TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups

A protocol proposal for Messaging Layer Security (MLS)

Karthikeyan Bhargavan Richard Barnes Eric Rescorla

May 3, 2018

The Messaging Layer Security (MLS) architecture [OBR+18] envisions a protocol that can establish a key shared by a *group* of *members*, where each member controls a number of *clients* (devices). Each client is identified by its own long-term key, and can participate in the protocol asynchronously, that is, without needing any other client to be online. Notably, each client can issue asynchronous group modification requests to add new members, remove members, and update its own keys, etc. The architecture document also states a series of security goals for the protocol. We begin this document by stating the desired functionality and security goals of MLS in our own notation. We then propose a new protocol that seeks to achieve the confidentiality goals of the MLS architecture.

1 Global Group Messaging Functionality

The global state of the messaging system $\mathcal{S}$ consists of the following elements:

- $\mathcal{G} = \{g_0, g_1, \ldots\}$: a set of group identifiers
- $\mathcal{D} = \{d_0, d_1, \ldots\} \subseteq \mathcal{G}$: a set of device identifiers
- $\mathsf{members} : \mathcal{G} \to 2^{\mathcal{D}}$: a function from groups to their members
- $\mathsf{secret\_key} : \mathcal{G} \to \mathcal{K}_s$: a function from groups to secret key material
- $\mathsf{public\_key} : \mathcal{G} \to \mathcal{P}$: a function from groups to public key material

For simplicity, we do not model the relationships between multiple devices owned by the same user, and instead, we treat each device as an independent group member. We treat each device as a singleton group that contains only the device ($\mathcal{D} \subseteq \mathcal{G}$), and so we can write $\mathsf{secret\_key}(d_i)$ to mean the secret key(s) known only to the device, and $\mathsf{public\_key}(d_i)$ for the corresponding public key(s).

Intuitively, the secret key of a group is only known to members of the group, whereas all other components of the state are public. (We may wish to restrict knowledge of group membership for privacy, but the above model does not address such privacy goals.)

The global state evolves over time:

$$\mathcal{S}_0 \longrightarrow \mathcal{S}_1 \longrightarrow \ldots \longrightarrow \mathcal{S}_k \longrightarrow \ldots$$

We write $(\mathcal{G}_k, \mathcal{D}_k, \mathsf{members}_k, \mathsf{secret\_key}_k, \mathsf{public\_key}_k)$ to refer to the components of a global state $\mathcal{S}_k$. A state transition from S_k to S_{k+1} involves one of the following group operations:

- CREATE$(d_i, g', \{d_0, \ldots, d_j\})$: the device d_i creates a new group g' whose members are $d_0, \ldots, d_j$.
  This operation results in a new state where $g' \in \mathcal{G}_{k+1} \backslash \mathcal{G}_k$, and $\mathsf{members}_{k+1}(g') = \{d_0, \ldots, d_j\}$, and $\mathsf{secret\_key}_{k+1}(g')$ and $\mathsf{public\_key}_{k+1}(g')$ are assigned. All other groups in the global state remain unchanged.

- ADD(d_i, g_j, d'): the device d_i adds a device d' to the group g_j.
  When $d_i = d'$ we call this a *user-initiated* ADD. Otherwise, we call this a *group-initiated* ADD, and require that $d_i \in \mathsf{members}_k(g_j)$. In both cases, we require that $\{d'\} = \mathsf{members}_{k+1}(g_j) \setminus \mathsf{members}_k(g_j)$.

- REMOVE(d_i, g_j, d'): the device d_i removed a device d' from the group g_j.
  When $d_i = d'$ we call this a user-initiated REMOVE (or a leave operation). Otherwise, we call this a group-initiated REMOVE (or evict operation) and require that $d_i \in \mathsf{members}_k(g_j)$. In both cases, we require that $\{d'\} = \mathsf{members}_k(g_j) \setminus \mathsf{members}_{k+1}(g_j)$. Note that while this notation implies that all the users exist upfront, that need not be true in practice. We can model a new user as one which previously existed but was never part of any group other than its own singleton group.

- UPDATE(d_i, g_j): the device d_i refreshes its local state and generates a fresh group key for g_j, hence updating $\mathsf{secret\_key}_{k+1}(g_j)$ and $\mathsf{public\_key}_{k+1}(g_j)$.

There is some overlap between the CREATE and ADD operations, since creating a group and adding some members should have the same overall effect as creating the whole group at once. In this discussion, we maintain the separation between the two because it is possible to perform a CREATE operation more efficiently than the corresponding sequence of ADD operations (as noted in [BMO+18]).

In addition to the above group-modifying operations, each device can call the following functions to exchange messages in the current state $\mathcal{S}_k$ (without modifying the global state):

- SEND$_k(d_i, g_j, m) \mapsto c$: the device d_i encrypts a message m to the group g_j, resulting in the ciphertext c.

- RECEIVE$_k(d_l, g_j, c) \mapsto (d_i, m)$: the device $d_l \in \mathsf{members}_k(g_j)$ decrypts a ciphertext c and extracts the message m and sender d_i.

Intuitively, if $d_i \in \mathsf{members}_k(g_j)$ it uses an encryption key derived from the secret group key $\mathsf{secret\_key}_k(g_j)$ to encrypt the message. Otherwise it uses the public key $\mathsf{public\_key}_k(g_j)$. In both cases, the recipient must be a member of the group and hence can decrypt using $\mathsf{secret\_key}_k(g_j)$.

2 Threat Model and Security Goals

Our threat model includes both outsider adversaries who control the network and can see all ciphertexts as well as insider adversaries who have compromised some devices. We introduce a new operation that an adversary can call to compromise a device in $\mathcal{S}_k$:

- $\textsc{compromise}_k(d_i)$: the adversary compromises device d_i and thereby obtains all the secret keys for the groups that d_i is a member of.

For now, we assume that the long-term authentication keys of d_i are not affected by compromise; put differently, we assume that each device obtains, stores, and manages the compromise of authentication credentials using some (unspecified, out-of-band) mechanism. Hence, we assume that each message sent by a device can be authenticated by other devices, even after the device has been compromised. This is clearly an undesirable gap in our model, and one we will seek to address as the authentication mechanisms of MLS evolve.

An MLS protocol that seeks to implement the global functionality specified above must satisfy the following security goals:

Message Secrecy If device d_i invokes $\textsc{send}_k(d_i, g_j, m)$ in a state $\mathcal{S}_k$, and neither d_i nor any member of g_j has been compromised (in the past or the future), then m is kept confidential from the adversary.
Confidentiality can be formally expressed in many different ways. For example, we can state that the adversary should not be able to obtain m, or that the adversary cannot distinguish between m and some other message m' that may have been encrypted by d_i.

