Messaging Layer Security
Internet-Draft
Intended status: Informational
Expires: 15 June 2025

J. Alwen
Amazon
R. Barnes
Cisco
R. Mahy
Rohan Mahy Consulting Services
M. Mularczyk
Amazon

12 December 2024

A Safe Application Interface to Messaging Layer Security
draft-barnes-mls-appsync-01

Abstract

The Messaging Layer Security protocol enables a group of participants to negotiate a common cryptographic state. While the primary function of MLS is to establish shared secret state for the group, an MLS group also captures authentication information for group participants and information on which the group has confirmed agreement. This document defines an interface interface by which multiple uncoordinated application functions may safely reuse the cryptographic state of an MLS group for application purposes.

About This Document

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

The latest revision of this draft can be found at https://bifurcation.github.io/mls-appsync/draft-barnes-mls-appsync.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-barnes-mls-appsync/.

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/bifurcation/mls-appsync.

Status of This Memo

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

Copyright Notice

Table of Contents

1. Introduction

The Messaging Layer Security protocol (MLS) is designed to be integrated into applications, in order to provide security services that the application requires [RFC9420]. There are two questions to answer when designing such an integration:

1. How does the application provide the services that MLS requires?

2. How does the application use MLS to get security benefits?

The MLS Architecture describes the requirements for the first of these questions [I-D.mls-architecture], namely the structure of the Delivery Service and Authentication Service that MLS requires. This document is focused on the second question.

MLS itself offers some basic functions that applications can use, such as the secure message encapsulation (PrivateMessage), the MLS exporter, and the epoch authenticator. Current MLS applications make use of these mechanisms to acheive a variety of confidentiality and authentication properties.

As application designers become more familiar with MLS, there is increasing interest in leveraging other cryptographic tools that an MLS group provides:

* HPKE and signature key pairs for each member, where the private key is known only to that member, and the public key is authenticated to the other members.

* A pre-shared key mechanism that can allow an application to inject data into the MLS key schedule.

* An exporter mechanism that allows applications to derive secrets from the MLS key schedule.

* Association of data with Commits as a synchronization mechanism.

* Binding of information to the GroupContext to confirm group agreement.

There is also interest in exposing an MLS group to multiple loosely-coordinated components of an application. To accommodate such cases, the above mechanisms need to be exposed in such a way that different components' usage will not conflict with each other, or with MLS itself.

This document defines a set of mechanisms that application components can use to ensure that their use of these facilities is properly domain-separated from MLS itself, and from other application components that might be using the same MLS group.

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.

We make heavy use of the terminology in the MLS specification [RFC9420].

Application: The system that instantiates, manages, and uses an MLS group. Each MLS group is used by exactly one application, but an application may maintain multiple groups.

Application component: A subsystem of an application that has access to an MLS group.

Component ID: An identifier for an application component. These identifiers are assigned by the application.

3. Protocol Overview

The mechansms in this document take MLS mechanisms that are either not inherently designed to be used by applications, or not inherently designed to be used by multiple application components, and adds a domain separator that separates application usage from MLS usage, and application components' usage from each other:

* Signing operations are tagged so that signatures will only verify in the context of a given component.

* Public-key encryption operations are similarly tagged so that encrypted data will only decrypt in the context of a given component.

* Pre-shared keys are identified as originating from a specific component, so that differnet components' contributions to the MLS key schedule will not collide.

* Exported values include an identifier for the component to which they are being exported, so that different components will get different exported values.

We also define new general mechanisms that allow applications to take advantage of the extensibility mechanisms of MLS without having to define extensions themselves:

* An application_data extension type that associates application data with MLS messages, or with the state of the group.

* An ApplicationData proposal type that enables arbitrary application data to be associated to a Commit.

* An ApplicationDataUpdate proposal type that enables efficient updates to an application_data GroupContext extension.

As with the above, information carried in these proposals and extension marked as belonging to a specific application component, so that components can manage their information independently.

The separation between components is acheived by the application assigning each component a unique component ID number. These numbers are then incorporated into the appopriate calculations in the protocol to achieve the required separation.

3.1. Component IDs

A component ID is a four-byte value that uniquely identifies a component within the scope of an application.

```
uint32 ComponentID;
```

> TODO: What are the uniqueness requirements on these? It seems like the more diversity, the better. For example, if a ComponentID is reused across applications (e.g., via an IANA registry), then there will be a risk of replay across applications. Maybe we should include a binder to the group/epoch as well, something derived from the key schedule.
>
> TODO: It might be better to frame these in terms of "data types" instead of components, to avoid presuming software architecture. Though that makes less sense for the more "active" portions of the API, e.g., signing and encryption.

When a label is required for an operation, the following data structure is used. The label field identifies the operation being performed. The component_id field identifies the component performing the operation. The context field is specified by the operation in question.

```
struct {
  opaque label<V>;
  ComponentID component_id;
  opaque context<V>;
} ComponentOperationLabel;
```

4. Hybrid Public Key Encryption (HPKE) Keys

This component of the API allows components to make use of the HPKE key pairs generated by MLS. An component identified by an CompnentID can use any HPKE key pair for any operation defined in [RFC9180], such as encryption, exporting keys and the PSK mode, as long as the info input to Setup<MODE>S and Setup<MODE>R is set to ComponentOperationLabel with component_id set to the appopriate ComponentID. The context can be set to an arbitrary Context specified by the application designer and can be empty if not needed. For example, a component can use a key pair PublicKey, PrivateKey to encrypt data as follows:

```
SafeEncryptWithContext(ComponentID, PublicKey, Context, Plaintext) =
    SealBase(PublicKey, ComponentOperationLabel, "", Plaintext)

SafeDecryptWithContext(ComponentID, PrivateKey, Context, KEMOutput, Ciphertext
) =
    OpenBase(KEMOutput, PrivateKey, ComponentOperationLabel, "", Ciphertext)
```

Where the fields of ComponentOperationLabel are set to

```
label = "MLS 1.0 Application"
component_id = ComponentID
context = Context
```

> TODO: Should this use EncryptWithLabel / DecryptWithLabel? That wouldn't cover other modes / exports, but you could say "mutatis mutandis".

