Network Working Group
Internet-Draft
Intended status: Informational
Expires: 8 January 2026

J. Alwen
K. Kohbrok
R. Robert
Phoenix R&D

7 July 2025

MLS Virtual Clients
draft-kohbrok-mls-virtual-clients-03

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.

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

* Client: Any MLS client including emulator clients, virtual clients and real clients.

* Real Client: An MLS client whose secret key material is held by a single agent.

* Virtual Client: A client for which the secret key material is held by one or more other 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. Emulator clients can be real or virtual clients.

* Heirarchical group: A generalization of an MLS group in which members can be either virtual or real clients. Heirarchical group members may also act as emulator clients to collaboratively emulate a virtual client representing the heirarchical group in one or more other heirarchical groups.

* Group representative: A group representative of (heirarchical group) G is a virtual client emulated by the clients in G. The group representative of group G in another group S is the representative of G that is a member S.

* Subgroup: A heirarchical group with a representative in one or more other groups.

* Supergroup: A heirarchical group with one or more virtual members.

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 case where each individual emulator client is a regular member of the group, each with its own leaf.

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 (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. Hidden subgroups

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.

3.3. Transparent subgroups

TODO: The following text assumes that we have some mechanism of adding one or more additional signatures to MLS messages.

While applications can choose to use virtual clients to hide the corresponding emulator clients, they don’t have to. When using the virtual client to send messages, the sending emulator client can provide an addition signature using either its leaf credential in the emulation group, or another AS-provided credential that allows higher-level group members to authenticate the message.

4. Limitations

The use of virtual clients comes with a few limitations when compared to MLS, where all emulator clients are themselves members of the higher-level groups.

4.1. External remove proposals

In some cases, it is desirable for an external sender (e.g. the messaging provider of a user) to be able to propose the removal of an individual (non-virtual) client from a group without requiring another client of the same user to be online. Doing so would allow another client to commit to said remove proposal and thus remove the client in question from the group.

This is not possible when using virtual clients. Here, the non-virtual client would be the emulator client of a virtual client in a higher-level group. While the server could propose the removal of the client from the emulation group, this would not effectively remove the client's access to the higher-level groups in which the virtual client is a member.

For such a removal to take place, another emulator client would have to be online to update the key material of the virtual client (in addition to the removal in the emulation group).

Another possibility would be for emulator clients to provision KeyPackages for which only a subset of emulator clients have access to. The external sender could then propose the removal of the virtual client, coupled with the immediate addition of a new one using one of the KeyPackages.

4.2. External joins

When there are no subgroups and all (emulator) clients are members of each higher-level group, new (emulator) clients would be able to join via external commit without influencing the operation of any other emulator client and without requiring another emulator client to be online.

When using virtual clients and a client wishes to externally join the emulator group, it will not have immediate access to the secrets of the virtual clients associated with that group.

This can be remedied via one of the following options:

* Another emulator client could provide it with the necessary secrets

* The new emulator client could have the virtual client rejoin all higher-level groups

While the first option has the benefit of not requiring an external commit in any higher-level groups (thus reducing overhead), it either requires another emulator client to be online to share the necessary secrets directly, or a way for the new emulator client to retrieve the necessary without the help of another client. The latter can be achieved, for example, by encrypting the relevant secrets such that the new client can retrieve and decrypt them.

The second option on the other hand additionally requires the new emulator client to re-upload all KeyPackages of the virtual client, thus further increasing the difficulty of coordinating actions between emulation group and higher-level groups.

5. Client emulation

A set C of emulator clients that want to emulate one or more virtual clients must first form an MLS heirarchical group G with membership C. The emulator clients use G to coordinate their shared virtual clients. Just like real clients, a virtual client V can create, join or participate in any group S, even acting as an emulator client itself for some other virtual client. If V joins a group S then this makes G a subgroup of supergroup G where V is called G’s representative in V. G may have 0 or more representatives which can each be a member of 0 or more supergroups. But G can have at most 1 representative in a given supergroup. Emulating clients in G MUST ensure that G and all of its supergroups have distinct group IDs.

