* Streamline some structs, e.g., folding GroupContext into GroupInfo
* Provide clearer rules for validating and applying commits
* Clarify tree hash and parent hash, and correct examples
* Clean up struct names and references to outdated structs
* Cite AEAD limits draft
draft-14
* Ensure that a signature public key is always intelligible (*)
* Clean up terminology of derived secrets/keys
* Fix parent hash (*)
* Specify compatibility behavior around new credentials
* Add Path Required to Proposal Type template
* Sub-group branching requires fresh key packages for each member
* Use asavg and typed code blocks
* Require init key and leaf key to be different
* Preconfigured senders extension and removal of signature key indirection
draft-13
* TLS syntax updates (including variable-header-length vectors) (*)
* Stop generating redundant PKE key pairs. (*)
* Move validation of identity change to the AS
* Add message/mls MIME type registration
* Split LeafNode from KeyPackage (*)
* Remove endpoint_id (*)
* Reorganize to make section layout more sane
* Forbid proposals by reference in external commits (*)
* Domain separation for KeyPackage and Proposal references (*)
* Downgrade MUST to SHOULD for commit senders including all valid commits
* Stronger parent hashes for authenticated identities (*)
* Move wire_format to a separate tagged-union structure MLSMessage
* Generalize tree extend/truncate algorithms
* Add algorithms for link-based trees
* Forbid self-Update entirely (*)
* Consolidate resumption PSK cases (*)
* 384 Ciphersuite Addition
* Remove explicit version pin on HPKE (*)
* Remove the requirement for Add in external commit (*)
* Use smaller, fixed-size hash-based identifiers (*)
* Be explicit that Credentials can attest to multiple identities (*)
draft-12
* Use the GroupContext to derive the joiner_secret (*)
* Make PreSharedKeys non optional in GroupSecrets (*)
* Update name for this particular key (*)
* Truncate tree size on removal (*)
* Use HPKE draft-08 (*)
* Clarify requirements around identity in MLS groups (*)
* Signal the intended wire format for MLS messages (*)
* Inject GroupContext as HPKE info instead of AAD (*)
* Clarify extension handling and make extension updatable (*)
* Improve extensibility of Proposals (*)
* Constrain proposal in External Commit (*)
* Remove the notion of a 'leaf index' (*)
* Add group_context_extensions proposal ID (*)
* Add RequiredCapabilities extension (*)
* Use cascaded KDF instead of concatenation to consolidate PSKs (*)
* Use key package hash to index clients in message structs (*)
* Don’t require PublicGroupState for external init (*)
* Make ratchet tree section clearer.
* Handle non-member sender cases in MLSPlaintextTBS
* Clarify encoding of signatures with NIST curves
* Remove OPEN ISSUEs and TODOs
* Normalize the description of the zero vector

draft-11
* Include subtree keys in parent hash (*)
* Pin HPKE to draft-07 (*)
* Move joiner secret to the end of the first key schedule epoch (*)
* Add an AppAck proposal
* Make initializations of transcript hashes consistent

draft-10
* Allow new members to join via an external Commit (*)
* Enable proposals to be sent inline in a Commit (*)
* Re-enable constant-time Add (*)
* Change expiration extension to lifetime extension (*)
* Make the tree in the Welcome optional (*)
* PSK injection, re-init, sub-group branching (*)
* Require the initial init_secret to be a random value (*)
* Remove explicit sender data nonce (*)
* Do not encrypt to joiners in UpdatePath generation (*)
* Move MLSPlaintext signature under the confirmation tag (*)
* Explicitly authenticate group membership with MLSPlaintext (*)
* Clarify X509Credential structure (*)
* Remove unneeded interim transcript hash from GroupInfo (*)
* IANA considerations
* Derive an authentication secret
* Use Extract/Expand from HPKE KDF
* Clarify that application messages MUST be encrypted

draft-09

* Remove blanking of nodes on Add (*)

* Change epoch numbers to uint64 (*)

* Add PSK inputs (*)

* Add key schedule exporter (*)

* Sign the updated direct path on Commit, using "parent hashes" and one signature per leaf (*)

* Use structured types for external senders (*)

* Redesign Welcome to include confirmation and use derived keys (*)

* Remove ignored proposals (*)

* Always include an Update with a Commit (*)

* Add per-message entropy to guard against nonce reuse (*)

* Use the same hash ratchet construct for both application and handshake keys (*)

* Add more ciphersuites

* Use HKDF to derive key pairs (*)

* Mandate expiration of ClientInitKeys (*)

* Add extensions to GroupContext and flesh out the extensibility story (*)

* Rename ClientInitKey to KeyPackage

draft-08

* Change ClientInitKeys so that they only refer to one ciphersuite (*)

* Decompose group operations into Proposals and Commits (*)

* Enable Add and Remove proposals from outside the group (*)
* Replace Init messages with multi-recipient Welcome message (*)

* Add extensions to ClientInitKeys for expiration and downgrade resistance (*)

* Allow multiple Proposals and a single Commit in one MLSPlaintext (*)

**draft-07**

* 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. (*)
* Initial specification proposal for a padding mechanism to improving protection of Application Messages against traffic analysis. (*)
* Inversion of the Group Init Add and Application Secret derivations in the Handshake Key Schedule to be ease chaining in case we switch design. (*)
* Removal of the UserAdd construct and split of GroupAdd into Add and Welcome messages (*)
* Initial proposal for authenticating handshake messages by signing over group state and including group state in the key schedule (*)
* Added an appendix with example code for tree math
* Changed the ECIES mechanism used by TreeKEM so that it uses nonces generated from the shared secret

2. Terminology

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

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

Group: A group represents a logical collection of clients that share a common secret value at any given time. Its state is represented as a linear sequence of epochs in which each epoch depends on its predecessor.

Epoch: A state of a group in which a specific set of authenticated clients hold shared cryptographic state.

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

Key Package: A signed object describing a client’s identity and capabilities, and including a hybrid public-key encryption (HPKE [RFC9180]) public key that can be used to encrypt to that client, and which other clients can use to introduce the client to a new group.

Group Context: An object that summarizes the state of the group. The group context is signed to bind a message to a particular group, and also provided to new members to help them join a group.

Signature Key: A signing key pair used to authenticate the sender of a message.

Handshake Message: An MLSPlaintext or MLSCiphertext message carrying an MLS Proposal or Commit object, as opposed to application data.
Application Message: An MLSCiphertext message carrying application data.

Terminology specific to tree computations is described in Section 5.1.

In general, symmetric values are referred to as "keys" or "secrets" interchangeably. Either term denotes a value that MUST be kept confidential to a Client. When labeling individual values, we typically use "secret" to refer to a value that is used to derive further secret values, and "key" to refer to a value that is used with an algorithm such as HMAC or an AEAD algorithm.

2.1. Presentation Language

We use the TLS presentation language [RFC8446] to describe the structure of protocol messages. In addition to the base syntax, we add two additional features, the ability for fields to be optional and the ability for vectors to have variable-size length headers.

2.1.1. Optional Value

An optional value is encoded with a presence-signaling octet, followed by the value itself if present. When decoding, a presence octet with a value other than 0 or 1 MUST be rejected as malformed.

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

2.1.2. Variable-size Vector Headers

In the TLS presentation language, vectors are encoded as a sequence of encoded elements prefixed with a length. The length field has a fixed size set by specifying the minimum and maximum lengths of the encoded sequence of elements.

In MLS, there are several vectors whose sizes vary over significant ranges. So instead of using a fixed-size length field, we use a variable-size length using a variable-length integer encoding based on the one in Section 16 of [RFC9000]. (They differ only in that the one here requires a minimum-size encoding.) Instead of presenting min and max values, the vector description simply includes a V. For example:
struct {
    uint32 fixed<0..255>;
    opaque variable<V>;
} StructWithVectors;

Such a vector can represent values with length from 0 bytes to 2^30 bytes. The variable-length integer encoding reserves the two most significant bits of the first byte to encode the base 2 logarithm of the integer encoding length in bytes. The integer value is encoded on the remaining bits, in network byte order. The encoded value MUST use the smallest number of bits required to represent the value. When decoding, values using more bits than necessary MUST be treated as malformed.

This means that integers are encoded on 1, 2, or 4 bytes and can encode 6-, 14-, or 30-bit values respectively.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Length</th>
<th>Usable Bits</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
<td>14</td>
<td>64</td>
<td>16383</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>30</td>
<td>16384</td>
<td>1073741823</td>
</tr>
<tr>
<td>11</td>
<td>invalid</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Summary of Integer Encodings

Vectors that start with "11" prefix are invalid and MUST be rejected.

For example, the four byte sequence 0x9d7f3e7d decodes to 494878333; the two byte sequence 0x7bbd decodes to 15293; and the single byte 0x25 decodes to 37.

The following figure adapts the pseudocode provided in [RFC9000] to add a check for minimum-length encoding:
ReadVarint(data):
  // The length of variable-length integers is encoded in the
  // first two bits of the first byte.
  v = data.next_byte()
  prefix = v >> 6
  if prefix == 3:
      raise Exception('invalid variable length integer prefix')

  length = 1 << prefix

  // Once the length is known, remove these bits and read any
  // remaining bytes.
  v = v & 0x3f
  repeat length-1 times:
      v = (v << 8) + data.next_byte()

  // Check that the encoder used the minimum bits required
  if prefix >= 1 && v < (1 << (8*(1 << (prefix-1))-2)):
      raise Exception('minimum encoding was not used')

  return v

The use of variable-size integers for vector lengths allows vectors
to grow very large, up to $2^{30}$ bytes. Implementations should take
care not to allow vectors to overflow available storage. To
facilitate debugging of potential interoperability problems,
implementations SHOULD provide a clear error when such an overflow
condition occurs.

3. Operating Context

 MLS is designed to operate in the context described in
[I-D.ietf-mls-architecture]. In particular, we assume that the
following services are provided:

* A Delivery Service that routes MLS messages among the participants
  in the protocol. The following types of delivery are typically
  required:
    - Pre-publication of KeyPackage objects for clients
    - Broadcast delivery of Proposal and Commit messages to members
      of a group
    - Unicast delivery of Welcome messages to new members of a group

* An Authentication Service that enables group members to
  authenticate the credentials presented by other group members.
4. Protocol Overview

The core functionality of MLS is continuous group authenticated key exchange (AKE). As with other authenticated key exchange protocols (such as TLS), the participants in the protocol agree on a common secret value, and each participant can verify the identity of the other participants. MLS provides group AKE in the sense that there can be more than two participants in the protocol, and continuous group AKE in the sense that the set of participants in the protocol can change over time.

The core organizing principles of MLS are _groups_ and _epochs_. A group represents a logical collection of clients that share a common secret value at any given time. The history of a group is divided into a linear sequence of epochs. In each epoch, a set of authenticated _members_ agree on an _epoch secret_ that is known only to the members of the group in that epoch. The set of members involved in the group can change from one epoch to the next, and MLS ensures that only the members in the current epoch have access to the epoch secret. From the epoch secret, members derive further shared secrets for message encryption, group membership authentication, and so on.

The creator of an MLS group creates the group’s first epoch unilaterally, with no protocol interactions. Thereafter, the members of the group advance their shared cryptographic state from one epoch to another by exchanging MLS messages:

* A _KeyPackage_ object describes a client’s capabilities and provides keys that can be used to add the client to a group.

* A _Proposal_ message proposes a change to be made in the next epoch, such as adding or removing a member.

* A _Commit_ message initiates a new epoch by instructing members of the group to implement a collection of proposals.

* A _Welcome_ message provides a new member to the group with the information to initialize their state for the epoch in which they were added or in which they want to add themselves to the group.

KeyPackage and Welcome messages are used to initiate a group or introduce new members, so they are exchanged between group members and clients not yet in the group.

Proposal and Commit messages are sent from one member of a group to the others. MLS provides a common framing layer for sending messages within a group: An _MLSPlaintext_ message provides sender
authentication for unencrypted Proposal and Commit messages. An _MLSCiphertext_ message provides encryption and authentication for both Proposal/Commit messages as well as any application data.

4.1. Cryptographic State and Evolution

The cryptographic state at the core of MLS is divided into three areas of responsibility:

Figure 1: Overview of MLS group evolution

* A _ratchet tree_ that represents the membership of the group, providing group members a way to authenticate each other and efficiently encrypt messages to subsets of the group. Each epoch has a distinct ratchet tree. It seeds the _key schedule_.

* A _key schedule_ that describes the chain of key derivations used to progress from epoch to epoch (mainly using the _init_secret_ and _epoch_secret_), as well as the derivation of a variety of other secrets (see Table 4), for example:

- An _encryption secret_ that is used to initialize the secret tree for the epoch.
- An _exporter secret_ that allows other protocols to leverage MLS as a generic authenticated group key exchange.

- A _resumption secret_ that members can use to prove their membership in the group, e.g., in the case of branching a subgroup.

* A _secret tree_ derived from the key schedule that represents shared secrets used by the members of the group for encrypting and authenticating messages. Each epoch has a distinct secret tree.

Each member of the group maintains a partial view of these components of the group’s state. MLS messages are used to initialize these views and keep them in sync as the group transitions between epochs.

Each new epoch is initiated with a Commit message. The Commit instructs existing members of the group to update their views of the ratchet tree by applying a set of Proposals, and uses the updated ratchet tree to distribute fresh entropy to the group. This fresh entropy is provided only to members in the new epoch and not to members who have been removed. Commits thus maintain the property that the epoch secret is confidential to the members in the current epoch.

For each Commit that adds one or more members to the group, there is a single corresponding Welcome message. The Welcome message provides all the new members with the information they need to initialize their views of the key schedule and ratchet tree, so that these views align with the views held by other members of the group in this epoch.

4.2. Example Protocol Execution

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

* Adding a member, initiated by a current member;

* Updating the leaf secret of a member;

* Removing a member.
Each of these operations is "proposed" by sending a message of the corresponding type (Add / Update / Remove). The state of the group is not changed, however, until a Commit message is sent to provide the group with fresh entropy. In this section, we show each proposal being committed immediately, but in more advanced deployment cases an application might gather several proposals before committing them all at once. In the illustrations below, we show the Proposal and Commit messages directly, while in reality they would be sent encapsulated in MLSPretext or MLSCiphertext objects.

Before the initialization of a group, clients publish KeyPackages to a directory provided by the Service Provider.

When a client A wants to establish a group with B and C, it first initializes a group state containing only itself and downloads KeyPackages for B and C. For each member, A generates an Add and Commit message adding that member, and broadcasts them to the group. It also generates a Welcome message and sends this directly to the new member (there’s no need to send it to the group). Only after A has received its Commit message back from the server does it update its state to reflect the new member’s addition.

Upon receiving the Welcome message, the new member will be able to read and send new messages to the group. However, messages sent before they were added to the group will not be accessible.
Subsequent additions of group members proceed in the same way. Any member of the group can download a KeyPackage for a new client and broadcast Add and Commit messages that the current group will use to update their state, and a Welcome message that the new client can use to initialize its state and join the group.

To enforce the forward secrecy and post-compromise security of messages, each member periodically updates the keys that represent them to the group. A member does this by sending a Commit (possibly with no proposals), or by sending an Update message that is committed by another member. Once the other members of the group have processed these messages, the group’s secrets will be unknown to an attacker that had compromised the sender’s prior leaf secret.
Update messages SHOULD be sent at regular intervals of time as long as the group is active, and members that don’t update SHOULD eventually be removed from the group. It’s left to the application to determine an appropriate amount of time between Updates.

Members are removed from the group in a similar way. Any member of the group can send a Remove proposal followed by a Commit message. The Commit message provides new entropy to all members of the group except the removed member. This new entropy is added to the epoch secret for the new epoch so that it is not known to the removed member. Note that this does not necessarily imply that any member is actually allowed to evict other members; groups can enforce access control policies on top of these basic mechanism.

Figure 4: Client B proposes to update its key, and client A commits the proposal. As a result, the keys for both B and A updated, so the group has post-compromise security with respect to both of them.
4.3. Relationships Between Epochs

A group has a single linear sequence of epochs. Groups and epochs are generally independent of one another. However, it can sometimes be useful to link epochs cryptographically, either within a group or across groups. MLS derives a resumption pre-shared key (PSK) from each epoch to allow entropy extracted from one epoch to be injected into a future epoch. This link guarantees that members entering the new epoch agree on a key if and only if they were members of the group during the epoch from which the resumption key was extracted.

MLS supports two ways to tie a new group to an existing group. Reinitialization closes one group and creates a new group comprising the same members with different parameters. Branching starts a new group with a subset of the original group’s participants (with no effect on the original group). In both cases, the new group is linked to the old group via a resumption PSK.
Applications may also choose to use resumption PSKs to link epochs in other ways. For example, the following figure shows a case where a resumption PSK from epoch n is injected into epoch n+k. This demonstrates that the members of the group at epoch n+k were also members at epoch n, irrespective of any changes to these members’ keys due to Updates or Commits.

5. Ratchet Tree Concepts

The protocol uses "ratchet trees" for deriving shared secrets among a group of clients. A ratchet tree is an arrangement of secrets and key pairs among the members of a group in a way that allows for secrets to be efficiently updated to reflect changes in the group.

Ratchet trees allow a group to efficiently remove any member by encrypting new entropy to a subset of the group. A ratchet tree assigns shared keys to subgroups of the overall group, so that, for example, encrypting to all but one member of the group requires only
log(N) encryptions, instead of the N-1 encryptions that would be needed to encrypt to each participant individually (where N is the number of members in the group).

This remove operation allows MLS to efficiently achieve post-compromise security. In an Update proposal or a full Commit message, an old (possibly compromised) representation of a member is efficiently removed from the group and replaced with a freshly generated instance.

5.1. Ratchet Tree 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’s 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, or if the node itself is the root of the tree. Nodes are _siblings_ if they share the same parent.

A _subtree_ of a tree is the tree given by any node (the _head_ of the subtree) and its descendants. 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_).

Every tree used in this protocol is a perfect binary tree, that is, a complete balanced binary tree with $2^d$ leaves all at the same depth d. This structure is unique for a given depth d.

There are multiple ways that an implementation might represent a ratchet tree in memory. A convenient property of left-balanced binary trees (including the complete trees used here) is that they can be represented as an array of nodes, with node relationships computed based on the nodes’ indices in the array. A more traditional representation based on linked node objects may also be used. Appendix C and Appendix D provide some details on how to implement the tree operations required for MLS in these representations. MLS places no requirements on implementations’ internal representations of ratchet trees. An implementation MAY use any tree representation and associated algorithms, as long as they produce correct protocol messages.
5.1.1. Ratchet Tree Nodes

Each leaf node in a ratchet tree is given an _index_ (or _leaf index_), starting at 0 from the left to 2^d - 1 at the right (for a tree with 2^d leaves). A tree with 2^d leaves has 2^(d+1) - 1 nodes, including parent nodes.

Each node in a ratchet tree is either _blank_ (containing no value) or it holds an asymmetric key pair with some associated data:

* A public key (for the HPKE scheme in use, see Section 6.1)
* A private key (only within the member’s direct path, see Section 5.2)
* A credential (only for leaf nodes)
* An ordered list of "unmerged" leaves (see Section 5.2)
* A hash of certain information about the node’s parent, as of the last time the node was changed (see Section 8.9).

Every node, regardless of whether the node is blank or populated, has a corresponding _hash_ that summarizes the contents of the subtree below that node. The rules for computing these hashes are described in Section 8.8.

