Causal TreeKEM: Post-Compromise Secure Group Key Agreement for Causal Broadcast

Matthew Weidner

March 19, 2019

1 Motivation

(This section describes some issues with concurrent messages that have been discussed before on the MLS list. Readers familiar with these issues can skip to the next section.)

The version of TreeKEM described in the current MLS protocol draft [2] implicitly assumes a consistent total order on state changes (key updates, adds, and removes). That is, there is a linear sequence of group states agreed upon by all group members, and each message leading to a state change can only be generated by a user who knows its preceding state.

However, group communication protocols do not naturally impose a total order on messages. While messages can be forced into a consistent total order, say by a central server or by following predetermined client-side rules [2, §8], this may result in messages being placed logically after messages that their authors had not yet received. These messages may then fail, since they assumed a different initial state. Failures can lead to starvation, in which one user's state changes repeatedly fail, and to violations of asynchronicity. Concurrency issues are especially troubling in ad-hoc networks, in which the group may frequently partition and re-merge, leading to concurrent messages that go undetected for long periods of time.

For example, suppose there are two concurrent key updates, each followed by adding a user. Then the two added users share no init secrets in common, so that they cannot securely communicate without setting up a separate peer-to-peer conversation or asking another user to mediate. To avoid this, the current draft suggests we require existing users to "confirm" an Add message before sending the added user's Welcome message. This violates asynchronicity.

TreeKEM does have some support for concurrent state change messages [3, §5]. For example, concurrent Remove messages result in an init secret unknown to any removed user but known to the remaining users, so long as the resulting root secrets are put into the init secret derivation chain in a consistent order.

In particular, Update messages can be merged at nodes where they do not conflict, with an arbitrary winner at conflicting nodes [3, pg. 9].

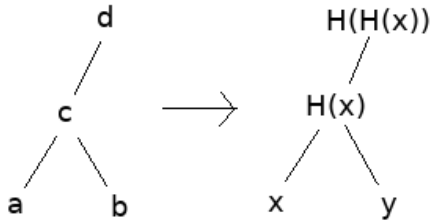

Figure 1: Merging updates in ordinary TreeKEM. Here $a \mapsto x$ and $b \mapsto y$ concurrently.

If we put the init secrets resulting from both updates into the init secret derivation chain, as suggested by Bhargavan et al. [3, Figure 6], then the resulting init secret is unknown to an adversary who had previously compromised either A or B.

However, this approach does not completely end the compromise. In the above example, suppose an adversary compromised B, the right leaf, before the concurrent key updates. Future Update messages by other users will reveal some node secrets to the adversary, who still knows $H(x)$. If another user C is compromised in the future, the adversary will learn the init secret resulting from A and B's updates. Then even if C performs a key update, the adversary will learn the new group secret, using $H(x)$, contrary to what we should expect. Unlike for the init secret, we cannot use a generic key derivation function of $H(x)$ and $H(y)$ to derive the secret for A and B's nearest common ancestor, because other users need to be able to compute the resulting public key without interaction.

This scenario is somewhat special, and may not significantly reduce our security guarantees in a real-life setting. However, it certainly complicates the precise statement of post-compromise security, which makes analyzing the protocol and devising key update schedules both more difficult.

2 Causal TreeKEM

We now describe "Causal TreeKEM", a proposed modification to TreeKEM that has better support for concurrent updates. This section describes the core of the approach, restricting to the case of static groups.

As in TreeKEM, fix some asymmetric cryptosystem with a Derive-Key-Pair function DKP mapping binary strings to asymmetric key pairs, and fix a preimage-resistant hash function H mapping binary strings to binary strings. As a convenience, let $priv(s)$ denote the private key of $DKP(s)$. We use a ratchet tree, i.e., a binary tree with key pairs at each node, precisely

as in TreeKEM [2, §5], except that we omit node secrets from the stored state. Node secrets will still play a role in key updates; this is their sole role in TreeKEM as well.

A ratchet tree is initialized similar to TreeKEM, with public keys for each user at the leaves, a random root key pair, and blanks elsewhere. However, throughout Section 2.1 we will assume that all nodes are already populated with key pairs, postponing our discussion of blank nodes until Section 3.1.

Additionally fix binary *key combination operators* $\star_1, \star_2$, acting on private keys and public keys, respectively. Let $\star = (\star_1, \star_2)$ denote their combination acting on key pairs. We require the following properties:

(1) If (x, X) and (y, Y) are valid key pairs, then so is $(x, X) \star (y, Y) = (x \star_1 y, X \star_2 Y)$.

(2) $\star$ is associative and commutative.

(3) $\star_1$ is cancellative: if $x \star_1 z = y \star_1 z$ for some z, then $x = y$.

