Messaging Layer Security
Internet-Draft
Intended status: Informational
Expires: 30 November 2026

R. Robert
K. Kohbrok
Phoenix R&D

29 May 2026

SlimMLS
draft-robert-mls-slim-00

Abstract

This document defines SlimMLS, an extension to the Messaging Layer Security (MLS) protocol for reducing wire and per-client storage overhead in groups that use ciphersuites with large public keys, ciphertexts, signatures, and credentials. SlimMLS replaces many large objects that appear in MLS authenticated or transcript-hashed structures with typed hash references. Clients resolve the referenced objects only when needed, using a per-message carrier, a local cache, or an application-specific retrieval channel. The extension is most useful when an independent Delivery Service can assist with retrieval and per-recipient delivery, although local caching and delayed fetching also help without such assistance. SlimMLS defines slim variants of KeyPackage, Commit, Welcome, GroupInfo, and message framing. When placing a signature outside an encrypted envelope would reveal the signer, SlimMLS keeps the signature inside the ciphertext.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-robert-mls-slim/.

Discussion of this document takes place on the Messaging Layer Security Working Group mailing list (mailto:mls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/mls/. Subscribe at https://www.ietf.org/mailman/listinfo/mls/.

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

Status of This Memo

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

This Internet-Draft will expire on 30 November 2026.

Copyright Notice

Table of Contents

1. Introduction

Post-quantum (PQ) signature and key encapsulation mechanism (KEM) primitives can have substantially larger keys, ciphertexts, and signatures than their classical counterparts. Large credentials can create similar pressure, including in deployments that do not otherwise use large ciphersuite objects. In an MLS group [RFC9420], these values appear as larger LeafNodes, parent nodes, Welcomes, Commits, GroupInfos, and messages. The increase is visible both on the wire and in client storage.

In MLS, many of these values are embedded in structures that are signed, covered by tree hashes or parent hashes, or fed into transcript hashes. A Delivery Service (DS) can already choose how to route messages, but it cannot remove or replace authenticated bytes without invalidating the object. This limits cache-aware delivery, server-assisted fanout, and operation with clients that hold only part of the group state.

SlimMLS addresses this by separating object identity from object delivery. The authenticated or transcript-hashed structure carries a typed hash reference, and the referenced object is delivered through a per-message carrier, a local cache, or an application-specific retrieval channel. Recipients verify each object by recomputing the reference before using it.

The largest benefits come from deployments where the DS is an independent service that can assist clients. In such deployments, the DS can omit objects a recipient already has, reduce update path ciphertexts to the subset each recipient needs, and supply GroupInfo separately from a SlimWelcome. Deployments without an assisting DS still benefit from smaller authenticated state, local caches of credentials and keys, and the ability to defer fetching large objects until they are needed.

The main protocol changes are:

* SlimKeyPackages replace KeyPackages and batch the signatures needed to authenticate multiple LeafNodes.

* SlimCommits carry commit content separately from update path delivery data, so the Delivery Service can deliver only the ciphertexts each recipient needs.

* SlimWelcomes adapt Welcome messages to slim KeyPackage references and server-assisted GroupInfo delivery.

* Slim message framing and SlimGroupInfo define when signatures are represented by references and when the raw signature must remain inside encrypted plaintext to avoid revealing the sender.

This pattern is not new in MLS. [RFC9420] already uses RefHash-based references such as KeyPackageRef and ProposalRef. SlimMLS generalizes the mechanism.

[[TODO: quantify wire and storage savings for representative PQ ciphersuites once the specification stabilizes.]]

2. Conventions and Definitions

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.

This document uses the TLS presentation language and notation of [RFC9420]. Familiarity with MLS [RFC9420] is assumed.

3. Overview

A SlimMLS group is an MLS group whose GroupContext carries the slim_mls extension (Section 12). In such a group:

* In structures derived by mechanical substitution, each HPKEPublicKey, SignaturePublicKey, Credential, HPKECiphertext, or signature is replaced with a hash reference of the corresponding type. Structures specified explicitly in this document define their own replacements and exceptions.

* The referenced large objects are retrieved per Section 6 if and when necessary.

4. Reference Computation

All new references are computed in the same style as KeyPackageRef in Section 5.2 of [RFC9420], i.e., using RefHash instantiated with the relevant ciphersuite hash function and a label unique to the referenced object type. Inputs are TLS-encoded per [RFC9420]. For group-bound structures, the relevant ciphersuite is the group's ciphersuite. For references contained in or computed over a SlimKeyPackage, the relevant ciphersuite is outer_key_package.cipher_suite of that SlimKeyPackage.

The following reference types are defined:

Referenced value	Reference type	RefHash label
HPKEPublicKey	HPKEPublicKeyRef	"MLS 1.0 SlimMLS HPKEPublicKey"
SignaturePublicKey	SignaturePublicKeyRef	"MLS 1.0 SlimMLS SignaturePublicKey"
Credential	CredentialRef	"MLS 1.0 SlimMLS Credential"
HPKECiphertext	HPKECiphertextRef	"MLS 1.0 SlimMLS HPKECiphertext"
Signature	SignatureRef	"MLS 1.0 SlimMLS Signature"
SlimKeyPackage	SlimKeyPackageRef	"MLS 1.0 SlimMLS KeyPackage Reference"

Table 1

Each reference is opaque<V> where V is the output length of the ciphersuite hash function.

For a SlimKeyPackageRef, the value input is the TLS-encoded SlimKeyPackage. SlimKeyPackageRef is used to identify the recipient of a SlimWelcome; it is not a large-object reference and has no corresponding LargeObjectCarrier entry.

SignatureRef is used for signature fields that appear in slim structures unless this document requires the raw signature to remain inside an encrypted envelope. It is also used for detached signatures referenced by slim structures, such as the SlimKeyPackage batch signature (Section 7).

Some signature fields use SignatureOrRef, whose variant is constrained by the wire envelope that carries the structure:

```
enum {
  reserved(0),
  signature(1),
  signature_ref(2),
  (255)
} SignatureOrRefType;

struct {
  SignatureOrRefType signature_type;
  select (SignatureOrRef.signature_type) {
    case signature:
      Signature signature;
    case signature_ref:
      SignatureRef signature_ref;
  };
} SignatureOrRef;
```

When signature_type = signature_ref, the raw signature bytes are resolved per Section 6. When signature_type = signature, the raw signature bytes are carried inline and MUST NOT be sent outside the encrypted envelope in a LargeObjectCarrier.