The _resolution_ of a node is an ordered list of non-blank nodes that collectively cover all non-blank descendants of the node. The resolution of a node is effectively a depth-first, left-first enumeration of the nearest non-blank nodes below the node:

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

For example, consider the following subtree, where the _ character represents a blank node and unmerged leaves are indicated in square brackets:
In this tree, we can see all of the above rules in play:

* The resolution of node X is the list [X, B]
* The resolution of leaf 2 or leaf 6 is the empty list []
* The resolution of top node is the list [X, B, Y, H]

5.1.2. Paths through a Ratchet Tree

The _direct path_ of a root is the empty list, and of any other node is the concatenation of that node’s parent along with the parent’s direct path.

The _copath_ of a node is the node’s sibling concatenated with the list of siblings of all the nodes in its direct path, excluding the root.

The _filtered direct path_ of a leaf node L is the node’s direct path, with any node removed whose child on the copath of L has an empty resolution (keeping in mind that any unmerged leaves of the copath child count toward its resolution). The removed nodes do not need their own key pairs because encrypting to the node’s key pair would be equivalent to encrypting to its non-copath child.

For example, consider the following tree (where blank nodes are indicated with _, but also assigned a label for reference):

Figure 9: A tree with blanks and unmerged leaves
Figure 10: A complete tree with seven members, with labels for blank parent nodes

In this tree, the direct paths, copaths, and filtered direct paths for the leaf nodes are as follows:

<table>
<thead>
<tr>
<th>Node</th>
<th>Direct path</th>
<th>Copath</th>
<th>Filtered Direct Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>T, U, W</td>
<td>B, V, Y</td>
<td>T, W</td>
</tr>
<tr>
<td>B</td>
<td>T, U, W</td>
<td>A, V, Y</td>
<td>T, W</td>
</tr>
<tr>
<td>E</td>
<td>X, Y, W</td>
<td>F, Z, U</td>
<td>X, Y, W</td>
</tr>
<tr>
<td>F</td>
<td>X, Y, W</td>
<td>E, Z, U</td>
<td>X, Y, W</td>
</tr>
<tr>
<td>G</td>
<td>Z, Y, W</td>
<td>H, X, U</td>
<td>Y, W</td>
</tr>
</tbody>
</table>

Table 2

5.2. Views of a Ratchet Tree

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

No participant in an MLS group knows the private key associated with every node in the tree. Instead, each member is assigned to a leaf of the tree, which determines the subset of private keys it knows. The credential stored at that leaf is one provided by the member.
In particular, MLS maintains the members’ views of the tree in such a way as to maintain the _tree invariant_: 

The private key for a node in the tree is known to a member of the group only if the node’s subtree contains that member’s leaf.

In other words, if a node is not blank, then it holds a public key. The corresponding private key is known only to members occupying leaves below that node.

The reverse implication is not true: A member may not know the private keys of all the intermediate nodes they’re below. Such a member has an _unmerged_ leaf. Encrypting to an intermediate node requires encrypting to the node’s public key, as well as the public keys of all the unmerged leaves below it. A leaf is unmerged when it is first added, because the process of adding the leaf does not give it access to all of the nodes above it in the tree. Leaves are "merged" as they receive the private keys for nodes, as described in Section 8.4.

For example, consider a four-member group (A, B, C, D) where the node above the right two members is blank. (This is what it would look like if A created a group with B, C, and D.) Then the public state of the tree and the views of the private keys of the tree held by each participant would be as follows, where _ represents a blank node, ? represents an unknown private key, and pk(X) represents the public key corresponding to the private key X:

```
Public Tree
============
pk(ABCD)
   /  \
  pk(AB)  _
     / \     /\
pk(A) pk(B) pk(C) pk(D)
```

```
Private @ A      Private @ B      Private @ C      Private @ D
----------------- ----------------- ----------------- -----------------
AB              AB              ABCD              ABCD
  / \          / \          / \          / \     
 / \ / \      / \ / \         / \ / \         / \ / \ / \
```

Note how the tree invariant applies: Each member knows only their own leaf, and the private key AB is known only to A and B.
6. Cryptographic Objects

6.1. Ciphersuites

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

* HPKE parameters:
  - A Key Encapsulation Mechanism (KEM)
  - A Key Derivation Function (KDF)
  - An AEAD encryption algorithm
* A hash algorithm
* A MAC algorithm
* A signature algorithm

MLS uses HPKE for public-key encryption [RFC9180]. The DeriveKeyPair function associated to the KEM for the ciphersuite maps octet strings to HPKE key pairs. As in HPKE, MLS assumes that an AEAD algorithm produces a single ciphertext output from AEAD encryption (aligning with [RFC5116]), as opposed to a separate ciphertext and tag.

Ciphersuites are represented with the CipherSuite type. HPKE public keys are opaque values in a format defined by the underlying protocol (see the Cryptographic Dependencies section of the HPKE specification for more information).

opaque HPKEPublicKey<V>;

The signature algorithm specified in the ciphersuite is the mandatory algorithm to be used for signatures in MLSContentAuthData and the tree signatures. It MUST be the same as the signature algorithm specified in the credentials in the leaves of the tree (including the leaf node information in KeyPackages used to add new members).

Like HPKE public keys, signature public keys are represented as opaque values in a format defined by the ciphersuite’s signature scheme.

opaque SignaturePublicKey<V>;

opaque SignaturePublicKey<V>;
For ciphersuites using Ed25519 or Ed448 signature schemes, the public key is in the format specified in [RFC8032]. For ciphersuites using ECDSA with the NIST curves (P-256, P-384, or P-521), the public key is the output of the uncompressed Elliptic-Curve-Point-to-Octet-String conversion according to [SECG].

The signatures used in this document are encoded as specified in [RFC8446]. In particular, ECDSA signatures are DER-encoded and EdDSA signatures are defined as the concatenation of r and s as specified in [RFC8032].

To disambiguate different signatures used in MLS, each signed value is prefixed by a label as shown below:

\[
\text{SignWithLabel}(\text{SignatureKey}, \text{Label}, \text{Content}) = \text{Signature.Sign}(\text{SignatureKey}, \text{SignContent})
\]

\[
\text{VerifyWithLabel}(\text{VerificationKey}, \text{Label}, \text{Content}, \text{SignatureValue}) = \text{Signature.Verify}(\text{VerificationKey}, \text{SignContent}, \text{SignatureValue})
\]

Where SignContent is specified as:

```c
struct {
    opaque label<V> = "MLS 1.0 " + Label;
    opaque content<V> = Content;
} SignContent;
```

Here, the functions Signature.Sign and Signature.Verify are defined by the signature algorithm.

The ciphersuites are defined in section Section 17.1.

6.2. Hash-Based Identifiers

Some MLS messages refer to other MLS objects by hash. For example, Welcome messages refer to KeyPackages for the members being welcomed, and Commits refer to Proposals they cover. These identifiers are computed as follows:

opaque HashReference<V>;

HashReference KeyPackageRef;
HashReference ProposalRef;
MakeKeyPackageRef(value) = RefHash("MLS 1.0 KeyPackage Reference", value)
MakeProposalRef(value)   = RefHash("MLS 1.0 Proposal Reference", value)

RefHash(label, value) = Hash(RefHashInput)

Where RefHashInput is defined as:

struct {
  opaque label<V> = label;
  opaque value<V> = value;
} RefHashInput;

For a KeyPackageRef, the value input is the encoded KeyPackage, and the ciphersuite specified in the KeyPackage determines the KDF used. For a ProposalRef, the value input is the MLSAuthenticatedContent carrying the proposal. In the latter two cases, the KDF is determined by the group’s ciphersuite.

6.3. Credentials

Each member of a group presents a credential that provides one or more identities for the member, and associates them with the member’s signing key. The identities and signing key are verified by the Authentication Service in use for a group.

It is up to the application to decide which identifier or identifiers to use at the application level. For example, a certificate in an X509Credential may attest to several domain names or email addresses in its subjectAltName extension. An application may decide to present all of these to a user, or if it knows a "desired" domain name or email address, it can check that the desired identifier is among those attested. Using the terminology from [RFC6125], a Credential provides "presented identifiers", and it is up to the application to supply a "reference identifier" for the authenticated client, if any.
// See IANA registry for registered values
uint16 CredentialType;

struct {
    opaque cert_data<V>;
} Certificate;

struct {
    CredentialType credential_type;
    select (Credential.credential_type) {
        case basic:
            opaque identity<V>;
        case x509:
            Certificate chain<V>;
    }
} Credential;

A BasicCredential is a bare assertion of an identity, without any additional information. The format of the encoded identity is defined by the application.

For an X.509 credential, each entry in the chain represents a single DER-encoded X.509 certificate. The chain is ordered such that the first entry (chain[0]) is the end-entity certificate and each subsequent certificate in the chain MUST be the issuer of the previous certificate. The public key encoded in the subjectPublicKeyInfo of the end-entity certificate MUST be identical to the signature_key in the LeafNode containing this credential.

6.3.1. Credential Validation

The application using MLS is responsible for specifying which identifiers it finds acceptable for each member in a group. In other words, following the model that [RFC6125] describes for TLS, the application maintains a list of "reference identifiers" for the members of a group, and the credentials provide "presented identifiers". A member of a group is authenticated by first validating that the member’s credential legitimately represents some presented identifiers, and then ensuring that the reference identifiers for the member are authenticated by those presented identifiers.
The parts of the system that perform these functions are collectively referred to as the Authentication Service (AS) [I-D.ietf-mls-architecture]. A member's credential is said to be validated with the AS when the AS verifies the credential's presented identifiers, and verifies that those identifiers match the reference identifiers for the member.

Whenever a new credential is introduced in the group, it MUST be validated with the AS. In particular, at the following events in the protocol:

* When a member receives a KeyPackage that it will use in an Add proposal to add a new member to the group.

* When a member receives a GroupInfo object that it will use to join a group, either via a Welcome or via an External Commit.

* When a member receives an Add proposal adding a member to the group.

* When a member receives an Update proposal whose LeafNode has a new credential for the member.

* When a member receives a Commit with an UpdatePath whose LeafNode has a new credential for the committer.

* When an external_senders extension is added to the group, or an existing external_senders extension is updated.

In cases where a member's credential is being replaced, such as Update and Commit cases above, the AS MUST also verify that the set of presented identifiers in the new credential is valid as a successor to the set of presented identifiers in the old credential, according to the application’s policy.

6.3.2. Uniquely Identifying Clients

MLS implementations will presumably provide applications with a way to request protocol operations with regard to other clients (e.g., removing clients). Such functions will need to refer to the other clients using some identifier. MLS clients have a few types of identifiers, with different operational properties.

Internally to the protocol, group members are uniquely identified by their leaf index. However, a leaf index is only valid for referring to members in a given epoch. The same leaf index may represent a different member, or no member at all, in a subsequent epoch.
The Credentials presented by the clients in a group authenticate application-level identifiers for the clients. However, these identifiers may not uniquely identify clients. For example, if a user has multiple devices that are all present in an MLS group, then those devices’ clients could all present the user’s application-layer identifiers.

If needed, applications may add application-specific identifiers to the extensions field of a LeafNode object with the application_id extension.

opaque application_id<V>;

However, applications SHOULD NOT rely on the data in an application_id extension as if it were authenticated by the Authentication Service, and SHOULD gracefully handle cases where the identifier presented is not unique.

7. Message Framing

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

In most of the protocol, messages are handled in the form of MLSAuthenticatedContent objects. These structures contain the content of the message itself as well as information to authenticate the sender (see Section 7.1). The additional protections required to transmit these messages over an untrusted channel (group membership authentication or AEAD encryption) are added by encoding the MLSAuthenticatedContent as an MLSP plaintext or MLSCiphertext message, which can then be sent as an MLSMessage. Likewise, these protections are enforced (via membership verification or AEAD decryption) when decoding an MLSP plaintext or MLSCiphertext into an MLSAuthenticatedContent object.

MLSCiphertext represents a signed and encrypted message, with protections for both the content of the message and related metadata. MLSP plaintext represents a message that is only signed, and not encrypted. Applications MUST use MLSCiphertext to encrypt application messages and SHOULD use MLSCiphertext to encode handshake messages, but MAY transmit handshake messages encoded as MLSP plaintext objects in cases where it is necessary for the Delivery Service to examine such messages.
enum {
    reserved(0),
    mls10(1),
    (255)
} ProtocolVersion;

enum {
    reserved(0),
    application(1),
    proposal(2),
    commit(3),
    (255)
} ContentType;

enum {
    reserved(0),
    member(1),
    external(2),
    new_member_proposal(3),
    new_member_commit(4),
    (255)
} SenderType;

struct {
    SenderType sender_type;
    select (Sender.sender_type) {
        case member:
            uint32 leaf_index;
        case external:
            uint32 sender_index;
        case new_member_commit:
        case new_member_proposal:
            struct{};
    }
} Sender;

enum {
    reserved(0),
    mls_plaintext(1),
    mls_ciphertext(2),
    mls_welcome(3),
    mls_group_info(4),
    mls_key_package(5),
    (255)
} WireFormat;

struct {
    opaque group_id<V>;
}
uint64 epoch;
Sender sender;
opaque authenticated_data<V>;

ContentType content_type;
select (MLSContent.content_type) {
  case application:
    opaque application_data<V>;
  case proposal:
    Proposal proposal;
  case commit:
    Commit commit;
}
} MLSContent;

struct {
  ProtocolVersion version = mls10;
  WireFormat wire_format;
  select (MLSMessages.wire_format) {
    case mls_plaintext:
      MLSPlaintext plaintext;
    case mls_ciphertext:
      MLSCiphertext ciphertext;
    case mls_welcome:
      Welcome welcome;
    case mls_group_info:
      GroupInfo group_info;
    case mls_key_package:
      KeyPackage key_package;
  }
} MLSMessage;

Messages from senders that aren’t in the group are sent as MLSPlaintext. See Section 13.1.8 and Section 13.4.3.2 for more details.

The following structure is used to fully describe the data transmitted in plaintexts or ciphertexts.

struct {
  WireFormat wire_format;
  MLSContent content;
  MLSContentAuthData auth;
} MLSAuthenticatedContent;

The following figure illustrates how the various structures described in this section relate to each other, and the high-level operations used to produce and consume them:
7.1. Content Authentication

MLSContent is authenticated using the MLSContentAuthData structure.
struct {
    ProtocolVersion version = mls10;
    WireFormat wire_format;
    MLSContent content;
    select (MLSContentTBS.content.sender.sender_type) {
        case member:
            GroupContext context;
            case new_member_commit:
                struct();
            break;
        case external:
            case new_member_proposal:
                struct();
            break;
    }
} MLSContentTBS;

opaque MAC<V>;

struct {
    // SignWithLabel(., "MLSContentTBS", MLSContentTBS)
    opaque signature<V>;
    select (MLSContent.content_type) {
        case commit:
            // MAC(confirmation_key,
            // GroupContext.confirmed_transcript_hash)
            MAC confirmation_tag;
            case application:
            case proposal:
                struct();
    }
} MLSContentAuthData;

The signature is computed using SignWithLabel with label "MLSContentTBS" and with a content that covers the message content and the wire format that will be used for this message. If the sender’s sender_type is member, the content also covers the GroupContext for the current epoch so that signatures are specific to a given group and epoch.

The sender MUST use the private key corresponding to the following signature key depending on the sender’s sender_type:

* member: The signature key contained in the LeafNode at the index indicated by leaf_index in the ratchet tree.

* external: The signature key at the index indicated by sender_index in the external_senders group context extension (see Section 13.1.8.1). The content_type of the message MUST be proposal.
* new_member_commit: The signature key in the LeafNode in the Commit’s path (see Section 13.4.3.2). The content_type of the message MUST be commit.

* new_member_proposal: The signature key in the LeafNode in the KeyPackage embedded in an External Add Proposal. The content_type of the message MUST be proposal and the proposal_type of the Proposal MUST be add.

Recipients of an MLSMessage MUST verify the signature with the key depending on the sender_type of the sender as described above.

The confirmation tag value confirms that the members of the group have arrived at the same state of the group. A MLSContentAuthData is said to be valid when both the signature and confirmation_tag fields are valid.

7.2. Encoding and Decoding a Plaintext

Plaintexts are encoded using the MLSPlaintext structure.

```
struct {
    MLSContent content;
    MLSContentAuthData auth;
    select (MLSPlaintext.content.sender.sender_type) {
        case member:
            MAC membership_tag;
        case external:
        case new_member_commit:
        case new_member_proposal:
            struct{];
    }
} MLSPlaintext;
```

The membership_tag field in the MLSPlaintext object authenticates the sender’s membership in the group. For messages sent by members, it MUST be set to the following value:

```
struct {
    MLSContentTBS content_tbs;
    MLSContentAuthData auth;
} MLSContentTBM;
```

`membership_tag = MAC(membership_key, MLSContentTBM)`

When decoding an MLSPlaintext into an MLSAuthenticatedContent, the application MUST check membership_tag and MUST check that the MLSContentAuthData is valid.
7.3. Encoding and Decoding a Ciphertext

Ciphertexts are encoded using the MLSCiphertext structure.

```c
struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<V>;
    opaque encrypted_sender_data<V>;
    opaque ciphertext<V>;
} MLSCiphertext;
```

encrypted_sender_data and ciphertext are encrypted using the AEAD function specified by the ciphersuite in use, using as input the structures MLSSenderData and MLSCiphertextContent.

7.3.1. Content Encryption

The ciphertext content is encoded using the MLSCiphertextContent structure.

```c
struct {
    select (MLSCiphertext.content_type) {
        case application:
            opaque application_data<V>;
        case proposal:
            Proposal proposal;
        case commit:
            Commit commit;
    }
    MLSContentAuthData auth;
    opaque padding[length_of_padding];
} MLSCiphertextContent;
```

The padding field is set by the sender, by first encoding the content (via the select) and the auth field, then appending the chosen number of zero bytes. A receiver identifies the padding field in a plaintext decoded from MLSCiphertext.ciphertext by first decoding the content and the auth field; then the padding field comprises any remaining octets of plaintext. The padding field MUST be filled with all zero bytes. A receiver MUST verify that there are no non-zero bytes in the padding field, and if this check fails, the enclosing MLSCiphertext MUST be rejected as malformed.
In the MLS key schedule, the sender creates two distinct key ratchets for handshake and application messages for each member of the group. When encrypting a message, the sender looks at the ratchets it derived for its own member and chooses an unused generation from either the handshake or application ratchet depending on the content type of the message. This generation of the ratchet is used to derive a provisional nonce and key.

Before use in the encryption operation, the nonce is XORed with a fresh random value to guard against reuse. Because the key schedule generates nonces deterministically, a client MUST keep persistent state as to where in the key schedule it is; if this persistent state is lost or corrupted, a client might reuse a generation that has already been used, causing reuse of a key/nonce pair.

To avoid this situation, the sender of a message MUST generate a fresh random four-byte "reuse guard" value and XOR it with the first four bytes of the nonce from the key schedule before using the nonce for encryption. The sender MUST include the reuse guard in the reuse_guard field of the sender data object, so that the recipient of the message can use it to compute the nonce to be used for decryption.

```
+----------++---------....---+
|       Key Schedule Nonce |
+----------++---------....---+
XOR
+----------++---------....---+
|     Guard   0      |
+----------++---------....---+
===
+----------++---------....---+
| Encrypt/Decrypt Nonce |
+----------++---------....---+
```

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

```c
struct {
    opaque group_id<V);
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<V>;
} MLSCiphertextContentAAD;
```

When decoding an MLSCiphertextContent, the application MUST check that the MLSContentAuthData is valid.
7.3.2. Sender Data Encryption

The "sender data" used to look up the key for content encryption is encrypted with the ciphersuite’s AEAD with a key and nonce derived from both the sender_data_secret and a sample of the encrypted content. Before being encrypted, the sender data is encoded as an object of the following form:

```c
struct {
    uint32 leaf_index;
    uint32 generation;
    opaque reuse_guard[4];
} MLSSenderData;
```

When constructing an MLSSenderData from a Sender object, the sender MUST verify Sender.sender_type is member and use Sender.leaf_index for MLSSenderData.leaf_index.

