MLS@IETF102

WG Info: https://datatracker.ietf.org/wg/mls/about/
Chairs: Nick Sullivan & Sean Turner

NOTE WELL

This is a reminder of IETF policies in effect on various topics such as patents or code of conduct. It is only meant to point you in the right direction. Exceptions may apply. The IETF's patent policy and the definition of an IETF "contribution" and "participation" are set forth in BCP 79; please read it carefully.

As a reminder:

- By participating in the IETF, you agree to follow IETF processes and policies.
- If you are aware that any IETF contribution is covered by patents or patent applications that are owned or controlled by you or your sponsor, you must disclose that fact, or not participate in the discussion.
- As a participant in or attendee to any IETF activity you acknowledge that written, audio, video, and photographic records of meetings may be made public.
- Personal information that you provide to IETF will be handled in accordance with the IETF Privacy Statement.

As a reminder:

- As a participant or attendee, you agree to work respectfully with other participants; please contact the ombudsteam (https://www.ietf.org/contact/ombudsteam/) if you have questions or concerns about this.

Definitive information is in the documents listed below and other IETF BCPs. For advice, please talk to WG chairs or ADs:

- BCP 9 (Internet Standards Process),
- BCP 25 (Working Group processes),
- BCP 25 (Anti-Harassment Procedures),
- BCP 54 (Code of Conduct),
- BCP 78 (Copyright),
- BCP 79 (Patents, Participation),
- https://www.ietf.org/privacy-policy/ (Privacy Policy)

Requests

Minute Taker(s)

Jabber Scribe(s)

Sign Blue Sheets

State your name @ the mic

Keep it professional @ the mic

Agenda

Time - Duration

10min Administrivia (Chairs)

5min Charter Review (10K ft level)

10min Architecture

25min Handshake message ordering / server trust

25min ART vs. TreeKEM vs. both

25min Message Protection

25min Authentication

25min A.O.B.

- Versioning / extensibility
- Interop testing framework
- Interim plans

Charter Review (10K ft level)

Charter Review (key excerpts)

The primary goal of this working group is to develop a standard messaging security protocol for human-to-human(s) communication with the above security and deployment properties so that applications can share code, and so that there can be shared validation of the protocol (as there has been with TLS 1.3).

It is not a goal of this group to enable interoperability/federation between messaging applications beyond the key establishment, authentication, and confidentiality services.

While authentication is a key goal of this working group, it is not the objective of this working group to develop new authentication technologies.