Message Integrity If device d_l invokes $\textsc{receive}_k(d_l, g_j, c)$ to obtain (d_i, m) in a state $\mathcal{S}_k$ where neither d_l nor d_i is currently compromised, then d_i must have previously invoked $\textsc{send}_k(d_i, g_j, m)$ in the same global state $\mathcal{S}_k$.

Forward Secrecy (FS) If device $d_i \in \mathsf{members}_{k+1}(g_j)$ is compromised in state $\mathcal{S}_{k+x}$, and the state transition from $\mathcal{S}_k$ to $\mathcal{S}_{k+1}$ modified group g_j (added, removed, or updated a key), then messages sent to g_j in states $\mathcal{S}_0, \ldots, \mathcal{S}_k$ remain confidential.

Post-Compromise Security (PCS) If device d_i is compromised in state $\mathcal{S}_k$, and in some subsequent state $\mathcal{S}_{k+x}$, either d_i executes $\textsc{update}(d_i, g_j)$ or some d_j executes $\textsc{remove}(d_j, g_j, d_i)$, then starting in state $\mathcal{S}_{k+x+1}$, messages sent to g_j are again confidential.

Both forward secrecy and post-compromise security are strengthened versions of the message secrecy goal. Ordinary message secrecy only holds for groups that are never compromised. FS says that it holds for all messages sent up to $\mathcal{S}_k$, even if the group is compromised at $\mathcal{S}_{k+x}$, as long as the group's keys have been updated between these states. Similarly, PCS says that message secrecy holds after $\mathcal{S}_{k+x}$ even if the group was compromised at $\mathcal{S}_k$, as long as the compromised device's keys were updated between these states.

3 Asynchronous Decentralized Protocols

The goal of an MLS protocol is to implement the global messaging functionality and achieve its stated security goals in a way that is decentralized (that is, devices can execute the protocol without relying on a trusted central authority) and asynchronous (that is, each device can advance the global state and send messages even if all other devices are offline.)

Local State Each MLS protocol needs to specify the local state at each device as its relationship to the abstract global state. In particular, the local state at device d_i, written $\mathcal{S} \mid_{d_i}$, must consist at least of the following elements:

- $\mathcal{G}_i$: the groups that d_i belongs to, including the singleton group $\{d_i\}$
- $\mathsf{members} : \mathcal{G}_i \to 2^{\mathcal{D}}$: members of the groups that d_i belongs to
- $\mathsf{secret\_key} : \mathcal{G}_i \to \mathcal{K}_s$: secret keys for groups that d_i belongs to
- $\mathsf{public\_key} : \mathcal{G}_i \to \mathcal{P}$: public keys for groups that d_i belongs to

In addition, the state at d_i may also contain public information about other groups that it does not belong to. For example, each device may store the public components for the full global state, but it is more likely that different protocols will try to optimize the storage that each device needs to keep.

Asynchronous Group Operations Each group operation involves a specific device (the sender) and a specific group (the target). Consequently, for each operation, the protocol needs to define a SEND operation for the sending device and a corresponding RECEIVE operation that is executed at each recipient.

For example, to update its key a device d_i calls $\textsc{sendupdate}(d_i, g_j)$ to generate an (encrypted) message c which it then sends to some set of devices $d_0, \ldots, d_m$. At this point, the global state has been changed by d_i and the local state at d_i is consistent with the global state, but the local states at other devices are out-of-date. As each recipient d_l receives the update and calls $\textsc{receiveupdate}(d_l, g_j, c)$ to process the update, it changes its own local state to become consistent with the new global state.

The full list of operations that an MLS protocol needs to implement are:

- $\textsc{sendcreate}(d_i, g', \{d_0, \ldots, d_j\}) : \mathcal{S}_k \mid_{d_i} \mapsto (c0, \ldots, c_m, \mathcal{S}_{k+1} \mid_{d_i})$
  $\textsc{receivecreate}(d_l, g', c_l) : \mathcal{S}_k \mid_{d_l} \mapsto (d_i, \mathcal{S}_{k+1} \mid_{d_l})$

- $\text{SENDADD}(d_i, g_j, d') : \mathcal{S}_k \mid_{d_i} \mapsto (c0, \ldots, c_m, \mathcal{S}_{k+1} \mid_{d_i})$
  $\text{RECEIVEADD}(d_l, g_j, c_l) : \mathcal{S}_k \mid_{d_l} \mapsto (d_i, \mathcal{S}_{k+1} \mid_{d_l})$
- $\text{SENDREMOVE}(d_i, g_j, d') : \mathcal{S}_k \mid_{d_i} \mapsto (c0, \ldots, c_m, \mathcal{S}_{k+1} \mid_{d_i})$
  $\text{RECEIVEREMOVE}(d_l, g_j, c_l) : \mathcal{S}_k \mid_{d_l} \mapsto (d_i, \mathcal{S}_{k+1} \mid_{d_l})$
- $\text{SENDUPDATE}(d_i, g_j) : \mathcal{S}_k \mid_{d_i} \mapsto (c0, \ldots, c_m, \mathcal{S}_{k+1} \mid_{d_i})$
  $\text{RECEIVEUPDATE}(d_l, g_j, c_l) : \mathcal{S}_k \mid_{d_l} \mapsto (d_i, \mathcal{S}_{k+1} \mid_{d_l})$

Crucially, the protocol needs to implement these local send and receive operations using just the locally available device state. If each device stores information about all members of the group, and if each group modification operation involves a distinct message for each member, the storage, bandwidth, and computational requirements for every operation becomes linear in the group size.

For large groups with thousands of members, linear growth does not scale. Consequently, the current draft MLS protocol [BMO+18] uses a tree structure for each group to construct a number of auxiliary sub-groups that help reduce the complexity of each group operation. It relies on a cryptographic construction called Asynchronous Ratcheting Trees (ART) [CGCG+17] to implement all operations using $O(log(n))$ storage, bandwidth, and complexity, both at the sender and the receiver. In the next section, we present a protocol that uses a similar tree structure but a different cryptographic construction to further reduce the computational complexity at the recipient.