For operations involving the secret key, ComponentID MUST be set to the ComponentID of the component performing the operation, and not to the ID of any other component. In particular, this means that a component cannot decrypt data meant for another component, while components can encrypt data that other components can decrypt.

In general, a ciphertext encrypted with a PublicKey can be decrypted by any entity who has the corresponding PrivateKey at a given point in time according to the MLS protocol (or application component). For convenience, the following list summarizes lifetimes of MLS key pairs.

* The key pair of a non-blank ratchet tree node. The PrivateKey of such a key pair is known to all members in the nodeâ\200\231s subtree. In particular, a PrivateKey of a leaf node is known only to the member in that leaf. A member in the subtree stores the PrivateKey for a number of epochs, as long as the PublicKey does not change. The key pair of the root node SHOULD NOT be used, since the external key pair recalled below gives better security.

* The external_priv, external_pub key pair used for external initialization. The external_priv key is known to all group members in the current epoch. A member stores external_priv only for the current epoch. Using this key pair gives better security guarantees than using the key pair of the root of the ratchet tree and should always be preferred.

* The init_key in a KeyPackage and the corresponding secret key. The secret key is known only to the owner of the KeyPackage and is deleted immediately after it is used to join a group.

5. Signature Keys

MLS session states contain a number of signature keys including the ones in the LeafNode structs. Application components can safely sign content and verify signatures using these keys via the SafeSignWithLabel and SafeVerifyWithLabel functions, respectively, much like how the basic MLS protocol uses SignWithLabel and VerifyWithLabel.

In more detail, a component identified by ComponentID should sign and verify using:

```
SafeSignWithLabel(ComponentID, SignatureKey, Label, Content) =
    SignWithLabel(SignatureKey, "ComponentOperationLabel", ComponentOperationL
abel)

SafeVerifyWithLabel(ComponentID, VerificationKey, Label, Content, SignatureVal
ue) =
    VerifyWithLabel(VerificationKey, "ComponentOperationLabel", ComponentOpera
tionLabel, SignatureValue)
```

Where the fields of ComponentOperationLabel are set to

```
label = Label
component_id = ComponentID
context = Content
```

For signing operations, the ComponentID MUST be set to the ComponentID of the component performing the signature, and not to the ID of any other component. This means that a component cannot produce signatures in place of other component. However, components can verify signatures computed by other components. Domain separation is ensured by explicitly including the ComponentID with every operation.

6. Pre-Shared Keys

PSKs represent key material that is injected into the MLS key schedule when creating or processing a commit as defined in Section 8.4 of [RFC9420]. Its injection into the key schedule means that all group members have to agree on the value of the PSK.

While PSKs are typically cryptographic keys which due to their properties add to the overall security of the group, the PSK mechanism can also be used to ensure that all members of a group agree on arbitrary pieces of data represented as octet strings (without the necessity of sending the data itself over the wire). For example, a component can use the PSK mechanism to enforce that all group members have access to and agree on a password or a shared file.

This is achieved by creating a new epoch via a PSK proposal. Transitioning to the new epoch requires using the information agreed upon.

To facilitate using PSKs in a safe way, this document defines a new PSKType for application components. This provides domain separation between pre-shared keys used by the core MLS protocol and applications, and between those used by different components.

```
enum {
  // ...
  application(3),
  (255)
} PSKType;

struct {
  PSKType psktype;
  select (PreSharedKeyID.psktype) {
    // ...
    case application:
      ComponentID component_id;
      opaque psk_id<V>;
  };
  opaque psk_nonce<V>;
} PreSharedKeyID;
```

TODO: It seems like you could also do this by structuring the external PSKType as (component_id, psk_id). I guess this approach separates this API from other external PSKs.

7. Exported Secrets

An application component can use MLS as a group key agreement protocol by exporting symmetric keys. Such keys can be exported (i.e. derived from MLS key material) in two phases per epoch: Either at the start of the epoch, or during the epoch. Derivation at the start of the epoch has the added advantage that the source key material is deleted after use, allowing the derived key material to be deleted later even during the same MLS epoch to achieve forward secrecy. The following protocol secrets can be used to derive key from for use by application components:

* exporter_secret at the beginning of an epoch

* application_export_secret during an epoch

The application_export_secret is an additional secret derived from the epoch_secret at the beginning of the epoch in the same way as the other secrets listed in Table 4 of [RFC9420] using the label "application_export".

Any derivation performed by an application component either from the exporter_secret or the application_export_secret has to use the following function:

```
DeriveApplicationSecret(Secret, Label) =
  ExpandWithLabel(Secret, "ApplicationExport " + ComponentID + " " + Label)
```

Where ExpandWithLabel is defined in Section 8 of [RFC9420] and where ComponentID MUST be set to the ComponentID of the component performing the export.

TODO: This section seems over-complicated to me. Why is it not sufficient to just use the exporter_secret? Or the MLS-Exporter mechanism with a label structured to include the ComponentID?

8. Attaching Application Data to MLS Messages

The MLS GroupContext, LeafNode, KeyPackage, and GroupInfo objects each have an extensions field that can carry additional data not defined by the MLS specification. The application_data extension provides a generic container that applications can use to attach application data to these messages. Each usage of the extension serves a slightly different purpose:

* GroupContext: Confirms that all members of the group agree on the application data, and automatically distributes it to new joiners.

* KeyPackage and LeafNode: Associates the application data to a particular client, and advertises it to the other members of the group.

* GroupInfo: Distributes the application data confidentially to the new joiners for whom the GroupInfo is encrypted (as a Welcome message).

The content of the application_data extension is a serialized ApplicationDataDictionary object:

```
struct {
    ComponentID component_id;
    opaque data<V>;
} ComponentData;

struct {
    ComponentData component_data<V>;
} ApplicationDataDictionary;
```

The entries in the component_data MUST be sorted by component_id, and there MUST be at most one entry for each component_id.

An application_data extension in a LeafNode, KeyPackage, or GroupInfo can be set when the object is created. An application_data extension in the GroupContext needs to be manage using the tools available to update GroupContext extensions: The creator of the group can set extensions unilaterally, and thereafter, the GroupContextExtensions

proposal can be used to update extensions. The ApplicationDataUpdate proposal described in Section 9 provides a more efficient way to update the application_data extension.