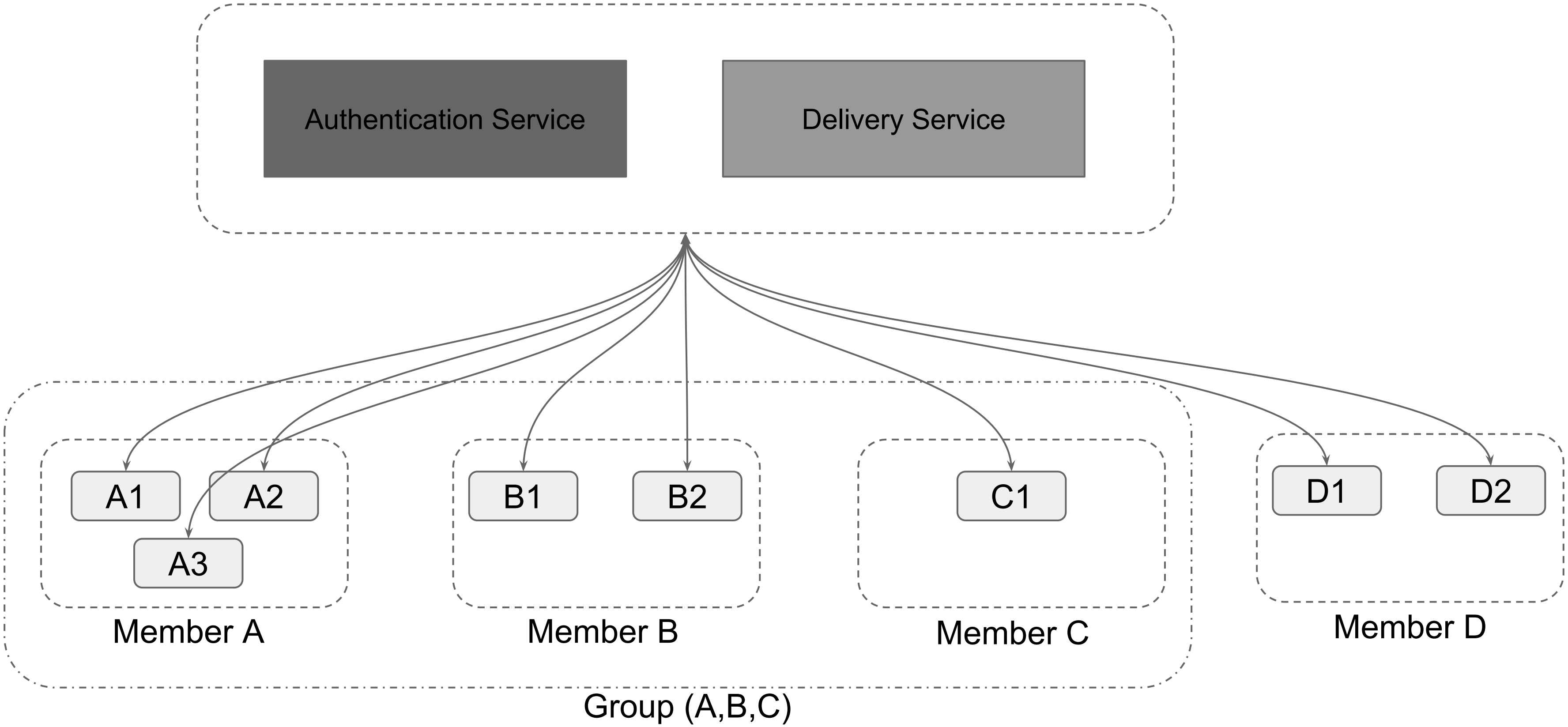
Architecture Review

Architecture

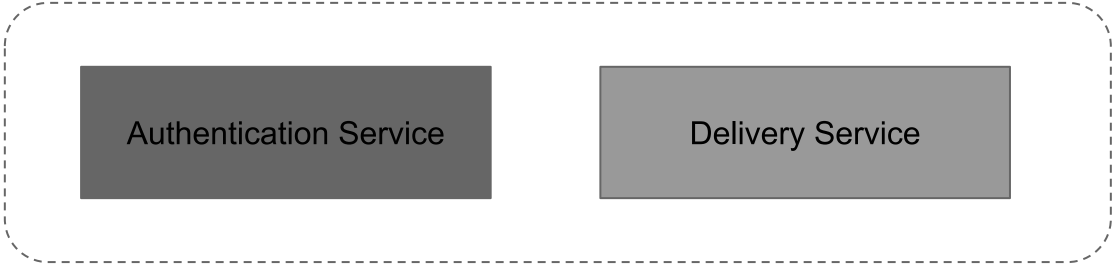
IETF 102

emadomara@google.com

System Overview



System Overview

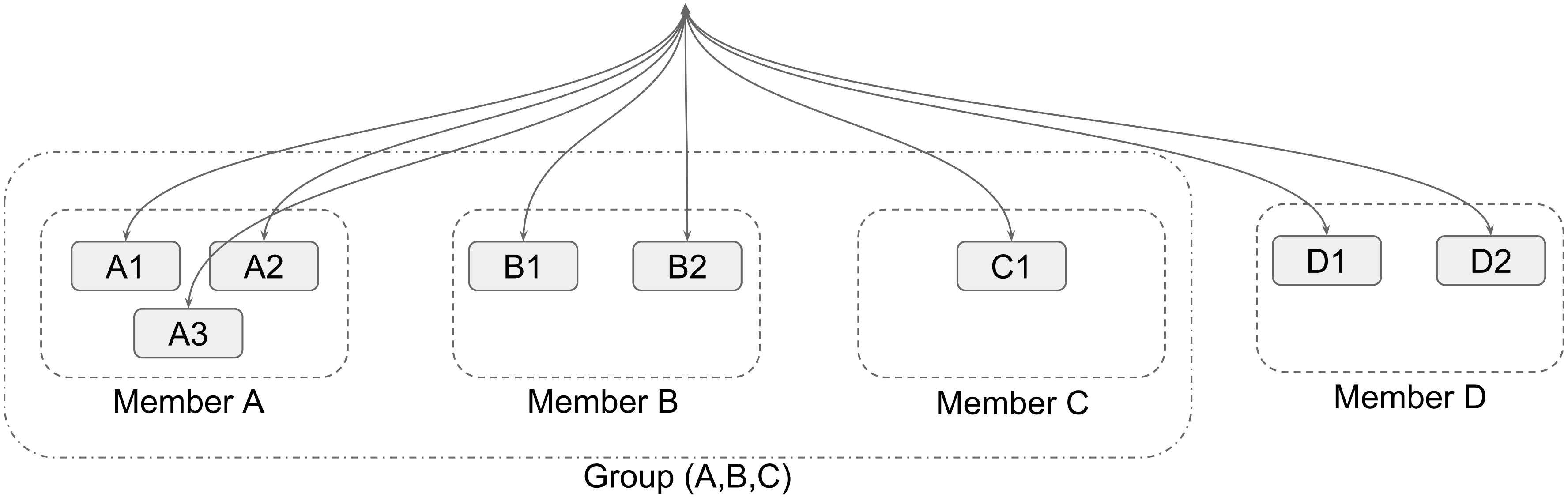


- Stores user ids to identity key mappings

- Distributes and delivers messages and attachments
- Stores initial key materials (initKeys)
- *Stores group membership