In addition to efficiency, another important consideration for a decentralized protocol is the ability for nodes to generate and consume messages independently, without requiring central coordination. In order to allow for uncoordinated operation, a protocol needs to accommodate the risk that two or more nodes will generate messages that evolve the group state in two different ways. One option, adopted in the current draft MLS protocol, is for members simply to reject all but the first update. This requires only that the participating nodes agree on the sequence of the messages, but requires a large number of retries before all nodes' changes are incorporated in the group state. In the next section, we present a protocol that can reduce the number of retries by allowing nodes to merge multiple received updates in some cases.

Protocol Consistency Goals A decentralized, asynchronous MLS protocol needs to implement the global functionality, which means that the local states at each device should map to a global state that achieves the functionality and security goals of MLS. In particular, the protocol needs to achieve the following:

Local State Consistency For each device d_i, there exists a sequence of global states $\mathcal{S}_0 \longrightarrow \ldots \longrightarrow S_k \ldots$ such that the local state at d_i is consistent with one of these global states ($\mathcal{S}_k \mid_{d_{d_i}}$).

Global State Convergence If all messages sent by all devices have been correctly received and processed at all recipient devices, and no messages are in-flight, then the state at all devices is consistent with the same global state $\mathcal{S}_k$.

A Simple Protocol with Linear Scaling All four group modification operations in MLS involve a similar pattern. Since each operation modifies (or creates) a group, it needs to generate a fresh group key and deliver it to the members of the group, and possibly also deliver a new public key for the sender (to acheive post-compromise security). Hence, each of these operations can be seen as a *key encapsulation mechanism* (KEM) applied to a group of public encapsulation keys, rather than a single public key. Following Smart [Sma05], we call this a multi-KEM or mKEM construction.

A naïve protocol implementating MLS would be as follows: each device keeps the public keys of all other members of the group. For each group operation, it encrypts a fresh key to all members of the group, using their public keys:

- CREATE$(d_i, g', \{d_0, \ldots, d_n\})$: the device d_i creates a fresh group secret key k' and sends a CREATE message containing this key encrypted to $\mathsf{public\_key}(d_0), \ldots, \mathsf{public\_key}(d_n)$. Each recipient d_l decrypts the message using its private key $\mathsf{secret\_key}(d_l)$ and updates its state.

- ADD(d_i, g_j, d'): the device d_i creates a fresh group secret key k' and sends an ADD message containing this key encrypted to the current participants' keys $\mathsf{public\_key}(d_0), \ldots, \mathsf{public\_key}(d_n)$ as well as the new participant's key $\mathsf{public\_key}(d')$. Each recipient d_i uses its secret key $\mathsf{secret\_key}(d_i)$ to decrypt the new secret key and update its state.

- REMOVE(d_i, g_j, d'): the device d_i creates a fresh group secret key k' and sends a REMOVE message containing this key encrypted to all members of g_j except for d'. If there is no key already established for this subgroup, d_i encrypts the key for the public key of each remaining member of the group $\mathsf{public\_key}(d_0), \ldots, \mathsf{public\_key}(d_n)$.

- UPDATE(d_i, g_j): the device d_i creates a fresh key-pair for itself $(\mathsf{secret\_key}_{k+1}(d_i), \mathsf{public\_key}_{k+1}(d_i))$ and a fresh group secret key k' and sends an UPDATE message containing this key encrypted to all members of g_j except for itself. If there is no key already established for this subgroup, d_i encrypts the key for the public key of each remaining member of the group $\mathsf{public\_key}(d_0), \ldots, \mathsf{public\_key}(d_n)$.

In this protocol, every group operation results in a fresh group key, and with some effort, we can prove that the protocol achieves message secrecy and integrity, forward secrecy and post-compromise security. If we forbid concurrent operations and assume that all operations are sent and received in some total order, then the local state at each node remains consistent to the global state obtained by executing each sent operation in this order.

However, this protocol requires $O(n)$ storage, bandwidth, and computation. Specifically, each create, add, update and remove operation requires $O(n)$ storage, computation and bandwidth at the sender, and $O(1)$ computation and bandwidth at the recipient. Linear storage and computation for sending new group operations is not scalable for large groups.

The ART protocol [CGCG+17] uses a tree-based structure to reduce the cost of each operation to $O(log(n))$ at both the sender and recipient. In the next section, we also use a tree based structure to build a protocol that is halfway between ART and the mKEM-based protocol above. It is as efficient as ART at senders, more efficient at recipients, and supports a greater degree of concurrent operations.

4 TreeKEM: an MLS protocol

We propose to organize the members of each user-visible group (a "messaging group") as a tree of subgroups so that protocol messages can update multiple subgroups at once. The resulting protocol is dubbed TreeKEM, and it borrows ideas from multi-KEM [Sma05], Asynchronous Ratchet Trees [CGCG+17], Decentralized Dynamic Broadcast Encryption [PPS11], and Group Key Exchange [ACMP10]. TreeKEM relies on a collision-resistant hash function (H), a public-key encryption mechanism ($pgen, penc, pdec$), a pseudo-random function used for key derivation function (kdf), and an authenticated encryption scheme (gen, enc, dec).

Trees of Sub-groups We assume that each messaging group in the global state is the root of a tree where each node of the tree corresponding to some sub-group. The leaves of the tree are devices (i.e., one-member groups). For simplicity, we will assume a left-balanced binary tree, but the protocol is easily generalizable to a tree of arbitrary arity and structure. Hence, each messaging group of n devices forms a tree of height $log(n)$, where each device (leaf node) is a member of up to $log(n)$ groups (its ancestors in the tree) and the root corresponds to the messaging group. We extend the global state $\mathcal{S}$ with the following elements that allow $\mathcal{G}$ to be treated as a forest, where each tree is rooted at one messaging group:

- $\mathcal{M} \subseteq \mathcal{G}$: a set of messaging groups
- $\mathsf{encryption\_key} : \mathcal{M} \to \mathcal{K}_{ae}$: a mapping from messaging groups to authenticated encryption keys
- $\mathsf{parent}(g_i) \mapsto g_j$: a mapping from a tree node to its parents (if it exists)
- $\mathsf{sibling}(g_i) \mapsto g_j$: a mapping from a tree node to its sibling (if it exists)
- $\mathsf{path}(d_i, g_j) \mapsto g_0, \ldots, g_j$: the sequence of groups from d_i to the root in the tree with root g_j, where $g_0 = \{d_i\}$, $g_{k+1} = \mathsf{parent}(g_k)$, and $g_j \in \mathcal{M}$.
- $\mathsf{copath}(d_i, g_j) \mapsto g'_0, \ldots, g'_j$: the siblings of each group in $\mathsf{path}(d_i, g_j)$

As we may expect for trees, the above functions satisfy several properties. For example, for any g_i, if $\mathsf{sibling}(g_i) \neq \bot$, then $\mathsf{parent}(\mathsf{sibling}(g_i)) = \mathsf{parent}(g_i)$ and $\mathsf{sibling}(g_i) \neq g_i$. Furthermore, if $d_i \in \mathsf{members}(g_j)$, then $d_i \in \mathsf{members}(\mathsf{parent}(g_j))$, and if $d_i \in \mathsf{members}(g_j)$, then $d_i \notin \mathsf{members}(\mathsf{sibling}(g_j))$.