9. Updating Application Data in the GroupContext

Updating the application_data with a GroupContextExtensions proposal is cumbersome. The application data needs to be transmitted in its entirety, along with any other extensions, whether or not they are being changed. And a GroupContextExtensions proposal always requires an UpdatePath, which updating application state never should.

The ApplicationDataUpdate proposal allows the application_data extension to be updated without these costs. Instead of sending the whole value of the extension, it sends only an update, which is interpreted by the application to provide the new content for the application_data extension. No other extensions are sent or updated, and no UpdatePath is required.

```
enum { invalid(0), update(1), remove(2), (255) }
ApplicationDataUpdateOperation;

struct { ComponentID component_id; ApplicationDataUpdateOperation op;

select (ApplicationDataUpdate.op) {
    case update: opaque update<V>;
    case remove: struct{}
} } ApplicationDataUpdate;
```

An ApplicationDataUpdate proposal is invalid if its component_id references a component that is not known to the application, or if it specifies the removal of state for a component_id that has no state present. A proposal list is invalid if it includes multiple ApplicationDataUpdate proposals that remove state for the same component_id, or proposals that both update and remove state for the same component_id. In other words, for a given component_id, a proposal list is valid only if it contains (a) a single remove operation or (b) one or more update operation.

TODO: Deconflict with GroupContextExtensions.

ApplicationDataUpdate proposals are processed after any default proposals (i.e., those defined in [RFC9420]), and any ApplicationData proposals.

A client applies ApplicationDataUpdate proposals by component ID. For each component_id field that appears in an ApplicationDataUpdate proposal in the Commit, the client assembles a list of

ApplicationDataUpdate proposals with that component_id, in the order in which they appear in the Commit, and processes them in the following way:

* If the list comprises a single proposal with the op field set to remove:
  - If there is an entry in the component_states vector in the application_state extension with the specified component_id, remove it.
  - Otherwise, the proposal is invalid.
* If the list comprises one or more proposals, all with op field set to update:
  - Provide the application logic registered to the component_id value with the content of the update field from each proposal, in the order specified.
  - The application logic returns either an opaque value new_data that will be stored as the new application data for this component, or else an indication that it considers this update invalid.
  - If the application logic considers the update invalid, the MLS client MUST consider the proposal list invalid.
  - If no application_data extension is present in the GroupContext, add one to the end of the extensions list in the GroupContext.
  - If there is an entry in the component_data vector in the application_data extension with the specified component_id, then set its data field to the specified new_data.
  - Otherwise, insert a new entry in the component_states vector with the specified component_id and the data field set to the new_data value. The new entry is inserted at the proper point to keep the component_states vector sorted by component_id.
* Otherwise, the proposal list is invalid.

  NOTE: An alternative design here would be to have the update operation simply set the new value for the application_data GCE, instead of sending a diff. This would be simpler in that the MLS stack wouldn't have to ask the application for the new state value, and would discourage applications from storing large state

in the GroupContext directly (which bloats Welcome messages). It would effectively require the state in the GroupContext to be a hash of the real state, to avoid large ApplicationDataUpdate proposals. This pushes some complexity onto the application, since the application has to define a hashing algorithm, and define its own scheme for initializing new joiners.

10. Attaching Application Data to a Commit

The ApplicationData proposal type allows an application component to associate application data to a Commit, so that the member processing the Commit knows that all other group members will be processing the same data. ApplicationData proposals are ephemeral in the sense that they do not change any persistent state related to MLS, aside from their appearance in the transcript hash.

The content of an ApplicationData proposal is the same as an application_data extension. The proposal type is set in Section 12.

```
struct {
    ComponentID component_id;
    opaque data<V>;
} ApplicationData;
```

An ApplicationData proposal is invalid if it contains a component_id that is unknown to the application, or if the application_data field contains any ComponentData entry whose data field is considered invalid by the application logic registered to the indicated component_id.

ApplicationData proposals MUST be processed after any default proposals (i.e., those defined in [RFC9420]), but before any ApplicationDataUpdate proposals.

A client applies an ApplicationData proposal by providing the contents of the application_data field to the component identified by the component_id. If a Commit references more than one ApplicationData proposal for the same component_id value, then they MUST be processed in the order in which they are specified in the Commit.

11. Security Considerations

The API defined in this document provides the following security guarantee: If an application uses MLS and all its components use this API, then the security guarantees of the base MLS protocol and the security guarantees of the components, each analyzed in isolation, still hold for the composed protocol. In other words, the API protects applications from careless component developers. As long as all the components use this API, it is not possible that some combination of components (the developers of which did not know about each other) impedes the security of the base MLS protocol or any used component. No further analysis of the combination is necessary. This also means that any security vulnerabilities introduced by one component do not spread to other component or the base MLS protocol.

12. IANA Considerations

TODO:

* Register application_data extension
* Register ApplicationData proposal
* Register ApplicationDataUpdate proposal

13. References

13.1. 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>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

[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>.

13.2. Informative References

[I-D.mls-architecture]
"*** BROKEN REFERENCE ***".

Acknowledgments

TODO: Acknowledgements.

Authors' Addresses

Joel Alwen
Amazon
Email: alwenjo@amazon.com

Richard Barnes
Cisco
Email: rlb@ipv.sx

Rohan Mahy
Rohan Mahy Consulting Services
Email: rohan.ietf@gmail.com

Marta Mularczyk
Amazon
Email: mulmarta@amazon.com

MLS
Internet-Draft
Intended status: Informational
Expires: 30 March 2025

J. Alwen
AWS
B. Hale
Naval Postgraduate School
M. Mularczyk
AWS
X. Tian
Naval Postgraduate School

26 September 2024

Flexible Hybrid PQ MLS Combiner
draft-hale-mls-combiner-01

Abstract

This document describes a protocol for combining a traditional MLS session with a post-quantum (PQ) MLS session to achieve flexible and efficient hybrid PQ security that amortizes the computational cost of PQ Key Encapsulation Mechanisms and Digital Signature Algorithms. Specifically, we describe how to use the exporter secret of a PQ MLS session, i.e. an MLS session using a PQ ciphersuite, to seed PQ guarantees into an MLS session using a traditional ciphersuite. By supporting on-demand traditional-only key updates (a.k.a. PARTIAL updates) or hybrid-PQ key updates (a.k.a. FULL updates), we can reduce the bandwidth and computational overhead associated with PQ operations while meeting the requirement of frequent key rotations.