System Overview

- Register
- Send message
- Invite member
- Join group
- Add device
- Create group
- Receive message
- Remove member
- Leave group
- Remove device



Functional Requirements

- Scalable
  - Support group size up to 50,000 clients
- Asynchronous
  - All client operations can be performed without waiting for the other clients to be online
- Multiple devices
  - Devices are considered separate clients
  - Restoring history after joining is not allowed by the protocol, but Application can provide that.
- State recovery
  - Lost/Corrupted state must be recovered without affecting the group state.
- Metadata collection
  - AS/DS must only store data required for message delivery
- Federation
  - Multiple implementation should be able to interoperate
- Versioning
  - Support version negotiation

Security Requirements

- Message secrecy, integrity and authentication
  - Only current group member can read messages
  - Messages are only accepted if it was sent by a current group member
  - *Message padding to protect against traffic analysis
- Forward secrecy and post compromise security
- Group membership security
  - Consistent view of group members
  - Added clients can’t read messages sent before joining
  - Removed clients can’t read messages sent after leaving
- Attachments security
- Data origin authentication and *deniability

Security Considerations

- Delivery service compromise
  - Must not be able to read or inject messages
  - Modified, reordered or replayed messages must be detected by the clients
  - It can mount various DoS attacks.

- Authentication service compromise
  - Can return incorrect identities to the client
  - Can’t be defeated without transparency logging such as KT

- Client compromise
  - Can read and send messages to the group for a period of time
  - It shouldn’t be able to perform DoS attack.
  - Will be defeated once the compromised client updates their key material

Open Questions ???

- Should the draft define the frequency of key update or keep it open to the application?
- Should the protocol hide the user devices to protect their privacy?
- Is the server trusted to store group membership?
- Should the draft include section for traffic analysis mitigation (ex message padding) ?

Handshake Message Ordering

Handshake Messages Ordering

IETF 102

emadomara@google.com

Handshake messages

User messages that change the group state

1. Group Init
2. User Add
3. Group Add
4. Remove
5. Update

- Each handshake message is premised on a given starting state, indicated in its "prior_epoch" field.

Conflicts: X--> ? <--Y

- Conflicts happen when two or more clients generate handshakes messages simultaneously, based on the same state.

- Conflicts can be resolved in two approaches
    a. Server-forced ordering
    b. Client-forced ordering

Starvation

- Both server-forced and client-forced ordering can cause starvation in a busy group where a given client may never be able to send a handshake message.

- This problem is specific to ART only. TreeKEM merges concurrent update without rejecting them.

Server-forced ordering

- Messages will have an authenticated sequential counter (epoch)
- The delivery server will dispatch them in order.
- If two messages share the same counter, the server is trusted to choose to process one of them and reject the other.
- The rejected client will retry after updating its state.

Client-forced ordering

- Two steps update protocol
- Step 1: Propose the update
- Step2: Send the update if it gets approved by 50%+ by other clients
- What if most of the clients are offline ?

Open Questions ???

- Should the Architecture document cover this problem ? (Currently discussed in the protocol document)

- Is the delivery server trusted to force the ordering?

- Is “Starvation” a real problem in practice?

ART vs. TreeKEM

Messaging Layer Security

ART vs TreeKEM

Jon Millican
jmillican@fb.com

Protocol Operations

- Group state at each point in time is stored in a key tree
- Each participant caches a view of the tree
- Protocol operations update the participants’ view of the tree
  - Group Creation
  - Group-initiated Add
  - User-initiated Add
  - Key Update
  - Remove

Asynchronous Ratcheting Tree

- (Cohn-Gordon et al., 2017, https://eprint.iacr.org/2017/666.pdf)
- Based on a Diffie-Hellman binary key tree.
- Updates to any leaf in logarithmic time.
- Asynchronous operation.
- Proofs of confidentiality of group keys in static groups.
- MLS defines some things that the original paper leaves out of scope:
  - More constraints on tree structure
  - Membership changes.
  - Race conditions.

DH output -> DH key pair

- `Derive-Key-Pair` maps random bit strings to DH key pairs
- Resulting private key known both original private key holders

```
  AB = Derive-Key-Pair(DH(A,B))
 /  \
A    B



e.g.:
  Derive-Key-Pair(X) = X25519-Priv(SHA-256(X))
```

DH Trees

```
             ABCDEFG
           /         \
          /           \
      ABCD             EFGH
      /  \             /  \
    AB    CD         EF    |
   / \   / \        / \    |
  A   B C   D      E   F   G
```

Part	Role
Root	Group Key
Direct Path	Update
Copath	Add
Frontier	Add

```
leaf + copath -> root
frontier = copath(next)
```