An emulator client E in G creates a new virtual client V of G by assigning the V a fresh virtual client ID (unique among all virtual clients of G) and a signature key pair. The new creation of V, its ID and key pair are communicated to rest of G via a commit in G sent by E. As an invariant, emulator client’s in G maintain a copy of the complete local MLS state of V. This includes all MLS related secrets currently held by V. Using this state, each emulator clients can independently process MLS messages sent to V to update their copy of V’s state. Emulator client’s in G can also act on behalf of V (subject to application policy) by taking a new action. Possible actions include anything an MLS client can perform such as generating and publishing a new key packages and sending commits, proposals, welcome messages or application messages. To help other members of G update their copies of V’s state according to the action, E anounces the action using a commit to G. Any secrets created by E as part of implementing the action are generated deterministically by exporting seeds from G. This allows other emulator clients in G to reproduce the same secrets and update their own copies of the V’s state maintaining the invariant.

OPEN QUESTION: It’s also conceivable that emulator clients announce their actions via application messages. This is sufficient for operations that are affect individual groups, because the DS of that group will enforce message ordering.

5.1. Generating Virtual Client Secrets

An emulator client V in a group G may sample four types of MLS-related secrets on behalf of a virtual client V which must be reproducable by the other clients in G: init_key KEM keys in KeyPackage structs, encryption_key KEM keys in LeafNode structs, path_secrets for an UpdatePath structs and signature key pairs. In each case, to do this V (and all other clients in G) do this by constructing an appropriate label for the new secret and exporting from G with the label.

5.2. DS/AS Details

Virtual client emulation should be largely agnostic to specific details of the AS and DS of the application. However, a few conditions must be met.

* Access control: All emulator clients must be able to act as the virtual client, including, for example, queue access and KeyPackage upload

* Queue compatibility: The queue system must allow all emulator clients to retrieve messages for the virtual clients. (Although workarounds like one emulator client retrieving messages and then sending them to the emulation group are possible.)

5.3. Adding emulator clients

If a client is added to the emulation group, it has to be provisioned with the private key material and the group states of all higher-level groups. While the latter might be able to be provisioned by the higher-level DS, the former has to be provided by another emulator client.

The other emulator client can provide the secret key material used to derive all key material relevant to the virtual client (higher-level group secrets, KeyPackage secrets, etc.)

TODO: This means that all such key material must be derived in a well-separated and forward-secure way. (See TODO above to specify further details on how to derive key material for the virtual client.)

Since the new emulator client can only emulate the virtual client if it has access to those secrets, it cannot join the emulation group via external commit, except if said secrets are provided asynchronously.

5.4. Sending application messages

MLS applications messages are encrypted using key material derived from the secret tree, where a unique key/nonce pair is derived for each message and irrevocably deleted after the message was encrypted or decrypted.

This poses a problem in the context of virtual client emulation, because the use of such key material cannot easily be coordinated between emulating clients. However, reusing a key/nonce pair for different application messages leaks information about the plaintexts. Moreover, any client receving the two would not be able to decrypt the second message as the requisit key would already be deleted.

5.5. Challenge-based application message encryption

This problem can be solved by introducing a new type of application message where the encryption keys are derived using a challenge-based approach.

Using a forward-secret exporter secret (provided by the safe extension API), each member creates a new secret tree. Whenever a group member wants to send a message, it creates a fresh random challenge (see Section 5.5.1) for that message. Each challenge is mapped to its own secret using a forward-secure KDF implimented using a new secret tree (see Section 5.5.6). The secret is used to derive the key/nonce used to encrypt a message. The sender includes the challenge in the AAD of the application message so that receivers can also derive the decryption key. Finally, to ensure forward secrecy of the challenge-based application message both sender and recipients apply the same deletion schedule as for the standard secret tree in normal MLS.

