Network Working Group
Internet-Draft
Intended status: Informational
Expires: 3 September 2026

J. Alwen
AWS
K. Kohbrok
Phoenix R&D
B. McMillion

M. Mularczyk
AWS
R. Robert
Phoenix R&D
2 March 2026

MLS Virtual Clients
draft-ietf-mls-virtual-clients-00

Abstract

This document describes a method that allows multiple MLS clients to emulate a virtual MLS client. A virtual client allows multiple emulator clients to jointly participate in an MLS group under a single leaf. Depending on the design of the application, virtual clients can help hide metadata and improve performance.

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 3 September 2026.

Copyright Notice

Table of Contents

1. Introduction

The MLS protocol facilitates communication between clients, where in an MLS group, each client is represented by the leaf to which it holds the private key material. In this document, we propose the notion of a virtual client that is jointly emulated by a group of emulator clients, where each emulator client holds the key material necessary to act as the virtual client.

The use of a virtual client allows multiple distinct clients to be represented by a single leaf in an MLS group. This pattern of shared group membership provides a new way for applications to structure groups, can improve performance and help hide group metadata. The effect of the use of virtual clients depends largely on how it is applied (see Section 3).

We discuss technical challenges and propose a concrete scheme that allows a group of clients to emulate a virtual client that can participate in one or more MLS groups.

2. Terminology

* Virtual Client: A client for which the secret key material is held by one or more emulator clients, each of which can act on behalf of the virtual client.

* Emulator Client: A client that collaborates with other emulator clients in emulating a virtual client.

* Emulation group: Group used by emulator clients to coordinate emulation of a virtual client.

* Higher-level group: A group that is not an emulation group and that may contain one or more virtual clients.

* Simple multi-client: A simple alternative to the concept of virtual clients, where entities that are represented by more than one client (e.g. a user with multiple devices) are implemented by including all of the entities' clients in all groups the entity is participating in.

TODO: Terminology is up for debate. We've sometimes called this "user trees", but since there are other use cases, we should choose a more neutral name. For now, it's virtual client emulation.

3. Applications

Virtual clients generally allow multiple emulator clients to share membership in an MLS group, where the virtual client is represented as a single leaf. This is in contrast to the simple multi-client scenario as defined above.

Depending on the application, the use of virtual clients can have different effects. However, in all cases, virtual client emulation introduces a small amount of overhead for the emulator clients and certain limitations when it comes to emulation group management (see Section 4).

3.1. Virtual clients for performance

If a group of emulator clients emulate a virtual client in more than one group, the overhead caused by the emulation process can be outweighed by two performance benefits.

On the one hand, the use of virtual clients makes the higher-level groups (in which the virtual client is a member) smaller. Instead of one leaf for each emulator client, it only has a single leaf for the virtual client. As the complexity of most MLS operations depends on the number of group members, this increases performance for all members of that group.

At the same time, the virtual client emulation process (see Section 5) allows emulator clients to carry the benefit of a single operation in the emulation group to all virtual clients emulated in that group.

3.2. Metadata hiding

Virtual clients can be used to hide the emulator clients from other members of higher-level groups. For example, removing group members of the emulator group will only be visible in the higher-level group as a regular group update. Similarly, when an emulator client wants to send a message in a higher-level group, recipients will see the virtual client as the sender and won’t be able to discern which emulator client sent the message, or indeed the fact that the sender is a virtual client at all.

Hiding emulator clients behind their virtual client(s) can, for example, hide the number of devices a human user has, or which device the user is sending messages from.

As hiding of emulator clients by design obfuscates the membership in higher-level groups, it also means that other higher-level group members can’t identify the actual senders and recipients of messages. From the point of view of other group members, the "end" of the end-to-end encryption and authentication provided by MLS ends with the virtual client. The relevance of this fact largely depends on the security goals of the application and the design of the authentication service.

If the virtual client is used to hide the emulator clients, the delivery service and other higher-level group members also lose the ability to enforce policies to evict stale clients. For example, an emulator client could become stale (i.e. inactive), while another keeps sending updates. From the point of view of the higher-level group, the virtual client would remain active.