Local State We specify the local state at each device as follows:

- $\mathcal{M}_i$: the messaging groups that d_i belongs to.
- $\mathsf{encryption\_key} : \mathcal{M}_i \rightarrow \mathcal{K}_{ae}$: authenticated encryption keys for the messaging groups of d_i
- $\mathcal{G}_i$: a set of sub-groups that d_i belongs to, including the singleton group $\{d_i\}$.
- $\mathsf{secret\_key} : \mathcal{G}_i \rightarrow \mathcal{K}_s$: secret keys for groups that d_i belongs to
- $\mathsf{public\_key} : \mathsf{sibling}(\mathcal{G}_i) \rightarrow \mathcal{P}$: public keys for the sibling groups of $\mathcal{G}_i$

Hence, for each messaging group g_j, d_i needs to keep the group encryption key, secret keys for all groups on $\mathsf{path}(d_i, g_j)$, and public keys for all groups on $\mathsf{copath}(d_i, g_j)$. So, the storage requirements for every group of size n that d_i decides to join is $O(log(n))$. To participate in m groups of size n each, d_i would have to keep $O(mlog(n))$ state.

It is worth noting that up to this point the group structures and local states stored in ART and TreeKEM are the same. The only difference is that in ART, the keys for each group $(\mathsf{secret\_key}(g), \mathsf{public\_key}(g))$ must correspond to a Diffie-Hellman keypair, whereas in TreeKEM, we can choose any keypair that support key encapsulation (KEM); that is, we should be able to encrypt a key to $\mathsf{public\_key}(g)$ in a way that it can be decrypted only using $\mathsf{secret\_key}(g)$.

Computing Tree Keys In order to compute group keys, we need to assign to each tree node a keypair that is known only to members of the group, that is, to the devices that appear as leaves in the current subtree. At the leaves, we generate fresh KEM key-pairs for each device $(\mathsf{secret\_key}(d_i), \mathsf{public\_key}(d_i))$. For each internal node, the secret key is computed as a hash of the secret key of one of the two children, intuitively the last child to have issued a group operation. The messaging group's authentication encryption key $(\mathsf{encryption\_key}(d_j))$ is derived (as a chain of *kdf* invocations) from the sequence of keys at the root of the tree (as in MLS [BMO+18]).

In ART, each internal node's secret key is computed (via a Diffie-Hellman shared secret computation) from both children's secret keys. In TreeKEM, on the other hand, internal node keys only depend on one of the two children. Hence, the keys for internal nodes in TreeKEM are *not contributive* (unlike ART) and this leads directly to some of TreeKEM's advantages with regard to concurrency. Note that the messaging group's encryption key is still contributive in the sense that it incorporates keying material from all nodes that have initiated some group operation.

As a running example, consider the sequence of transitions in Figures 2-7 (summarized in Figure 1):

- Figure 2 represents a freshly created group of five devices (A,B,C,D,E). Each node is annotated with the group $\mathsf{secret\_key}$ assigned to that node

(for simplicity, our figures use the device name to also refer to its secret key). The tree keys are computed as if the nodes have been added in left-to-right order so that the last device (E) determines the key of its ancestors.

- Figure 3 shows the tree after device B has issued an update, generating a fresh key B' and installing a sequence of hashed keys up the tree.
- Figure 4 shows the tree after a new device F has been added to the group with a fresh key F and installing a sequence of hashed keys up the tree.
- Figure 5 shows the tree after the device C is removed from the group, and all its (previous) groups are given a sequence of hashed keys starting with a fresh key C' that is unknown to C.
- Figure 6 shows the tree after A and D have issued simultaneous updates, and other nodes have applied them in sequence. In this state, compromise of A or D individually will not reveal the group key, but compromise of both will.
- Figure 7 shows the tree after B has issued another update, following the application of the concurrent updates from A and D. The full post-compromise security of the group is restored.

In TreeKEM, if two operations execute concurrently, they are many ways they can be merged. For example, we may assume that operations coming from devices in a left subtree must be executed before devices on the right. Or else, we can enforce more subtle policies like: updates must be processed before removes. We may also rely on the delivery service to totally order all operations so that all devices process them in the same order. The key property of TreeKEM is that most operations are "mergeable" in the sense that any device who receives two concurrent operations will be able to process and execute them both without having to reject one of them or asking for more information.

Figure 6 shows how two concurrent updates may be merged by using the order of devices as a tie-breaker. Since both updates were issued against the old global state, however, the new tree is still not fully up to date, in the sense that an attacker who knows the old keys at both A and D is able to compute the new group key. However, as soon as any device close to A or D issues an update against the new, merged state, the old keys will be locked out, as shown in Figure 7. In other words, TreeKEM allows concurrent updates to be immediately executed, but their post-compromise security benefits do not apply until another update is issued against the merged state.

Because processing an update involves overwriting keys, in order for concurrent updates to work properly, implementations need to retain a set of historical keys so that they can process updates which were sent based on the same initial state $\mathcal{S}_k$. This is compatible with a number of state retention algorithms, as long as implementations agree on which updates are to be processed and which are to be rejected (and hence when keys can be discarded).