The reuse_guard field contains a fresh random value used to avoid nonce reuse in the case of state loss or corruption, as described in Section 7.3.1.

The key and nonce provided to the AEAD are computed as the KDF of the first KDF.Nh bytes of the ciphertext generated in the previous section. If the length of the ciphertext is less than KDF.Nh, the whole ciphertext is used. In pseudocode, the key and nonce are derived as:

```c
ciphertext_sample = ciphertext[0..KDF.Nh-1]

sender_data_key = ExpandWithLabel(sender_data_secret, "key",
                                   ciphertext_sample, AEAD.Nk)

sender_data_nonce = ExpandWithLabel(sender_data_secret, "nonce",
                                     ciphertext_sample, AEAD.Nn)
```

The Additional Authenticated Data (AAD) for the SenderData ciphertext is the first three fields of MLSCiphertext:

```c
struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
} MLSSenderDataAAD;
```

When parsing a SenderData struct as part of message decryption, the recipient MUST verify that the leaf index indicated in the leaf_index field identifies a non-blank node.
8. Ratchet Tree Operations

The ratchet tree for an epoch describes the membership of a group in that epoch, providing public-key encryption (HPKE) keys that can be used to encrypt to subsets of the group as well as information to authenticate the members. In order to reflect changes to the membership of the group from one epoch to the next, corresponding changes are made to the ratchet tree. In this section, we describe the content of the tree and the required operations.

8.1. Parent Node Contents

As discussed in Section 5.1.1, the nodes of a ratchet tree contain several types of data describing individual members (for leaf nodes) or subgroups of the group (for parent nodes). Parent nodes are simpler:

```c
struct {
    HPKEPublicKey encryption_key;
    opaque parent_hash<V>;
    uint32 unmerged_leaves<V>;
} ParentNode;
```

The encryption_key field contains an HPKE public key whose private key is held only by the members at the leaves among its descendants. The parent_hash field contains a hash of this node’s parent node, as described in Section 8.9. The unmerged_leaves field lists the leaves under this parent node that are unmerged, according to their indices among all the leaves in the tree. The entries in the unmerged_leaves vector MUST be sorted in increasing order.

8.2. Leaf Node Contents

A leaf node in the tree describes all the details of an individual client’s appearance in the group, signed by that client. It is also used in client KeyPackage objects to store the information that will be needed to add a client to a group.

```c
enum {
    reserved(0),
    key_package(1),
    update(2),
    commit(3),
    (255)
} LeafNodeSource;
```

```c
struct {
    ProtocolVersion versions<V>;
```
CipherSuite ciphersuites<V>;
ExtensionType extensions<V>;
ProposalType proposals<V>;
CredentialType credentials<V>;
} Capabilities;

struct {
    uint64 not_before;
    uint64 not_after;
} Lifetime;

// See IANA registry for registered values
uint16 ExtensionType;

struct {
    ExtensionType extension_type;
    opaque extension_data<V>;
} Extension;

struct {
    HPKEPublicKey encryption_key;
    SignaturePublicKey signature_key;
    Credential credential;
    Capabilities capabilities;
    LeafNodeSource leaf_node_source;
    select (LeafNode.leaf_node_source) {
        case key_package:  
        lifetime:
            Lifetime lifetime;

        case update:
            struct{};

        case commit:
        parent_hash:
            opaque parent_hash<V>;
    }

    Extension extensions<V>;
    // SignWithLabel(., "LeafNodeTBS", LeafNodeTBS)
    opaque signature<V>;
} LeafNode;

struct {
    HPKEPublicKey encryption_key;
    SignaturePublicKey signature_key;
    Credential credential;
    Capabilities capabilities;
}
LeafNodeSource leaf_node_source;
select (LeafNodeTBS.leaf_node_source) {
    case key_package:
        Lifetime lifetime;
        
case update:
        struct{};
        
case commit:
        opaque parent_hash<V>;
}
}
Extension extensions<V>;

select (LeafNodeTBS.leaf_node_source) {
    case key_package:
        struct{};
        
case update:
        opaque group_id<V>;
        
case commit:
        opaque group_id<V>;
}
}
LeafNodeTBS;

The encryption_key field contains an HPKE public key whose private key is held only by the member occupying this leaf (or in the case of a LeafNode in a KeyPackage object, the issuer of the KeyPackage). The signature_key field contains the member’s public signing key. The credential field contains information authenticating both the member’s identity and the provided signing key, as described in Section 6.3.

The capabilities field indicates what protocol versions, ciphersuites, extensions, credential types, and non-default proposal types are supported by a client. Proposal and extension types defined in this document are considered "default" and thus need not be listed, while any credential types the application wishes to use MUST be listed. Extensions that appear in the extensions field of a LeafNode MUST be included in the extensions field of the capabilities field, and the credential type used in the LeafNode MUST be included in the credentials field of the capabilities field.

The leaf_node_source field indicates how this LeafNode came to be added to the tree. This signal tells other members of the group whether the leaf node is required to have a lifetime or parent_hash, and whether the group_id is added as context to the signature.
Whether these fields can be computed by the client represented by the LeafNode depends on when the LeafNode was created. For example, a KeyPackage is created before the client knows which group it will be used with, so its signature can’t bind to a group_id.

In the case where the leaf was added to the tree based on a pre-published KeyPackage, the lifetime field represents the times between which clients will consider a LeafNode valid. These times are represented as absolute times, measured in seconds since the Unix epoch (1970-01-01T00:00:00Z). Applications MUST define a maximum total lifetime that is acceptable for a LeafNode, and reject any LeafNode where the total lifetime is longer than this duration.

In the case where the leaf node was inserted into the tree via a Commit message, the parent_hash field contains the parent hash for this leaf node (see Section 8.9).

The LeafNodeTBS structure covers the fields above the signature in the LeafNode. In addition, when the leaf node was created in the context of a group (the update and commit cases), the group ID of the group is added as context to the signature.

LeafNode objects stored in the group’s ratchet tree are updated according to the evolution of the tree. Each modification of LeafNode content MUST be reflected by a change in its signature. This allows other members to verify the validity of the LeafNode at any time, particularly in the case of a newcomer joining the group.

8.3. Leaf Node Validation

The validity of a LeafNode needs to be verified at a few stages:

* When a LeafNode is downloaded in a KeyPackage, before it is used to add the client to the group

* When a LeafNode is received by a group member in an Add, Update, or Commit message

* When a client validates a ratchet tree, e.g., when joining a group or after processing a commit

The client verifies the validity of a LeafNode using the following steps:

* Verify that the credential in the LeafNode is valid as described in Section 6.3.1.
* Verify that the signature on the LeafNode is valid using signature_key.

* Verify that the LeafNode is compatible with the group’s parameters. If the GroupContext has a required_capabilities extension, then the required extensions, proposals, and credential types MUST be listed in the LeafNode’s capabilities field.

* Verify that the credential type is supported by all members of the group, as specified by the capabilities field of each member’s LeafNode, and that the capabilities field of this LeafNode indicates support for all the credential types currently in use by other members.

* Verify the lifetime field:
  - If the LeafNode appears in a message being sent by the client, e.g., a proposal or a commit, then the client MUST verify that the current time is within the range of the lifetime field.
  - If instead the LeafNode appears in a message being received by the client, e.g., a proposal, a commit, or a ratchet tree of the group the client is joining, it is RECOMMENDED that the client verifies that the current time is within the range of the lifetime field.

* Verify that the extensions in the LeafNode are supported by checking that the ID for each extension in the extensions field is listed in the capabilities.extensions field of the LeafNode.

* Verify the leaf_node_source field:
  - If the LeafNode appears in a KeyPackage, verify that leaf_node_source is set to key_package.
  - If the LeafNode appears in an Update proposal, verify that leaf_node_source is set to update.
  - If the LeafNode appears in the leaf_node value of the UpdatePath in a Commit, verify that leaf_node_source is set to commit.

* Verify that the following fields are unique among the members of the group:
  - signature_key
  - encryption_key
8.4. Ratchet Tree Evolution

Whenever a member initiates an epoch change (i.e., commits; see Section 13.4), they may need to refresh the key pairs of their leaf and of the nodes on their leaf’s direct path in order to maintain forward secrecy and post-compromise security.

The member initiating the epoch change generates the fresh key pairs using the following procedure. The procedure is designed in a way that allows group members to efficiently communicate the fresh secret keys to other group members, as described in Section 8.6.

A member updates the nodes along its direct path as follows:

* Blank all the nodes on the direct path from the leaf to the root.
* Generate a fresh HPKE key pair for the leaf.
* Generate a sequence of path secrets, one for each node on the leaf’s filtered direct path, as follows. In this setting, path_secret[0] refers to the first parent node in the filtered direct path, path_secret[1] to the second parent node, and so on.

path_secret[0] is sampled at random
path_secret[n] = DeriveSecret(path_secret[n-1], "path")

* Compute the sequence of HPKE key pairs (node_priv,node_pub), one for each node on the leaf’s direct path, as follows.

node_secret[n] = DeriveSecret(path_secret[n], "node")
note_priv[n], node_pub[n] = KEM.DeriveKeyPair(node_secret[n])

The node secret is derived as a temporary intermediate secret so that each secret is only used with one algorithm: The path secret is used as an input to DeriveSecret and the node secret is used as an input to DeriveKeyPair.

For example, suppose there is a group with four members, with C an unmerged leaf at Z:
If member B subsequently generates an UpdatePath based on a secret "leaf_secret", then it would generate the following sequence of path secrets:


path_secret[0] ---> node_secret[0] -------> node_priv[0], node_pub[0]

leaf_secret -------> leaf_node_secret -------> leaf_priv, leaf_pub

After applying the UpdatePath, the tree will have the following structure, where lp and np[i] represent the leaf_priv and node_priv values generated as described above:

node_priv[1] -------> Y'

node_priv[0] ----> X'      Z[C]

leaf_priv -----------+--->
       0   1   2   3
8.5. Synchronizing Views of the Tree

After generating fresh key material and applying it to ratchet forward their local tree state as described in the Section 8.4, the generator broadcasts this update to other members of the group in a Commit message, who apply it to keep their local views of the tree in sync with the sender’s. More specifically, when a member commits a change to the tree (e.g., to add or remove a member), it transmits an UpdatePath containing a set of public keys and encrypted path secrets for intermediate nodes in the filtered direct path of its leaf. The other members of the group use these values to update their view of the tree, aligning their copy of the tree to the sender’s.

An UpdatePath contains the following information for each node in the filtered direct path of the sender’s leaf, including the root:

* The public key for the node
* One or more encrypted copies of the path secret corresponding to the node

The path secret value for a given node is encrypted to the subtree rooted at the parent’s non-updated child, i.e., the child on the copath of the sender’s leaf node. There is one encryption of the path secret to each public key in the resolution of the non-updated child.

A member of the group updates their direct path by computing new values for their leaf node and the nodes along their filtered direct path:

1. Blank all nodes along the direct path of the sender’s leaf.

2. Compute updated path secrets and public keys for the nodes on the sender’s filtered direct path.
   * Generate a sequence of path secrets of the same length as the filtered direct path, as defined in Section 8.4
   * For each node in the filtered direct path, replace the node’s public key with the node_pub[n] value derived from the corresponding path secret path_secret[n].

3. Compute the new parent hashes for the nodes along the filtered direct path and the sender’s leaf node.

4. Update the leaf node for the sender.
* Set the leaf_node_source to commit.

* Set the encryption_key to the public key of a freshly sampled key pair

* Set the parent hash to the parent hash for the leaf.

* Re-sign the leaf node with its new contents

Since the new leaf node effectively updates an existing leaf node in the group, it MUST adhere to the same restrictions as LeafNodes used in Update proposals (aside from leaf_node_source). The application MAY specify other changes to the leaf node, e.g., providing a new signature key, updated capabilities, or different extensions.

The member then _encrypts path secrets to the group_. For each node in the member’s filtered direct path, the member takes the following steps:

1. Compute the resolution of the node’s child that is on the copath of the sender (the child that is not in the direct path of the sender). Any new member (from an Add proposal) added in the same Commit MUST be excluded from this resolution.

2. For each node in the resolution, encrypt the path secret for the direct path node using the public key of the resolution node, as defined in Section 8.6

The recipient of an UpdatePath performs the corresponding steps.
First, the recipient _merges UpdatePath into the tree_:

1. Blank all nodes on the direct path of the sender’s leaf.

2. For all nodes on the filtered direct path of the sender’s leaf,

   * Set the public key to the public key in the UpdatePath.

   * Set the list of unmerged leaves to the empty list.

3. Compute parent hashes for the nodes in the sender’s filtered direct path, and verify that the parent_hash field of the leaf node matches the parent hash for the first node in its filtered direct path.

   * Note that these hashes are computed from root to leaf, so that each hash incorporates all the non-blank nodes above it. The root node always has a zero-length hash for its parent hash.
Second, the recipient _decrypts the path secrets_: 

1. Identify a node in the filtered direct path for which the recipient is in the subtree of the non-updated child.

2. Identify a node in the resolution of the copath node for which the recipient has a private key.

3. Decrypt the path secret for the parent of the copath node using the private key from the resolution node.

4. Derive path secrets for ancestors of that node in the sender’s filtered direct path using the algorithm described above.

5. Derive the node secrets and node key pairs from the path secrets.

6. Verify that the derived public keys are the same as the corresponding public keys sent in the UpdatePath.

7. Store the derived private keys in the corresponding ratchet tree nodes.

For example, in order to communicate the example update described in Section 8.4, the member at node B would transmit the following values:

<table>
<thead>
<tr>
<th>Public Key</th>
<th>Ciphertext(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_pub[1]</td>
<td>E(pk(Z), path_secret[1]), E(pk(C), path_secret[1])</td>
</tr>
<tr>
<td>node_pub[0]</td>
<td>E(pk(A), path_secret[0])</td>
</tr>
</tbody>
</table>

Table 3

In this table, the value node_pub[i] represents the public key derived from node_secret[i], pk(X) represents the current public key of node X, and E(K, S) represents the public-key encryption of the path secret S to the public key K (using HPKE).

A recipient at node A would decrypt E(pk(A), path_secret[0]) to obtain path_secret[0], then use it to derive path_secret[1] and the resulting node secrets and key pairs. Thus A would have the private keys to nodes X’ and Y’, in accordance with the tree invariant.
Similarly, a recipient at node D would decrypt $E(pk(Z), path_secret[1])$ to obtain $path_secret[1]$, then use it to derive the node secret and key pair for the node $Y'$. As required to maintain the tree invariant, node D does not receive the private key for the node $X'$, since $X'$ is not an ancestor of D.

After processing the update, each recipient MUST delete outdated key material, specifically:

* The path secrets and node secrets used to derive each updated node key pair.

* Each outdated node key pair that was replaced by the update.

8.6. Update Paths

As described in Section 13.4, each Commit message may optionally contain an UpdatePath, with a new LeafNode and set of parent nodes for the sender’s filtered direct path. For each parent node, the UpdatePath contains a new public key and encrypted path secret. The parent nodes are kept in the same order as the filtered direct path.

```c
struct {
    opaque kem_output<V>;
    opaque ciphertext<V>;
} HPKECiphertext;

struct {
    HPKEPublicKey encryption_key;
    HPKECiphertext encrypted_path_secret<V>;
} UpdatePathNode;

struct {
    LeafNode leaf_node;
    UpdatePathNode nodes<V>;
} UpdatePath;
```

For each UpdatePathNode, the resolution of the corresponding copath node MUST exclude all new leaf nodes added as part of the current Commit. The length of the encrypted_path_secret vector MUST be equal to the length of the resolution of the copath node (excluding new leaf nodes), with each ciphertext being the encryption to the respective resolution node.

The HPKECiphertext values are computed as

```c
kem_output, context = SetupBaseS(node_public_key, group_context)
ciphertext = context.Seal("", path_secret)
```
where node_public_key is the public key of the node that the path secret is being encrypted for, group_context is the provisional GroupContext object for the group, and the functions SetupBaseS and Seal are defined according to [RFC9180].

Decryption is performed in the corresponding way, using the private key of the resolution node.

context = SetupBaseR(kem_output, node_private_key, group_context)
path_secret = context.Open("", ciphertext)

8.7. Adding and Removing Leaves

In addition to the path-based updates to the tree described above, it is also necessary to add and remove leaves of the tree in order to reflect changes to the membership of the group (see Section 13.1.1 and Section 13.1.3). Since the tree is always full, adding or removing leaves corresponds to increasing or decreasing the depth of the tree, resulting in the number of leaves being doubled or halved. These operations are also known as _extending_ and _truncating_ the tree.

Leaves are always added and removed at the right edge of the tree. When the size of the tree needs to be increased, a new blank root node is added, whose left subtree is the existing tree and right subtree is a new all-blank subtree. This operation is typically done when adding a member to the group.

```
        <-- new blank root
       __|__
      /     \
     X    ===>   X  <-- new blank subtree ===>   X
   / \        / \                           / \     / \
  A   B       A   B   _   _                     A   B   C   _
      ^                  ^                      ^
           |                        |                    |
         +-- new member             +-- new member
```

Figure 12: Extending the tree to make room for a third member

When the right subtree of the tree no longer has any non-blank nodes, it can be safely removed. The root of the tree and the right subtree are discarded (whether or not the root node is blank). The left child of the root becomes the new root node, and the left subtree becomes the new tree. This operation is typically done after removing a member from the group.
Concrete algorithms for these operations on array-based and link-based trees are provided in Appendix C and Appendix D. The concrete algorithms are non-normative. An implementation MAY use any algorithm that produces the correct tree in its internal representation.

8.8. Tree Hashes

MLS hashes the contents of the tree in two ways to authenticate different properties of the tree. This section defines _tree hashes_, and _parent hashes_ are defined in Section 8.9.

Each node in a ratchet tree has a tree hash that summarizes the subtree below that node. The tree hash of the root is used in the GroupContext to confirm that the group agrees on the whole tree. Tree hashes are computed recursively from the leaves up to the root.

\[
P \rightarrow \text{th}(P) \\
\quad \quad \quad \quad ^{\text{th}(L)} \quad \text{th}(R)
\]

The tree hash of an individual node is the hash of the node’s TreeHashInput object, which may contain either a LeafNodeHashInput or a ParentNodeHashInput depending on the type of node. LeafNodeHashInput objects contain the leaf_index and the LeafNode (if any). ParentNodeHashInput objects contain the ParentNode (if any) and the tree hash of the node’s left and right children. For both parent and leaf nodes, the optional node value MUST be absent if the node is blank and present if the node contains a value.
enum {
    reserved(0),
    leaf(1),
    parent(2),
    (255)
} NodeType;

struct {
    NodeType node_type;
    select (TreeHashInput.node_type) {
        case leaf:   LeafNodeHashInput leaf_node;
        case parent: ParentNodeHashInput parent_node;
    }
} TreeHashInput;

struct {
    uint32 leaf_index;
    optional<LeafNode> leaf_node;
} LeafNodeHashInput;

struct {
    optional<ParentNode> parent_node;
    opaque left_hash<V>;
    opaque right_hash<V>;
} ParentNodeHashInput;

The tree hash of an entire tree corresponds to the tree hash of the root node, which is computed recursively by starting at the leaf nodes and building up.