5. SlimMLS Structs

For every [RFC9420] struct not given an explicit replacement in this document that embeds an HPKEPublicKey, SignaturePublicKey, Credential, HPKECiphertext, or signature, SlimMLS defines a corresponding "Slim*" struct that is identical to the original except that each occurrence of a large object is replaced by the reference type from Section 4 and that each occurrence of a struct for which there exists a Slim equivalent is replaced by that equivalent. Tree

hashes and parent hashes are likewise computed over the slim encodings, so the reference values stand in for the corresponding large objects in these computations.

The following signatures use SignatureOrRef because their correct presentation depends on whether the signature is carried inside an encrypted envelope:

* an encrypted GroupInfo carried in a SlimWelcome, whose signature remains inline inside the encrypted GroupInfo plaintext, and

* a SlimPrivateMessage framing signature, which remains inline inside the encrypted SlimPrivateMessageContent.

When either signature is carried inside an encrypted envelope, it MUST use the signature variant.

Welcome, KeyPackage, and Commit are not derived by mechanical substitution. They are replaced by the SlimWelcome struct (Section 8), the SlimKeyPackage struct (Section 7), and the SlimCommit struct (Section 9) respectively. As a consequence, the UpdatePath struct does not appear in a SlimMLS group. Its contents are split between SlimCommit (which carries the committer’s SlimLeafNode) and SlimUpdatePath (which carries the path nodes).

In a SlimMLS group, the slim struct is sent on the wire wherever [RFC9420] would specify the original struct. Validation rules of [RFC9420] apply in the same way. References are only replaced by the corresponding large objects if functionally necessary (e.g. to verify a signature, encrypt a ciphertext, or retrieve referenced signature bytes).

The [RFC9420] structs affected, the large objects they embed, and the SlimMLS replacements are listed below. Structs whose only embedded large objects appear via a nested struct (e.g., proposals that carry a LeafNode or KeyPackage) inherit slim variants implicitly via the slim nested struct.

RFC 9420 struct	Embedded large object(s)	SlimMLS replacement(s)
LeafNode	HPKEPublicKey, SignaturePublicKey, Credential, signature	HPKEPublicKeyRef, SignaturePublicKeyRef, CredentialRef, SignatureRef
ParentNode	HPKEPublicKey	HPKEPublicKeyRef
ParentHashInput	HPKEPublicKey	HPKEPublicKeyRef
UpdatePathNode	HPKEPublicKey, HPKECiphertext (vector)	HPKEPublicKeyRef, HPKECiphertextRef (vector)
Signature-bearing structs except SignatureOrRef cases	signature	SignatureRef
Add proposal	KeyPackage	SlimKeyPackage
Update proposal	LeafNode	SlimLeafNode
ratchet_tree extension	LeafNode, ParentNode	slim_ratchet_tree extension with SlimLeafNode, SlimParentNode

Table 2

The exact TLS presentation of each remaining slim struct is obtained by mechanical substitution against [RFC9420] and is not repeated here.

[[TODO: provide explicit TLS presentations for each slim variant in a later revision.]]

5.1. SlimLeafNode

SlimLeafNode replaces LeafNode. SlimLeafNodeTBS replaces the LeafNodeTBS input defined in Section 7.2 of [RFC9420].

SlimMLS uses the app_data_dictionary extension specified in Section 7.2.1 of [I-D.ietf-mls-extensions] to carry SlimMLS-specific components in a SlimLeafNode's extensions field.

```
struct {
  opaque sibling_hash<V>;
} SlimKeyPackageMerkleProofNode;

struct {
  uint32                          leaf_index;
  uint32                          tree_size;
  SlimKeyPackageMerkleProofNode  path<V>;
} SlimKeyPackageMerkleProof;

struct {
  HPKEPublicKeyRef      encryption_key_ref;
  SignaturePublicKeyRef signature_key_ref;
  CredentialRef         credential_ref;
  Capabilities          capabilities;

  LeafNodeSource leaf_node_source;
  select (SlimLeafNodeTBS.leaf_node_source) {
    case key_package:
      Lifetime lifetime;

    case update:
      struct{};

    case commit:
      opaque parent_hash<V>;
  };

  Extension extensions<V>;

  select (SlimLeafNodeTBS.leaf_node_source) {
    case key_package:
      struct{};

    case update:
      opaque group_id<V>;
      uint32 leaf_index;

    case commit:
      opaque group_id<V>;
      uint32 leaf_index;
  };
} SlimLeafNodeTBS;
```

```
struct {
  HPKEPublicKeyRef      encryption_key_ref;
  SignaturePublicKeyRef signature_key_ref;
  CredentialRef         credential_ref;
  Capabilities          capabilities;

  LeafNodeSource leaf_node_source;
  select (SlimLeafNode.leaf_node_source) {
    case key_package:
      Lifetime lifetime;

    case update:
      struct{};

    case commit:
      opaque parent_hash<V>;
  };

  Extension extensions<V>;

  /*
    For key_package leaves, this is a SignatureRef to the batch signature.
    For update and commit leaves, this is a SignatureRef to the normal
    SlimLeafNode signature over SlimLeafNodeTBS.
  */
  SignatureRef signature;

  select (SlimLeafNode.leaf_node_source) {
    case key_package:
      SlimKeyPackageMerkleProof proof;

    case update:
      struct{};

    case commit:
      struct{};
  };
} SlimLeafNode;
```

The proof field is present only when leaf_node_source = key_package. It is not part of SlimLeafNodeTBS to avoid a circulare dependency. Update and commit leaf nodes do not carry this field.

For leaf_node_source = key_package, the signature field references the SlimKeyPackage batch signature and validation MUST verify both this signature and the embedded proof as described in Section 7.1. This allows clients to validate a key-package SlimLeafNode even when it appears in a ratchet tree without the enclosing SlimKeyPackage.

For leaf nodes whose source is update or commit, the signature field is the normal SlimLeafNode signature over SlimLeafNodeTBS, verified as in Section 7.3 of [RFC9420] after applying the slim substitutions in this document.

5.2. SlimParentNode and SlimParentHashInput

The following slim tree structures are used in tree-hash and parent-hash computations:

```
struct {
  HPKEPublicKeyRef encryption_key_ref;
  opaque           parent_hash<V>;
  uint32           unmerged_leaves<V>;
} SlimParentNode;

struct {
  HPKEPublicKeyRef encryption_key_ref;
  opaque           parent_hash<V>;
  opaque           original_sibling_tree_hash<V>;
} SlimParentHashInput;
```

SlimParentNode replaces ParentNode. SlimParentHashInput replaces the ParentHashInput defined in Section 7.9 of [RFC9420].