Group Evolution

```
             +-> Msg Secret               +-> Msg Secret               +-> Msg Secret
             |                            |                            |
... --> KDF -+-> Init Secret --> KDF -+-> Init Secret --> KDF -+-> Init Secret -->
...
         ^                            ^                            ^
         |                            |                            |
       Update                       Update                       Update
       Secret                       Secret                       Secret
         ^                            ^                            ^
         |                            |                            |
        Tree                         Tree                         Tree
        Root                         Root                         Root
```

Operation 0: Create group

- Can be created directly.
- Can be created by starting with an one-member group, then doing add operations.
- Current draft does the latter, so there’s no protocol for creation.
- ART paper specifies the former, but we don’t use in the draft yet.

```
             ABCDEFG
            /       \
           /         \
       ABCD           EFGH
      /    \         /    \
    AB      CD     EF      |
   /  \    /  \   /  \     |
  A    B  C    D E    F    G
```

Operation 1: Group-Initiated Add

```
struct {
    UserInitKey init_key;
} GroupAdd;

// Pre-published UserInitKey for
// asynchronicity

// NB: Add Key has implications
// for removals; “double join”
```

```
          ABCD
         /    \
       AB      CD
      /  \    /  \
     A    B  C    D
                 / \
         Add Key    Init
```

Operation 2: User-Initiated Add

```
struct {
    DHPublicKey add_path<1..2^16-1>;
} UserAdd;

// Pre-published frontier in
// GroupInitKey for asynchronicity
```

```
        ABCD
      /      \
    AB        CD
   /  \      /  \
  A    B    C    D
```

Operation 3: Key Update (for PCS)

```
struct {
    DHPublicKey
ratchetPath<1..2^16-1>;
} Update;

// This approach to confidentiality
// is proved in [ART]
```

```
          ABCD
         /    \
      AB        CD
     /  \      /  \
    A    B    C    D
```

Operation 4: Remove

```
struct {
    uint32 deleted;
    DHPublicKey path<1..2^16-1>;
} Delete;

// To lock out, update to a key the
// deleted node doesn't know

// "Double join" issues similar to
// GroupAdd
```

```
        AXCD
       /    \
    AX       CD
   /  \     /  \
  A    X   C    D
```

TreeKEM - an alternative ratcheting tree

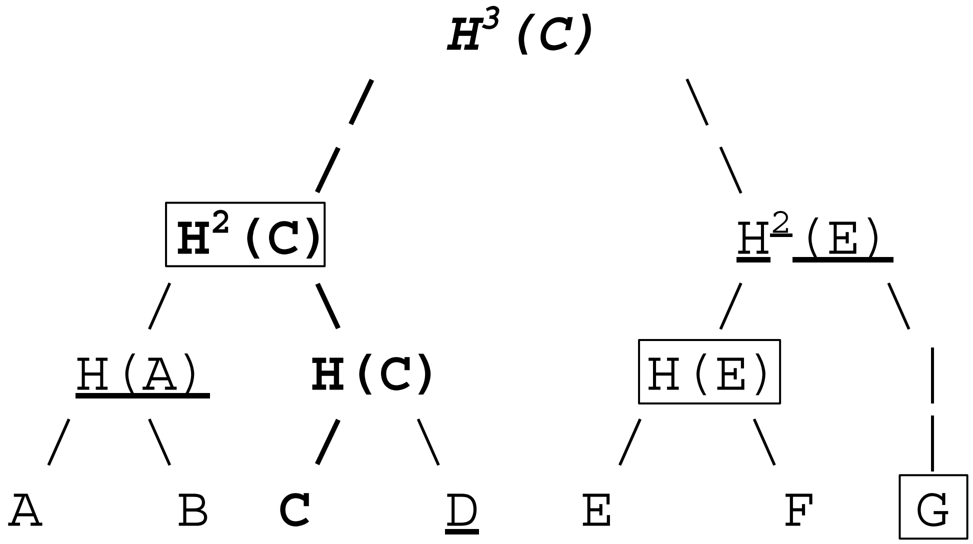
- (Barnes, Bhargaven, Rescorla, 2018, https://www.ietf.org/mail-archive/web/mls/current/msg00117.html)
- New tree-based primitive.
- Based on non-contributive hashing, instead of Diffie-Hellman.
- Encrypts parent nodes to their children, instead of deriving them.
- Most properties directly analogous to ART.
- Key potential improvements over ART:
  - Merging simultaneous updates might be better supported.
  - Receive updates only O(1) public key operations.