4. Emulation group management

Emulation group is more elaborate than performing simple MLS operation within the emulation group.

When adding a new emulator client, there are several pieces of cryptographic state that need to be synchronized before the new emulator client can start using the virtual client. The emulator client can either get this state from another emulator client, or if all other emulator clients are offline, the emulator client can use a series of external joins to onboard itself.

4.1. Adding an emulator client

When an emulator client adds a new client to the emulation group, the GroupInfo in that Welcome message needs to contain a NewEmulatorClientState component.

TODO: Define component properly. Things that need to be included:

* signing keys (we need to account for per-group and global signature key setups, maybe even leave this to the application?)

* init and encryption keys for active KeyPackages

* epoch ids, (punctured) epoch base secrets and epoch encryption key (as defined in Section 5.2) for active emulation group epochs

* All MLS group secrets for active virtual client groups

  - epoch secrets

- secret tree nodes (including those of potential secret trees of past epochs)

- encryption keys of ratchet tree nodes (including the leaf)

- encryption keys for sent, but uncommitted update proposals

4.2. Joining externally

Without another online emulator client to bootstrap from, a new emulator can join the emulation group externally. A prerequisite for this external join is that the new client has the ability to learn which groups the virtual client is in and to externally join those groups.

If those prerequisites are met, the new client needs to follow these steps:

1. Request a fresh credential for the virtual client with a new signing key

2. Perform an External Join to the emulation group. Send an application message containing a ResyncMessage to the emulation group with the new key.

3. Replace all active KeyPackages with new KeyPackages, generated from the new emulation group epoch.

4. Perform an External Join to all of the groups that the virtual client is a member of, using LeafNodes generated from the new emulation group epoch (see Section 5.2). Welcome messages which were unprocessed by the offline devices are discarded, and these groups are Externally Joined instead (potentially being queued for user approval first).

```
struct {
  opaque signature_private_key<V>;
} ResyncMessage;
```

4.3. Removing emulator clients

To effectively remove an emulator client, it needs to be removed from the emulation group _and_ a commit with an update path needs to be sent into every higher level group by another emulator client using the new emulation group’s epoch to generate the necessary secrets (see Section 5.2). The latter step is required to ensure that the removed emulator client loses its access to any active virtual client secrets.

A corollary of this slightly more elaborate removal procedure is that the removal of an emulator client requires another emulator client to be online and perform the necessary updates. This is in contrast to the simple multi-client setup, where an external sender can effectively remove individual clients.

5. Client emulation

To ensure that all emulator clients can act through the virtual client, they have to coordinate some of its actions.

5.1. Delivery Service

Client emulation requires that any message sent by an emulator client on behalf of a virtual client be delivered not just to the rest of the supergroup to which the the message is sent, but also to all other clients in the emulator group.

5.2. Generating Virtual Client Secrets

Generally, secrets for virtual client operations are derived from the emulation group. To that end, emulator clients derive an epoch_base_secret with every new epoch of that group.

```
emulator_epoch_secret = SafeExportSecret(XXX)
```

TODO: Replace XXX with the component ID.

The emulator_epoch_secret is in turn used to derive four further secrets, after which it is deleted.

```
epoch_id =
  DeriveSecret(emulator_epoch_secret, "Epoch ID")
epoch_base_secret =
  DeriveSecret(emulator_epoch_secret, "Base Secret")
epoch_encryption_key =
  DeriveSecret(emulator_epoch_secret, "Encryption Key")
generation_id_secret =
  DeriveSecret(emulator_epoch_secret, "Generation ID Secret")
```

The epoch_base_secret is then used to key an instance of the PPRF defined in [I-D.ietf-mls-extensions] using a tree with 2^32 leaves.

Secrets are derived from the PPRF as follows:

```
VirtualClientSecret(Input) = tree_node_[LeafNode(Input)]_secret
```

Emulator client MUST store both the (punctured) epoch_base_secret and the epoch_id until no key material derived from it is actively used anymore. This is required for the addition of new clients to the emulator group as described in Section Section 5.4.