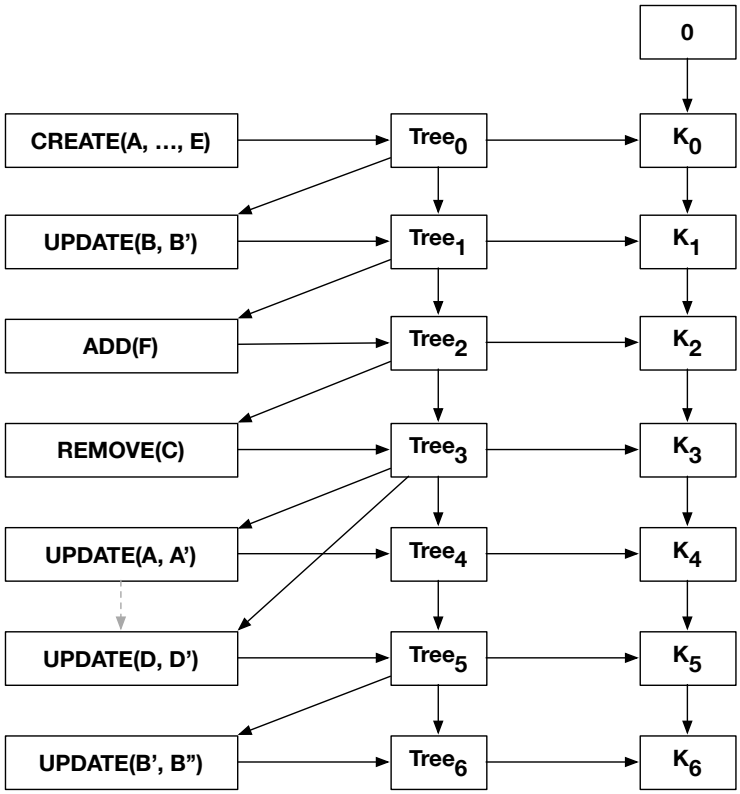

Figure 1: The sequence of changes depicted in Figures 2-7. Columns represent protocol messages, states of the tree, and group keys, respectively. Solid arrows indicate causality. The dashed arrow between the two updates indicates that the group has agreed on an ordering between them.

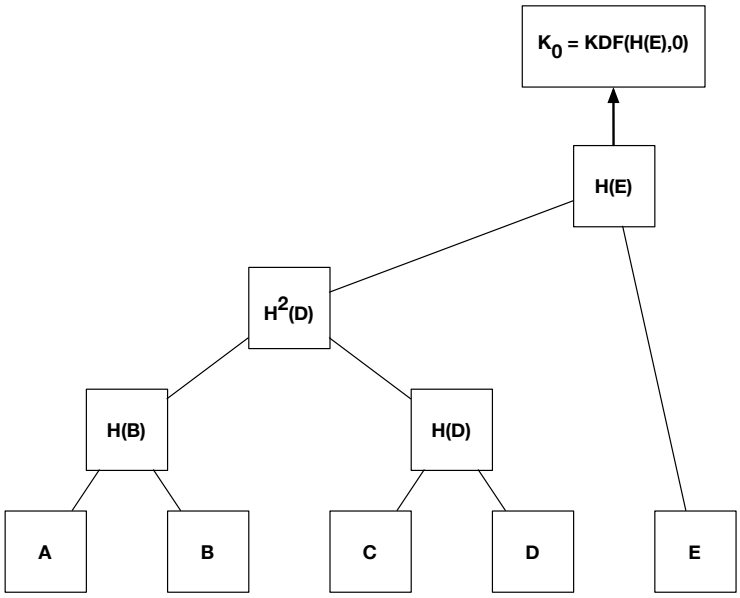

Figure 2: A newly created messaging group with 5 devices and initial group encryption key K_0.

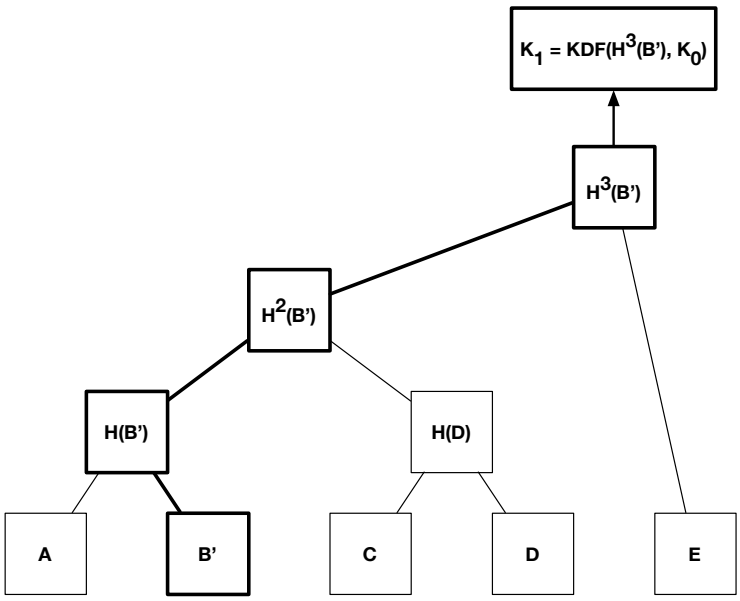

Figure 3: Device B updates the keys for all the groups it belongs to, resulting in a new group key K_1.

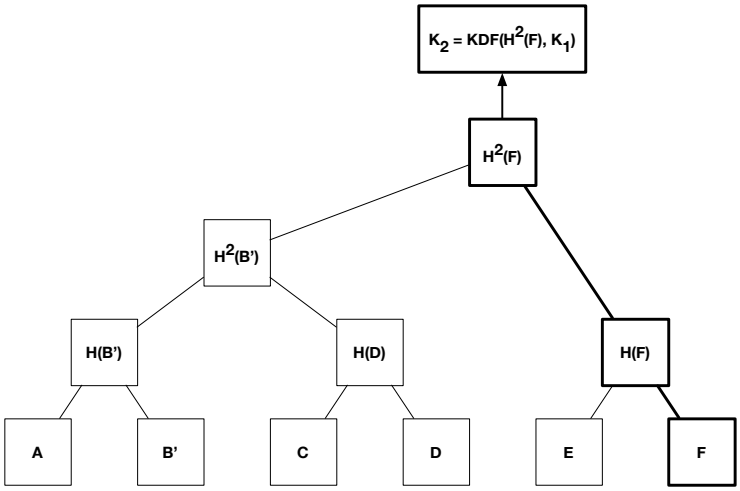

Figure 4: Device F is added to the group, resulting in a new group key K_2.