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 30 March 2025.

Table of Contents

1. Introduction

A fully capable quantum adversary has the ability to break fundamental underlying cryptographic assumptions of traditional Key Encapsulation Mechanisms (KEMs) and Digital Signature Algorithms (DSAs). This has led to the development of post-quantum (PQ) cryptographically secure KEMs and DSAs by the cryptographic research community which have been formally adopted by the National Institute of Standards and Technology (NIST), including the Module Lattice KEM (ML-KEM) and Module Lattice DSA (ML-DSA) algorithms. While these provide PQ security, ML-KEM and ML-DSA have significantly worse overhead in terms of public key size, signature size, ciphertext size, and CPU time than their traditional counterparts. Moreover, research arms on side-channel attacks, etc., have motivated uses of hybrid-PQ combiners that draw security from both the underlying PQ and underlying traditional components. A variety of hybrid security treatments have arisen across IETF working groups to bridge the gap between performance and security to encourage the adoption of PQ security in existing protocols, including the MLS protocol [RFC9420].

Within the MLS working group, there are several topic areas that make use of PQ security extensions:

1. A single MLS ciphersuite for a hybrid post-quantum/traditional KEM. The ciphersuite can act as a drop-in replacement for the KEM, focusing on hybrid confidentiality but not authenticity, and does not incur changes elsewhere in the MLS stack. As a confidentiality focus, it addresses the the harvest-now / decrypt-later threat model. However, every key epoch incurs a PQ overhead cost.

2. Hybrid PQ signature ciphersuites that address hybrid authenticity, including construction and security considerations of hybrid signatures.

3. Mechanisms that leverage hybridization as a means to not only address the security balance between PQ and traditional components and achieve resistance to harvest-now / decrypt-later attacks, but also use it as a means to improve performance of PQ use.

This document addresses the third topic of these work items.

2. About This Document

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

Status information for this document may be found at _[Todo]_.

Discussion of this document takes place on the MLS 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/PairedMLS/draft-pairedMLS.

3. 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."

4. Copyright Notice

5. 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], and [RFC8174] when, and only when, they appear in all capitals, as shown here.

The terms MLS client, MLS member, MLS group, Leaf Node, GroupContext, KeyPackage, Signature Key, Handshake Message, Private Message, Public Message, and RequiredCapabilities have the same meanings as in the [MLS protocol] https://www.rfc-editor.org/rfc/rfc9420.html (https://www.rfc-editor.org/rfc/rfc9420.html).

6. Notation

We use terms from from MLS [RFC9420] and PQ Hybrid Terminology [I-D.ietf-pquip-pqt-hybrid-terminology]. Below, we have restated relevant terms and define new ones:

Application Message: A PrivateMessage carrying application data.

Handshake Message: A PublicMessage or PrivateMessage carrying an MLS Proposal or Commit object, as opposed to application data.

Key Derivation Function (KDF): A Hashed Message Authentication Code (HMAC)-based expand-and-extract key derivation function (HKDF) as described in RFC5869.

Key Encapsulation Mechanism (KEM): A key transport protocol that allows two parties to obtain a shared secret based on the receiver's public key.

Post-Quantum (PQ) MLS Session: An MLS session that uses a PQ-KEM construction, such as described by FIPS 203 from NIST. It may optionally also use a PQ-DSA construction, such as described by FIPS 204 from NIST.

Traditional MLS Session: An MLS session that uses a Diffie-Hellman (DH) based KEM as described in RFC9180.

PQ/T: A Post-Quantum and Traditional hybrid (protocol).

7. The Combiner Protocol Execution

The hybrid PQ MLS (HPQMLS) combiner protocol runs two MLS sessions in parallel, synchronizing their group memberships. The two sessions are combined by exporting a secret from the PQ session and importing it as a Pre-Shared Key (PSK) into the traditional session. This combination process is mandatory for Commits of Add and Remove proposals in order to maintain synchronization between the sessions. However, it is optional for any other Commits (e.g. to allow for less computationally expensive traditional key rotations). Due to the higher computational costs and output sizes of PQ KEM (and signature) operations, it may be desirable to issue PQ combined (a.k.a. FULL) Commits less frequently than the traditional-only (a.k.a. PARTIAL)

Commits. Since FULL Commits introduce PQ security into the MLS key schedule, the overall key schedule remains PQ-secure even when PARTIAL Commits are used. The FULL Commit rate establishes the post-quantum Post-Compromise Security (PCS) window, while the PARTIAL Commit rate can tighten the traditional PCS window even while maintaining PQ security more generally. The combiner protocol design treats both sessions as black-box interfaces so we only highlight operations requiring synchronizations in this document.

The default way to start a HPQMLS combined session is to create a PQ MLS session and then start a traditional MLS session with the exported PSK from the PQ session, as previously mentioned. Alternatively, a combined session can also be created after a traditional MLS session has already been running. This is done through creating a PQ MLS session with the same group members, sending a Welcome message containing the HPQMLSInfo struct in the GroupContext, and then making a FULL Commit as described in in the Commit Flow (Section 7.1) section.

7.1. Commit Flow

Commits to proposals MAY be _PARTIAL_ or _FULL_. For a PARTIAL Commit, only the traditional session's epoch is updated following the Propose-Commit sequence from Section 12 of RFC9420. For a FULL Commit, a Commit is first applied to the PQ session and another Commit is applied to the traditional session using a PSK derived from the PQ session using the DeriveExtensionSecret and hpqmls_psk label (see Key Schedule (Section 9.1)). To ensure the correct PSK is imported into the traditional session, the sender includes information about the PSK in a PreSharedKey proposal for the traditional session's Commit list of proposals. The information about the exported PSK is captured (shown '=' in the figures below for illustration purposes) by the PreSharedKeyID struct as detailed in RFC9420 (https://www.rfc-editor.org/rfc/rfc9420.html#name-pre-shared-keys). Receivers process the PQ Commit to derive a new epoch in the PQ session and then the traditional Commit (which also includes the PSK proposal) to derive the new epoch in the traditional session.