When deriving a secret for a virtual client, e.g. for use in a KeyPackage or LeafNode update, the deriving client samples a random octet string random and hashes it with its leaf index in the emulation group using the hash function of the emulation group's ciphersuite.

```
struct {
  u32 leaf_index;
  opaque random<V>;
} HashInput
```

```
pprf_input = Hash(HashInput)
```

TODO: We could also hash in the specific operation to further separate domains.

The pprf_input is then used to derive an operation_secret.

```
operation_secret = VirtualClientSecret(pprf_input)
```

Given an epoch_id, random and the leaf_index of the emulator client performing the virtual client operation, other emulator clients can derive the operation_secret and use it to perform the same operation.

Depending on the operation, the acting emulator client will have to derive one or more secrets from the operation_secret.

There are four types of MLS-related secrets that can be derived from an operation_secret.

* signature_key_secret: Used to derive the signature key in a virtual client's leaf
* init_key_secret: Used to derive the init_key HPKE key in a KeyPackage
* encryption_key_secret: Used to derive the encryption_key HPKE key in the LeafNode of a virtual client
* path_generation_secret: Used to generate path_secrets for the UpdatePath of a virtual client

```
signature_key_secret =
  DeriveSecret(epoch_base_secret, "Signature Key")

encryption_key_secret =
  DeriveSecret(epoch_base_secret, "Encryption Key")

init_key_secret =
  DeriveSecret(epoch_base_secret, "Init Key")

path_generation_secret =
  DeriveSecret(epoch_base_secret, "Path Generation")
```

From these secrets, the deriving client can generate the corresponding keypair by using the secret as the randomness required in the key generation process.

5.3. Creating LeafNodes and UpdatePaths

When creating a LeafNode, either for a Commit with path, an Update proposal or a KeyPackage, the creating emulator client MUST derive the necessary secrets from the current epoch of the emulator group as described in Section Section 5.2.

Similarly, if an emulator client generates an Commit with an update path, it MUST use path_generation_secret as the path_secret for the first parent_node instead of generating it randomly.

To signal to other emulator clients which epoch to use to derive the necessary secrets to recreate the key material, the emulator client includes a DerivationInfoComponent in the LeafNode.

```
struct {
  opaque epoch_id<V>;
  opaque ciphertext<V>;
} DerivationInfoComponent

struct {
  uint32 leaf_index;
  opaque random<V>;
} EpochInfoTBE
```

The ciphertext is the serialized EpochInfoTBE encrypted under the epoch’s epoch_encryption_key with the epoch_id as AAD using the AEAD scheme of the emulation group’s ciphersuite.

When other emulator clients receive an Update (i.e. either an Update proposal or a Commit with an UpdatePath) in group that the virtual client is a member in it uses the epoch_id to determine the epoch of the emulator group from which to derive the secrets necessary to re-create the key material of the LeafNode and a potential UpdatePath.

5.4. Adding emulator clients

There are two ways of adding new clients to the emulation group. Either new clients get sent the secret key material of all groups that the virtual client is currently in, or it joins into all of the virtual client’s groups, either via a regular or an external commit.

TODO: Specify protocol

5.5. Virtual client actions

There are two occasions where emulator clients need to communicate directly to operate the virtual client. In both cases, the acting emulator client sends a Commit to the emulation group before taking an action with the virtual client.

The commit serves two purposes: First, the agreement on message ordering facilitated by the DS prevents concurrent conflicting actions by two or more emulator clients. Second, the acting emulator client can attach additional information to the commit using the SafeAAD mechanism described in Section 4.9 of [I-D.ietf-mls-extensions].

```
enum {
  reserved(0),
  key_package_upload(1),
  external_join(2),
  255,
} ActionType;

struct {
  ActionType action_type;
  select (VirtualClientAction.action_type) {
    case key_package_upload:
      KeyPackageUpload key_package_upload;
    case external_join:
      ExternalJoin external_join;
  };
} VirtualClientAction;
```

5.5.1. Creating and uploading KeyPackages

When creating a KeyPackage, the creating emulator client derives the init_secret as described in Section 5.2.

Before uploading one or more KeyPackages for a virtual client, the uploading emulator client MUST create a KeyPackageUpload message and send it to the emulator group as described in Section 5.5.