8.9. Parent Hashes

While tree hashes summarize the state of a tree at point in time, parent hashes capture information about how keys in the tree were populated.

When a client sends a commit to change a group, it can include an UpdatePath to assign new keys to the nodes along its filtered direct path. When a client computes an UpdatePath (as defined in Section 8.5), it computes and signs a parent hash that summarizes the state of the tree after the UpdatePath has been applied. These summaries are constructed in a chain from the root to the member’s leaf so that the part of the chain closer to the root can be overwritten as nodes set in one UpdatePath are reset by a later UpdatePath.
As a result, the signature over the parent hash in each member’s leaf effectively signs the subtree of the tree that hasn’t been changed since that leaf was last changed in an UpdatePath. A new member joining the group uses these parent hashes to verify that the parent nodes in the tree were set by members of the group, not chosen by an external attacker. For an example of how this works, see Appendix B.

Consider a ratchet tree with a non-blank parent node P and children D and S (for "parent", "direct path", and "sibling"), with D and P in the direct path of a leaf node L (for "leaf"):

![Diagram of a ratchet tree](image)

Figure 14: Inputs to a parent hash.

The parent hash of P changes whenever an UpdatePath object is applied to the ratchet tree along a path from a leaf L traversing node D (and hence also P). The new "Parent hash of P (with copath child S)" is obtained by hashing P’s ParentHashInput struct.

```c
struct {
    HPKEPublicKey encryption_key;
    opaque parent_hash<V>;
    opaque original_sibling_tree_hash<V>;
} ParentHashInput;
```

The field encryption_key contains the HPKE public key of P. If P is the root, then the parent_hash field is set to a zero-length octet string. Otherwise, parent_hash is the Parent Hash of the next node.
after P on the filtered direct path of the leaf L. This way, P’s Parent Hash fixes the new HPKE public key of each non-blank node on the path from P to the root. Note that the path from P to the root may contain some blank nodes that are not fixed by P’s Parent Hash. However, for each node that has an HPKE key, this key is fixed by P’s Parent Hash.

Finally, original_sibling_tree_hash is the tree hash of S in the ratchet tree modified as follows: For each leaf L in P.unmerged_leaves, blank L and remove it from the unmerged_leaves sets of all parent nodes.

Observe that original_sibling_tree_hash does not change between updates of P. This property is crucial for the correctness of the protocol.

Note that original_sibling_tree_hash is the tree hash of S, not the parent hash. The parent_hash field in ParentHashInput captures information about the nodes above P. The original_sibling_tree_hash captures information about the subtree under S that is not being updated (and thus the subtree to which a path secret for P would be encrypted according to Section 8.5).

For example, in the following tree:

```
        W [F]
          /   /
        U   Y [F]
          /     /
        T     /
          /   /
         A   B   C   D   E   F   G   _
```

Figure 15: A tree illustrating parent hash computations.

With P = W and S = Y, original_sibling_tree_hash is the tree hash of the following tree:

```
        Y
          /   /
        /     /
       E     _
```

Barnes, et al. Expires 12 January 2023
Because W.unmerged_leaves includes F, F is blanked and removed from Y.unmerged_leaves.

Note that no recomputation is needed if the tree hash of S is unchanged since the last time P was updated. This is the case for computing or processing a Commit whose UpdatePath traverses P, since the Commit itself resets P. (In other words, it is only necessary to recompute the original sibling tree hash when validating a group’s tree on joining.) More generally, if none of the entries in P.unmerged_leaves is in the subtree under S (and thus no leaves were blanked), then the original tree hash at S is the tree hash of S in the current tree.

If it is necessary to recompute the original tree hash of a node, the efficiency of recomputation can be improved by caching intermediate tree hashes, to avoid recomputing over the subtree when the subtree is included in multiple parent hashes. A subtree hash can be reused as long as the intersection of the parent’s unmerged leaves with the subtree is the same as in the earlier computation.

8.9.1. Using Parent Hashes

In ParentNode objects and LeafNode objects with leaf_node_source set to commit, the value of the parent_hash field is the parent hash of the next non-blank parent node above the node in question (the next node in the filtered direct path). Using the node labels in Figure 14, the parent_hash field of D is equal to the parent hash of P with copath child S. This is the case even when the node D is a leaf node.

The parent_hash field of a LeafNode is signed by the member. The signature of such a LeafNode thus also attests to which keys the group member introduced into the ratchet tree and to whom the corresponding secret keys were sent. This prevents malicious insiders from constructing artificial ratchet trees with a node D whose HPKE secret key is known to the insider yet where the insider isn’t assigned a leaf in the subtree rooted at D. Indeed, such a ratchet tree would violate the tree invariant.

8.9.2. Verifying Parent Hashes

Parent hashes are verified at two points in the protocol: When joining a group and when processing a Commit.

The parent hash in a node D is valid with respect to a parent node P if the following criteria hold. Here C and S are the children of P (for "child" and "sibling"), with C being the child that is on the direct path of D (possibly D itself) and S the other child:
* D is a descendant of P in the tree.

* The parent_hash field of D is equal to the parent hash of P with copath child S.

* D is in the resolution of C, and the intersection of P’s unmerged_leaves with the subtree under C is equal to the resolution of C with D removed.

These checks verify that D and P were updated at the same time (in the same UpdatePath), and that they were neighbors in the UpdatePath because the nodes in between them would have omitted from the filtered direct path.

A parent node P is "parent-hash valid" if it can be chained back to a leaf node in this way. That is, if there is leaf node L and a sequence of parent nodes P_1, ..., P_N such that P_N = P and each step in the chain is authenticated by a parent hash: L’s parent hash is valid with respect to P_1, P_1’s parent hash is valid with respect to P_2, and so on.

When joining a group, the new member MUST authenticate that each non-blank parent node P is parent-hash valid. This can be done "bottom up" by building chains up from leaves and verifying that all non-blank parent nodes are covered by exactly one such chain, or "top down" by verifying that there is exactly one descendant of each non-blank parent node for which the parent node is parent-hash valid.

When processing a Commit message that includes an UpdatePath, clients MUST recompute the expected value of parent_hash for the committer’s new leaf and verify that it matches the parent_hash value in the supplied leaf_node. After being merged into the tree, the nodes in the UpdatePath form a parent-hash chain from the committer’s leaf to the root.

9. Key Schedule

Group keys are derived using the Extract and Expand functions from the KDF for the group’s ciphersuite, as well as the functions defined below:

```
ExpandWithLabel(Secret, Label, Context, Length) =
    KDF.Expand(Secret, KDFLabel, Length)

DeriveSecret(Secret, Label) =
    ExpandWithLabel(Secret, Label, "", KDF.Nh)
```

Where KDFLabel is specified as:

```
struct {
    uint16 length = Length;
    opaque label<V> = "MLS 1.0 " + Label;
    opaque context<V> = Context;
} KDFLabel;

The value KDF.Nh is the size of an output from KDF.Extract, in bytes. In the below diagram:

* KDF.Extract takes its salt argument from the top and its Input Key Material (IKM) argument from the left
* DeriveSecret takes its Secret argument from the incoming arrow
* 0 represents an all-zero byte string of length KDF.Nh.

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

* The init secret from the previous epoch
* The commit 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:
A number of values are derived from the epoch secret for different purposes:
<table>
<thead>
<tr>
<th>Label</th>
<th>Secret</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;sender data&quot;</td>
<td>sender_data_secret</td>
<td>Deriving keys to encrypt sender data</td>
</tr>
<tr>
<td>&quot;encryption&quot;</td>
<td>encryption_secret</td>
<td>Deriving message encryption keys (via the secret tree)</td>
</tr>
<tr>
<td>&quot;exporter&quot;</td>
<td>exporter_secret</td>
<td>Deriving exported secrets</td>
</tr>
<tr>
<td>&quot;external&quot;</td>
<td>external_secret</td>
<td>Deriving the external init key</td>
</tr>
<tr>
<td>&quot;confirm&quot;</td>
<td>confirmation_key</td>
<td>Computing the confirmation MAC for an epoch</td>
</tr>
<tr>
<td>&quot;membership&quot;</td>
<td>membership_key</td>
<td>Computing the membership MAC for an MLSPlaintext</td>
</tr>
<tr>
<td>&quot;resumption&quot;</td>
<td>resumption_psk</td>
<td>Proving membership in this epoch (via a PSK injected later)</td>
</tr>
<tr>
<td>&quot;authentication&quot;</td>
<td>epoch_authenticator</td>
<td>Confirming that two clients have the same view of the group</td>
</tr>
</tbody>
</table>

Table 4: Epoch-derived secrets

The external_secret is used to derive an HPKE key pair whose private key is held by the entire group:

\[
\text{external_priv, external_pub} = \text{KEM.DeriveKeyPair(external_secret)}
\]

The public key external_pub can be published as part of the GroupInfo struct in order to allow non-members to join the group using an external commit.

9.1. Group Context

Each member of the group maintains a GroupContext object that summarizes the state of the group:
struct {
    ProtocolVersion version = mls10;
    CipherSuite cipher_suite;
    opaque group_id<V>;
    uint64 epoch;
    opaque tree_hash<V>;
    opaque confirmed_transcript_hash<V>;
    Extension extensions<V>;
} GroupContext;

The fields in this state have the following semantics:

* The cipher_suite is the cipher suite used by the group.

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

* The epoch field represents the current version of the group.

* The tree_hash field contains a commitment to the contents of the group’s ratchet tree and the credentials for the members of the group, as described in Section 8.8.

* The confirmed_transcript_hash field contains a running hash over the messages that led to this state.

* The extensions field contains the details of any protocol extensions that apply to the group.

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

Different changes to the group will have different effects on the group state. These effects are described in their respective subsections of Section 13.1. The following general rules apply:

* The group_id field is constant.

* The epoch field increments by one for each Commit message that is processed.

* The tree_hash is updated to represent the current tree and credentials.
* The confirmed_transcript_hash field is updated with the data for an MLSAuthenticatedContent encoding a Commit message as described below.

* The extensions field changes when a GroupContextExtensions proposal is committed.

9.2. Transcript Hashes

The transcript hashes computed in MLS represent a running hash over all Proposal and Commit messages that have ever been sent in a group. Commit messages are included directly. Proposal messages are indirectly included via the Commit that applied them. Both types of message are included by hashing the MLSAuthenticatedContent object in which they were sent.

The transcript hash comprises two individual hashes:

* A confirmed_transcript_hash that represents a transcript over the whole history of Commit messages, up to and including the signature of the most recent Commit.

* An interim_transcript_hash that covers the confirmed transcript hash plus the confirmation_tag of the most recent Commit.

New members compute the interim transcript hash using the confirmation_tag field of the GroupInfo struct, while existing members can compute it directly.

Each Commit message updates these hashes by way of its enclosing MLSAuthenticatedContent. The MLSAuthenticatedContent struct is split into ConfirmedTranscriptHashInput and InterimTranscriptHashInput. The former is used to update the confirmed transcript hash and the latter to update the interim transcript hash.

```c
struct {
    WireFormat wire_format;
    MLSContent content; // with content.content_type == commit
    opaque signature<V>;
} ConfirmedTranscriptHashInput;

struct {
    MAC confirmation_tag;
} InterimTranscriptHashInput;
```
interim_transcript_hash_[0] = ""; // zero-length octet string

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

interim_transcript_hash_[n+1] =
    Hash(confirmed_transcript_hash_[n] ||
        InterimTranscriptHashInput_[n]);

9.3. External Initialization

In addition to initializing a new epoch via KDF invocations as described above, an MLS group can also initialize a new epoch via an asymmetric interaction using the external key pair for the previous epoch. This is done when a new member is joining via an external commit.

In this process, the joiner sends a new init_secret value to the group using the HPKE export method. The joiner then uses that init_secret with information provided in the GroupInfo and an external Commit to initialize their copy of the key schedule for the new epoch.

kem_output, context = SetupBaseS(external_pub, "")
init_secret = context.export("MLS 1.0 external init secret", KDF.Nh)

Members of the group receive the kem_output in an ExternalInit proposal and perform the corresponding calculation to retrieve the init_secret value.

context = SetupBaseR(kem_output, external_priv, "")
init_secret = context.export("MLS 1.0 external init secret", KDF.Nh)

In both cases, the info input to HPKE is set to the GroupInfo for the previous epoch, encoded using the TLS serialization.

9.4. Pre-Shared Keys

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

Injecting an external PSK can improve security in the case where having a full run of Updates across members is too expensive, or if the external group key establishment mechanism provides stronger security against classical or quantum adversaries.
Note that, as a PSK may have a different lifetime than an Update, it does not necessarily provide the same Forward Secrecy (FS) or Post-Compromise Security (PCS) guarantees as a Commit message. Unlike the key pairs populated in the tree by an Update or Commit, which are always freshly generated, PSKs may be pre-distributed and stored. This creates the risk that a PSK may be compromised in the process of distribution and storage. The security that the group gets from injecting a PSK thus depends on both the entropy of the PSK and the risk of compromise. These factors are outside of the scope of this document, but should be considered by application designers relying on PSKs.

Each PSK in MLS has a type that designates how it was provisioned. External PSKs are provided by the application, while resumption PSKs are derived from the MLS key schedule and used in cases where it is necessary to authenticate a member’s participation in a prior epoch.

The injection of one or more PSKs into the key schedule is signaled in two ways: Existing members are informed via PreSharedKey proposals covered by a Commit, and new members added in the Commit are informed by the GroupSecrets object in the Welcome message corresponding to the Commit. To ensure that existing and new members compute the same PSK input to the key schedule, the Commit and GroupSecrets objects MUST indicate the same set of PSKs, in the same order.
enum {
    reserved(0),
    external(1),
    resumption(2),
    (255)
} PSKType;

enum {
    reserved(0),
    application(1),
    reinit(2),
    branch(3),
} ResumptionPSKUsage;

struct {
    PSKType psktype;
    select (PreSharedKeyID.psktype) {
        case external:
            opaque psk_id<V>;

        case resumption:
            ResumptionPSKUsage usage;
            opaque psk_group_id<V>;
            uint64 psk_epoch;
    }
    opaque psk_nonce<V>;
} PreSharedKeyID;

Each time a client injects a PSK into a group, the psk_nonce of its PreSharedKeyID MUST be set to a fresh random value of length KDF.Nh, where KDF is the KDF for the ciphersuite of the group into which the PSK is being injected. This ensures that even when a PSK is used multiple times, the value used as an input into the key schedule is different each time.

Upon receiving a Commit with a PreSharedKey proposal or a GroupSecrets object with the psks field set, the receiving Client includes them in the key schedule in the order listed in the Commit, or in the psks field respectively. For resumption PSKs, the PSK is defined as the resumption_psk of the group and epoch specified in the PreSharedKeyID object. Specifically, psk_secret is computed as follows:

struct {
    PreSharedKeyID id;
    uint16 index;
    uint16 count;
} PSKLabel;
psk_extracted_[i] = KDF.Extract(0, psk_[i])
psk_input_[i] = ExpandWithLabel(psk_extracted_[i], "derived psk",
                      PSKLabel, KDF.Nh)

psk_secret_[0] = 0
psk_secret_[i] = KDF.Extract(psk_input_[i-1], psk_secret_[i-1])
psk_secret = psk_secret_[n]

Here 0 represents the all-zero vector of length KDF.Nh. The index
field in PSKLabel corresponds to the index of the PSK in the psk
array, while the count field contains the total number of PSKs. In
other words, the PSKs are chained together with KDF.Extract
invocations (labeled "Extract" for brevity in the diagram), as
follows:

```
  0       0       = psk_secret_[0]
    |               |
    V               V
psk_[0] --> Extract --> ExpandWithLabel --> Extract = psk_secret_[1]
    |               |
    V               V
psk_[1] --> Extract --> ExpandWithLabel --> Extract = psk_secret_[2]
    |       ...           |
    V               V
psk_[n-1] --> Extract --> ExpandWithLabel --> Extract = psk_secret_[n]
```

In particular, if there are no PreSharedKey proposals in a given
Commit, then the resulting psk_secret is psk_secret_[0], the all-zero
vector.

9.5. Exporters

The main MLS key schedule provides an exporter_secret which can be
used by an application to derive new secrets for use outside of MLS.

```
MLS-Exporter(Label, Context, Length) =
  ExpandWithLabel(DeriveSecret(exporter_secret, Label),
                    "exporter", Hash(Context), Length)
```

Applications SHOULD provide a unique label to MLS-Exporter that
identifies the secret’s intended purpose. This is to help prevent
the same secret from being generated and used in two different
places.
The exported values are bound to the group epoch from which the exporter_secret is derived, and hence reflect a particular state of the group.

It is RECOMMENDED for the application generating exported values to refresh those values after a Commit is processed.

9.6. Resumption PSK

The main MLS key schedule provides a resumption_psk that is used as a PSK to inject entropy from one epoch into another. This functionality is used in the reinitialization and branching processes described in Section 12.2 and Section 12.3, but may be used by applications for other purposes.

Some uses of resumption PSKs might call for the use of PSKs from historical epochs. The application SHOULD specify an upper limit on the number of past epochs for which the resumption_psk may be stored.

9.7. Epoch Authenticators

The main MLS key schedule provides a per-epoch epoch_authenticator. If one member of the group is being impersonated by an active attacker, the epoch_authenticator computed by their client will differ from those computed by the other group members.

This property can be used to construct defenses against impersonation attacks that are effective even if members’ signature keys are compromised. As a trivial example, if the users of the clients in an MLS group were to meet in person and reliably confirm that their epoch authenticator values were equal (using some suitable user interface), then each user would be assured that the others were not being impersonated in the current epoch. As soon as the epoch changed, though, they would need to re-do this confirmation. The state of the group would have changed, possibly introducing an attacker.

More generally, in order for the members of an MLS group to obtain concrete authentication protections using the epoch_authenticator, they will need to use it in some secondary protocol (such as the face-to-face protocol above). The details of that protocol will then determine the specific authentication protections provided to the MLS group.
10. Secret Tree

For the generation of encryption keys and nonces, the key schedule begins with the encryption_secret at the root and derives a tree of secrets with the same structure as the group’s ratchet tree. Each leaf in the Secret Tree is associated with the same group member as the corresponding leaf in the ratchet tree.

If N is a parent node in the Secret Tree then the secrets of the children of N are defined as follows (where left(N) and right(N) denote the children of N):

\[
\text{tree_node}_\{N\}_\text{secret} \\
\downarrow \\
\text{ExpandWithLabel}(. , "tree", "left", KDF.Nh) \\
= \text{tree_node}_\{\text{left}(N)\}_\text{secret} \\
\downarrow \\
\text{ExpandWithLabel}(. , "tree", "right", KDF.Nh) \\
= \text{tree_node}_\{\text{right}(N)\}_\text{secret}
\]

The secret in the leaf of the Secret Tree is used to initiate two symmetric hash ratchets, from which a sequence of single-use keys and nonces are derived, as described in Section 10.1. The root of each ratchet is computed as:

\[
\text{tree_node}_\{N\}_\text{secret} \\
\downarrow \\
\text{ExpandWithLabel}(. , "handshake", "", KDF.Nh) \\
= \text{handshake_ratchet_secret}_\{N\}_\{0\} \\
\downarrow \\
\text{ExpandWithLabel}(. , "application", "", KDF.Nh) \\
= \text{application_ratchet_secret}_\{N\}_\{0\}
\]

10.1. Encryption Keys

As described in Section 7, MLS encrypts three different types of information:

* Metadata (sender information)
* Handshake messages (Proposal and Commit)
* Application messages
The sender information used to look up the key for content encryption is encrypted with an AEAD where the key and nonce are derived from both sender_data_secret and a sample of the encrypted message content.

For handshake and application messages, a sequence of keys is derived via a "sender ratchet". Each sender has their own sender ratchet, and each step along the ratchet is called a "generation".

The following figure shows a secret tree for a four-member group, with the handshake and application ratchets that member D will use for sending and the first two application keys and nonces.