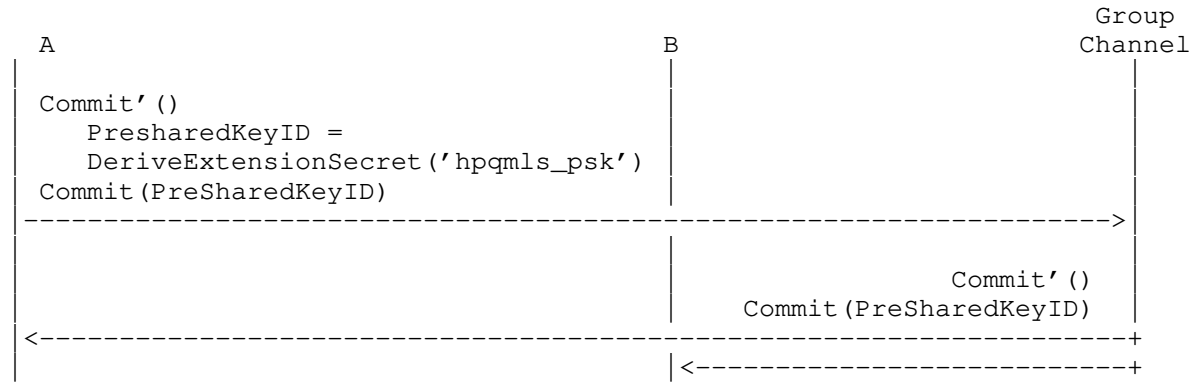

Fig 1a. FULL Commit to an empty proposal list.
Messages with ' are sent in the the PQ session.
PreSharedKeyID identifies a PSK exported from the PQ session in the new epoch following a Commit'(). The PreSharedKeyID is implicitly included in the commit in the classical session via the PreSharedKey Proposal.

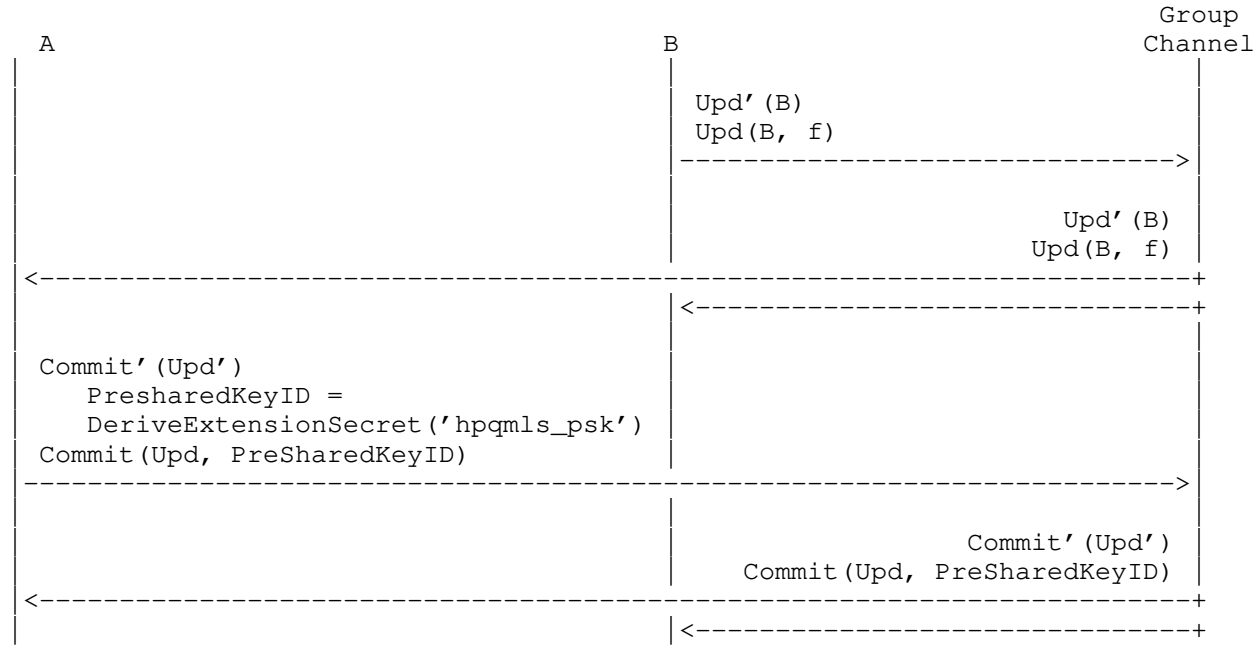

Fig 1b. FULL Commit to an Update proposal from Client B.
Messages with ' are sent in the the PQ session.

Remark: Fig 1b shows Client A accepting the update proposals from Client B as a FULL Commit. The flag f in the classical update proposal Upd(B, f) indicates B's intention for a FULL Commit to whomever Commits to its proposal.

7.2. Adding a User

User leaf nodes are first added to the PQ session following the sequence described in Section 3 of RFC9420 except using PQ algorithms where HPKE algorithms exist. For example, a PQ-DSA signed PQ KeyPackage, i.e. containing a PQ public key, must first be published via the Distribution Service (DS). Then the associated Commit and Welcome messages will be sent and processed in the PQ session according to Section 12 of RFC9420. The same sequence is repeated in the standard session except following the FULL Commit combining sequence where a PreSharedKeyID proposal is additionally committed. The joiner MUST issue a FULL Commit as soon as possible after joining to achieve PCS.