5.5.1. Challenge generation

To send an application message the sender must first sample a challenge. It is crucial for application message confidentiality that challenges have high entropy and are never used more than once. The following method for sampling challenges ensures this by introducing sufficient entropy and guaranting that two challenges can only ever be the same if they are sampled by the same sender, in the same epoch, for the same message generation using the same entropy.

For each epoch in which a client wants to send challenge-based application messages it maintains a local uint32 message generation counter for the epoch. The counter is initilized to 0 and incremented after each challenge is sampled. If the counter wraps around to 0 then all subsequent attempts by the client to send in the epoch MUST result in a failure.

uint32 generation;

To sample a challenge the sender first samples AEAD.Nk uniform random octets called the challenge-seed. Next the they populate a ChallengeContext including their leaf index in the group in which the message is sent, the current generation counter and the confirmation tag of the current epoch (which in turn contains the hashed group context).

Additionally, the sender also includes their leaf index and the confirmation tag of each hierarchical group between the sending (virtual) client and the real client that actually samples the entropy.

The application may supply further context in the applicaiton_context field. Finally, the challenge is derived from the challenge-seed and ChallengeContext using the FS-KDF and the generation counter is incremented.

```
struct {
  optional<GroupChallengeContext> subgroup_context;
  uint32 leaf_index;
  MAC confirmation_tag;
} GroupChallengeContext

struct {
  GroupChallengeContext group_challenge_context;
  uint32 generation;
  opaque application_context<V>;
} ChallengeContext

challenge = FS-KDF.Expand(challenge-seed, ChallengeContext, KDF.Nh)
```

5.5.2. Message Framing

The following enum and structs define the wire format for challenge-based application messages.

```
   struct {
     opaque challenge<V>;
     uint32 sender_index;
   } CBAMSender;

   struct {
       ProtocolVersion version = mls10;
       WireFormat wire_format;
       CBAMPrivateMessage private_message;
   } CBAMMLSMessage;
```

5.5.3. Message Authentication

The following structs are used to authenticate data in a challenge-based application message.

```
  // See the "MLS Wire Formats" IANA registry for values
  uint16 WireFormat;

  struct {
    opaque group_id<V>;
    uint64 epoch;
    CBAMSender sender;
    opaque authenticated_data<V>;
    opaque application_data<V>;
  } CBAMFramedContent

  struct {
    ProtocolVersion version = mls10;
    WireFormat wire_format;
    CBAMFramedContent content;
    GroupContext context;
  } CBAMFramedContentTBS;

  struct {
    /* SignWithLabel(., "CBAMFramedContentTBS", CBAMFramedContentTBS) */
    opaque signature<V>;
  } CBAMFramedContentAuthData;

  struct {
    WireFormat wire_format;
    CBAMFramedContent content;
    CBAMFramedContentAuthData auth_data;
  } CBAMAuthenticatedContent;
```

Challenge-based application messages are encoded, authenticated and encrypted much like MLS private messages using the CBAMPrivateMessage struct.

```
struct {
    opaque group_id<V>;
    uint64 epoch;
    opaque authenticated_data<V>;
    opaque encrypted_cbam_sender_data<V>;
    opaque cbam_encrypted_cbam_private_message_content<V>;
} CBAMPrivateMessage;
```

5.5.4. Content Encryption

Content to be encrypted is encoded with a CBAMPrivateMessageContent and the Additional Authenticated data is encoded with a CBAMPrivateContentAAD. The key and nonce used for encryption are derived from the encryption_secret and the challenge C using the challenge-based secret tree as described in Section 5.5.6.

```
struct {
  opaque application_data<V>;
  CBAMFramedContentAuthData auth_data;
  opaque padding[length_of_padding];
} CBAMPrivateMessageContent;

struct {
  opaque group_id<V>;
  uint64 epoch;
  opaque cbam_authenticated_data<V>;
} CBMAPrivateContentAAD;

aead_key = aead_key[C]

aead_nonce = aead_nonce[C]
```