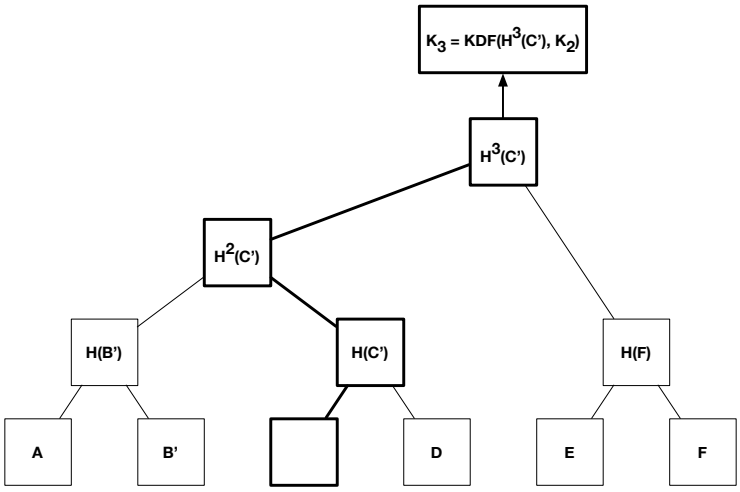

Figure 5: Device C is removed from the group, resulting in a new group key K_3.

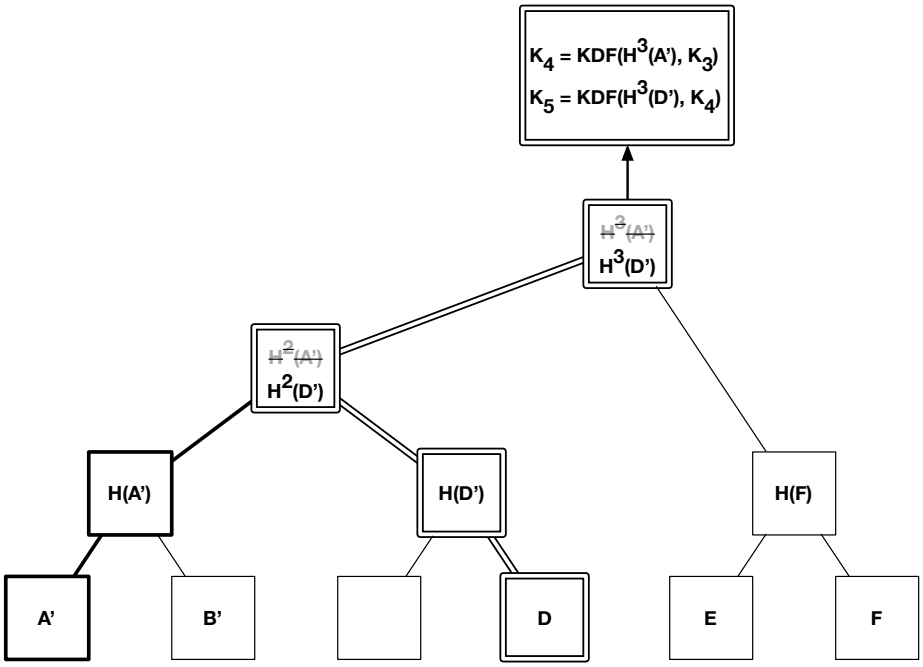

Figure 6: Devices A and D issue updates at the same time, and the tree merges these updates, choosing to execute D's update after A's. Note that some of A's updates are overwritten by D's, but the group key K_5 incorporates both updates. At this stage, we obtain PCS against the compromise of one of the two devices but not both: an adversary who compromises the old keys at both device A and D will still be able to compute K_5, since $H^2(D')$ and $H^2(A')$ were sent encrypted to the old public keys of A and D.

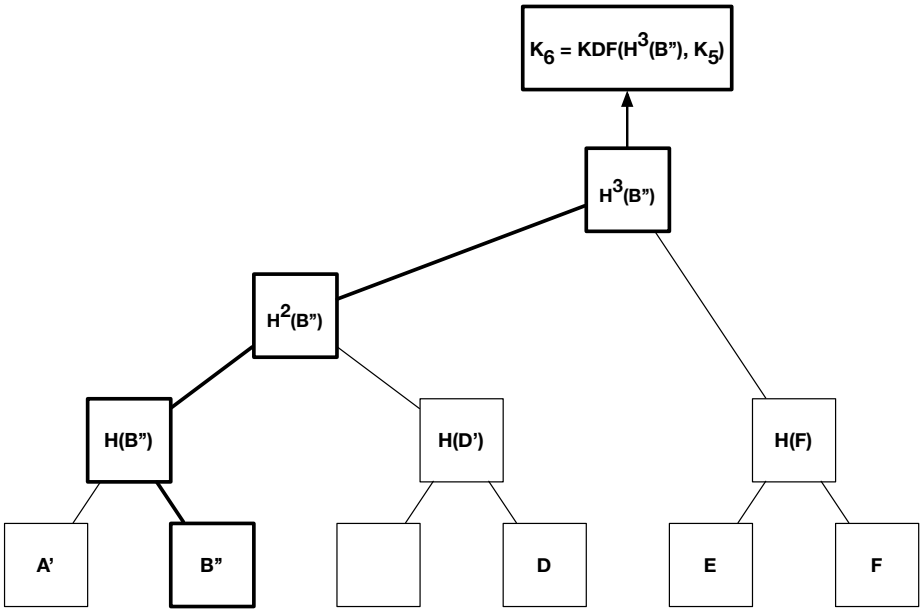

Figure 7: After processing the concurrent updates from devices A and D, device B sends a new update. This update propagates up the tree, thereby “healing” all tree nodes into a consistent merged state. Once this new update is processed, we obtain PCS against compromise of all devices; that is, an adversary who compromises the old states of A, B, and D can no longer compute the new group key.

Sending Messages to Subsets of a Group Because the groups are arranged in a tree, it is possible to efficiently encrypt messages to subsets of the group members. To send to any group, g_j we merely encrypt to $\mathsf{public\_key}(g)$. In order to send to $g_j \setminus d_i$ we encrypt to the the public keys corresponding to the $\mathsf{copath}(g_j, d_i)$. For example, if we wished to encrypt a message to nodes a, b, c, d, and f in 4, we would need to encrypt to nodes abcd and f. This technique can be extended to subsets with more missing members, though of course efficiency rapidly falls off.