The recipients can use the leaf_index of the sender, as well as the random and epoch_id to derive the init_key for each KeyPackageRef. If the recipients receive a Welcome, they can then check which init_key to use based on the KeyPackageRef.

```
struct {
  KeyPackageRef key_package_ref<V>;
  opaque random<V>;
} KeyPackageInfo

struct {
  opaque epoch_id<V>;
  KeyPackageInfo key_package_info<V>;
} KeyPackageUpload
```

After successfully sending the message, the sender MUST then upload the corresponding KeyPackages.

The key_package_refs allow emulator clients to identify which KeyPackage to use and how to derive it when the virtual client receives a Welcome message.

5.5.2. Externally joining groups with the virtual client

Before an emulator client uses an external commit to join a group with the virtual client, it MUST send an ExternalJoin message to the emulation group as described in Section 5.5.

```
struct {
  opaque group_id<V>;
} ExternalJoin
```

The sender MUST then use an external join to join the group with GroupID group_id. When creating the commit to join the group externally, it MUST generate the LeafNode and path as described in Section 5.3.

5.6. Sending PrivateMessages

Given that MLS generates the encryption keys and nonces for PrivateMessages sequentially, but multiple emulator clients may send messages through the virtual client simultaneously, this can create a situation where encryption keys and nonces are reused inappropriately. Critically, if two emulator clients encrypt a message with both the same key and nonce simultaneously, this could compromise the message's confidentiality and integrity. Emulator clients MUST prevent this by computing the reuse_guard, as described below instead of sampling it randomly.

5.7. Small-Space PRP

A small-space pseudorandom permutation (PRP) is a cryptographic algorithm that works similar to a block cipher, while also being able to adhere to format constraints. In particular, it is able to perform a psuedorandom permutation over an arbitrary input and output space.

This document uses the FF1 mode from [NIST] with the input-output space of 32-bit integers, instantiated with AES-128.

output = SmallSpacePRP.Encrypt(key, input)
input = SmallSpacePRP.Decrypt(key, output)

5.8. Reuse Guard

MLS clients typically generate the bytes for the reuse_guard randomly. When sending a message with a virtual client, however, emulator clients choose a random value x such that x modulo the number of leaves in the emulation group is equal to its leaf_index. They then calculate:

```
prp_key = ExpandWithLabel(leaf_node_secret, "reuse guard", key_schedule_nonce, 16)
reuse_guard = SmallSpacePRP.Encrypt(prp_key, x)
```

ExpandWithLabel is computed with the emulation group's ciphersuite's algorithms. leaf_node_secret is the secret corresponding to the virtual client's LeafNode in the higher level group and key_schedule_nonce is the nonce provided by the key schedule for encrypting this message.

prp_key is computed in a way that it is unique to the key-nonce pair and computable by all emulator clients (but nobody else). reuse_guard is computed in a way that it appears random to outside observers (in particular, it does not leak which emulator client sent the message), but two emulator clients will never generate the same value.

5.9. Delivery Service

The method discussed above for computing reuse_guard prevents emulator clients from ever reusing the same key-nonce pair, as this would compromise the message. However, it does not prevent different emulator clients from attempting to encrypt messages with the same key but different nonces. While this doesn't create any security issues, it is a functionality issue due to the MLS deletion schedule. Other higher level group members (or indeed emulator clients) will delete the encryption key after using it to decrypt the first message they receive and will be unable to decrypt subsequent messages.

The best solution depends on whether the Delivery Service is strongly or eventually consistent [RFC9750]. Emulator clients communicating with a strongly-consistent DS SHOULD prevent this issue by coordinating the use of individual ratchet generations for encryption through the DS. Emulator clients MAY send a generation ID to the DS whenever they fan out a private message. The generation ID is derived as follow.

```
enum {
  reserved(0),
  application(1),
  handshake(2),
  (255)
} RatchetType

struct {
  uint32 generation;
  RatchetType ratchet_type;
} PrivateMessageContext

generation_id = ExpandWithLabel(generation_id_secret, "generation id",
                      PrivateMessageContext, Kdf.Nh)
```

* generation is the generation of the ratchet used for encryption

* ratchet_type is the type of ratchet used to encrypt the PrivateMessage

* ExpandWithLabel as defined in [RFC9420]

* generation_id_secret is derived as specified in Section 5.2

* Kdf.Nh is from the emulation group's ciphersuite

Attaching the generation ID to the PrivateMessage allows the DS to detect collisions between generations per epoch and per ratchet type.