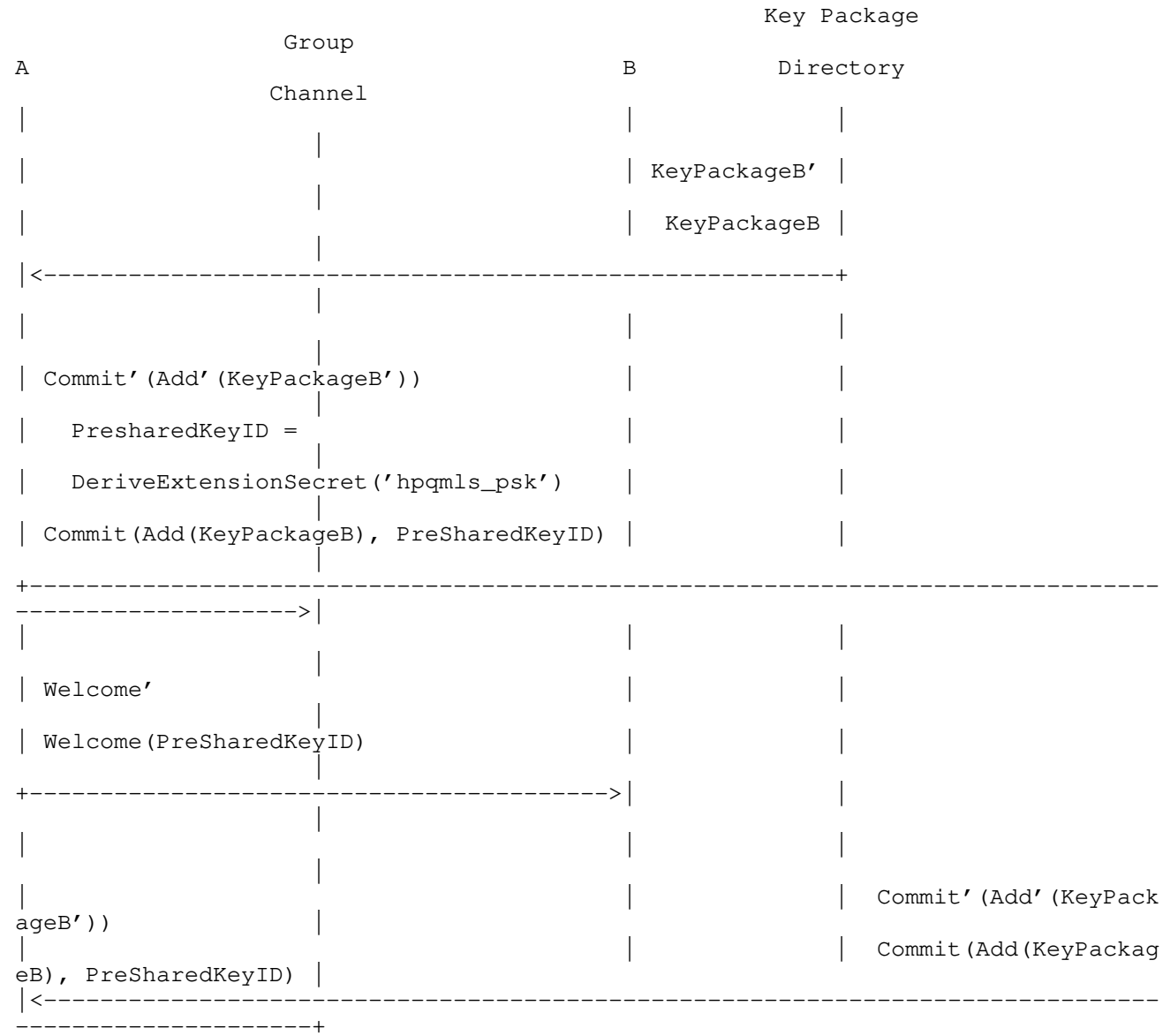

Figure 2:
Client A adds client B to the group.

Messages with ’ come from the PQ session. Processing Welcome and Commit in the traditional
sessio requires the PSK exported exported from the PQ session.

7.2.1. Welcome Message Validation

Since a client must join two sessions, the Welcome messages it receives to each session must indicate that it is not sufficient to join only one or the other. Therefore, the HPQMLSInfo struct indicating the GroupID and ciphersuites of the two sessions MUST be included in the Welcome message via serialization as a GroupContext Extension in order to validate joining the combined sessions. All members MUST verify group membership is consistent in both sessions after a join and the new member MUST issue a FULL Commit as described in Fig 1b.

7.2.2. External Joins

External joins are used by members who join a group without being explicitly added (via an Add-Commit sequence) by another existing member. The external user MUST join both the PQ session and the traditional session. As stated previously, the GroupInfo used to create the External Commit MUST contain the HPQMLSInfo struct. After joining, the new member MUST issue a FULL Commit as described in Fig 1b.

7.3. Removing a Group Member

User removals MUST be done in both PQ and traditional sessions followed by a FULL Commit Update as as described in Fig 1b. Members MUST verify group membership is consistent in both sessions after a removal.

7.4. Application Messages

HPQMLS combiner provides PQ security to the traditional MLS session. Application messages are therefore only sent in the traditional session using the encryption_secret provided by the key schedule of the traditional session according to Section 15 of RFC9420.

8. Modes of Operation

Security needs vary by organizations and system-specific risk tolerance and/or constraints. While this combiner protocol targets combining a PQ session and a traditional session the degree of PQ security may be tuned depending on the use-case: i.e., as PQ/T Confidentiality Only or both PQ/T Confidentiality and PQ/T Authenticity. For PQ/T Confidentiality Only, the PQ session MUST use a PQ KEM, while for PQ authenticity, the PQ session MUST use both a PQ KEM and a PQ DSA. The modes of operation are specified by the mode flag in HPQMLSInfo struct and are listed below.

8.1. PQ/T Confidentiality Only

The default mode of operation is PQ/T Confidentiality Only mode. This mode provides confidentiality and limited authenticity against quantum attackers. More precisely, it provides PQ authenticity against "outsiders", that is, against quantum attackers who do not have acces to (signature) secret keys of any group member. (Authenticity comes from the fact that the traditional session adds AEAD / MAC tags which are not available to outsiders with CRQC.) This mode does not prevent quantum impersonation attacks by other group members. That is, a group member with a CRQC can successfully impersonate another group member.