5.3. SlimGroupInfo

SlimGroupInfo is the slim replacement for the [RFC9420] GroupInfo struct. It uses the same fields and verification rules as GroupInfo, except that GroupInfo extensions use SlimMLS replacements and the signature is represented as SignatureOrRef.

```
struct {
  GroupContext group_context;
  Extension extensions<V>;
  MAC confirmation_tag;
  uint32 signer;
} SlimGroupInfoTBS;

struct {
  GroupContext group_context;
  Extension extensions<V>;
  MAC confirmation_tag;
  uint32 signer;

  /*
    SignWithLabel(., "GroupInfoTBS", SlimGroupInfoTBS)
  */
  SignatureOrRef signature;
} SlimGroupInfo;
```

The signature is computed and verified as in [RFC9420], except that the to-be-signed content is SlimGroupInfoTBS. A standalone SlimGroupInfo, or a SlimGroupInfo carried in WelcomeGroupInfo.info_type = plaintext, MUST use signature_type = signature_ref. A SlimGroupInfo encrypted into WelcomeGroupInfo.info_type = encrypted MUST use signature_type = signature so that the raw signature is protected by the SlimWelcome encryption.

6. Large Object Retrieval

References to large objects in SlimMLS structures, in their associated companion structures (e.g., SlimUpdatePath, Section 9), and in GroupInfo extensions defined by this document MUST be resolved to the corresponding large objects when necessary for MLS operations or validation checks.

SlimMLS defines three retrieval channels:

* The LargeObjectCarrier (Section 6.1), specified in this document.

* A client-local cache of previously resolved objects.

* An application-specific fetch mechanism.

Applications can decide how or if they use one or more retrieval channels.

On receipt of a SlimMLS structure, a client:

1. For every large-object *Ref it needs to process the structure or any associated companion structure, locates a candidate object through any of the three channels above.

2. Computes the reference of each candidate object under the appropriate label (Section 4) and verifies that it equals the *Ref being resolved. A candidate object whose reference does not match MUST NOT be used for any cryptographic operation.

3. If a required *Ref cannot be resolved through any channel, the client MUST either request the missing object through an application-specific mechanism, for example from the DS, or drop the message.

6.1. Large Object Carrier

A client sending a SlimMLS struct over the wire MAY also send a LargeObjectCarrier struct that contains a subset of the large objects referenced by the SlimMLS struct or by an associated structure such as SlimUpdatePath (Section 9).

This document does not define a single MLSMessage wrapper for LargeObjectCarrier. When a carrier is sent on the wire, its encoding, multiplexing, and association with the SlimMLS message are provided by the application or DS protocol using SlimMLS.

For each large-object reference type that appears in a slim structure or an associated structure, the LargeObjectCarrier contains a vector of the corresponding large objects:

```
struct {
  HPKEPublicKey      hpke_public_keys<V>;
  SignaturePublicKey signature_public_keys<V>;
  Credential         credentials<V>;
  HPKECiphertext     hpke_ciphertexts<V>;
  Signature          signatures<V>;
} LargeObjectCarrier;
```

The carrier is NOT part of the signed structure. The DS MAY add, remove, reorder, or substitute entries on a per-recipient basis, e.g., to omit objects the recipient already has cached, or to distribute SlimUpdatePath ciphertexts (see Section 9). The LargeObjectCarrier is optional in the SlimMLS wire protocol. The DS MAY reject a message based on a missing LargeObjectCarrier, or on a LargeObjectCarrier that is missing the large objects that clients will need to process a message, if its local deployment policy requires senders to provide those objects proactively.

7. SlimKeyPackage

SlimKeyPackage applies the slim reference replacement to KeyPackage and amortizes KeyPackage authentication across a batch of KeyPackages uploaded by the same client. The signature around the KeyPackage is omitted, and the per-KeyPackage LeafNode signature is replaced by a detached batch signature over the root of a Merkle tree. The leaves of this tree are the SlimLeafNodeTBS values of the SlimKeyPackages in the batch.

In a SlimMLS group, an [RFC9420] KeyPackage MUST NOT be used to add a new member: Add proposals MUST carry a SlimKeyPackage, and standalone KeyPackage publication MUST use the mls_slim_key_package wire format (Section 11). A recipient MUST reject any Add proposal or wire-format message carrying an [RFC9420] KeyPackage in the context of a group with the slim_mls extension.

SlimLeafNodeTBS is the unsigned part of a SlimLeafNode. It carries neither the signature field nor the key-package Merkle proof.

A SlimKeyPackage is partitioned into an OuterSlimKeyPackage (the fields of a KeyPackage other than the LeafNode and signature) and a SlimLeafNode. The SlimLeafNode carries the SignatureRef of the detached batch signature and its Merkle inclusion proof:

```
struct {
  ProtocolVersion  version;
  CipherSuite      cipher_suite;
  HPKEPublicKeyRef init_key_ref;
  Extension        extensions<V>;
} OuterSlimKeyPackage;

struct {
  ProtocolVersion       version;
  CipherSuite           cipher_suite;
  SignaturePublicKeyRef signature_key_ref;
  uint32                tree_size;
  opaque                merkle_root<V>;
} SlimKeyPackageBatchTBS;

struct {
  OuterSlimKeyPackage              outer_key_package;
  SlimLeafNode                     leaf_node;
} SlimKeyPackage;
```

A SlimKeyPackage carries no outer signature and no per-package LeafNode signature. Authenticity of outer_key_package is provided by an OuterKeyPackageHash component (Section 7.2) placed in the

SlimLeafNode’s app_data_dictionary extension. Authenticity of the SlimLeafNodeTBS, including this component, is provided by the batch signature referenced by leaf_node.signature and the Merkle inclusion proof in leaf_node.proof.

The HPKEPublicKey corresponding to init_key_ref, together with any large objects referenced by the SlimLeafNode, is resolved per Section 6.

7.1. KeyPackage Batch Merkle Tree

The Merkle tree for a SlimKeyPackage batch is computed under the hash function of the SlimKeyPackage ciphersuite. A batch MUST contain at least one leaf. All SlimKeyPackages in a batch MUST have the same version, cipher_suite, and signature_key_ref.

The leaf hash for a SlimLeafNodeTBS is computed over the TLS-encoded SlimLeafNodeTBS:

```
HashWithLabel("SlimKeyPackageLeaf", SlimLeafNodeTBS)
```

The parent hash for two child hashes is computed over their TLS-encoded ordered pair:

```
struct {
  opaque left<V>;
  opaque right<V>;
} SlimKeyPackageMerkleParentInput;

HashWithLabel("SlimKeyPackageNode", SlimKeyPackageMerkleParentInput)
```

The tree is built bottom-up from the ordered list of leaf hashes. At each level, adjacent nodes are paired from left to right. If the final node at a level has no sibling, it is promoted unchanged to the next level. The single node remaining after this process is the Merkle root.