Alternatively, devices communicating with an eventually-consistent DS may need to simply retain messages and encryption keys for a short period of time after sending, in case it becomes necessary to decrypt another device's message and re-encrypt and re-send their original message with another encryption key.

6. Security considerations

TODO: Detail security considerations once the protocol has evolved a little more. Starting points:

Some of the performance benefits of this scheme depend on the fact that one can update once in the emulation group and "re-use" the new randomness for updates in multiple higher-level groups. At that point, clients only really recover when they update the emulation group, i.e. re-using somewhat old randomness of the emulation group won't provide real PCS in higher-level groups.

7. Privacy considerations

TODO: Specify the metadata hiding properties of the protocol. The details depend on how we solve some of the problems described throughout this document. However, using a virtual client should mask add/remove activity in the underlying emulation group. If it actually hides the identity of the members may depend on the details of the AS, as well as how we solve the application messages problem.

8. Performance considerations

There are several use cases, where a specific group of clients represents a higher-level entity such as a user, or a part of an organization. If that group of clients shares membership in a large number of groups, where its sole purpose is to represent the higher-level entity, then instead emulating a virtual client can yield a number of performance benefits, especially if this strategy is employed across an implementation. Generally, the more emulator clients are hidden behind a single virtual client and the more clients are replaced by virtual clients, the higher the potential performance benefits.

8.1. Smaller Trees

As a general rule, groups where one or more sets of clients are replaced by virtual clients have fewer members, which leads to cheaper MLS operations where the cost depends on the group size, e.g., commits with a path, the download size of the group state for new members, etc. This increase in performance can offset performance penalties, for example, when using a PQ-secure cipher

suite, or if the application requires high update frequencies (deniability).

8.2. Fewer blanks

Blanks are typically created in the process of client removals. With virtual clients, the removal of an emulator client will not cause the leaf of the virtual client (or indeed any node in the virtual client's direct path) to be blanked, except if it is the last remaining emulator client. As a result, fluctuation in emulator clients does not necessarily lead to blanks in the group of the corresponding virtual clients, resulting in fewer overall blanks and better performance for all group members.

9. Emulation costs

From a performance standpoint, using virtual clients only makes sense if the performance benefits from smaller trees and fewer blanks outweigh the performance overhead incurred by emulating the virtual client in the first place.

10. IANA considerations

This document requests the addition of a new value under the heading "Messaging Layer Security" in the "MLS Component Types" regsitry.

10.1. DerivationInfoComponent

A component meant to communicate information on how to derive secrets for a given commit.

* Value: TBD
* Name: DerivationInfoComponent
* Where: LN
* Recommended: True

10.2. VirtualClientAction

A component meant to communicate which virtual client action is taken in conjunction with the given commit in the emulation group.

* Value: TBD
* Name: VirtualClientAction

* Where: Ad

* Recommended: True

11. Normative References

[I-D.ietf-mls-extensions]
Robert, R., "The Messaging Layer Security (MLS) Extensions", Work in Progress, Internet-Draft, draft-ietf-mls-extensions-08, 21 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-mls-extensions-08>.

[NIST] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", n.d., <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf>.

[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>.

[RFC9750] Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., and A. Duric, "The Messaging Layer Security (MLS) Architecture", RFC 9750, DOI 10.17487/RFC9750, April 2025, <https://www.rfc-editor.org/rfc/rfc9750>.

Authors' Addresses

Jol Alwen
AWS
Email: alwenjo@amazon.com

Konrad Kohbrok
Phoenix R&D
Email: konrad.kohbrok@datashrine.de

Brendan McMillion
Email: brendanmcmillion@gmail.com

Marta Mularczyk
AWS
Email: mulmarta@amazon.ch

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