Hash output -> public key pair

- `Derive-Key-Pair-TreeKEM` maps random bit strings to public key pairs
- TreeKEM abstracts away the specific algorithm

```
  H(A) = Derive-Key-Pair-TreeKEM(Hash(A))
 /    \
A      B


e.g.:
  Derive-Key-Pair(X) = X25519-Priv(SHA-256(X))
```

TreeKEM Trees - same structure as ART



Part	Role
Root	Group Key
Direct Path	Update
Copath	Add
Frontier	Add

leaf + copath -> root

frontier = copath(next)

Group Evolution - matches that of ART

```
             +-> Msg Secret                +-> Msg Secret                +-> Msg Secret
             |                             |                             |
... --> KDF -+-> Init Secret --> KDF -+-> Init Secret --> KDF -+-> Init Secret -->
...
         ^                            ^                            ^
         |                            |                            |
       Update                       Update                       Update
       Secret                       Secret                       Secret
         ^                            ^                            ^
         |                            |                            |
        Tree                         Tree                         Tree
        Root                         Root                         Root
```

TreeKEM Operation 1: Group-Initiated Add

```
struct {
    ciphertext PathKeys<1..2^16-1>;
    ciphertext AddKey;
    DHPublicKey NewUserIdentity;
} GroupAdd;

// AddKey transmitted to pre-published
// UserInitKey

// NB: Add Key still has implications
// for removals; “double join”
```

```
      H^2(Add Key)
        /      \
  H(A)    H(Add Key)
  /   \      /   \
 A     B    C   Add Key
```

Operation 2: User-Initiated Add

```
struct {
    ciphertext PathKeys<1..2^16-1>;
} UserAdd;

// Pre-published frontier in
// GroupInitKey for asynchronicity
```

```
        H²(D)
       /     \
   H(A)       H(D)
   /  \       /  \
  A    B     C    D
```

Operation 3: Key Update (for PCS)

```
struct {
    ciphertext PathKeys<1..2^16-1>;
} Update;
```

```
          H^2(B)
         /      \
    H(B)        H(C)
    /   \       /   \
   A     B     C     D
```

Operation 4: Remove

```
struct {
    uint32 deleted;
    ciphertext PathKeys<1..2^16-1>;
} Delete;

// To lock out, update to the deleted
// node’s path to keys that it doesn’t
// know.

// “Double join” issues similar to
// GroupAdd
```

```
        H^2(X)
        /    \
    H(X)      H(C)
    /  \      /  \
   A    X    C    D
```

Key Comparisons

- Update complexity
  - ART is O(log n) public key operations for every participant.
  - TreeKEM is O(log n) public key operations for sender, O(1) public key operation for everyone else - with O(log n) hashes.
- Concurrent updates
  - Hashing is non-contributive, so TreeKEM can usually sequence in a manner computable to everyone.
  - ART cannot usually handle concurrent updates.
  - Note: TreeKEM’s PCS properties here need to be studied.

Open Issues with both

- Tuning up, proving FS and PCS properties of the operations
  - … especially Add, Remove
- Logistical details, especially around Remove
- Message sequencing
- Message protection, transcript integrity
- Authentication
  - Current draft has a very basic scheme, needs elaboration
  - Deniable authentication?
- *Attachments

Message Protection

MLS Message Protection

Benjamin Beurdouche & al.
benjamin.beurdouche@inria.fr

What's in the draft today...

Nothing yet...

What do we need ?

A. Define an Application Key Schedule (AKS)
   to go from the Epoch secret to the Application messages encryption keys.

B. Define what algorithms to use and on which objects
   to protect Application messages against active network attackers,
   improve resistance to traffic analysis and compromised insiders, if possible.

Handshake Key Schedule

```
              Init Secret [n-1] (or 0)
                    |
                    V
Update Secret -> HKDF-Extract = Epoch Secret
                    |
                    |
                    +--> Derive-Secret(., "msg", ID, Epoch, Msg)
                    |       = message_master_secret          “Application secret”
                    |
                    +--> Derive-Secret(., "add", ID, Epoch, Msg)
                    |       |
                    |       V
                    |    Derive-Key-Pair(.) = Add Key Pair
                    |
                    V
              Derive-Secret(., "init", ID, Epoch, Msg)
                    |
                    V
              Init Secret [n]
```

Application Key Schedule

Two main ways of ratcheting secrets: interleaving or in parallel
independently from what we ratchet (Application secret or message encryption key)

- Group ratcheting of the secret (interleaving)
- Participant/Sender ratcheting of the secret (parallel)

Both need to provide security properties such as:

- Forward Secrecy for secrets / keys / messages
- PCS for secrets / keys / messages
- (Key Compromise Impersonation resistance)

We could relax these properties for keys/messages if we really need to...