```
G
 / \
E F
 / \ / \A B C D
 / \HR0 AR0---K0
| ||--N0
AR1---K1
| |-N1
AR2
```

Figure 16: Secret tree for a four-member group

A sender ratchet starts from a per-sender base secret derived from a Secret Tree, as described in Section 10. The base secret initiates a symmetric hash ratchet which generates a sequence of keys and nonces. The sender uses the j-th key/nonce pair in the sequence to encrypt (using the AEAD) the j-th message they send during that epoch. Each key/nonce pair MUST NOT be used to encrypt more than one message.

Keys, nonces, and the secrets in ratchets are derived using DeriveTreeSecret. The context in a given call consists of the current position in the ratchet.

```
DeriveTreeSecret(Secret, Label, Generation, Length) = ExpandWithLabel(Secret, Label, Generation, Length)
```

Where Generation is encoded as a big endian uint32.
ratchet_secret_[N]_[j]

    --- DeriveTreeSecret(., "nonce", j, AEAD.Nn)
    = ratchet_nonce_[N]_[j]

    --- DeriveTreeSecret(., "key", j, AEAD.Nk)
    = ratchet_key_[N]_[j]

| V DeriveTreeSecret(., "secret", j, KDF.Nh)
| = ratchet_secret_[N]_[j+1]

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

10.2. Deletion Schedule

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

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

* a key, nonce, or secret derived from $S$ has been consumed. (This goes for values derived via DeriveSecret as well as ExpandWithLabel.)

Here, $S$ may be the init_secret, commit_secret, epoch_secret, encryption_secret as well as any secret in a Secret Tree or one of the ratchets.

As soon as a group member consumes a value they MUST immediately delete (all representations of) that value. This is crucial to ensuring forward secrecy for past messages. Members MAY keep unconsumed values around for some reasonable amount of time to handle out-of-order message delivery.

For example, suppose a group member encrypts or (successfully) decrypts an application message using the $j$-th key and nonce in the ratchet of leaf node $L$ in some epoch $n$. Then, for that member, at least the following values have been consumed and MUST be deleted:

* the commit_secret, joiner_secret, epoch_secret, encryption_secret of that epoch $n$ as well as the init_secret of the previous epoch $n-1$,

* all node secrets in the Secret Tree on the path from the root to the leaf with node $L$. 
* the first j secrets in the application data ratchet of node L and
  * application_ratchet_nonce_[L][j] and
    application_ratchet_key_[L][j].

Concretely, consider the Secret Tree shown in Figure 16. Client A, B, or C would generate the illustrated values on receiving a message from D with generation equal to 1, having not received a message with generation 0 (e.g., due to out-of-order delivery). In such a case, the following values would be consumed:

* The key K1 and nonce N1 used to decrypt the message
* The application ratchet secrets AR1 and AR0
* The tree secrets D, F, G (recall that G is the encryption_secret for the epoch)
* The epoch_secret, commit_secret, psk_secret, and joiner_secret for the current epoch

Other values may be retained (not consumed):

* K0 and N0 for decryption of an out-of-order message with generation 0
* AR2 for derivation of further message decryption keys and nonces
* HR0 for protection of handshake messages from D
* E and C for deriving secrets used by senders A, B, and C

11. Key Packages

In order to facilitate the asynchronous addition of clients to a group, key packages are pre-published that provide some public information about a user. A KeyPackage object specifies:

1. A protocol version and ciphersuite that the client supports,
2. a public key that others can use to encrypt a Welcome message to this client (an "init key"), and
3. the content of the leaf node that should be added to the tree to represent this client.
KeyPackages are intended to be used only once and SHOULD NOT be reused except in the case of last resort (see Section 16.4). Clients MAY generate and publish multiple KeyPackages to support multiple ciphersuites.

The value for init_key MUST be a public key for the asymmetric encryption scheme defined by cipher_suite, and it MUST be unique among the set of KeyPackages created by this client. Likewise, the leaf_node field MUST be valid for the ciphersuite, including both the encryption_key and signature_key fields. The whole structure is signed using the client’s signature key. A KeyPackage object with an invalid signature field MUST be considered malformed.

The signature is computed by the function SignWithLabel with a label KeyPackageTBS and a content comprising of all of the fields except for the signature field.

struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey init_key;
    LeafNode leaf_node;
    Extension extensions<V>;
    // SignWithLabel(., "KeyPackageTBS", KeyPackageTBS)
    opaque signature<V>;
} KeyPackage;

struct {
    ProtocolVersion version;
    CipherSuite cipher_suite;
    HPKEPublicKey init_key;
    LeafNode leaf_node;
    Extension extensions<V>;
} KeyPackageTBS;

If a client receives a KeyPackage carried within an MLSMessage object, then it MUST verify that the version field of the KeyPackage has the same value as the version field of the MLSMessage. The version field in the KeyPackage provides an explicit signal of the intended version to the other members of group when they receive the KeyPackage in an Add proposal.
The field leaf_node.capabilities indicates what protocol versions, ciphersuites, credential types, and non-default proposal/extension types are supported by the client. (Proposal and extension types defined in this document are considered "default" and not listed.) This information allows MLS session establishment to be safe from downgrade attacks on the parameters described (as discussed in Section 12), while still only advertising one version / ciphersuite per KeyPackage.

The field leaf_node.leaf_node_source of the LeafNode in a KeyPackage MUST be set to key_package.

Extension included in the extensions or leaf_node.extensions fields MUST be included in the leaf_node.capabilities field.

11.1. KeyPackage Validation

The validity of a KeyPackage needs to be verified at a few stages:

* When a KeyPackage is downloaded by a group member, before it is used to add the client to the group
* When a KeyPackage is received by a group member in an Add message

The client verifies the validity of a KeyPackage using the following steps:

* Verify that the ciphersuite and protocol version of the KeyPackage match those in the GroupContext.
* Verify that the leaf_node of the KeyPackage is valid for a KeyPackage according to Section 8.3.
* Verify that the signature on the KeyPackage is valid using the public key in leaf_node.credential.
* Verify that the value of leaf_node.encryption_key is different from the value of the init_key field.

12. Group Creation

A group is always created with a single member, the "creator". The other members are added when the creator effectively sends itself Add proposals, commits them, and then sends the corresponding Welcome message to the new participants. These processes are described in detail in Section 13.1.1, Section 13.4, and Section 13.4.3.1.
The creator of a group MUST take the following steps to initialize the group:

* Fetch KeyPackages for the members to be added, and select a version and ciphersuite according to the capabilities of the members. To protect against downgrade attacks, the creator MUST use the capabilities information in these KeyPackages to verify that the chosen version and ciphersuite is the best option supported by all members.

* Initialize a one-member group with the following initial values:
  - Ratchet tree: A tree with a single node, a leaf containing an HPKE public key and credential for the creator
  - Group ID: A value set by the creator
  - Epoch: 0
  - Tree hash: The root hash of the above ratchet tree
  - Confirmed transcript hash: The zero-length octet string
  - Interim transcript hash: The zero-length octet string
  - Init secret: A fresh random value of size KDF.Nh
  - Extensions: Any values of the creator’s choosing

* For each member, construct an Add proposal from the KeyPackage for that member (see Section 13.1.1)

* Construct a Commit message that commits all of the Add proposals, in any order chosen by the creator (see Section 13.4)

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

* Transmit the Welcome message to the other new members

Group IDs SHOULD be constructed in such a way that there’s an overwhelmingly low probability of honest group creators generating the same group ID, even without assistance from the Delivery Service. For example, by making the group ID a freshly generated random value of size KDF.Nh. The Delivery Service MAY attempt to ensure that group IDs are globally unique by rejecting the creation of new groups with a previously used ID.
The recipient of a Welcome message processes it as described in Section 13.4.3.1.

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

12.1. Required Capabilities

The configuration of a group imposes certain requirements on clients in the group. At a minimum, all members of the group need to support the ciphersuite and protocol version in use. Additional requirements can be imposed by including a required_capabilities extension in the GroupContext.

```c
struct {
  ExtensionType extension_types<V>;
  ProposalType proposal_types<V>;
  CredentialType credential_types<V>;
} RequiredCapabilities;
```

This extension lists the extensions, proposals, and credential types that must be supported by all members of the group. The "default" proposal and extension types defined in this document are assumed to be implemented by all clients, and need not be listed in RequiredCapabilities in order to be safely used. Note that this is not true for credential types.

For new members, support for required capabilities is enforced by existing members during the application of Add commits. Existing members should of course be in compliance already. In order to ensure this continues to be the case even as the group’s extensions are updated, a GroupContextExtensions proposal is deemed invalid if it contains a required_capabilities extension that requires non-default capabilities not supported by all current members.

12.2. Reinitialization

A group may be reinitialized by creating a new group with the same membership and different parameters, and linking it to the old group via a resumption PSK. The members of a group reinitialize it using the following steps:

1. A member of the old group sends a ReInit proposal (see Section 13.1.5)
2. A member of the old group sends a Commit covering the ReInit proposal.

3. A member of the old group creates an initial Commit setting up a new group that matches the ReInit and sends a Welcome message:

   * The group_id, version, cipher_suite, and extensions fields in the Welcome message MUST be the same as the corresponding fields in the ReInit proposal. The epoch in the Welcome message MUST be 1.

   * The Welcome MUST specify a PreSharedKeyID of type resumption with usage reinit, where the group_id field matches the old group and the epoch field indicates the epoch after the Commit covering the ReInit.

Note that these three steps may be done by the same group member or different members. For example, if a group member sends a Commit with an inline ReInit proposal (steps 1 and 2) but then goes offline, another group member may recreate the group instead. This flexibility avoids situations where a group gets stuck between steps 2 and 3.

Resumption PSKs with usage reinit MUST NOT be used in other contexts. A PreSharedKey proposal with type resumption and usage reinit MUST be considered invalid.

12.3. Subgroup Branching

A new group can be formed from a subset of an existing group’s members, using the same parameters as the old group.

A member can create a subgroup by performing the following steps:

1. Fetch a new KeyPackage for each group member that should be included in the subgroup.

2. Create an initial Commit message that sets up the new group and contains a PreSharedKey proposal of type resumption with usage branch. To avoid key re-use, the psk_nonce included in the PreSharedKeyID object MUST be a randomly sampled nonce of length KDF.Nh.

3. Send the corresponding Welcome message to the subgroup members.

A client receiving a Welcome including a PreSharedKey of type resumption with usage branch MUST verify that the new group reflects a subgroup branched from the referenced group by checking:
* The version and ciphersuite values in the Welcome are the same as those used by the old group.

* The epoch in the Welcome message MUST be 1.

* Each LeafNode in a new subgroup MUST match some LeafNode in the original group. In this context, a pair of LeafNodes is said to "match" if the identifiers presented by their respective credentials are considered equivalent by the application.

Resumption PSKs with usage branch MUST NOT be used in other contexts. A PreSharedKey proposal with type resumption and usage branch MUST be considered invalid.

13. Group Evolution

Over the lifetime of a group, its membership can change, and existing members might want to change their keys in order to achieve post-compromise security. In MLS, each such change is accomplished by a two-step process:

1. A proposal to make the change is broadcast to the group in a Proposal message

2. A member of the group or a new member broadcasts a Commit message that causes one or more proposed changes to enter into effect

In cases where the Proposal and Commit are sent by the same member, these two steps can be combined by sending the proposals in the commit.

The group thus evolves from one cryptographic state to another each time a Commit message is sent and processed. These states are referred to as "epochs" and are uniquely identified among states of the group by eight-octet epoch values. When a new group is initialized, its initial state epoch is 0x0000000000000000. Each time a state transition occurs, the epoch number is incremented by one.

13.1. Proposals

Proposals are included in an MLSContent by way of a Proposal structure that indicates their type:
/ See IANA registry for registered values
uint16 ProposalType;

struct {
  ProposalType msg_type;
  select (Proposal.msg_type) {
    case add:                      Add;
    case update:                   Update;
    case remove:                   Remove;
    case psk:                      PreSharedKey;
    case reinit:                   ReInit;
    case external_init:           ExternalInit;
    case group_context_extensions: GroupContextExtensions;
  }
} Proposal;

On receiving an MLSContent containing a Proposal, a client MUST verify the signature inside MLSContentAuthData and that the epoch field of the enclosing MLSContent is equal to the epoch field of the current GroupContext object. If the verification is successful, then the Proposal should be cached in such a way that it can be retrieved by hash (as a ProposalOrRef object) in a later Commit message.

13.1.1. Add

An Add proposal requests that a client with a specified KeyPackage be added to the group.

struct {
  KeyPackage key_package;
} Add;

An Add proposal is invalid if the KeyPackage is invalid according to Section 11.1.

An Add is applied after being included in a Commit message. The position of the Add in the list of proposals determines the leaf node where the new member will be added. For the first Add in the Commit, the corresponding new member will be placed in the leftmost empty leaf in the tree, for the second Add, the next empty leaf to the right, etc. If no empty leaf exists, the tree is extended to the right.

* Identify the leaf L for the new member: if there are empty leaves in the tree, L is the leftmost empty leaf. Otherwise, the tree is extended to the right by one leaf node and L is the new leaf.
For each non-blank intermediate node along the path from the leaf L to the root, add L’s leaf index to the unmerged_leaves list for the node.

* Set the leaf node L to a new node containing the LeafNode object carried in the leaf_node field of the KeyPackage in the Add.

13.1.2. Update

An Update proposal is a similar mechanism to Add with the distinction that it replaces the sender’s LeafNode in the tree instead of adding a new leaf to the tree.

```c
struct {
    LeafNode leaf_node;
} Update;
```

An Update proposal is invalid if the LeafNode is invalid for an Update proposal according to Section 8.3.

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

* Replace the sender’s LeafNode with the one contained in the Update proposal

* Blank the intermediate nodes along the path from the sender’s leaf to the root

13.1.3. Remove

A Remove proposal requests that the member with the leaf index removed be removed from the group.

```c
struct {
    uint32 removed;
} Remove;
```

A Remove proposal is invalid if the removed field does not identify a non-blank leaf node.

A member of the group applies a Remove message by taking the following steps:
* Identify a leaf node matching removed. This lookup MUST be done on the tree before any non-Remove proposals have been applied (the "old" tree in the terminology of Section 13.4), since proposals such as Update can change the LeafNode stored at a leaf. Let L be this leaf node.

* Replace the leaf node L with a blank node

* Blank the intermediate nodes along the path from L to the root

* Truncate the tree by removing the right subtree until there is at least one non-blank leaf node in the right subtree. If the rightmost non-blank leaf has index L, then this will result in the tree having $2^d$ leaves, where $d$ is the smallest value such that $2^d > L$.

13.1.4. PreSharedKey

A PreSharedKey proposal can be used to request that a pre-shared key be injected into the key schedule in the process of advancing the epoch.

```c
struct {
    PreSharedKeyID psk;
} PreSharedKey;
```

A PreSharedKey proposal is invalid if any of the following is true:

* The psktype in the PreSharedKeyID struct is set to resumption and the usage is reinit or branch.

* The psk_nonce is not of length KDF.Nh.

The psk_nonce MUST be randomly sampled. When processing a Commit message that includes one or more PreSharedKey proposals, group members derive psk_secret as described in Section 9.4, where the order of the PSKs corresponds to the order of the PreSharedKey proposals in the Commit.

13.1.5. ReInit

A ReInit proposal represents a request to reinitialize the group with different parameters, for example, to increase the version number or to change the ciphersuite. The reinitialization is done by creating a completely new group and shutting down the old one.
struct {
    opaque group_id<V>;
    ProtocolVersion version;
    CipherSuite cipher_suite;
    Extension extensions<V>;
} ReInit;

A ReInit proposal is invalid if the version field is less than the version for the current group.

A member of the group applies a ReInit proposal by waiting for the committer to send the Welcome message that matches the ReInit, according to the criteria in Section 12.2.

13.1.6. ExternalInit

An ExternalInit proposal is used by new members that want to join a group by using an external commit. This proposal can only be used in that context.

struct {
    opaque kem_output<V>;
} ExternalInit;

A member of the group applies an ExternalInit message by initializing the next epoch using an init secret computed as described in Section 9.3. The kem_output field contains the required KEM output.

13.1.7. GroupContextExtensions

A GroupContextExtensions proposal is used to update the list of extensions in the GroupContext for the group.

struct { Extension extensions<V>; } GroupContextExtensions;

A GroupContextExtensions proposal is invalid if it includes a required_capabilities extension and some members of the group do not support some of the required capabilities (including those added in the same commit, and excluding those removed).

A member of the group applies a GroupContextExtensions proposal with the following steps:

* Remove all of the existing extensions from the GroupContext object for the group and replacing them with the list of extensions in the proposal. (This is a wholesale replacement, not a merge. An extension is only carried over if the sender of the proposal includes it in the new list.)
Note that once the GroupContext is updated, its inclusion in the confirmation_tag by way of the key schedule will confirm that all members of the group agree on the extensions in use.

13.1.8. External Proposals

Add and Remove proposals can be constructed and sent to the group by a party that is outside the group in two cases. One case, indicated by an external SenderType is useful in cases where, for example, an automated service might propose to remove a member of a group who has been inactive for a long time, or propose adding a newly-hired staff member to a group representing a real-world team.

ReInit proposals can also be sent to the group by an external sender, for example to enforce a changed policy regarding MLS version or ciphersuite.

The external SenderType requires that signers are pre-provisioned to the clients within a group and can only be used if the external_senders extension is present in the group’s GroupContext.

The other case, indicated by a new_member_proposal SenderType is useful when existing members of the group can independently authorize the addition of an MLS client proposing it be added to the group. External proposals which are not authorized are considered invalid.

An external proposal MUST be sent as an MLSPlaintext object, since the sender will not have the keys necessary to construct an MLSCiphertext object.

13.1.8.1. External Senders Extension

The external_senders extension is a group context extension that contains the credentials and signature keys of senders that are permitted to send external proposals to the group.