8.2. PQ/T Confidentiality + Authenticity

The elevated mode of operation is the PQ/T Confidentiality + Authenticity mode. Under a use environment of a cryptographically relevant quantum computer (CRQC), the threat model used in the default mode would be too weak and assurance about update authenticity is required. Recall that authenticity in MLS refers to three types of guarantees: 1) that messages were sent by a member of the group provided by the computed symmetric group key used in AEAD, 2) that key updates were performed by a valid member of the group, and 3) that a message was sent by a particular user (i.e. non-repudiation) provided by digital signatures on messages. While the symmetric group key used for AEAD in the traditional session remains protected from a CRQC adversary through the PSK from the PQ session, signatures would not be secure against forgery without using a PQ DSA to sign handshake messages nor are application messages assured to have non-repudiation against a CRQC adversary. Therefore, in the PQ/T Confidentiality + Authenticity mode, the PQ session MUST use a PQ DSA in addition to PQ KEM ciphersuites for handshake messages (the traditional session remains unchanged).

This version of PQ authenticity provides PQ authenticity to the PQ session's MLS commit messages, strengthening assurance for (1) and ensuring (2). These in turn provide PQ assurance for the key schedule from which application keys are derived in the traditional session. Application keys are used in an AEAD for protection of MLS application messages and thereby inherit the PQ security. However, it should be noted that PQ non-repudation security for application messages as described by (3) is not achieved by this mode. Achieving PQ non-repudiation on application messages would require hybrid signatures in the traditional session, with considerations to options described in [I-D.hale-pquip-hybrid-signature-spectrums].

9. Extension Requirements to MLS

The HPQMLSInfo struct contains characterizing information to signal to users that they are participating in a hybrid session. This is necessary both functionally to allow for group synchronization and as a security measure to prevent downgrading attacks to coax users into parcipating in just one of the two sessions. The group_id, cipher_suite, and epoch from both sessions (t for the traditional session and pq for the PQ session) are used as bookkeeping values to validate and synchronize group operations. The mode is a boolean value: 0 for the default PQ/T Confidentiality Only mode and 1 for the PQ/T Confidentiality + Authenticity mode.

The HPQMLSInfo struct conforms to the Safe Extensions API (see [I-D.ietf-mls-extensions]). Recall that an extension is called _safe_ if it does not modify base MLS protocol or other MLS extensions beyond using components of the Safe Extension API. This allows security analysis of our HPQMLS Combiner protocol in isolation of the security guarantees of the base MLS protocol to enable composability of guarantees. The HPMLSInfo extension struct SHALL be in the following format:

```
struct{
    ExtensionType HPQMLS;
    opaque extension_data<V>;
    } ExtensionContent;

struct{
    opaque t_session_group_id<V>;
    opaque PQ_session_group_id<V>;
    bool mode;
    CipherSuite t_cipher_suite;
    CipherSuite pq_cipher_suite;
    uint64 t_epoch;
    uint64 pq_epoch;
} HPQMLSInfo
```

9.1. Key Schedule

The hpqmls_psk exporter key derived in the PQ session MUST be derived in accordance with the Safe Extensions API guidance (see 2.1.5 Exporting Secrets in [I-D.ietf-mls-extensions]). In particular, it SHALL NOT use the extension_secret and MUST be derived from only the epoch_secret from the key schedule in [RFC9420] (https://www.rfc-editor.org/rfc/rfc9420.html). This is to ensure forward secrecy guarantees (see Security Considerations (Section 10)).

Even though the hpqmls_psk PSK is not sent over the wire, members of the HPQMLS session must agree on the value of which PSK to use. In alignment with the Safe Extensions API policy for PSKs, HPQMLS PSKs used SHALL set PSKType = 3 and extension_type = HPQMLS (see Section 2.1.6 Pre-Shared Keys in [I-D.ietf-mls-extensions]).