Group Ratchet

(TLS-like)

TLS-like per-Group ratchet chain of Application Secret

```
~~~
Application Secret [0] (Group)
...
Application Secret [N-1] (Group)
|
+ ---> HKDF_Expand_Label (., "aead key", [senderA], length)
|      = write_key_[N-1]_[senderA]
|
+ ---> HKDF_Expand_Label (., "aead iv", [senderA], length)
|      = write_iv_[N-1]_[senderA]
|
Derive_Secret(.) =
Application Secret [N] (Group)
|
+ ---> HKDF_Expand_Label (., "aead key", [senderB], length)
|      = write_key_[N]_[senderB]
|
+ ---> HKDF_Expand_Label (., "aead iv", [senderB], length)
|      = write_iv_[N]_[senderB]
|
Derive_Secret(.) =
Application Secret [N+1] (Group)
...
~~~
```

Participant/Sender ratcheting of the secret

TLS-like per-participant Application Key Schedule

Double-Ratchet-like per-participant Application Key Schedule

TLS-like

Initial Participant Application Secret derivation:

```
~~~
Application Secret [0] (Group)
      |      |
      |      ---> HKDF_Extract([participant A]) =
      |                Application Secret [0] (A)
      |
      ----------> HKDF_Extract([participant B]) =
                       Application Secret [0] (B)
~~~
```

Then, for each participant ratchet chain of Application secret:

```
~~~
Application Secret [N-1] (Participant)
|
+ ---> HKDF_Expand_Label (., "aead key", "", length)
|       = write_key_[N-1] (Participant)
|
+ ---> HKDF_Expand_Label (., "aead iv", "", length)
|       = write_iv_[N-1] (Participant)
|
Derive_Secret(.) =
Application Secret [N] (Participant)
...
~~~
```

Double Ratchet-like

```
            message_master_secret [PN]
                      |
                      V
ClientIndex -> Derive-Client-Secret(., .)
               = Client Secret
                      |
                      V
     Constant -> HKDF-Expand
               = Client Chain Secret [n-1] (or 0), Message key, Nonce
```

Alexey Ermishkin

Application Key Schedule

Group ratcheting of the secret (interleaving)

+ Reduced complexity (storage)
+ Improved Forward Secrecy (no unused key stored for a long time)
- Reduced ability to handle out-of-order messaging for high frequency transmissions

Participant/Sender ratcheting of the secret (parallel)

+ Well-known design
+ Able to handle out-of-order messaging
- Higher complexity (storage)
- Weakened Forward Secrecy (if participant never sends)

Application Key Schedule

The choice of the AKS will balance between security, typically FS, and sending rates...

Both these solution have their own incompatible benefits/drawbacks.

- Group ratcheting of the secret (interleaving)
- Participant/Sender ratcheting of the secret (parallel)

There might be a good intermediate based on more Trees...

Message Encryption

Choosing algorithms and the objects to encrypt is somehow less controversial...

We need to handle:

- AEAD for integrity/confidentiality/weak authentication
- Padding of messages to improve resistance to Traffic Analysis
- Encrypt the optional strong authenticating value for privacy considerations (currently a signature but we could do better)

Proposal...

We need to move forward without committing too fast to a design.

A “safe” approach would be to define an initial message protection text based on an approach we expect will work...

Typically, something like using the per-participant ratcheting scheme and AEAD ciphers to encrypt the *optionally padded* *optionally signed* plaintext.

There are two existing PRs that could used for this…

https://github.com/ekr/mls-protocol/pull/54/files
https://github.com/ekr/mls-protocol/pull/50/files