The path entries in SlimKeyPackageMerkleProof are ordered from the leaf level toward the root. To verify a proof, a recipient starts with the leaf hash of leaf_node_tbs, leaf_index, and tree_size. At each level, if the current level width is odd and the current index is width - 1, the current hash is promoted unchanged and no path entry is consumed. Otherwise, the next path entry is consumed as the sibling hash and the parent hash is computed with the sibling on the left when the current index is odd, and on the right when the current index is even. The index is then divided by two, rounding down, and the level width is divided by two, rounding up. A proof is valid only if tree_size is nonzero, leaf_index < tree_size, all path entries are consumed, and the final computed hash is the Merkle root.

The batch signature is computed over:

SignWithLabel(., "SlimKeyPackageBatchTBS", SlimKeyPackageBatchTBS)

where version and cipher_suite are taken from outer_key_package, signature_key_ref is taken from leaf_node_tbs, tree_size is taken from the proof in leaf_node, and merkle_root is the Merkle root computed as above.

7.2. OuterKeyPackageHash component

```
struct {
  opaque outer_key_package_hash<V>;
} OuterKeyPackageHash;
```

outer_key_package_hash is the hash, under the SlimKeyPackage’s ciphersuite hash function, of the TLS-encoded outer_key_package of the SlimKeyPackage in which the SlimLeafNode appears.

An OuterKeyPackageHash is valid only if the SlimLeafNode’s leaf_node_source is key_package and outer_key_package_hash equals the hash of the enclosing SlimKeyPackage’s outer_key_package.

The app_data_dictionary extension MUST contain exactly one OuterKeyPackageHash component under component identifier TBD. A missing, malformed, or duplicated OuterKeyPackageHash component makes the enclosing SlimKeyPackage invalid.

7.3. Creation and Processing

A sender constructs a SlimKeyPackage as follows:

1. Construct an OuterSlimKeyPackage with the desired version, cipher_suite, HPKEPublicKeyRef for the init key, and extensions.

2. Construct a SlimLeafNodeTBS with leaf_node_source = key_package. Add an app_data_dictionary extension containing an OuterKeyPackageHash whose value is the hash of the OuterSlimKeyPackage from step 1.

3. Construct all other SlimLeafNodeTBS values in the batch in the same way.

4. Build the KeyPackage batch Merkle tree over the SlimLeafNodeTBS values.

5. Construct a SlimKeyPackageBatchTBS for the root, sign it with the signature private key corresponding to signature_key_ref, and compute the SignatureRef over the raw signature bytes.

6. Emit each SlimKeyPackage with the common SignatureRef in leaf_node.signature and the leaf's inclusion proof in leaf_node.proof, optionally accompanied by a LargeObjectCarrier holding the referenced HPKEPublicKey and the large objects referenced by the SlimLeafNode and the batch signature bytes.

A recipient processes a SlimKeyPackage like a KeyPackage with the following exceptions:

* There is no outer signature to verify.

* There is no per-package LeafNode signature to verify.

* The SlimLeafNode MUST contain an app_data_dictionary extension with a valid OuterKeyPackageHash component.

* The SlimLeafNode MUST have leaf_node_source = key_package and therefore carry a proof field.

* The recipient resolves the batch signature bytes using leaf_node.signature, computes SignatureRef over those bytes, and verifies that it equals leaf_node.signature.

* The recipient verifies the Merkle inclusion proof in leaf_node.proof, reconstructs the SlimKeyPackageBatchTBS, resolves the SignaturePublicKey corresponding to signature_key_ref, and verifies the raw batch signature bytes identified by leaf_node.signature using VerifyWithLabel label "SlimKeyPackageBatchTBS".

* All large-object *Ref values are resolved per Section 6.

8. SlimWelcome

SlimWelcome makes two changes relative to the [RFC9420] Welcome:

1. The per-recipient EncryptedGroupSecrets is replaced by a SlimEncryptedGroupSecrets (Section 8.2), which permits the recipient to be identified either by SlimKeyPackageRef or by leaf index.

2. The encrypted_group_info field is replaced by an optional<WelcomeGroupInfo> (Section 8.3), where WelcomeGroupInfo is a tagged union over an encrypted form and a plaintext form.

With the exception of changes described in this section, SlimWelcomes are processed just like regular Welcome messages.

```
struct {
  CipherSuite                cipher_suite;
  SlimEncryptedGroupSecrets  secrets<V>;
  optional<WelcomeGroupInfo> group_info;
} SlimWelcome;
```

8.1. Sender and DS Behavior

The sender MUST construct SlimWelcome.secrets to contain one SlimEncryptedGroupSecrets for every recipient it intends to add, in the same order it would use under [RFC9420].

The DS MAY remove entries from SlimWelcome.secrets on a per-recipient basis, e.g., to deliver to each recipient only its own entry.

If the sender omits the group_info field (presence octet 0), the DS MUST populate it with a plaintext WelcomeGroupInfo before delivering the SlimWelcome to a recipient (Section 8.3). In this mode, the GroupInfo used by the sender to encrypt the SlimEncryptedGroupSecrets and the GroupInfo populated by the DS MUST be byte-for-byte identical SlimGroupInfo objects.

A DS that supplies large objects alongside a SlimWelcome can distinguish between basic-processing delivery and update-capable delivery. Basic-processing delivery contains the large objects needed for the recipient to validate the SlimWelcome, derive the epoch secrets, receive subsequent MLS messages, and send application messages. It can omit HPKE public keys that are only needed to construct a future Commit with an update path. Update-capable delivery additionally includes enough HPKE public keys for the recipient to construct an update path without a later fetch, such as the HPKE public keys for the recipient’s copath resolution in the

delivered tree. A recipient that has only basic-processing state MUST resolve the missing HPKEPublicKeyRefs before sending a Commit with an update path.

8.2. SlimEncryptedGroupSecrets

SlimEncryptedGroupSecrets extends the [RFC9420] EncryptedGroupSecrets to allow either of two recipient identifiers:

```
enum {
  reserved(0),
  slim_key_package_ref(1),
  leaf_node_index(2),
  (255)
} RecipientIdentifierType;

struct {
  RecipientIdentifierType recipient_type;
  select (SlimEncryptedGroupSecrets.recipient_type) {
    case slim_key_package_ref:
      SlimKeyPackageRef new_member;
    case leaf_node_index:
      uint32 leaf_node_index;
  };
  HPKECiphertext encrypted_group_secrets;
} SlimEncryptedGroupSecrets;
```

A recipient locates the entry intended for it by matching either its own SlimKeyPackageRef or its leaf in the group’s ratchet tree via the leaf index.

The leaf_node_index recipient type MUST be used only when the SlimEncryptedGroupSecrets is encrypted using a plaintext SlimGroupInfo as context and the delivered SlimWelcome carries WelcomeGroupInfo.info_type = plaintext. Otherwise, the sender MUST use slim_key_package_ref.