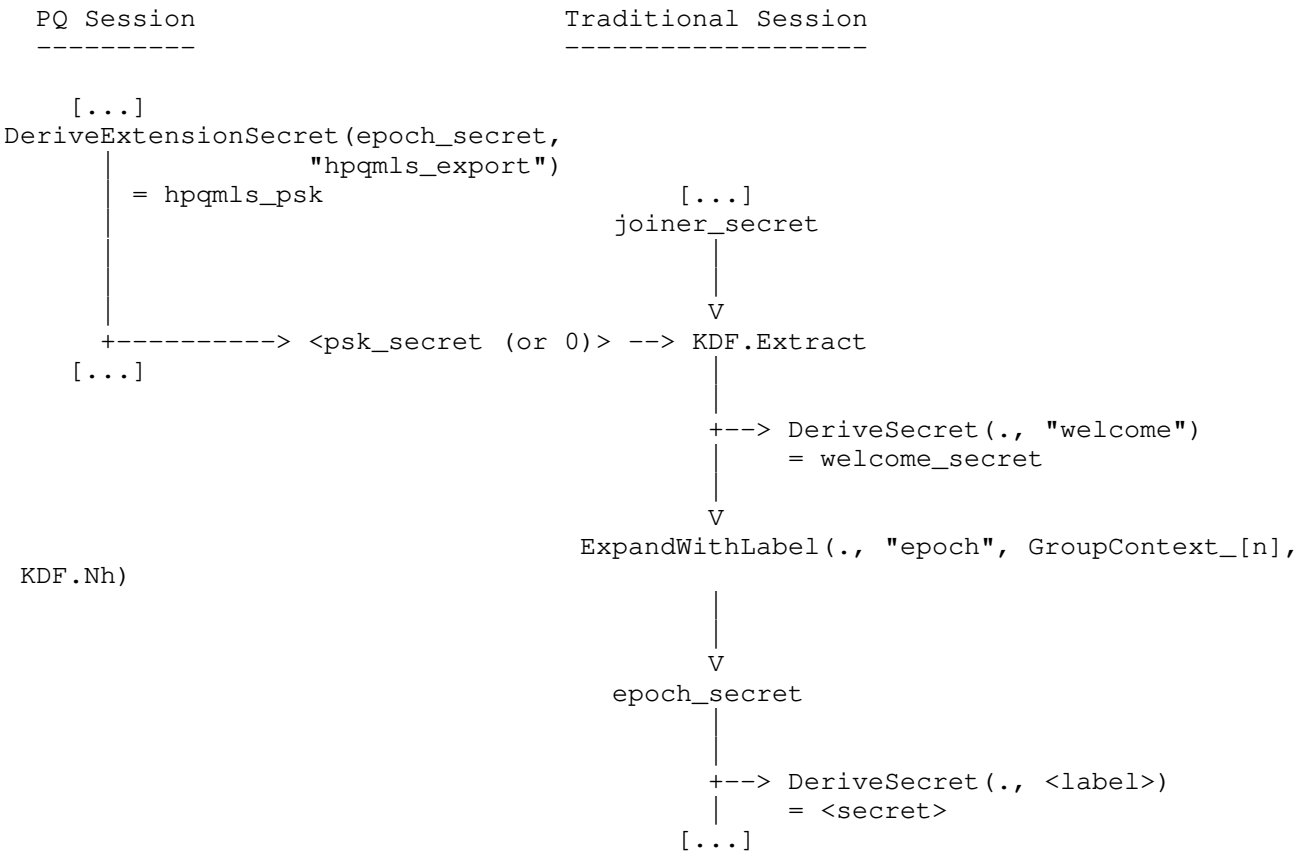

Fig 3: The hpqmls_psk of the PQ session is injected into the key schedule of the traditional session using the safe extensions API DeriveExtensionSecret.

10. Security Considerations

10.1. FULL Commit Frequency

So long as the FULL Commit flow is followed for group administration actions, PQ security is extended to the traditional session. Therefore, FULL Commits can occur as frequently or infrequently as desired by any given security policy. This results in a flexible and efficient use of compute, storage, and bandwidth resources for the host by mainly calling partial updates on the traditional MLS session, given that the group membership is stable. Thus, our

protocol provides PQ security and can maintain a tighter PCS window against traditional attackers as well as forward secrecy window against traditional or quantum attackers with lower overhead when compared to running a single MLS session that only uses PQ KEMs or PQ KEM/DSAs. Furthermore, the PQ PCS window against quantum attackers can be selected based on an application and even variable over time, ranging from e.g. a single FULL Commit in PQ/T Confidentiality Only mode followed by PARTIAL Commits from that point onwards (enabling general PQ/traditional confidentiality, traditional update authenticity, traditional PCS, and PQ/traditional forward secrecy) to frequent FULL Commits in the same mode (enabling general PQ/traditional confidentiality, traditional update authenticity, PQ/traditional PCS, and PQ/traditional forward secrecy). In PQ/T Confidentiality + Authenticity mode with frequent FULL Commits, the latter case would enable general PQ/traditional confidentiality, PQ/traditional update authenticity, PQ/traditional PCS, and PQ/traditional forward secrecy.

10.2. Attacks on Authentication

While PQ message integrity is provided by the symmetric key used in AEAD, attacks on non-repudiation (e.g., source forgery) on application messages may still be possible by a CRQC adversary since only traditional signatures on used after the AEAD. However, in terms of group key agreement, this is insufficient to mount anything more than a denial-of-service attack (e.g. via group state desynchronization). In terms of application messages, a traditional DSA signature may be forged by an external CRQC adversary, but the content (including sender information) is still protected by AEAD which uses the symmetric group key. Thus, an external CRQC adversary can only conduct a false-framing attack, where group members are assured of the authenticity of a message being sent by a group member for the adversary has changed the signature to imply a different sender; it would require an insider CRQC adversary to actually mount a masquerading or forgery attack, which is beyond the scope of this protocol.

If this is a concern, Hybrid PQ DSAs can be used in the traditional session to sign application messages. Since this would negate much of the efficiency gains from using this protocol and denial-of-service attacks can be achieve through more expeditious means, such a option is not considered here.

10.3. Forward Secrecy

Recall that one of the ways MLS achieves forward secrecy is by deleting security sensitive values after they are consumed (e.g. to encrypt or derive other keys/nonces) and the key schedule has entered a new epoch. For example, values such as the init_secret or epoch_secret are deleted at the _start_ of a new epoch. If the MLS exporter_secret or the extension_secret from the PQ session is used directly as a PSK for the traditional session, against the requirements set above, then there is a potential scenario in which an adversary can break forward secrecy because these keys are derived _during_ an epoch and are not deleted. Therefore, the hpqmls_psk MUST be derived from the epoch_secret of the PQ session (see Figure 3) to ensure forward secrecy.

10.4. Transport Security

Recommendations for preventing denial-of-service attacks or restricting transmitted messages are inherited from MLS.

11. IANA Considerations

The MLS sessions combined by this protocol conform to the IANA registries listed for MLS RFC9420.

12. References

12.1. Normative References (i.e. RFCs)

[I-D.ietf-mls-extensions] Robert, R., "The Messaging Layer Security (MLS) Extensions", Work in Progress, Internet-Draft, draft-ietf-mls-extensions-04, 24 April 2024, https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-04 (https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-04)

[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 (https://www.rfc-editor.org/rfc/rfc2119).

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, https://www.rfc-editor.org/rfc/rfc8174 (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 (https://www.rfc-editor.org/rfc/rfc9420).

[I-D.hale-pquip-hybrid-signature-spectrums] Bindel, N., Hale, B., Connolly, D., and F. D, "Hybrid signature spectrums", Work in Progress, Internet-Draft, draft-hale-pquip-hybrid-signature-spectrums-01, 6 November 2023, https://datatracker.ietf.org/doc/html/draft-hale-pquip-hybrid-signature-spectrums-01 (https://datatracker.ietf.org/doc/html/draft-hale-pquip-hybrid-signature-spectrums-01).

[I-D.ietf-pquip-pqt-hybrid-terminology] Driscoll, F., Parsons, M., and Hale, B., "Terminology for Post-Quantum Traditional Hybrid Schemes", Work in Progress, Internet-Draft, draft-ietf-pquip-pqt-hybrid-terminology-04, 18 September 2024, https://datatracker.ietf.org/doc/draft-ietf-pquip-pqt-hybrid-terminology/ (https://datatracker.ietf.org/doc/draft-ietf-pquip-pqt-hybrid-terminology/).

12.2. Informational References

TODO

13. Acknowledgments

Contributors

13.1. Authors

Authors' Addresses

JoÃ«l Alwen
AWS
Email: alwenjo@amazon.com

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

Marta Mularczyk
AWS
Email: mulmarta@amazon.ch

Xisen Tian
Naval Postgraduate School
Email: xisen.tian1@nps.edu