Authentication

MLS Authentication

@ IETF 102

What’s in the draft today

Each participant has a long term identity key

Each UserInitKey is signed by the participant’s identity key

GroupInitKeys include a Merkle tree head over the identity keys in the group

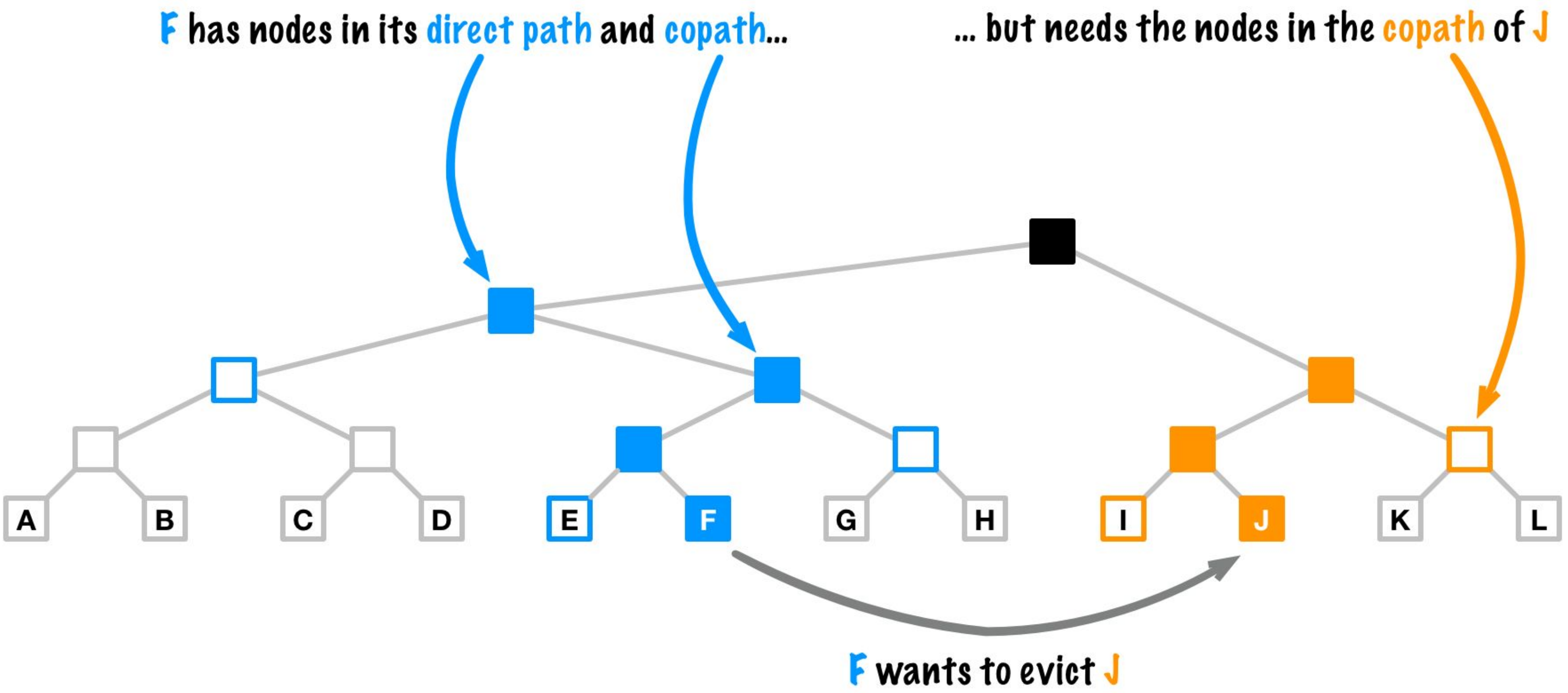
Handshake messages are signed by the sender’s identity key and a Merkle proof of group membership

No credentials => no real identity

What do we need to do here?

```
class Client {

  onconnect(group) { /* verify identities of other members */ }

  onmessage(msg) { /* verify sender identity */ }

  onnewmember(joiner) { /* verify new member identity */ }

  remove(other) {
    /* fetch and verify copath for other */
  }

}
```

Remove() requires knowing more of the tree



“Post-connect” cases are easier

Message authentication: Signatures [+ membership proofs]

Proofs not needed if endpoints cache a validated list of public keys

See Message Protection discussion

Authenticating new joiner: Signature + credential in UserInitKey / UserAdd

Should probably do something SIGMA-like (see next slide)

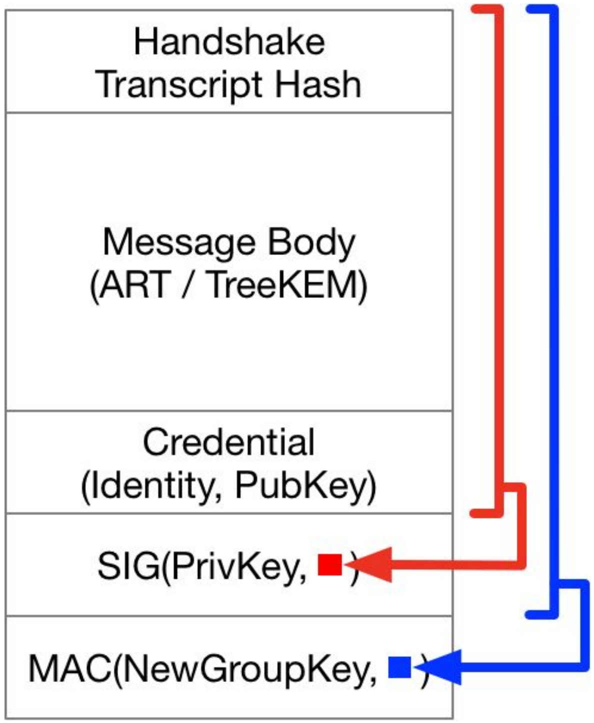
“SIGMA-like”

Incorporate handshake transcript hash

Signature covering prior handshake plus new message

MAC by new group key

Analogous to TLS 1.3 authentication with Certificate + CertificateVerify + Finished