TreeKEM Operations The TreeKEM protocol defines the following group operations:

- When a device calls SENDCREATE$(d_i, g', \{d_0, \ldots, d_n\})$, the function generates fresh leaf keys $K_0, \ldots, K_n$ for $d_0, \ldots, d_n$. It then computes the rest of the tree keys by choosing a hash of one of the child keys (say the right child). Finally, the device computes a ciphertext for each device that consists of the secret keys for its groups (encrypted under its private key) and public keys for its co-path to the root.
  For example, to create the group in Figure 2, the creator would send:
  - to a: $penc_a(K_a, H(K_b), H(H(K_d)), H(H(H(K_e))))$
    as well as $\mathsf{public\_key}(\text{co-path}) = \mathsf{public\_key}(K_b), \mathsf{public\_key}(K_{cd}), \mathsf{public\_key}(H(H(K_e)))$
  - to b: $penc_b(K_b, H(H(K_d)), H(H(H(K_e))))$
    $+ \mathsf{public\_key}(K_a), \mathsf{public\_key}(H(K_d)), \mathsf{public\_key}(H(H(K_e)))$
  - to c: $penc_c(K_c, H(K_d), H(H(H(K_e))))$
    $+ \mathsf{public\_key}(K_d), \mathsf{public\_key}(H(K_b)), \mathsf{public\_key}(H(H(K_e)))$
  - to d: $penc_d(K_d, H(H(H(K_e))))$
    $+ \mathsf{public\_key}(K_c), \mathsf{public\_key}(H(K_b)), \mathsf{public\_key}(H(H(K_e)))$
  - to e: $penc_e(K_e) + \mathsf{public\_key}(H(H(K_d)))$

  Hence, for a group of size n the creator needs to send n encryptions, each encryption containing between 1 and $log(n)$ keys, coming to a total ciphertext size of $2n$. It also needs to send $2n$ public keys (corresponding to the whole tree.)

- The delivery service needs to deliver to each device one of these encryptions (ciphertext size $\leq log(n)$) and $log(n)$ public keys for the co-path. On receiving this creation message, each member of the new group calls RECEIVECREATE(d, c) which performs one decryption and computes $log(n)$ hashes to obtain all the secret and public keys needed to compute the new state.

- When a device calls SENDUPDATE(d_i, g_j), the function generates a fresh key K' for d_i and computes a sequence of hashes $H(K'), H(H(K')), \ldots, H^L(K')$ for all groups on the path from d_i to the root. It then computes a sequence

of ciphertexts for each group g_j on the co-path from d_i to the root by encrypting for g_j the new key for its parent.
For example, to update the group as in Figure 3, device b would send:

- to device a: $penc_{\mathsf{public\_key}(K_a)}(H(K_b))$ + $\mathsf{public\_key}(K_b)$
- to the group cd: $penc_{\mathsf{public\_key}(K_{cd})}(H(H(K_b)))$ + $\mathsf{public\_key}(H(K_b))$
- to the device e: $penc_{\mathsf{public\_key}(H(H(K_e)))}(H(H(H(K_b))))$ + $\mathsf{public\_key}(H(H(K_b)))$

Hence, for a group of size n, each updater needs to compute $log(n)$ encryptions, each containing one (hashed) key, and compute $log(n)$ public keys from secret keys. The message it sends has size $2log(n)$.

- The delivery service delivers to each device a single ciphertext and a single public key. The recipient then calls RECEIVEUPDATE(d_l, c) to compute the new key for its lowest shared ancestor with the updating node d_i, and then hashes the key to obtain the keys for all nodes from this ancestor to the root. The recipient hence has to compute a single decryption, then $log(n)$ hashes.

- When a device calls SENDADD(d_i, g_j, d_l), the function behaves the same way as if d_l had asked for an update. A fresh key K'_l is generated for d_l and hashed up the tree to the root. Each of these keys is then encrypted to the corresponding group on the co-path. If $d_i = d_l$ then only the new device knows these keys. As in update, the complexity for sending an ADD is $log(n)$ encryptions whereas the complexity for receiving an ADD is one decryption and $log(n)$ hashes.

- When a device calls SENDREMOVE(d_i, g_j, d_l), the function behaves as if d_l has asked for an update, except that d_l is not given its new key K'. Again, the complexity for sending a REMOVE is $log(n)$ encryptions whereas the complexity for receiving a REMOVE is one decryption and $log(n)$ hashes.

In summary, the complexity of creating a group in TreeKEM is roughly $O(n)$ where each recipient needs to do about $O(log(n))$ work. The complexity of issuing other group operations is roughly $O(log(n))$ encryptions and public-key derivations, where each recipient needs to only perform one encryption and $O(log(n))$ hashes. Hence, on the recipient side, TreeKEM appears to be more efficient than ART, which requires $O(log(n))$ variable-base DH operations + public key derivations at each recipient.

For the messaging operations on the full group, we use a key derived from the full sequence of root keys, by feeding each root key into *kdf* along with the prior root key, as in MLS [BMO+18]. The final result of this chain of derivations is again fed into *kdf* to obtain an authenticated encryption key which can be used to encrypt group messages within the group.

	Send		Receive		ART
Operation	Hash	Pub	Hash	Pub	Pub
Create	n	$2n$	$log(n)$	1	$2n$
Update	$log(n)$	$2log(n)$	$log(n)$	1	$2log(n)$
Add	$log(n)$	$2log(n)$	$log(n)$	1	$2log(n)$
Remove	$log(n)$	$2log(n)$	$log(n)$	1	$2log(n)$

Table 1: Complexity of TreeKEM operations, in units of the number of hashes and public-key operations required (including both KEM and public-key derivation). For ART, the send and receive operations have the same complexity.

A consequence of this design is that a device needs to know both the old and new keys in order to compute the new key for communicating within the group. When a new device is added, it will not know the old key and hence will not be able to derive the new group key. So, we assume that some member of the group delivers to the new device the new group messaging key, obtained by hashing the prior group key with the new root key.

5 Handling Concurrent Group Operations

A key feature of TreeKEM is its support for merging concurrent group operations. Notably, any node that recieves two concurrent updates is able to compute a consistent set of secret and public keys for its view of the global state.

A simple way of merging concurrent operations is to pick an arbitrary order of devices; for example in Figure 6 we choose the update from a right subtree over an update coming from a left subtree. Alternatively, we may rely on a centralized delivery service to serialize all concurrent updates into a consistent total order. Any such uniform, global ordering will result in a consistent global state that converges on all devices. However, not all operations can be merged in this way, and not all merges will have the desired effect, hence the protocol has to be careful on how it merges such operations. Some tricky examples:

- If two devices are added at the same location in the tree at the same time, only one of these can succeed, and the other add must be rejected.
- If two devices are added at different locations in the tree at the same time, and we order one of these (d_i) before the other (d_j). Then the final key will be determined by the ADD operation for d_j. However, since the device that added d_j was not aware of d_i when issuing its ADD operation, it would not have encrypted the right key to d_i and hence d_i remains in an incomplete state until it receives the additional data it needs.
- If two devices remove each other, we can remove both, or we may reject one or both of these operations as being contradictory.