8.3. WelcomeGroupInfo

WelcomeGroupInfo is a tagged union over two presentations of the SlimGroupInfo: an encrypted form or a plaintext form:

```
enum {
  reserved(0),
  encrypted(1),
  plaintext(2),
  (255)
} WelcomeGroupInfoType;

struct {
  WelcomeGroupInfoType info_type;
  select (WelcomeGroupInfo.info_type) {
    case encrypted:
      opaque encrypted_group_info<V>;
    case plaintext:
      SlimGroupInfo group_info;
  };
} WelcomeGroupInfo;
```

The encrypted variant is encrypted under a key derived from the joiner secret, as in the encrypted_group_info field of the [RFC9420] Welcome. Its plaintext is a SlimGroupInfo whose SignatureOrRef uses signature_type = signature. A sender MUST NOT encrypt a SlimGroupInfo with signature_type = signature_ref into this field.

The plaintext variant carries a SlimGroupInfo in the clear. Its SignatureOrRef MUST use signature_type = signature_ref, which is resolved per Section 6 before signature verification.

The HPKE context used to encrypt and decrypt SlimEncryptedGroupSecrets.encrypted_group_secrets depends on the GroupInfo presentation. For the encrypted variant, the context is the encrypted_group_info value, as in [RFC9420]. For the plaintext variant, and for SlimWelcomes sent with group_info absent, the context is the TLS-encoded SlimGroupInfo.

The outer optional<WelcomeGroupInfo> in the SlimWelcome additionally allows the sender to omit the GroupInfo. This mode is intended for server-assisted deployments where the sender and recipient can obtain the exact GroupInfo by some channel other than the SlimWelcome.

A recipient that receives a SlimWelcome whose group_info field is absent MUST consider the SlimWelcome invalid.

9. SlimCommit

SlimCommit enables split delivery for SlimMLS Commits. The DS can deliver to each recipient only the HPKECiphertextRefs and HPKECiphertexts intended for that recipient. It also saves one signature when a commit contains a path but does not rotate the sender's signature key.

A SlimMLS-aware sender MUST use a SlimCommit in place of an MLS Commit in a group with the slim_mls extension.

A SlimCommit is carried in either a SlimPublicMessage or SlimPrivateMessage (Section 9.2). These message structures allow the unsigned SlimUpdatePath to be carried alongside the framed SlimCommit without being included in the transcript hash, the membership tag, or the framing signature.

A SlimCommit carries the normal [RFC9420] confirmation tag in its SlimFramedContentAuthData. When the single-signature construction of Section 9.4 applies, the authentication data contains the confirmation tag but omits the framing signature field. Otherwise, the authentication data contains the confirmation tag and a SignatureOrRef whose variant is determined by the message envelope.

```
struct {
  ProposalOrRef          proposals<V>;
  optional<SlimLeafNode> leaf_node;
} SlimCommit;
```

leaf_node, when present, is the committer's new SlimLeafNode.

9.1. SlimUpdatePath

The path is conveyed separately from the framed SlimCommit:

```
struct {
  HPKEPublicKeyRef  encryption_key_ref;
  HPKECiphertextRef encrypted_path_secret_refs<V>;
} SlimUpdatePathNode;

struct {
  SlimUpdatePathNode nodes<V>;
} SlimUpdatePath;
```

Each SlimUpdatePathNode carries an HPKEPublicKeyRef for the new ParentNode public key and a vector of HPKECiphertextRefs. In the sender-to-DS SlimUpdatePath, the encrypted_path_secret_refs vector has the same order and cardinality as the encrypted_path_secret

vector in the corresponding [RFC9420] UpdatePathNode. In a DS-to-recipient SlimUpdatePath, the DS MAY reduce each encrypted_path_secret_refs vector to the subset the recipient needs. The reduced path MUST contain enough HPKECiphertextRefs for the recipient to decrypt one path secret and derive the remaining path secrets it needs to process the Commit. The corresponding HPKECiphertexts, and any HPKEPublicKeys that are functionally needed, are resolved per Section 6.

The SlimUpdatePath is not signed. Its HPKEPublicKeyRefs are authenticated by the parent hash in the committer's SlimLeafNode, as in Section 7.9 of [RFC9420], with HPKEPublicKeyRef replacing HPKEPublicKey in the parent-hash computation. Its HPKECiphertextRefs are unauthenticated delivery objects: a recipient validates them by decrypting the referenced HPKECiphertext, deriving the path public keys, checking that the derived public keys hash to the authenticated HPKEPublicKeyRefs, and verifying the Commit confirmation tag.

9.2. Slim Framing

In a SlimMLS group, public and private message framing uses dedicated SlimMLS structures. They are the same as [RFC9420] PublicMessage and PrivateMessage, except that:

* the framed content carries a SlimCommit or SlimProposal where [RFC9420] carries a Commit or Proposal,

* the authentication data is a SlimFramedContentAuthData, and

* when the content is a SlimCommit, the unsigned SlimUpdatePath is carried alongside the framed content, outside the authenticated content.

SlimProposal is the slim variant of the [RFC9420] Proposal struct obtained by the mechanical-substitution rule of Section 5: an Add proposal carries a SlimKeyPackage and an Update proposal carries a SlimLeafNode. Other proposal variants are unchanged.

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

  ContentType content_type;
  select (SlimFramedContent.content_type) {
    case application:
      opaque application_data<V>;
```

```
      case proposal:
        SlimProposal proposal;
      case commit:
        SlimCommit commit;
    };
  } SlimFramedContent;

  struct {
    /*
      SignWithLabel(., "FramedContentTBS", SlimFramedContentTBS)
    */
    optional<SignatureOrRef> signature;

    select (SlimFramedContent.content_type) {
      case commit:
        /*
          MAC(confirmation_key,
              GroupContext.confirmed_transcript_hash)
        */
        MAC confirmation_tag;
      case application:
      case proposal:
        struct{};
    };
  } SlimFramedContentAuthData;

  struct {
    ProtocolVersion    version = mls10;
    WireFormat         wire_format;
    SlimFramedContent  content;
    select (SlimFramedContentTBS.content.sender.sender_type) {
      case member:
      case new_member_commit:
        GroupContext context;
      case external:
      case new_member_proposal:
        struct{};
    };
  } SlimFramedContentTBS;

  struct {
    WireFormat         wire_format;
    SlimFramedContent  content;
    SlimFramedContentAuthData auth;
  } SlimAuthenticatedContent;

  struct {
    SlimFramedContent         content;
```

```
    SlimFramedContentAuthData auth;
    select (SlimPublicMessage.content.sender.sender_type) {
      case member:
        MAC membership_tag;
      case external:
      case new_member_commit:
      case new_member_proposal:
        struct{};
    };
    select (SlimPublicMessage.content.content_type) {
      case commit:
        optional<SlimUpdatePath> path;
      case application:
      case proposal:
        struct{};
    };
  } SlimPublicMessage;