```c
struct {
    SignaturePublicKey signature_key;
    Credential credential;
} ExternalSender;

ExternalSender external_senders<V>;
```
13.2. Proposal List Validation

A group member creating a commit and a group member processing a commit MUST verify that the list of committed proposals is valid using one of the following procedures, depending on whether the commit is external or not.

For a regular, i.e. not external, commit the list is invalid if any of the following occurs:

* It contains an individual proposal that is invalid as specified in Section 13.1.
* It contains an Update proposal generated by the committer.
* It contains a Remove proposal that removes the committer.
* It contains multiple Update and/or Remove proposals that apply to the same leaf. If the committer has received multiple such proposals they SHOULD prefer any Remove received, or the most recent Update if there are no Removes.
* It contains multiple Add proposals that contain KeyPackages that represent the same client according to the application (for example, identical signature keys).
* It contains an Add proposal with a KeyPackage that represents a client already in the group according to the application, unless there is a Remove proposal in the list removing the matching client from the group.
* It contains multiple PreSharedKey proposals that reference the same PreSharedKeyID.
* It contains multiple GroupContextExtensions proposals.
* It contains a ReInit proposal together with any other proposal. If the committer has received other proposals during the epoch, they SHOULD prefer them over the ReInit proposal, allowing the ReInit to be resent and applied in a subsequent epoch.
* It contains an ExternalInit proposal.
* It contains a proposal with a non-default proposal type that is not supported by some members of the group that will process the Commit (i.e., members being added or removed by the Commit do not need to support the proposal type).
* After processing the commit the ratchet tree is invalid, in particular, if it contains any leaf node that is invalid according to Section 8.3.

An application may extend the above procedure by additional rules, for example, requiring application-level permissions to add members, or rules concerning non-default proposal types.

For an external commit, the list is valid if it contains only the following proposals (not necessarily in this order):

* Exactly one ExternalInit
* At most one Remove proposal, with which the joiner removes an old version of themselves. If a Remove proposal is present, then the LeafNode in the path field of the external commit MUST meet the same criteria as would the LeafNode in an Update for the removed leaf (see Section 13.1.2). In particular, the credential in the LeafNode MUST present a set of identifiers that is acceptable to the application for the removed participant.
* Zero or more PreSharedKey proposals.
* No other proposals.

Proposal types defined in the future may make updates to the above validation logic to incorporate considerations related to proposals of the new type.

13.3. Applying a Proposal List

The sections above defining each proposal type describe how each individual proposals is applied. When creating or processing a Commit, a client applies a list of proposals to the ratchet tree and GroupContext. The client MUST apply the proposals in the list in the following order:

* If there is a GroupContextExtensions proposal, replace the extensions field of the GroupContext for the group with the contents of the proposal. The new extensions MUST be used for evaluating other proposals in this list. For example, if a GroupContextExtensions proposal adds a required_capabilities extension, then any Add proposals need to indicate support for those capabilities.
* Apply any Update proposals to the ratchet tree, in any order.
* Apply any Remove proposals to the ratchet tree, in any order.
* Apply any Add proposals to the ratchet tree, in the order they appear in the list.

* Look up the PSK secrets for any PreSharedKey proposals, in the order they appear in the list. These secrets are then used to advance the key schedule later in Commit processing.

* If there is an ExternalInit proposal, use it to derive the init_secret for use later in Commit processing.

* If there is a ReInit proposal, note its parameters for application later in Commit processing.

Proposal types defined in the future MUST specify how the above steps are to be adjusted to accommodate the application of proposals of the new type.

13.4. Commit

A Commit message initiates a new epoch for the group, based on a collection of Proposals. It instructs group members to update their representation of the state of the group by applying the proposals and advancing the key schedule.

Each proposal covered by the Commit is included by a ProposalOrRef value, which identifies the proposal to be applied by value or by reference. Commits that refer to new Proposals from the committer can be included by value. Commits for previously sent proposals from anyone (including the committer) can be sent by reference. Proposals sent by reference are specified by including the hash of the MLSAuthenticatedContent object in which the proposal was sent (see Section 6.2).
enum {
    reserved(0),
    proposal(1),
    reference(2),
    (255)
} ProposalOrRefType;

struct {
    ProposalOrRefType type;
    select (ProposalOrRef.type) {
        case proposal:  Proposal proposal;
        case reference: ProposalRef reference;
    }
} ProposalOrRef;

struct {
    ProposalOrRef proposals<V>;
    optional<UpdatePath> path;
} Commit;

A group member that has observed one or more valid proposals within an epoch MUST send a Commit message before sending application data. This ensures, for example, that any members whose removal was proposed during the epoch are actually removed before any application data is transmitted.

A sender and a receiver of a Commit MUST verify that the committed list of proposals is valid as specified in Section 13.2. A list is invalid if, for example, it includes an Update and a Remove for the same member, or an Add when the sender does not have the application-level permission to add new users.

The sender of a Commit SHOULD include all proposals that it has received during the current epoch, that are valid according to the rules for their proposal types and according to application policy, as long as this results in a valid proposal list.

Due to the asynchronous nature of proposals, receivers of a Commit SHOULD NOT enforce that all valid proposals sent within the current epoch are referenced by the next Commit. In the event that a valid proposal is omitted from the next Commit, and that proposal is still valid in the current epoch, the sender of the proposal MAY resend it after updating it to reflect the current epoch.

A member of the group MAY send a Commit that references no proposals at all, which would thus have an empty proposals vector. Such a Commit resets the sender’s leaf and the nodes along its direct path, and provides forward secrecy and post-compromise security with regard
to the sender of the Commit. An Update proposal can be regarded as a "lazy" version of this operation, where only the leaf changes and intermediate nodes are blanked out.

By default, the path field of a Commit MUST be populated. The path field MAY be omitted if (a) it covers at least one proposal and (b) none of the proposals covered by the Commit are of "path required" types. A proposal type requires a path if it cannot change the group membership in a way that requires the forward secrecy and post-compromise security guarantees that an UpdatePath provides. The only proposal types defined in this document that do not require a path are:

* add
* psk
* reinit

New proposal types MUST state whether they require a path. If any instance of a proposal type requires a path, then the proposal type requires a path. This attribute of a proposal type is reflected in the "Path Required" field of the proposal type registry defined in Section 17.3.

Update and Remove proposals are the clearest examples of proposals that require a path. An UpdatePath is required to evict the removed member or the old appearance of the updated member.

In pseudocode, the logic for validating the path field of a Commit is as follows:
pathRequiredTypes = [
    update,
    remove,
    external_init,
    group_context_extensions
]

pathRequired = false

for i, id in commit.proposals:
    proposal = proposalCache[id]
    assert(proposal != null)

    pathRequired = pathRequired ||
    (proposal.msg_type in pathRequiredTypes)

if len(commit.proposals) == 0 || pathRequired:
    assert(commit.path != null)

To summarize, a Commit can have three different configurations, with different uses:

1. An "empty" Commit that references no proposals, which updates the committer’s contribution to the group and provides PCS with regard to the committer.

2. A "partial" Commit that references proposals that do not require a path, and where the path is empty. Such a commit doesn’t provide PCS with regard to the committer.

3. A "full" Commit that references proposals of any type, which provides FS with regard to any removed members and PCS for the committer and any updated members.

13.4.1. Creating a Commit

When creating or processing a Commit, a client updates the ratchet tree and GroupContext for the group. These values advance from an "old" state reflecting the current epoch to a "new" state reflecting the new epoch initiated by the Commit. When the Commit includes an UpdatePath, a "provisional" group context is constructed that reflects changes due to the proposals and UpdatePath, but with the old confirmed transcript hash.

A member of the group creates a Commit message and the corresponding Welcome message at the same time, by taking the following steps:
* Verify that the list of proposals to be committed is valid as specified in Section 13.2.

* Construct an initial Commit object with the proposals field populated from Proposals received during the current epoch, and an empty path field.

* Create the new ratchet tree and GroupContext by applying the list of proposals to the old ratchet tree and GroupContext, as defined in Section 13.3.

* Decide whether to populate the path field: If the path field is required based on the proposals that are in the commit (see above), then it MUST be populated. Otherwise, the sender MAY omit the path field at its discretion.

* If populating the path field:
  
  - If this is an external commit, assign the sender the leftmost blank leaf node in the new ratchet tree. If there are no blank leaf nodes in the new ratchet tree, add a blank leaf to the right side of the new ratchet tree and assign it to the sender.

  - Update the sender’s direct path in the ratchet tree as described in Section 8.5. Define commit_secret as the value path_secret[n+1] derived from the last path secret value (path_secret[n]) derived for the UpdatePath.

  - Construct a provisional GroupContext object containing the following values:
    
    o group_id: Same as the old GroupContext
    o epoch: The epoch number for the new epoch
    o tree_hash: The tree hash of the new ratchet tree
    o confirmed_transcript_hash: Same as the old GroupContext
    o extensions: The new GroupContext extensions (possibly updated by a GroupContextExtensions proposal)

  - Encrypt the path secrets resulting from the tree update to the group as described in Section 8.5, using the provisional group context as the context for HPKE encryption.
- Create an UpdatePath containing the sender’s new leaf node and the new public keys and encrypted path secrets along the sender’s filtered direct path. Assign this UpdatePath to the path field in the Commit.

* If not populating the path field: Set the path field in the Commit to the null optional. Define commit_secret as the all-zero vector of length KDF.Nh (the same length as a path_secret value would be).

* Derive the psk_secret as specified in Section 9.4, where the order of PSKs in the derivation corresponds to the order of PreSharedKey proposals in the proposals vector.

* Construct an MLSContent object containing the Commit object. Sign the MLSContent using the old GroupContext as context.

- Use the MLSContent to update the confirmed transcript hash and update the new GroupContext.

- Use the init_secret from the previous epoch, the commit_secret and the psk_secret as defined in the previous steps, and the new GroupContext to compute the new joiner_secret, welcome_secret, epoch_secret, and derived secrets for the new epoch.

- Use the confirmation_key for the new epoch to compute the confirmation_tag value.

- Calculate the interim transcript hash using the new confirmed transcript hash and the confirmation_tag from the MLSContentAuthData.

* Protect the MLSAuthenticatedContent object using keys from the old epoch:

- If encoding as MLSPlaintext, compute the membership_tag value using the membership_key.

- If encoding as an MLSCiphertext, encrypt the message using the sender_data_secret and the next (key, nonce) pair from the sender’s handshake ratchet.

* Construct a GroupInfo reflecting the new state:

- Group ID, epoch, tree, confirmed transcript hash, interim transcript hash, and group context extensions from the new state
- The confirmation_tag from the MLSContentAuthData object
- Other extensions as defined by the application
- Optionally derive an external keypair as described in Section 9 (required for External Commits, see Section 13.4.3.2)
- Sign the GroupInfo using the member’s private signing key
- Encrypt the GroupInfo using the key and nonce derived from the joiner_secret for the new epoch (see Section 13.4.3.1)
  * For each new member in the group:
    - Identify the lowest common ancestor in the tree of the new member’s leaf node and the member sending the Commit
    - If the path field was populated above: Compute the path secret corresponding to the common ancestor node
    - Compute an EncryptedGroupSecrets object that encapsulates the init_secret for the current epoch and the path secret (if present).
  * Construct a Welcome message from the encrypted GroupInfo object, the encrypted key packages, and any PSKs for which a proposal was included in the Commit. The order of the psks MUST be the same as the order of PreSharedKey proposals in the proposals vector.
  * If a ReInit proposal was part of the Commit, the committer MUST create a new group with the parameters specified in the ReInit proposal, and with the same members as the original group. The Welcome message MUST include a PreSharedKeyID with the following parameters:
    - psktype: resumption
    - usage: reinit
    - group_id: The group ID for the current group
    - epoch: The epoch that the group will be in after this Commit

13.4.2. Processing a Commit

A member of the group applies a Commit message by taking the following steps:
* Verify that the epoch field of the enclosing MLSContent is equal to the epoch field of the current GroupContext object.

* Unprotect the Commit using the keys from the current epoch:
  - If the message is encoded as MLSPlaintext, verify the membership MAC using the membership_key.
  - If the message is encoded as MLSCiphertext, decrypt the message using the sender_data_secret and the (key, nonce) pair from the step on the sender’s hash ratchet indicated by the generation field.

* Verify that the signature on the MLSContent message as described in Section 7.1.

* Verify that the proposals vector is valid as specified in Section 13.2.

* Verify that all PreSharedKey proposals in the proposals vector are available.

* Create the new ratchet tree and GroupContext by applying the list of proposals to the old ratchet tree and GroupContext, as defined in Section 13.3.

* Verify that the path value is populated if the proposals vector contains any Update or Remove proposals, or if it’s empty. Otherwise, the path value MAY be omitted.

* If the path value is populated, validate it and apply it to the tree:
  - If this is an external commit, assign the sender the leftmost blank leaf node in the new ratchet tree. If there are no blank leaf nodes in the new ratchet tree, add a blank leaf to the right side of the new ratchet tree and assign it to the sender.
  - Validate the LeafNode as specified in Section 8.3. The leaf_node_source field MUST be set to commit.
  - Verify that the encryption_key value in the LeafNode is different from the committer’s current leaf node.
  - Merge the UpdatePath into the new ratchet tree as described in Section 8.5.
- Construct a provisional GroupContext object containing the following values:
  o group_id: Same as the old GroupContext
  o epoch: The epoch number for the new epoch
  o tree_hash: The tree hash of the new ratchet tree
  o confirmed_transcript_hash: Same as the old GroupContext
  o extensions: The new GroupContext extensions (possibly updated by a GroupContextExtensions proposal)

- Decrypt the path secrets for UpdatePath as described in Section 8.5, using the provisional GroupContext as the context for HPKE decryption.

- Define commit_secret as the value path_secret[n+1] derived from the last path secret value (path_secret[n]) derived for the UpdatePath.

* If the path value is not populated: Define commit_secret as the all-zero vector of length KDF.Nh (the same length as a path_secret value would be).

* Update the confirmed and interim transcript hashes using the new Commit, and generate the new GroupContext.

* Derive the psk_secret as specified in Section 9.4, where the order of PSKs in the derivation corresponds to the order of PreSharedKey proposals in the proposals vector.

* Use the init_secret from the previous epoch, the commit_secret and the psk_secret as defined in the previous steps, and the new GroupContext to compute the new joiner_secret, welcome_secret, epoch_secret, and derived secrets for the new epoch.

* Use the confirmation_key for the new epoch to compute the confirmation tag for this message, as described below, and verify that it is the same as the confirmation_tag field in the MLSContentAuthData object.

* If the above checks are successful, consider the new GroupContext object as the current state of the group.
* If the Commit included a ReInit proposal, the client MUST NOT use the group to send messages anymore. Instead, it MUST wait for a Welcome message from the committer meeting the requirements of Section 12.2.

13.4.3. Adding Members to the Group

New members can join the group in two ways. Either by being added by a group member, or by adding themselves through an external Commit. In both cases, the new members need information to bootstrap their local group state.

```c
struct {
    GroupContext group_context;
    Extension extensions<V>;
    MAC confirmation_tag;
    uint32 signer;
    // SignWithLabel(., "GroupInfoTBS", GroupInfoTBS)
    opaque signature<V>;
} GroupInfo;
```

New members MUST verify that group_id is unique among the groups they’re currently participating in.

New members also MUST verify the signature using the public key taken from the leaf node of the ratchet tree with leaf index signer. The signature covers the following structure, comprising all the fields in the GroupInfo above signature:

```c
struct {
    GroupContext group_context;
    Extension extensions<V>;
    MAC confirmation_tag;
    uint32 signer;
} GroupInfoTBS;
```

13.4.3.1. Joining via Welcome Message

The sender of a Commit message is responsible for sending a single Welcome message to all the new members added via Add proposals. The Welcome message provides the new members with the current state of the group after the application of the Commit message. The new members will not be able to decrypt or verify the Commit message, but will have the secrets they need to participate in the epoch initiated by the Commit message.
In order to allow the same Welcome message to be sent to all new members, information describing the group is encrypted with a symmetric key and nonce derived from the joiner_secret for the new epoch. The joiner_secret is then encrypted to each new member using HPKE. In the same encrypted package, the committer transmits the path secret for the lowest (closest to the leaf) node which is contained in the direct paths of both the committer and the new member. This allows the new member to compute private keys for nodes in its direct path that are being reset by the corresponding Commit.

If the sender of the Welcome message wants the receiving member to include a PSK in the derivation of the epoch_secret, they can populate the psks field indicating which PSK to use.

```plaintext
struct {
    opaque path_secret<V>;
} PathSecret;

struct {
    opaque joiner_secret<V>;
    optional<PathSecret> path_secret;
    PreSharedKeyID psks<V>
} GroupSecrets;

struct {
    KeyPackageRef new_member;
    HPKECipherText encrypted_group_secrets;
} EncryptedGroupSecrets;