- If a device is removed while another device is updated, and if the REMOVE is chosen to be executed before the UPDATE, then the fresh group key created by the update will be sent to the removed node, which would then know the new group key even though it has been removed.

- If two devices are updated at the same time, then the new updated group key will be delivered to the old keys at these devices, and hence an attacker who knows the old keys will be able to continue reading group messages, even though both devices have issued updates, intuitively violating PCS.

Some of these cases can be avoided by employing a strict merge policy. For example, we may decide that ADD operations cannot be concurrent and hence are not asynchronous in the sense that they may fail and the adding device needs to be aware of this, and be willing to send a fresh ADD operation on failure. Updates and removes may be performed concurrently but updates will always be ordered before removes. Hence, in each time slot, we allow a series of updates, followed by removes, followed by a single add.

Even with these rules, we need to be aware that the guarantees of TreeKEM will always be stronger for sequential operations than for concurrent ones. For example, if two nodes are removed one after the other, they will be removed from the group and the adversary won't have access to the new group key even if both devices were compromised. However, if these devices were removed concurrently, an adversary who has compromised both devices still has temporary access to the full group key, although it will gradually be deauthorized as these or other nodes start sending updates.

Consequently, the right way of looking at the state of TreeKEM is to look at the versions of each key in the tree. If there is any node which is out-of-date with respect to one of its children, then the global state of the messaging system is "un-merged". As soon as any new (non-concurrent) operation flows up from any of this node's children, the tree becomes up-to-date and can be considered to be "merged", and hence consistent with a merged global state. In other words, TreeKEM provides immediate consistency for sequential operations and eventual consistency for concurrent operations.

An Encrypting Serializer The guarantees of TreeKEM for concurrent operations can be strengthened if we are willing to invest more trust into the delivery service.

First, the delivery service can define a total order on group operations, based for example on the order in which it receives these operations, making the merge policies described above unnecessary.

Second, the delivery service can guarantee immediate consistency for concurrent updates by using encryption to enforce the serial order of operations. In particular, if operation o is serialized before o' then the ciphertexts of o' need to be encrypted with keys that incorporate the new keys generated by o. If o' and o were concurrent, then this property does not hold, but the delivery service can enforce it by adding another layer of encryption with the new keys.

Suppose the delivery service receives and executes each group operation in a total order and keeps track of the current public keys of each group. When the delivery service receives a concurrent operation, this operation will contain ciphertexts encrypted to the old public keys for certain groups. In this case, the delivery service encrypts these ciphertexts again with the new public keys for the groups, so that only devices who know both the old and new secret keys can read the contents.

Note that such an encrypting serializer does not get to read the plaintexts but can enforce a sequential interpretation of concurrent updates using public key encryption. Still, the above proposed mechanism does require a higher degree of trust in the delivery service; if the service colludes with a compromised device, the service can delay the removal or updates for the device, weakening the PCS guarantee. However, one may argue that any delivery service can already do this by selectively delaying or ignoring some operations.

Preventing Double Join In TreeKEM, when a device d_i adds or removes another device d_j, d_i controls the new keys for the path from d_j to the root, even though d_i is itself not a member of these subtrees. In a sense, d_i now controls two leaves in the tree, a phenomenon called *double join*. A similar scenario occurs in ART for group-add and remove operations.

Double join is undesirable because it gives d_i control over sub-groups that it was not invited to join. In particular, if we then wanted to delete d_i, we should also delete (or update the keys for) the path from d_j to the root, since this path has been "tainted" by d_i.

MLS protocols can employ various mechanisms to detect and protect against malicious nodes that have double-joined a group. For example, the protocol may keep track of all the leaves controlled by each device, so that when the device is deleted, all these other leaves can be deleted as well.

A stronger alternative would be to prevent double-join in the first place. User-initiated ADD and REMOVE operations do not result in double-join. For group-initiated REMOVE, the protocol may rely on algebraic properties of the public-key encryption to avoid double-join. For example, instead of overwriting the keys on the path from d_j to the root with a fresh key K', we may instead combine K' to the previous key on each node from d_j to the root. Hence, the removing node d_i would not know the new keys on the path, preventing double join. The key challenge is that d_i must still be able to compute the new public keys for this path. If we used Diffie-Hellman based encryption (e.g. El-Gamal or DHIES), then we can use an algebraic combination to achieve this goal. We add K' to the previous DH private key at each node, and compute the new DH public key by multiplying $g^{K'}$ with the previous public key.

Finally, preventing double-join efficiently in group-initiated ADD remains an open problem. The only way to do this that we know of is to generalize the tree structure to allow each node to hold a lists of keys, and encrypt every message to a subgroup under all the keys in the subgroup node. This prevent double-join at the cost of a significant number of extra encryptions.

Provably Consistent Group Operations A final desirable goal for an MLS protocol is that group operations should be provably consistent; that is, if a device were to encrypt fresh keys for two groups in a tree, anybody can check that these keys are mutually consistent, or at least that they map to the same full group key.

In TreeKEM, we propose to add strong consistency checking for all group operations by employing a classical Schnorr-like non-interactive zero-knowledge proof of consistency. We are working out the details of this construction, but informally, every group operation includes an El-Gamal encryption (with the recipient's public key) of the full messaging group key and a proof that this set of group keys encrypted to all recipients is consistent. Any device or server can verify this proof (the verifier does not have to be a member of the group) and consequently, the burden of verifying consistency can be off-loaded onto the delivery servers without impacting the recipients' computational load.

References

[ACMP10] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. Flexible Group Key Exchange with On-Demand Computation of Subgroup Keys. In *Third African International Conference on Cryptology (AfricaCrypt '10)*, volume 6055 of *LNCS*, pages 351–368, Stellenbosch, South Africa, 2010. Springer.

[BMO+18] R. Barnes, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security protocol. IETF Inernet Draft, February 2018.

[CGCG+17] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. Cryptology ePrint Archive, Report 2017/666, 2017. `https://eprint.iacr.org/2017/666`.

[OBR+18] E. Omara, B. Beurdouche, E. Rescorla, S. Inguva, A. Kwon, and A.Duric. Messaging layer security architecture. IETF Inernet Draft, February 2018.

[PPS11] Duong Hieu Phan, David Pointcheval, and Mario Strefler. Decentralized dynamic broadcast encryption. Cryptology ePrint Archive, Report 2011/463, 2011. `https://eprint.iacr.org/2011/463`.

[Sma05] N. P. Smart. *Efficient Key Encapsulation to Multiple Parties*, pages 208–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.