  struct {
    select (SlimPrivateMessage.content_type) {
      case application:
        opaque application_data<V>;
      case proposal:
        SlimProposal proposal;
      case commit:
        SlimCommit commit;
    };
    SlimFramedContentAuthData auth;
    opaque                    padding[length_of_padding];
  } SlimPrivateMessageContent;

  struct {
    opaque group_id<V>;
    uint64 epoch;
    ContentType content_type;
    opaque authenticated_data<V>;
    opaque encrypted_sender_data<V>;
    opaque ciphertext<V>;
    select (SlimPrivateMessage.content_type) {
      case commit:
        optional<SlimUpdatePath> path;
      case application:
      case proposal:
        struct{};
    };
  } SlimPrivateMessage;
```

The optional signature field in SlimFramedContentAuthData MUST be present whenever content_type is application or proposal. For content_type = commit, signature MUST be absent if and only if the single-signature construction of Section 9.4 applies.

When present in a SlimPublicMessage, signature MUST use signature_type = signature_ref. When present in a SlimPrivateMessage, signature MUST use signature_type = signature. This ensures that a SlimPrivateMessage framing signature is encrypted as part of SlimPrivateMessageContent.

When a framing signature is present, it is computed and verified as in [RFC9420], except that the to-be-signed content is SlimFramedContentTBS and the WireFormat is either mls_slim_public_message or mls_slim_private_message.

For SlimPrivateMessage, the ciphertext field encrypts a SlimPrivateMessageContent. The sender data encryption, content encryption, padding, and decryption rules are otherwise the same as for PrivateMessage in [RFC9420].

After decrypting a SlimPrivateMessage, the recipient reconstructs the SlimFramedContent from the outer group_id, epoch, content_type, and authenticated_data fields, the decrypted sender, and the decrypted content. The reconstructed SlimFramedContent is used with the decrypted SlimFramedContentAuthData for signature verification and, for commits, for OuterUpdateHash verification, transcript hash computation, and confirmation tag verification.

For SlimPublicMessage, the membership tag is computed over the following structure:

```
struct {
  SlimFramedContentTBS       content_tbs;
  SlimFramedContentAuthData  auth;
} SlimAuthenticatedContentTBM;
```

For commits, the input to the confirmed transcript hash is:

```
struct {
  WireFormat        wire_format;
  SlimFramedContent content;
} SlimConfirmedTranscriptHashInput;
```

The SlimUpdatePath is not part of SlimFramedContent, SlimAuthenticatedContentTBM, or SlimConfirmedTranscriptHashInput. It is therefore not authenticated by the membership tag, the framing signature, or the transcript hash. In a SlimPrivateMessage carrying a commit, the path is carried outside the ciphertext so the DS can reduce it per recipient.

9.3. SlimMessage

SlimMessage is the generic term for the two concrete wire presentations, SlimPublicMessage and SlimPrivateMessage. Both presentations are used for sender-to-DS transport and for DS-to-recipient delivery. For commits, the difference between the two is the cardinality of the HPKECiphertextRef vectors in path.

In a SlimMLS group, the [RFC9420] PublicMessage and PrivateMessage wire formats MUST NOT be used. A SlimMLS-aware sender MUST emit a SlimPublicMessage or SlimPrivateMessage in their place.

9.4. Single Signature Construction

When a SlimCommit is sent by a member, contains a SlimLeafNode, and the sender’s signature key is unchanged, the framing signature is omitted, and authenticity is provided by the SlimLeafNode’s own signature in combination with an OuterUpdateHash component placed in the SlimLeafNode’s app_data_dictionary extension. The confirmation tag is still present and processed as in [RFC9420].

The OuterUpdateHash binds the framed SlimCommit (excluding the SlimLeafNode itself, to avoid a circular dependency) and the GroupContext to the SlimLeafNode:

```
struct {
  opaque outer_update_hash<V>;
} OuterUpdateHash;

struct {
  opaque       group_id<V>;
  uint64       epoch;
  Sender       sender;
  opaque       authenticated_data<V>;
  ContentType  content_type;
  select (OuterFramedContent.content_type) {
    case commit:
      ProposalOrRef proposals<V>;
  };
} OuterFramedContent;