5.5.5. Sender Data Encryption

The encrypted_cbam_sender_data is obtained by encrypting the CBAMSenderData using keys derived from the cbam_sender_data_secret. This secret is exported using the safe API’s forward-secure exporter function MLS-FS-Exporter using the label "CBAM Sender Data Secret" and an empty context. Other than that, the same method is used to encrypt sender data as for standard application messages.

```
cbam_sender_data_secret = MLS-FS-Export("CBAM Sender Data Secret", "", KDF.Nk)

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

sender_data_key = ExpandWithLabel(cbam_sender_data_secret, "key",
                      ciphertext_sample, AEAD.Nk)
sender_data_nonce = ExpandWithLabel(cbam_sender_data_secret, "nonce",
                      ciphertext_sample, AEAD.Nn)
```

5.5.6. Forward Secure KDF

A consequence of the secret tree structure in MLS is that deriving the key/nonce for a given application message requires knowing the leaf node of the client.

The symmetric ratchets in MLS require performing as many (KDF and storage) operations as application messages are being skipped. The challenge-based secret tree (CBST) described in this section avoids these issues. Like the secret tree in MLS, it consists of a binary tree of secrets. However, leaves are indexed by challenges instead of leaf nodes which means the tree now has depth KDF.Nh.

Nodes in the CBST are identified by the string encoding the path from the root to the node. The root is identified by the empty string "". If a node is identified by string N then its left child is identified the string N||0 and the right child by the string N||1. Each node is assigned a secret. The root is assigned the cbam_encryption_secret which is exported from the MLS session using the safe API's FS-Export function. All other nodes in the CBST are assigned a secrety by applying ExpandWithLabel to its parents secret with appropriate labels.

```
cbst_encryption_secret = MLS-FS-Export("CBST", "", KDF.Nh)

cbst_tree_node_[""]_secret = cbst_encryption_secret

cbst_tree_node_[N]_secret
        |
        |
        +--> ExpandWithLabel(., "CBST", "left", KDF.Nh)
        |    = cbst_tree_node_[left(N)]_secret
        |
        +--> ExpandWithLabel(., "CBST", "right", KDF.Nh)
             = cbst_tree_node_[right(N)]_secret
```

The key and nonce for a KDF.Nh octet long challenge C are derived from the secret for leaf node identified by C.

```
aead_key[C] = ExpandWithLabel(cbst_tree_node[C]_secret, "CBST", "key", KDF.Nh)

nonce_key[C] = ExpandWithLabel(cbst_tree_node[C]_secret, "CBST", "nonce",
KDF.Nh)
```

The same deletion schedule applies to the CBST (including the cbst_encryption_secret) as for the secret tree in MLS.

5.6. Rotation of authentication key material

If the design of the AS specifies the use of cross-group authentication key material, emulator clients must coordinate the rotation of said key material in the emulation group to avoid multiple emulator clients rotating a key at the same time. Details depend on the design of the AS.

5.7. Example protocol flow

Virtual clients can, for example, be used by users with multiple devices. Here, each device acts as an emulator client that emulates the virtual client which represents Alice towards other users.

A group with Alice and Bob would thus still only have two members, regardless of the number of clients Alice and Bob have.

Each of Alice’s devices would thus keep the state of one emulator client, as well as the virtual client jointly emulated by all of Alice’s clients.

If one of Alice’s devices wanted to update its key material to achieve post-compromise security, it would first perform a commit in the emulation group, both to signal the action to other emulator clients and to update the key material from which the randomness for the virtual client is sampled. From the updated emulation group, the emulator client would then export the randomness to perform an update in each group in which Alice (through the virtual client) is a member.

Alice’s other clients would receive and process the commit in the emulation group. Using the information included about the virtual client operation, they also update their virtual client state.

Bob (or other members in the higher level group) will only see the update in the higher-level group, which they can simply process with their (virtual) clients.

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.

Authors' Addresses

J. Alwen
Email: alwenjo@amazon.com

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

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