“Initialization” case is more expensive

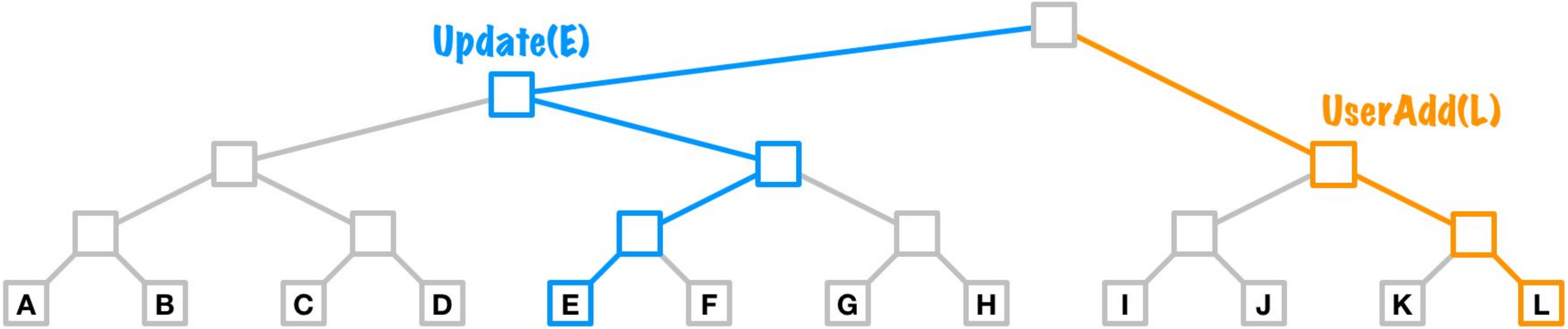
Need to provision new members with O(N) information about the tree…

List of group members’ identities and identity keys (N)

List of public keys for all tree nodes (~2N)

… and enable them to verify that the information is correct

Recall: Handshake messages carry public keys along a direct path



Messages can be used to distribute the tree

If a participant can see the last message sent by each participant, then he has all the direct paths => full view of the tree

O(N log N) data to download

Message signatures authenticate sender and membership

Need to ensure continuity of the sequence, reject injected messages

=> Need all messages until you've covered all participants

… C A A A B A B A B B B A A B A => Ω(N log N) data

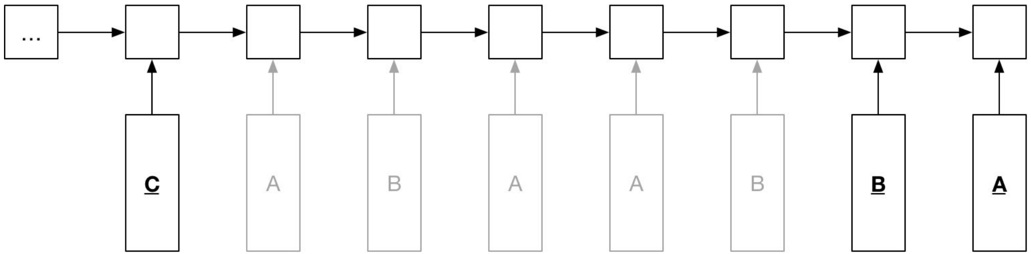
Don’t be afraid of commitment

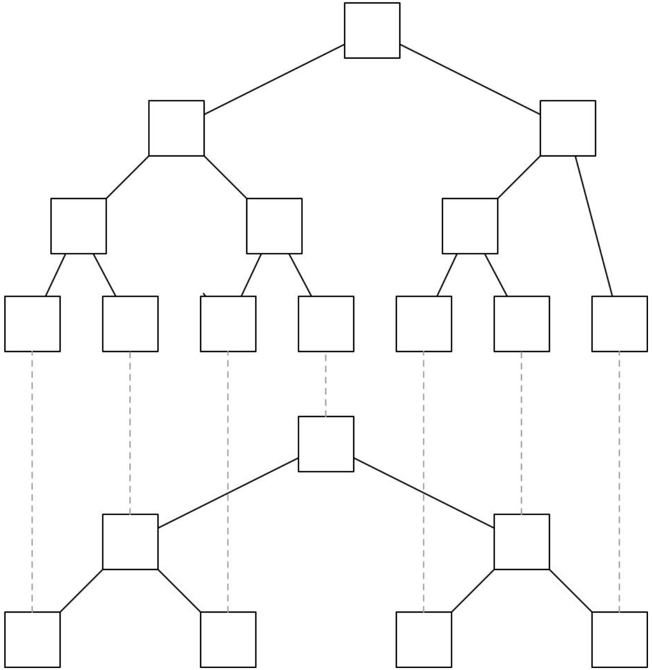
Include in GroupInitKey:

Commitment to the handshake history

Commitment to the current tree

This allows the distribution of the messages / tree nodes to be untrusted, e.g., P2P





Summary

Handshake messages are signed with credentials, SIGMA-like

On joining, new joiner receives GroupInitKey

GroupInitKey contains commitment to handshake history, tree

Joiner downloads last message from each sender from somewhere

Plausible? What’s missing / wrong?

Deniability?

Open Questions

Deniability?

Can we avoid the server knowing the whole membership of the group?

Commit to tree + identities in GroupInitKey

Versioning / extensibility

Interop testing framework

Interim plans

MLS@IETF102

WG Info: https://datatracker.ietf.org/wg/mls/about/
Chairs: Nick Sullivan & Sean Turner