struct {
  ProtocolVersion    version = mls10;
  WireFormat         wire_format;
  OuterFramedContent content;
  GroupContext       context;
} SlimFramedContentTBH;
```

outer_update_hash is the hash, under the group's ciphersuite hash function, of the TLS-encoded SlimFramedContentTBH. Fields of OuterFramedContent are populated from the SlimCommit being framed. The SlimLeafNode is omitted to prevent the circular dependency that would arise from including the very component being computed.

The app_data_dictionary extension MUST contain exactly one OuterUpdateHash component under component identifier TBD. A missing, malformed, or duplicated OuterUpdateHash component makes the single-signature construction invalid.

The OuterUpdateHash authenticates the non-path commit contents that the omitted framing signature would otherwise cover. The path's HPKEPublicKeyRefs are authenticated separately by parent hash validation. The path's HPKECiphertextRefs are not authenticated by the signature and do not contribute to the group state. Tampering with them can only cause decryption, derived-key, or confirmation-tag validation to fail.

When the construction applies, the SignaturePublicKeyRef in the SlimLeafNode MUST equal the sender's current SignaturePublicKeyRef. A sender that wishes to change its signature key MUST instead emit a SlimCommit whose authentication data carries a framing signature, so that the new key is bound by a signature under the old one.

A SlimCommit without a SlimLeafNode (e.g., a commit containing only Add or Remove proposals) does not use this construction. Its authentication data carries a framing signature.

A SlimCommit whose sender type is not member does not use this construction. Its authentication data carries a framing signature.

9.5. Sender, DS, and Recipient Behavior

A committer constructs a SlimMessage carrying a SlimCommit as follows:

1. Perform the steps of an [RFC9420] commit, deriving the path's HPKEPublicKeys, HPKECiphertexts, parent hashes, and the new epoch's key schedule. Parent hashes are computed over the slim parent-hash inputs, with HPKEPublicKeyRef replacing HPKEPublicKey.

2. Build a SlimCommit containing the committed proposals and, if the commit contains a path, the committer's new SlimLeafNode. The SlimLeafNode's parent hash MUST authenticate the HPKEPublicKeyRefs in the SlimUpdatePath.

3. Compute the confirmation tag for the new epoch as in [RFC9420].

4. If the single-signature construction applies, include a valid OuterUpdateHash in the SlimLeafNode and leave the optional signature field absent. If the construction does not apply, include the normal framing signature in SlimFramedContentAuthData, using signature_type = signature_ref for SlimPublicMessage and signature_type = signature for SlimPrivateMessage.

5. Build a SlimUpdatePath whose SlimUpdatePathNodes carry the HPKEPublicKeyRefs and HPKECiphertextRefs of the path, and emit a SlimPublicMessage or SlimPrivateMessage, optionally accompanied by a LargeObjectCarrier holding the corresponding HPKEPublicKeys, HPKECiphertexts, and any public-message framing signature referenced by the SignatureOrRef.

The DS, knowing the ratchet tree before and after the commit, produces a per-recipient SlimMessage by reducing each SlimUpdatePathNode's HPKECiphertextRef vector to the subset needed by that recipient. The DS MUST retain the HPKEPublicKeyRef in every SlimUpdatePathNode, and MUST retain enough HPKECiphertextRefs for the recipient to decrypt one path secret and derive the remaining path secrets it needs to process the Commit. If a LargeObjectCarrier is present, the DS reduces it accordingly, retaining only any

HPKEPublicKeys the recipient must receive and the HPKECiphertexts corresponding to the retained HPKECiphertextRefs. If the commit removes the recipient, path is omitted.

For basic-processing delivery of a Commit with an update path, a recipient needs enough ciphertext material to derive the epoch secrets and verify the Commit, but does not necessarily need every HPKE public key referenced by the path. Such a recipient can process the Commit, receive subsequent MLS messages, and send application messages, but might not be able to construct its own Commit with an update path until it resolves additional HPKEPublicKeyRefs. To make the recipient immediately update-capable, the DS MAY include any updated HPKE public keys in the recipient's copath that the recipient cannot derive from its retained path secret. In a full tree with no blank nodes, this is one HPKE public key for each non-committing recipient.

A recipient processes a SlimMessage carrying a SlimCommit by:

1. Resolving the HPKECiphertextRefs required for the recipient and any HPKEPublicKeyRefs that are functionally needed, per Section 6. For SlimPublicMessage, this also includes any SignatureRef in the SlimFramedContentAuthData SignatureOrRef. For SlimPrivateMessage, the recipient decrypts SlimPrivateMessageContent and uses the inline SignatureOrRef value when one is present.

2. If the authentication data contains a framing signature, verifying it per [RFC9420] using SlimFramedContentTBS. If the SignatureOrRef uses signature_type = signature_ref, the recipient first resolves the referenced signature. If it uses signature_type = signature, the recipient uses the inline signature. If the framing signature is absent, verifying the SlimLeafNode signature and OuterUpdateHash per Section 9.4.

3. Decrypting the resolved HPKECiphertext, deriving the path public keys, and verifying that the derived public keys hash to the authenticated HPKEPublicKeyRefs.

4. Applying the path update and verifying parent hashes over the slim encodings per Section 7.9.2 of [RFC9420].

5. Processing the commit per Section 12.4 of [RFC9420], deriving the new epoch, and verifying the confirmation tag. If confirmation tag validation fails, the recipient MUST reject the commit.

The input to the confirmed transcript hash is SlimConfirmedTranscriptHashInput (Section 9.2). The interim transcript hash is computed from the confirmed transcript hash and the confirmation tag as in [RFC9420].

DSs that do not maintain the ratchet tree cannot perform the per-recipient reduction described above. Strategies for such deployments are out of scope.

10. Optimizing Payload Sizes

The separation of large objects and their references in structs that are sent over the wire or stored locally allows a few performance optimizations. They are up to the application to implement.

10.1. Storing Partial Trees

SlimMLS clients need the slim public tree for MLS tree computations, but they do not need to fetch every large object referenced by that tree. Tree hashes, parent hashes, and GroupInfo validation are computed over SlimLeafNode and SlimParentNode encodings, so the references to public keys and credentials are the values committed to by those computations. A client can therefore defer fetching large objects that are not functionally needed, such as HPKE public keys outside the client’s copath. This is especially relevant when joining a group, where the DS can selectively include the public keys and credentials the joiner needs immediately.

10.2. Omitting Cached Large Objects

If the DS keeps track of the group’s public state, clients that send a commit with an update path only need to proactively provide the signature public key and credential if either of them change as part of the commit.

10.3. Cross-group Credential Caching

If clients use the same credential across multiple groups, other clients can cache them and pull new credentials selectively when they encounter credential references they can’t resolve. Whether this is a performance increase depends on the context of the application.

The DS can similarly deduplicate stored credentials.

10.4. Delayed Fetching of Large Objects

A client that has been offline for some time can fetch and process slim messages first. It can wait to fetch large objects that are not relevant for processing (such as HPKE public keys) until it has arrived at the current group state. The client thus avoids downloading stale, intermediate large objects.

11. Wire Formats

SlimMLS defines new WireFormat values for the SlimMLS variants of each [RFC9420] wire format. The MLSMessage struct defined in [RFC9420] is correspondingly extended with the following cases:

```
struct {
  ProtocolVersion version = mls10;
  WireFormat wire_format;
  select (MLSMessage.wire_format) {
    case mls_slim_welcome:          SlimWelcome        slim_welcome;
    case mls_slim_key_package:      SlimKeyPackage     slim_key_package;
    case mls_slim_group_info:       SlimGroupInfo      slim_group_info;
    case mls_slim_public_message:   SlimPublicMessage  slim_public_message;
    case mls_slim_private_message:  SlimPrivateMessage slim_private_message;
  };
} MLSMessage;
```

In a SlimMLS group, the [RFC9420] wire formats mls_public_message, mls_private_message, mls_welcome, mls_group_info, and mls_key_package MUST NOT be used. A SlimMLS-aware sender MUST emit the corresponding SlimMLS wire format instead: mls_slim_public_message or mls_slim_private_message for member-originated messages, mls_slim_welcome for Welcomes, mls_slim_group_info for standalone GroupInfos, and mls_slim_key_package for KeyPackages. A recipient MUST reject any RFC 9420 wire format received in the context of a group that carries the slim_mls extension.

12. The slim_mls Extension

SlimMLS is signaled by a GroupContext extension named slim_mls. Presence of this extension in the GroupContext means that all wire formats within the group use SlimMLS replacements (Section 11).

13. The slim_ratchet_tree Extension

The [RFC9420] ratchet_tree GroupInfo extension carries LeafNode and ParentNode objects. SlimMLS defines a slim_ratchet_tree GroupInfo extension that conveys the same tree using SlimLeafNode and SlimParentNode objects:

```
struct {
    NodeType node_type;
    select (SlimNode.node_type) {
        case leaf:   SlimLeafNode leaf_node;
        case parent: SlimParentNode parent_node;
    };
} SlimNode;

optional<SlimNode> slim_ratchet_tree<V>;
```

The ordering, truncation, and validation rules are the same as for the ratchet_tree extension in Section 12.4.3.3 of [RFC9420], except that tree hash and parent hash computations use the slim node encodings. The large objects referenced by the slim nodes are resolved per Section 6.

In a SlimMLS group, the [RFC9420] ratchet_tree extension MUST NOT be used. A sender that would include ratchet_tree MUST instead include the slim_ratchet_tree extension.

14. The slim_external_pub Extension

The [RFC9420] external_pub GroupInfo extension carries an HPKEPublicKey that external joiners use when issuing an External Init proposal (Section 12.4.3.2 of [RFC9420]). SlimMLS defines a slim_external_pub GroupInfo extension that conveys the same information using an HPKEPublicKeyRef:

```
struct {
  HPKEPublicKeyRef external_pub_ref;
} SlimExternalPub;
```

In a SlimMLS group, the [RFC9420] external_pub extension MUST NOT be used. A sender that would include external_pub MUST instead include the slim_external_pub extension, populated with the reference to the relevant HPKEPublicKey. The actual public key is resolved per Section 6, as for any other HPKEPublicKeyRef.

15. Security Considerations

Outside the SlimCommit split path described below, the signing and transcript-hashing rules of [RFC9420] are preserved by SlimMLS: every struct that was authenticated under [RFC9420] remains authenticated, with large objects bound through the collision-resistant hash references defined in Section 4. The strength of the binding is therefore at most the collision resistance of the ciphersuite hash.

SlimCommit (Section 9) changes the exact authentication surface of Commits with paths. The path's HPKEPublicKeyRefs are not covered directly by the framing signature or OuterUpdateHash, but are authenticated by the parent hash chain rooted in the signed SlimLeafNode. The path's HPKECiphertextRefs are not authenticated by the signature and are treated as delivery objects. Tampering with an HPKECiphertextRef or the referenced HPKECiphertext can only cause the recipient to fail reference resolution, decryption, derived-public-key matching, parent-hash validation, or confirmation-tag validation.

Large-object retrieval channels (Section 6) may be unauthenticated. The hash-reference verification in step 2 of that section is what provides authentication of the resolved objects, and is what binds them to the signed and transcript-hashed slim structures. A Delivery Service or network attacker that withholds, modifies, or substitutes entries in any channel can only cause a recipient to fail to resolve a reference, which is functionally equivalent to dropping the message, which the DS can already do under [RFC9420]. An attacker may also surface unsolicited or malformed objects through any channel. Recipients MUST NOT treat the contents of any retrieval channel as authoritative metadata and MUST ignore objects whose hash does not match any reference the recipient needs to resolve.

A client that caches resolved large objects across groups MUST index its cache by the tuple (reference type, ciphersuite hash function, reference value). The same underlying object yields different reference values under different hash functions, and a cached object MUST NOT be treated as authoritative across ciphersuites whose hash functions differ.

SlimKeyPackage batch signatures (Section 7) replace independent LeafNode signatures with one signature over a Merkle root. A recipient MUST verify both the Merkle inclusion proof and the batch signature before treating the SlimLeafNodeTBS as authenticated. The signed SlimKeyPackageBatchTBS binds the protocol version, ciphersuite, tree size, Merkle root, and SignaturePublicKeyRef. Together with the SignWithLabel label and the Merkle leaf and parent hash labels, this prevents a valid batch signature from being replayed across protocols, ciphersuites, or signing keys. The Merkle proof is carried in the SlimLeafNode only for leaf_node_source = key_package, so group members that later validate the ratchet tree can verify the authenticity of the leaf without having received the original SlimKeyPackage.

All SlimKeyPackages whose key-package SlimLeafNodes share the same signature field are linkable as members of the same publication batch. Deployments that consider this linkability sensitive can reduce batch sizes or publish independently signed batches.

SlimWelcome (Section 8) introduces two confidentiality changes relative to the [RFC9420] Welcome:

* When WelcomeGroupInfo.info_type is plaintext, the GroupInfo is visible to the DS and to anyone observing the SlimWelcome on the wire. This variant is appropriate only in deployments where the GroupInfo is not considered confidential with respect to those parties (e.g., where the DS already maintains group state).

* When SlimWelcome.group_info is absent (presence octet 0), the DS chooses which WelcomeGroupInfo a joiner ultimately receives.

In both cases authenticity is unchanged: the joiner MUST verify the SlimGroupInfo signature as under [RFC9420], and a SlimWelcome that reaches a recipient with group_info still absent MUST be rejected (Section 8.3).

SlimPrivateMessage and encrypted SlimWelcome GroupInfo signatures are carried inside the encrypted plaintext using the SignatureOrRef signature variant. This prevents a visible LargeObjectCarrier signature from being tested against known signature public keys to identify the hidden signer.

16. IANA Considerations

16.1. SlimMLS MLS Extension Types

IANA is requested to add the following entries to the "MLS Extension Types" registry defined in Section 17.3 of [RFC9420]:

Value	Name	Message(s)	Recommended	Reference
0x000C	slim_mls	GC	Y	RFC XXXX
0x000D	slim_ratchet_tree	GI	Y	RFC XXXX
0x000E	slim_external_pub	GI	Y	RFC XXXX

Table 3

(Values are SlimMLS’s suggested allocations. IANA may pick others.)

The "Message(s)" abbreviations are those used in Section 17.3 of [RFC9420]. "GC" denotes a GroupContext extension and "GI" denotes a GroupInfo extension. RFC XXXX is to be replaced with the RFC number assigned to this document upon publication.

16.2. SlimMLS Wire Formats

IANA is requested to add the following entries to the "MLS Wire Formats" registry defined in Section 17.2 of [RFC9420]:

Value	Name	Recommended	Reference
0x0007	mls_slim_welcome	Y	RFC XXXX
0x0008	mls_slim_key_package	Y	RFC XXXX
0x0009	mls_slim_group_info	Y	RFC XXXX
0x000A	mls_slim_public_message	Y	RFC XXXX
0x000B	mls_slim_private_message	Y	RFC XXXX

Table 4

(Values are SlimMLS’s suggested allocations. IANA may pick others.)

17. References

17.1. Normative References

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

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

[RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <https://www.rfc-editor.org/rfc/rfc9420>.

17.2. Informative References

[I-D.ietf-mls-extensions]
Robert, R., "The Messaging Layer Security (MLS) Extensions", Work in Progress, Internet-Draft, draft-ietf-mls-extensions-09, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-09>.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Raphael Robert
Phoenix R&D
Email: ietf@raphaelrobert.com

Konrad Kohbrok
Phoenix R&D
Email: konrad@ratchet.ing