struct {
    CipherSuite cipher_suite;
    EncryptedGroupSecrets secrets<V>;
    opaque encrypted_group_info<V>;
} Welcome;
```

The client processing a Welcome message will need to have a copy of the group’s ratchet tree. The tree can be provided in the Welcome message, in an extension of type ratchet_tree. If it is sent otherwise (e.g., provided by a caching service on the Delivery Service), then the client MUST download the tree before processing the Welcome.

On receiving a Welcome message, a client processes it using the following steps:
* Identify an entry in the secrets array where the new_member value corresponds to one of this client’s KeyPackages, using the hash indicated by the cipher_suite field. If no such field exists, or if the ciphersuite indicated in the KeyPackage does not match the one in the Welcome message, return an error.

* Decrypt the encrypted_group_secrets value with the algorithms indicated by the ciphersuite and the private key corresponding to init_key in the referenced KeyPackage.

* If a PreSharedKeyID is part of the GroupSecrets and the client is not in possession of the corresponding PSK, return an error. Additionally, if a PreSharedKeyID has type resumption with usage reinit or branch, verify that it is the only such PSK.

* From the joiner_secret in the decrypted GroupSecrets object and the PSKs specified in the GroupSecrets, derive the welcome_secret and using that the welcome_key and welcome_nonce. Use the key and nonce to decrypt the encrypted_group_info field.

welcome_nonce = KDF.Expand(welcome_secret, "nonce", AEAD.Nn)
welcome_key = KDF.Expand(welcome_secret, "key", AEAD.Nk)

* Verify the signature on the GroupInfo object. The signature input comprises all of the fields in the GroupInfo object except the signature field. The public key is taken from the LeafNode of the ratchet tree with leaf index signer. If the node is blank or if signature verification fails, return an error.

* Verify that the group_id is unique among the groups that the client is currently participating in.

* Verify that the cipher_suite in the GroupInfo matches the cipher_suite in the KeyPackage.

* Verify the integrity of the ratchet tree.
  - Verify that the tree hash of the ratchet tree matches the tree_hash field in GroupInfo.
  - For each non-empty parent node, verify that it is "parent-hash valid", as described in Section 8.9.2.
  - For each non-empty leaf node, validate the LeafNode as described in Section 8.3.
For each non-empty parent node, verify that each entry in the node’s unmerged_leaves represents a non-blank leaf node that is a descendant of the parent node.

* Identify a leaf whose LeafNode is identical to the one in the KeyPackage. If no such field exists, return an error. Let my_leaf represent this leaf in the tree.

* Construct a new group state using the information in the GroupInfo object.

  - The GroupContext is the group_context field from the GroupInfo object.

  - The new member’s position in the tree is at the leaf my_leaf, as defined above.

  - Update the leaf my_leaf with the private key corresponding to the public key in the node.

  - If the path_secret value is set in the GroupSecrets object:
    Identify the lowest common ancestor of the leaf node my_leaf and of the node of the member with leaf index GroupInfo.signer. Set the private key for this node to the private key derived from the path_secret.

  - For each parent of the common ancestor, up to the root of the tree, derive a new path secret and set the private key for the node to the private key derived from the path secret. The private key MUST be the private key that corresponds to the public key in the node.

* Use the joiner_secret from the GroupSecrets object to generate the epoch secret and other derived secrets for the current epoch.

* Set the confirmed transcript hash in the new state to the value of the confirmed_transcript_hash in the GroupInfo.

* Verify the confirmation tag in the GroupInfo using the derived confirmation key and the confirmed_transcript_hash from the GroupInfo.

* Use the confirmed transcript hash and confirmation tag to compute the interim transcript hash in the new state.

* If a PreSharedKeyID was used that has type resumption with usage reinit or branch, verify that the epoch field in the GroupInfo is equal to 1.
- For usage reinit, verify that the last Commit to the referenced group contains a ReInit proposal and that the group_id, version, cipher_suite, and group_context.extensions fields of the GroupInfo match the ReInit proposal. Additionally, verify that all the members of the old group are also members of the new group, according to the application.

- For usage branch, verify that the version and cipher_suite of the new group match those of the old group, and that the members of the new group compose a subset of the members of the old group, according to the application.

13.4.3.2. Joining via External Commits

External Commits are a mechanism for new members (external parties that want to become members of the group) to add themselves to a group, without requiring that an existing member has to come online to issue a Commit that references an Add Proposal.

Whether existing members of the group will accept or reject an External Commit follows the same rules that are applied to other handshake messages.

New members can create and issue an External Commit if they have access to the following information for the group’s current epoch:

* group ID
* epoch ID
* ciphersuite
* public tree hash
* confirmed transcript hash
* confirmation tag of the most recent Commit
* group extensions
* external public key

In other words, to join a group via an External Commit, a new member needs a GroupInfo with an ExternalPub extension present in its extensions field.
Thus, a member of the group can enable new clients to join by making a GroupInfo object available to them. Note that because a GroupInfo object is specific to an epoch, it will need to be updated as the group advances. In particular, each GroupInfo object can be used for one external join, since that external join will cause the epoch to change.

Note that the tree_hash field is used the same way as in the Welcome message. The full tree can be included via the ratchet_tree extension Section 13.4.3.3.

The information in a GroupInfo is not generally public information, but applications can choose to make it available to new members in order to allow External Commits.

In principle, External Commits work like regular Commits. However, their content has to meet a specific set of requirements:

* External Commits MUST contain a path field (and is therefore a "full" Commit). The joiner is added at the leftmost free leaf node (just as if they were added with an Add proposal), and the path is calculated relative to that leaf node.

* The Commit MUST NOT include any proposals by reference, since an external joiner cannot determine the validity of proposals sent within the group

* External Commits MUST be signed by the new member. In particular, the signature on the enclosing MLSAuthenticatedContent MUST verify using the public key for the credential in the leaf_node of the path field.

* When processing a Commit, both existing and new members MUST use the external init secret as described in Section 9.3.

* The sender type for the MLSAuthenticatedContent encapsulating the External Commit MUST be new_member_commit.

External Commits come in two "flavors" -- a "join" commit that adds the sender to the group or a "resync" commit that replaces a member’s prior appearance with a new one.
Note that the "resync" operation allows an attacker that has compromised a member's signature private key to introduce themselves into the group and remove the prior, legitimate member in a single Commit. Without resync, this can still be done, but requires two operations, the external Commit to join and a second Commit to remove the old appearance. Applications for whom this distinction is salient can choose to disallow external commits that contain a Remove, or to allow such resync commits only if they contain a "reinit" PSK proposal that demonstrates the joining member's presence in a prior epoch of the group. With the latter approach, the attacker would need to compromise the PSK as well as the signing key, but the application will need to ensure that continuing, non-resynchronizing members have the required PSK.

13.4.3.3. Ratchet Tree Extension

By default, a GroupInfo message only provides the joiner with a hash of the group's ratchet tree. In order to process or generate handshake messages, the joiner will need to get a copy of the ratchet tree from some other source. (For example, the DS might provide a cached copy.) The inclusion of the tree hash in the GroupInfo message means that the source of the ratchet tree need not be trusted to maintain the integrity of tree.

In cases where the application does not wish to provide such an external source, the whole public state of the ratchet tree can be provided in an extension of type ratchet_tree, containing a ratchet_tree object of the following form:

```c
struct {
    NodeType node_type;
    select (Node.node_type) {
        case leaf:   LeafNode leaf_node;
        case parent: ParentNode parent_node;
    };
} Node;

optional<Node> ratchet_tree<V>;
```

Each entry in the ratchet_tree vector provides the value for a node in the tree, or the null optional for a blank node.

The nodes are listed in the order specified by a left-to-right in-order traversal of the ratchet tree. Each node is listed between its left subtree and its right subtree. (This is the same ordering as specified for the array-based trees outlined in Appendix C.)
If the tree has $2^d$ leaves, then it has $2^{(d+1)} - 1$ nodes. The
ratchet_tree vector logically has this number of entries, but the
sender SHOULD NOT include blank nodes after the last non-blank node.
If a receiver encounters a vector whose length $L$ is not of the form
$2^{(d+1)} - 1$, then the receiver MUST extend it to the right with blank
values until it has such a length, adding the minimum number of blank
values possible. (Obviously, this may be done "virtually", by
synthesizing blank nodes when required, as opposed to actually
changing the structure in memory.)

The leaves of the tree are stored in even-numbered entries in the
array (the leaf with index $L$ in array position $2*L$). The root node
of the tree is at position $2^d - 1$ of the array. Intermediate parent
nodes can be identified by performing the same calculation to the
subarrays to the left and right of the root, following something like
the following algorithm:

```python
# Assuming a class Node that has left and right members
def subtree_root(nodes):
    # If there is only one node in the array return it
    if len(nodes) == 1:
        return Node(nodes[0])
    # Otherwise, the length of the array MUST be odd
    if len(nodes) % 2 == 0:
        raise Exception("Malformed node array ", len(nodes))
    # Identify the root of the subtree
    d = 0
    while (2**(d+1)) < len(nodes):
        d += 1
    R = 2**d - 1
    root = Node(nodes[R])
    root.left = subtree_root(nodes[:R])
    root.right = subtree_root(nodes[(R+1):])
    return root
```

(Note that this is the same ordering of nodes as in the array-based
tree representation described in Appendix C. The algorithms in that
section may be used to simplify decoding this extension into other
representations.)

For example, the following tree with six non-blank leaves would be
represented as an array of eleven elements, [A, W, B, X, C, _, D, Y, E, Z, F]. The above decoding procedure would identify the subtree
roots as follows (using $R$ to represent a subtree root):
The presence of a ratchet_tree extension in a GroupInfo message does not result in any changes to the GroupContext extensions for the group. The ratchet tree provided is simply stored by the client and used for MLS operations.

If this extension is not provided in a Welcome message, then the client will need to fetch the ratchet tree over some other channel before it can generate or process Commit messages. Applications should ensure that this out-of-band channel is provided with security protections equivalent to the protections that are afforded to Proposal and Commit messages. For example, an application that encrypts Proposal and Commit messages might distribute ratchet trees encrypted using a key exchanged over the MLS channel.

Regardless of how the client obtains the tree, the client MUST verify that the root hash of the ratchet tree matches the tree_hash of the GroupContext before using the tree for MLS operations.

Extensibility

The base MLS protocol can be extended in a few ways. New ciphersuites can be added to enable the use of new cryptographic algorithms. New types of proposals can be used to perform new actions within an epoch. Extension fields can be used to add additional information to the protocol. In this section, we discuss some constraints on these extensibility mechanisms that are necessary to ensure broad interoperability.
13.5. Ciphersuites

As discussed in Section 6.1, MLS allows the participants in a group to negotiate the cryptographic algorithms used within the group. This extensibility is important for maintaining the security of the protocol over time [RFC7696]. It also creates a risk of interoperability failure due to clients not supporting a common ciphersuite.

The ciphersuite registry defined in Section 17.1 attempts to strike a balance on this point. On the one hand, the base policy for the registry is Specification Required, a fairly low bar designed to avoid the need for standards work in cases where different ciphers are needed for niche applications. There is a higher bar (Standards Action) for ciphers to set the Recommended field in the registry. This higher bar is there in part to ensure that the interoperability implications of new ciphersuites are considered.

MLS ciphersuites are defined independent of MLS versions, so that in principle the same ciphersuite can be used across versions. Standards work defining new versions of MLS should consider whether it is desirable for the new version to be compatible with existing ciphersuites, or whether the new version should rule out some ciphersuites. For example, a new version could follow the example of HTTP/2, which restricted the set of allowed TLS ciphers (see Section 9.2.2 of [RFC7540]).

13.6. Proposals

Commit messages do not have an extension field because the set of proposals is extensible. As discussed in Section 13.4, Proposals with a non-default proposal type MUST NOT be included in a commit unless the proposal type is supported by all the members of the group that will process the Commit.

13.7. Credential Extensibility

In order to ensure that MLS provides meaningful authentication it is important that each member is able to authenticate some identity information for each other member. Identity information is encoded in Credentials, so this property is provided by ensuring that members use compatible credential types.

The types of credential that may be used in a group is restricted to what all members of the group support, as specified by the capabilities field of each LeafNode in the ratchet tree. An application can introduce new credential types by choosing an unallocated identifier from the registry in Section 17.4 and
indicating support for the credential type in published LeafNodes, whether in Update proposals to existing groups or KeyPackages that are added to new groups. Once all members in a group indicate support for the credential type, members can start using LeafNodes with the new credential. Application may enforce that certain credential types always remain supported by adding a required_capabilities extension to the group’s GroupContext, which would prevent any member from being added to the group that doesn’t support them.

In future extensions to MLS, it may be useful to allow a member to present more than one credential. For example, such credentials might present different attributes attested by different authorities. To be consistent with the general principle stated at the beginning of this section, such an extension would need to ensure that each member can authenticate some identity for each other member. For each pair of members (Alice, Bob), Alice would need to present at least one credential of a type that Bob supports.

13.8. Extensions

This protocol includes a mechanism for negotiating extension parameters similar to the one in TLS [RFC8446]. In TLS, extension negotiation is one-to-one: The client offers extensions in its ClientHello message, and the server expresses its choices for the session with extensions in its ServerHello and EncryptedExtensions messages. In MLS, extensions appear in the following places:

- In KeyPackages, to describe additional information related to the client
- In LeafNodes, to describe additional information about the client or its participation in the group (once in the ratchet tree)
- In the GroupInfo, to tell new members of a group what parameters are being used by the group, and to provide any additional details required to join the group
- In the GroupContext object, to ensure that all members of the group have the same view of the parameters in use
In other words, an application can use GroupContext extensions to ensure that all members of the group agree on a set of parameters. Clients indicate their support for parameters in the capabilities field of their LeafNode. New members of a group are informed of the group’s GroupContext extensions via the extensions field in the group_context field of the GroupInfo object. The extensions field in a GroupInfo object (outside of the group_context field) can be used to provide additional parameters to new joiners that are used to join the group.

This extension mechanism is designed to allow for the secure and forward-compatible negotiation of extensions. For this to work, implementations MUST correctly handle extensible fields:

* A client that posts a KeyPackage MUST support all parameters advertised in it. Otherwise, another client might fail to interoperate by selecting one of those parameters.

* A client initiating a group MUST ignore all unrecognized ciphersuites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients.

* Any field containing a list of extensions MUST NOT have more than one extension of any given type.

* A client adding a new member to a group MUST verify that the LeafNode for the new member is compatible with the group’s extensions. The capabilities field MUST indicate support for each extension in the GroupContext.

* If any extension in a GroupInfo message is unrecognized (i.e., not contained in the capabilities of the corresponding KeyPackage), then the client MUST reject the Welcome message and not join the group.

Note that the latter two requirements mean that all MLS extensions are mandatory, in the sense that an extension in use by the group MUST be supported by all members of the group.

The parameters of a group may be changed by sending a GroupContextExtensions proposal to enable additional extensions, or by reinitializing the group as described in Section 12.2.
14. Sequencing of State Changes

Each Commit message is premised on a given starting state, indicated by the epoch field of the enclosing MLSContent. If the changes implied by a Commit message are made starting from a different state, the results will be incorrect.

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

Applications MUST have an established way to resolve conflicting Commit messages for the same epoch. They can do this either by preventing conflicting messages from occurring in the first place, or by developing rules for deciding which Commit out of several sent in an epoch will be canonical. The approach chosen MUST minimize the amount of time that forked or previous group states are kept in memory, and promptly delete them once they’re no longer necessary to ensure forward secrecy.

The generation of Commit messages MUST NOT modify a client’s state, since the client doesn’t know at that time whether the changes implied by the Commit message will conflict with another Commit or not. Similarly, the Welcome message corresponding to a Commit MUST NOT be delivered to a new joiner until it’s clear that the Commit has been accepted.

Regardless of how messages are kept in sequence, there is a risk that in a sufficiently busy group, a given member may never be able to send a Commit message because they always lose to other members. The degree to which this is a practical problem will depend on the dynamics of the application.

15. Application Messages

The primary purpose of handshake messages are to provide an authenticated group key exchange to clients. In order to protect application messages sent among the members of a group, the encryption_secret provided by the key schedule is used to derive a sequence of nonces and keys for message encryption. Every epoch moves the key schedule forward which triggers the creation of a new secret tree, as described in Section 10, along with a new set of symmetric ratchets of nonces and keys for each member.
Each client maintains their own local copy of the 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.

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. The sender identifier and content type is used to identify which symmetric ratchet to use from the secret tree. The generation counter determines how far into the ratchet to iterate in order to produce the required nonce and key for encryption or decryption.

15.1. Padding

Application messages MAY 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?", then the answer "Wednesday" could be leaked to an adversary solely by the ciphertext length.

The length of the padding field in MLSCiphertextContent can be chosen at the time of message encryption by the sender. Senders may use padding to reduce the ability of attackers outside the group to infer the size of the encrypted content.

15.2. Restrictions

During each epoch, senders MUST NOT encrypt more data than permitted by the security bounds of the AEAD scheme used [I-D.irtf-cfrg-aead-limits].

Note that each change to the group through a handshake message will also set a new encryption_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.
15.3. Delayed and Reordered Application messages

Since each application message contains the group identifier, the epoch, and a generation counter, a client can receive messages out of order. When messages are received out of order, the client moves the sender ratchet forward to match the received generation counter. Any unused nonce and key pairs from the ratchet are potentially stored so that they can be used to decrypt the messages which were delayed or reordered.

Applications SHOULD define a policy on how long to keep unused nonce and key pairs for a sender, and the maximum number to keep. This is in addition to ensuring that these nonce and key pairs are promptly deleted when the epoch ends. Applications SHOULD also define a policy limiting the maximum number of steps that clients will move the ratchet forward in response to a new message. Messages received with a generation counter that’s too much higher than the last message received would then be rejected. This avoids causing a denial-of-service attack by requiring the recipient to perform an excessive number of key derivations. For example, a malicious group member could send a message with generation = 0xffffffff at the beginning of a new epoch, forcing recipients to perform billions of key derivations.

16. Security Considerations

The security goals of MLS are described in [I-D.ietf-mls-architecture]. We describe here how the protocol achieves its goals at a high level, though a complete security analysis is outside of the scope of this document.


Group secrets are partly derived from the output of a ratchet tree. Ratchet trees work by assigning each member of the group to a leaf in the tree and maintaining the following property: the private key of a node in the tree is known only to members of the group that are assigned a leaf in the node’s subtree. This is called the _ratchet tree invariant_ and it makes it possible to encrypt to all group members except one, with a number of ciphertexts that’s logarithmic in the number of group members.

The ability to efficiently encrypt to all members except one allows members to be securely removed from a group. It also allows a member to rotate their keypair such that the old private key can no longer be used to decrypt new messages.
16.2. Authentication

The first form of authentication we provide is that group members can verify a message originated from one of the members of the group. For encrypted messages, this is guaranteed because messages are encrypted with an AEAD under a key derived from the group secrets. For plaintext messages, this is guaranteed by the use of a membership_tag which constitutes a MAC over the message, under a key derived from the group secrets.

The second form of authentication is that group members can verify a message originated from a particular member of the group. This is guaranteed by a digital signature on each message from the sender’s signature key.

The signature keys held by group members are critical to the security of MLS against active attacks. If a member’s signature key is compromised, then an attacker can create LeafNodes and KeyPackages impersonating the member; depending on the application, this can then allow the attacker to join the group with the compromised member’s identity. For example, if a group has enabled external parties to join via external commits, then an attacker that has compromised a member’s signature key could use an external commit to insert themselves into the group -- even using a "resync"-style external commit to replace the compromised member in the group.

Applications can mitigate the risks of signature key compromise using pre-shared keys. If a group requires joiners to know a PSK in addition to authenticating with a credential, then in order to mount an impersonation attack, the attacker would need to compromise the relevant PSK as well as the victim’s signature key. The cost of this mitigation is that the application needs some external arrangement that ensures that the legitimate members of the group to have the required PSKs.

16.3. Forward Secrecy and Post-Compromise Security

Post-compromise security is provided between epochs by members regularly updating their leaf key in the ratchet tree. Updating their leaf key prevents group secrets from continuing to be encrypted to previously compromised public keys.

Forward-secrecy between epochs is provided by deleting private keys from past versions of the ratchet tree, as this prevents old group secrets from being re-derived. Forward secrecy _within_ an epoch is provided by deleting message encryption keys once they’ve been used to encrypt or decrypt a message.
Post-compromise security is also provided for new groups by members regularly generating new KeyPackages and uploading them to the Delivery Service, such that compromised key material won’t be used when the member is added to a new group.

16.4. KeyPackage Reuse

KeyPackages are intended to be used only once. That is, once a KeyPackage has been used to introduce the corresponding client to a group, it SHOULD be deleted from the KeyPackage publication system. Reuse of KeyPackages can lead to replay attacks.

An application MAY allow for reuse of a "last resort" KeyPackage in order to prevent denial-of-service attacks. Since a KeyPackage is needed to add a client to a new group, an attacker could prevent a client being added to new groups by exhausting all available KeyPackages. To prevent such a denial-of-service attack, the KeyPackage publication system SHOULD rate-limit KeyPackage requests, especially if not authenticated.

16.5. Group Fragmentation by Malicious Insiders

It is possible for a malicious member of a group to "fragment" the group by crafting an invalid UpdatePath. Recall that an UpdatePath encrypts a sequence of path secrets to different subtrees of the group’s ratchet trees. These path secrets should be derived in a sequence as described in Section 8.4, but the UpdatePath syntax allows the sender to encrypt arbitrary, unrelated secrets. The syntax also does not guarantee that the encrypted path secret for a given node corresponds to the public key provided for that node.

Both of these types of corruption will cause processing of a Commit to fail for some members of the group. If the public key for a node does not match the path secret, then the members that decrypt that path secret will reject the commit based on this mismatch. If the path secret sequence is incorrect at some point, then members that can decrypt nodes before that point will compute a different public key for the mismatched node than the one in the UpdatePath, which also causes the Commit to fail. Applications SHOULD provide mechanisms for failed commits to be reported, so that group members who were not able to recognize the error themselves can reinitialize the group if necessary.

Even with such an error reporting mechanism in place, however, it is still possible for members to get locked out of the group by a malformed commit. Since malformed Commits can only be recognized by certain members of the group, in an asynchronous application, it may be the case that all members that could detect a fault in a Commit
are offline. In such a case, the Commit will be accepted by the group, and the resulting state possibly used as the basis for further Commits. When the affected members come back online, they will reject the first commit, and thus be unable to catch up with the group. These members will either need to add themselves back with an external Commit, or reinitialize the group from scratch.

Applications can address this risk by requiring certain members of the group to acknowledge successful processing of a Commit before the group regards the Commit as accepted. The minimum set of acknowledgements necessary to verify that a Commit is well-formed comprises an acknowledgement from one member per node in the UpdatePath, that is, one member from each subtree rooted in the copath node corresponding to the node in the UpdatePath.

17. IANA Considerations

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

* MLS Ciphersuites (Section 17.1)
* MLS Extension Types (Section 17.2)
* MLS Proposal Types (Section 17.3)
* MLS Credential Types (Section 17.4)

All of these registries should be under a heading of "Messaging Layer Security", and assignments are made via the Specification Required policy [RFC8126]. See Section 17.5 for additional information about the MLS Designated Experts (DEs).

RFC EDITOR: Please replace XXXX throughout with the RFC number assigned to this document

17.1. MLS Ciphersuites

A ciphersuite is a combination of a protocol version and the set of cryptographic algorithms that should be used.

Ciphersuite names follow the naming convention:

CipherSuite MLS_LVL_KEM_AEAD_HASH_SIG = VALUE;

Where VALUE is represented as a sixteen-bit integer:

uint16 CipherSuite;
### Table 5

<table>
<thead>
<tr>
<th>Component</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVL</td>
<td>The security level</td>
</tr>
<tr>
<td>KEM</td>
<td>The KEM algorithm used for HPKE in ratchet tree operations</td>
</tr>
<tr>
<td>AEAD</td>
<td>The AEAD algorithm used for HPKE and message protection</td>
</tr>
<tr>
<td>HASH</td>
<td>The hash algorithm used for HPKE and the MLS transcript hash</td>
</tr>
<tr>
<td>SIG</td>
<td>The Signature algorithm used for message authentication</td>
</tr>
</tbody>
</table>

The columns in the registry are as follows:

* **Value**: The numeric value of the ciphersuite
* **Name**: The name of the ciphersuite
* **Recommended**: Whether support for this ciphersuite is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.
* **Reference**: The document where this ciphersuite is defined

Initial contents:
<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>R</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>RESERVED</td>
<td>-</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0001</td>
<td>MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0002</td>
<td>MLS_128_DHKEMP256_AES128GCM_SHA256_P256</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0003</td>
<td>MLS_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0004</td>
<td>MLS_256_DHKEMX448_AES256GCM_SHA512_Ed448</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0005</td>
<td>MLS_256_DHKEMP521_AES256GCM_SHA512_P521</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0006</td>
<td>MLS_256_DHKEMX448_CHACHA20POLY1305_SHA512_P521</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0007</td>
<td>MLS_256_DHKEMP384_AES256GCM_SHA384_P384</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xff00</td>
<td>Reserved for Private Use</td>
<td>-</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xffff</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6

All of these ciphersuites use HMAC [RFC2104] as their MAC function, with different hashes per ciphersuite. The mapping of ciphersuites to HPKE primitives, HMAC hash functions, and TLS signature schemes is as follows [RFC9180] [RFC8446]:
Table 7

The hash used for the MLS transcript hash is the one referenced in the ciphersuite name. In the ciphersuites defined above, "SHA256", "SHA384", and "SHA512" refer to the SHA-256, SHA-384, and SHA-512 functions defined in [SHS].

It is advisable to keep the number of ciphersuites low to increase the chances clients can interoperate in a federated environment, therefore the ciphersuites only include modern, yet well-established algorithms. Depending on their requirements, clients can choose between two security levels (roughly 128-bit and 256-bit). Within the security levels clients can choose between faster X25519/X448 curves and FIPS 140-2 compliant curves for Diffie-Hellman key negotiations. Additionally clients that run predominantly on mobile processors can choose ChaCha20Poly1305 over AES-GCM for performance reasons. Since ChaCha20Poly1305 is not listed by FIPS 140-2 it is not paired with FIPS 140-2 compliant curves. The security level of symmetric encryption algorithms and hash functions is paired with the security level of the curves.

The mandatory-to-implement ciphersuite for MLS 1.0 is MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519 which uses Curve25519 for key exchange, AES-128-GCM for HPKE, HKDF over SHA2-256, and Ed25519 for signatures.

Values with the first byte 255 (decimal) are reserved for Private Use.
New ciphersuite values are assigned by IANA as described in Section 17.

17.2. MLS Extension Types

This registry lists identifiers for extensions to the MLS protocol. The extension type field is two bytes wide, so valid extension type values are in the range 0x0000 to 0xffff.

Template:

* Value: The numeric value of the extension type

* Name: The name of the extension type

* Message(s): The messages in which the extension may appear, drawn from the following list:
  - KP: KeyPackage objects
  - LN: LeafNode objects
  - GC: GroupContext objects
  - GI: GroupInfo objects

* Recommended: Whether support for this extension is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.

* Reference: The document where this extension is defined

Initial contents:
### Table 8

#### 17.3. MLS Proposal Types

This registry lists identifiers for types of proposals that can be made for changes to an MLS group. The extension type field is two bytes wide, so valid extension type values are in the range 0x0000 to 0xffff.

**Template:**

* **Value:** The numeric value of the proposal type
* **Name:** The name of the proposal type
* **Recommended:** Whether support for this extension is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.
* **Path Required:** Whether a Commit covering a proposal of this type is required to have its path field populated (see Section 13.4).
* **Reference:** The document where this extension is defined

---

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Message(s)</th>
<th>Recommended</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>RESERVED</td>
<td>N/A</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0001</td>
<td>application_id</td>
<td>LN</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0002</td>
<td>ratchet_tree</td>
<td>GI</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0003</td>
<td>required_capabilities</td>
<td>GC</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0004</td>
<td>external_pub</td>
<td>GI</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0005</td>
<td>external_senders</td>
<td>GC</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xff00</td>
<td>Reserved for Private Use</td>
<td>N/A</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 9

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Recommended</th>
<th>Path Required</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>RESERVED</td>
<td>N/A</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0001</td>
<td>add</td>
<td>Y</td>
<td>N</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0002</td>
<td>update</td>
<td>Y</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0003</td>
<td>remove</td>
<td>Y</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0004</td>
<td>psk</td>
<td>Y</td>
<td>N</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0005</td>
<td>reinit</td>
<td>Y</td>
<td>N</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0006</td>
<td>external_init</td>
<td>Y</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0007</td>
<td>group_context_extensions</td>
<td>Y</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xff00</td>
<td>Reserved for Private Use</td>
<td>N/A</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xffff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 17.4. MLS Credential Types

This registry lists identifiers for types of credentials that can be used for authentication in the MLS protocol. The credential type field is two bytes wide, so valid credential type values are in the range 0x0000 to 0xffff.

Template:

* Value: The numeric value of the credential type
* Name: The name of the credential type
* Recommended: Whether support for this credential is recommended by the IETF MLS WG. Valid values are "Y" and "N". The "Recommended" column is assigned a value of "N" unless explicitly requested, and adding a value with a "Recommended" value of "Y" requires Standards Action [RFC8126]. IESG Approval is REQUIRED for a Y->N transition.
* Reference: The document where this credential is defined
Initial contents:

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Recommended</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>RESERVED</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0001</td>
<td>basic</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0x0002</td>
<td>x509</td>
<td>Y</td>
<td>RFC XXXX</td>
</tr>
<tr>
<td>0xff00 - 0xffff</td>
<td>Reserved for Private Use</td>
<td>N/A</td>
<td>RFC XXXX</td>
</tr>
</tbody>
</table>

Table 10

17.5. MLS Designated Expert Pool

Specification Required [RFC8126] registry requests are registered after a three-week review period on the MLS DEs’ mailing list: mls-reg-review@ietf.org (mailto:mls-reg-review@ietf.org), on the advice of one or more of the MLS DEs. However, to allow for the allocation of values prior to publication, the MLS DEs may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the MLS DEs mailing list for review SHOULD use an appropriate subject (e.g., "Request to register value in MLS Bar registry").

Within the review period, the MLS DEs will either approve or deny the registration request, communicating this decision to the MLS DEs mailing list and IANA. Denials SHOULD include an explanation and, if applicable, suggestions as to how to make the request successful. Registration requests that are undetermined for a period longer than 21 days can be brought to the IESG’s attention for resolution using the iesg@ietf.org (mailto:iesg@ietf.org) mailing list.

Criteria that SHOULD be applied by the MLS DEs includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or useful only for a single application, and whether the registration description is clear. For example, the MLS DEs will apply the ciphersuite-related advisory found in Section 6.1.

IANA MUST only accept registry updates from the MLS DEs and SHOULD direct all requests for registration to the MLS DEs’ mailing list.
It is suggested that multiple MLS DEs be appointed who are able to represent the perspectives of different applications using this specification, in order to enable broadly informed review of registration decisions. In cases where a registration decision could be perceived as creating a conflict of interest for a particular MLS DE, that MLS DE SHOULD defer to the judgment of the other MLS DEs.

17.6. The "message/mls" MIME Type

This document registers the "message/mls" MIME media type in order to allow other protocols (e.g., HTTP [RFC7540]) to convey MLS messages.

Media type name: message

Media subtype name: mls

Required parameters: none

Optional parameters: version

version: The MLS protocol version expressed as a string <major>.<minor>. If omitted the version is "1.0", which corresponds to MLS ProtocolVersion mls10. If for some reason the version number in the MIME type parameter differs from the ProtocolVersion embedded in the protocol, the protocol takes precedence.

Encoding scheme: MLS messages are represented using the TLS presentation language [RFC8446]. Therefore MLS messages need to be treated as binary data.

Security considerations: MLS is an encrypted messaging layer designed to be transmitted over arbitrary lower layer protocols. The security considerations in this document (RFC XXXX) also apply.

18. References

18.1. Normative References

18.2. Informative References


Barnes, et al. Expires 12 January 2023 [Page 125]
Appendix A. Protocol Origins of Example Trees

Protocol operations in MLS give rise to specific forms of ratchet tree, typically affecting a whole direct path at once. In this section, we describe the protocol operations that could have given rise to the various example trees in this document.

To construct the tree in Figure 10:

* A creates a group with B, ..., G
* F sends an empty Commit, setting X, Y, W
* G removes C and D, blanking V, U, and setting Y, W
* B sends an empty Commit, setting T and W

To construct the tree in Figure 9:

* A creates a group with B, ..., H, as well as some members outside this subtree
* F sends an empty Commit, setting Y and its ancestors
* D removes B and C, with the following effects:
  - Blank the direct paths of B and C
- Set X, the top node, and any further nodes in the direct path of D

* Someone outside this subtree removes G, blanking the direct path of G

* A adds a new member at B with a partial Commit, adding B as unmerged at X

To construct the tree in Figure 11:

* A creates a group with B, C, D
* B sends a full Commit, setting X and Y
* D removes C, setting Z and Y
* B adds a new member at C with a full Commit
  - The Add proposal adds C as unmerged at Z and Y
  - The path in the Commit resets X and Y, clearing Y’s unmerged leaves

To construct the tree in Figure 15:

* A creates a group with B, ..., G
* A removes F in a full Commit, setting T, U, and W
* E sends an empty Commit, setting Y and W
* A adds a new member at F in a partial Commit, adding F as unmerged at Y and W

Appendix B. Evolution of Parent Hashes

To better understand how parent hashes are maintained, let’s look in detail at how they evolve in a small group. Consider the following sequence of operations:

1. A initializes a new group
2. A adds B to the group with a full Commit
3. B adds C and D to the group with a full Commit
4. C sends an empty Commit.
Then the parent hashes associated to the nodes will be updated as follows (where we use the shorthand ph for parent hash, th for tree hash, and osth for original sibling tree hash):

1. A adds B: set $X$
   
   * $A'.parent_hash = ph(X) = H(X, ph="", osth=th(B))$

2. B adds C, D: set $B', X', Y$
   
   * $X'.parent_hash = ph(Y) = H(Y, ph="", osth=th(Z)), where th(Z) covers (C, _, D)$
   
   * $B'.parent_hash = ph(X') = H(X', ph=X'.parent_hash, osth=th(A))$

3. C sends empty Commit: set $C', Z', Y'$
   
   * $Z'.parent_hash = ph(Y') = H(Y', ph="", osth=th(X')), where th(X') covers (A, X', B')$
   
   * $C'.parent_hash = ph(Z') = H(Z', ph=Z'.parent_hash, osth=th(D))$

When a new member joins, they will receive a tree that has the following parent hash values, and compute the indicated parent-hash validity relationships:
<table>
<thead>
<tr>
<th>Node</th>
<th>Parent hash value</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>H(X, ph=&quot;, osth=th(B))</td>
<td>No, B changed</td>
</tr>
<tr>
<td>B'</td>
<td>H(X', ph=X'.parent_hash, osth=th(A))</td>
<td>Yes</td>
</tr>
<tr>
<td>C'</td>
<td>H(Z', ph=Z'.parent_hash, osth=th(D))</td>
<td>Yes</td>
</tr>
<tr>
<td>D</td>
<td>(none, never sent an UpdatePath)</td>
<td>N/A</td>
</tr>
<tr>
<td>X'</td>
<td>H(Y, ph=&quot;, osth=th(Z))</td>
<td>No, Y and Z changed</td>
</tr>
<tr>
<td>Z'</td>
<td>H(Y', ph=&quot;, osth=th(X'))</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 11

In other words, the joiner will find the following path-hash links in the tree:

```
Y'  
|  
\--
   \   
A  B'  C'  D  
   /   
X'  Z'  
```

Since these chains collectively cover all non-blank parent nodes in the tree, the tree is parent-hash valid.

Note that this tree, though valid, contains invalid parent-hash links. If a client were checking parent hashes top-down from Y', for example, they would find that X' has an invalid parent hash relative to Y', but that Z' has valid parent hash. Likewise, if the client were checking bottom-up, they would find that the chain from B' ends in an invalid link from X' to Y'. These invalid links are the natural result of multiple clients having committed.

Note also the way the tree hash and the parent hash interact. The parent hash of node C' includes the tree hash of node D. The parent hash of node Z' includes the tree hash of X', which covers nodes A and B' (including the parent hash of B'). Although the tree hash and the parent hash depend on each other, the dependency relationships are structured so that there’s never a circular dependency.
In the particular case where a new member first receives the tree for a group (e.g., in a ratchet tree GroupInfo extension Section 13.4.3.3), the parent hashes will be expressed in the tree representation, but the tree hash need not be. Instead, the new member will recompute the tree hashes for all the nodes in the tree, verifying that this matches the tree hash in the GroupInfo object. Then, if the tree is valid, then the subtree hashes computed in this way will align with the inputs needed for parent hash validation (except where recomputation is needed to account for unmerged leaves).

Appendix C. Array-Based Trees

One benefit of using complete 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, the tree with 8 leaves has the following structure:

```
  X
 /|
/ X
| X
 X X X X X X X X
```

Node: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Leaf: 0 1 2 3 4 5 6 7

This allows us to compute relationships between tree nodes simply by manipulating indices, rather than having to maintain complicated structures in memory. The basic rule is that the high-order bits of parent and child nodes indices have the following relation (where x is an arbitrary bit string):

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

Since node relationships are implicit, the algorithms for adding and removing nodes at the right edge of the tree are quite simple. If there are N nodes in the array:
* Add: Append \( N + 1 \) blank values to the end of the array.

* Remove: Truncate the array to its first \((N-1) / 2\) entries.

The following python code demonstrates the tree computations necessary to use an array-based tree for MLS.

```python
# The exponent of the largest power of 2 less than x. 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 levels,
# 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):
    if n == 0:
        return 0
    else:
        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.
def left(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')
```

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

# The right child of an intermediate node.
def right(x):
    k = level(x)
    if k == 0:
        raise Exception('leaf node has no children')

    return x ^ (0x03 << (k - 1))

# The parent of a node.
def parent(x, n):
    if x == root(n):
        raise Exception('root node has no parent')

    k = level(x)
    b = (x >> (k + 1)) & 0x01
    return (x | (1 << k)) ^ (b << (k + 1))

# The other child of the node’s parent.
def sibling(x, n):
    p = parent(x, n)
    if x < p:
        return right(p)
    else:
        return left(p)

# The direct path of a node, ordered from leaf to root.
def direct_path(x, n):
    r = root(n)
    if x == r:
        return []

    d = []
    while x != r:
        x = parent(x, n)
        d.append(x)
    return d

# The copath of a node, ordered from leaf to root.
def copath(x, n):
    if x == root(n):
        return []

    d = direct_path(x, n)
    d.insert(0, x)
    d.pop()
    return [sibling(y, n) for y in d]
def common_ancestor_semantic(x, y, n):
    dx = set([x]) | set(direct_path(x, n))
    dy = set([y]) | set(direct_path(y, n))
    dxy = dx & dy
    if len(dxy) == 0:
        raise Exception('failed to find common ancestor')
    return min(dxy, key=level)

def common_ancestor_direct(x, y, _):
    # Handle cases where one is an ancestor of the other
    lx, ly = level(x)+1, level(y)+1
    if (lx <= ly) and (x>>ly == y>>ly):
        return y
    elif (ly <= lx) and (x>>lx == y>>lx):
        return x
    # Handle other cases
    xn, yn = x, y
    k = 0
    while xn != yn:
        xn, yn = xn >> 1, yn >> 1
        k += 1
    return (xn << k) + (1 << (k-1)) - 1

Appendix D. Link-Based Trees

An implementation may choose to store ratchet trees in a "link-based" representation, where each node stores references to its parents and/or children. (As opposed to the array-based representation suggested above, where these relationships are computed from relationships between nodes' indices in the array.) Such an implementation needs to update these links to maintain the balanced structure of the tree as the tree is extended to add new members, or truncated when members are removed.

The following code snippet shows how these algorithms could be implemented in Python.
class Node:
    def __init__(self, value, left=None, right=None):
        self.value = value    # Value of the node
        self.left = left      # Left child node
        self.right = right    # Right child node

@staticmethod
def blank_subtree(depth):
    if depth == 1:
        return Node(None)
    L = Node.blank_subtree(depth-1)
    R = Node.blank_subtree(depth-1)
    return Node(None, left=L, right=R)

def empty(self):
    L_empty = (self.left == None) or self.left.empty()
    R_empty = (self.left == None) or self.left.empty()
    return (self.value == None) and L_empty and R_empty

class Tree:
    def __init__(self):
        self.depth = 0    # Depth of the tree
        self.root = None  # Root node of the tree, initially empty

    # Add a blank subtree to the right
    def extend(self):
        if self.depth == 0:
            self.depth = 1
            self.root = Node(None)
        L = self.root
        R = Node.blank_subtree(self.depth)
        self.root = Node(None, left=self.root, right=R)
        self.depth += 1

    # Truncate the right subtree
    def truncate(self):
        if self.root == None or self.root.right == None:
            raise Exception("Cannot truncate a tree with 0 or 1 nodes")
        if not self.root.right.empty():
            raise Exception("Cannot truncate non-blank subtree")
        self.depth -= 1
        self.root = self.root.left
Contributors

Joel Alwen
Amazon
Email: alwenjo@amazon.com

Karthikeyan Bhargavan
Inria
Email: karthikeyan.bhargavan@inria.fr

Cas Cremers
CISPA
Email: cremers@cispa.de

Alan Duric
Wire
Email: alan@wire.com

Britta Hale
Naval Postgraduate School
Email: britta.hale@nps.edu

Srinivas Inguva
Twitter
Email: singuva@twitter.com

Konrad Kohbrok
Aalto University
Email: konrad.kohbrok@datashrine.de

Albert Kwon
MIT
Email: kwonal@mit.edu

Tom Leavy
Amazon
Email: tomleavy@amazon.com

Brendan McMillion