Example 2.1. Let g be the generator of a Diffie-Hellman group with order $|g|$. Suppose we use a Diffie-Hellman based asymmetric cryptosystem, in which key pairs have the form (x, g^x). Then we can define $\star$ on key pairs by

$$(x, g^x) \star (y, g^y) := (x + y \pmod{|g|}, g^x g^y).$$

2.1 Key Updates

Users generate Update messages precisely as in ordinary TreeKEM. However, users process Update messages differently. Instead of treating an Update message as an instruction to *overwrite* keys with new key material, we treat it as an instruction to "$\star$ in" the new key material.

Specifically, a user B receiving an Update message from a user A uses it to update their own view of the ratchet tree as follows:

- B updates the public keys for the nodes on A's direct path as

  (new public key) := (current public key)$\star_2$(public key in Update message).

- B updates the private key for their nearest common ancestor with A as

  (new private key) := (current private key)$\star_1 priv$(secret in Update message).

  B can decrypt this secret because by definition of a node's resolution, A has sent the secret encrypted under the public key K of some node on B's direct path. Because B knows the private key corresponding to K, they can decrypt the secret. However, this depends on B knowing which state A was in when they sent the Update message, so that they know which version of the private key to use; we discuss this in Section 2.2 below.

- B computes the secret contribution for all ancestors of their nearest common ancestor with A, using H. B uses these secrets to compute the new private keys in the same way as they compute the nearest common ancestor's new private key. That is, they set

  (new private key) := (current private key)$\star_1 priv(H^i$(secret in Update message)),

  where i is the distance from the node to the nearest common ancestor.

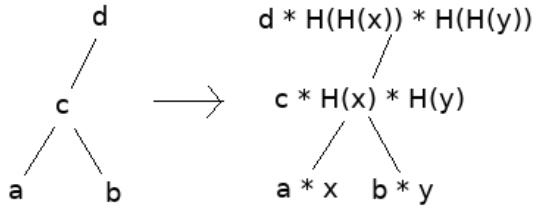

Figure 2: Merging updates in causal TreeKEM. Here the left leaf updates with x and the right leaf updates with y concurrently.

Observe that the key update protocol preserves the usual invariant: every user knows all private keys on their leaf's path to the root, as well as all public keys in the tree. In addition, they are not supposed to know any other private keys.

Additionally, concurrent key updates are handled in a natural way, with no need to discriminate between the updates.

Furthermore, because $\star$ is associative and commutative, the order in which users process Update messages does not affect their final state. Any sequence of updates, applied in any order, yields the same tree of private and public keys.

2.2 Causal Ordering

The previous section suggests that users can process Update messages in any order. This is incorrect: a user must actually process Update messages in *causal* order, so that they have the private key needed to decrypt each Update message's nearest common ancestor secret.

To describe this precisely, we borrow some definitions from the concurrency literature.

Definition 2.2. The *causal order* on state change messages is the partial order $<$ in which $m < m'$ if the author of m' received and processed m before sending m'. This includes the case that m and m' have the same author and m was sent before m'. Two messages are *concurrent* if they are incomparable under $<$, i.e., neither $m < m'$ nor $m' < m$. If $m < m'$, then m

is a *causal predecessor* of m', and m' is a *causal successor* of m. The *direct causal predecessors* of m are the maximal elements of $\{m' \mid m' < m\}$.

Definition 2.3. A *configuration* is a set S of state change messages that is *downwards-closed*: if $m' \in S$ and $m < m'$ in the causal order, then $m \in S$. We define a *user configuration* to be a pair (A, S), where A is a user and S is a configuration. The *ratchet tree of* (A, S) is A's view of the ratchet tree after processing exactly the messages in S. The *root private key of* S is the root's private key in the ratchet tree of (A, S) for any user A.

We now mandate that when a user A sends a state change message m, such as an Update message, they include a description of the set S of state change messages that they have already processed. Note that S is necessarily a configuration, and it is precisely the set of causal predecessors of m. When a user B receives this message, they compute the ratchet tree of (B, S) and use its private keys to decrypt the nearest common ancestor secret in m, assuming they have received all of the messages in S.

One way to describe a configuration S is to include some identifier, such as a hash, of each of the maximal elements of S. In other words, to each state change message m, we attach a description of m's direct causal predecessors.

Now all that we require of the messaging layer is *causal broadcast*, in which messages are delivered in causal order to all users. This is much easier to implement than totally ordered broadcast. The problem of causal broadcast in an asynchronous environment is well studied. For example, Eugster et al. state formal security games for causal broadcast and exhibit a simple protocol achieving it [5]. As another example, Depot [6] attains *consistent* causal broadcast, in which all users see the same messages with the same causal relations, even in the face of malicious users in an asynchronous peer-to-peer setting. Causal broadcast is a desirable security property in group messaging applications [5], so it is likely that applications will already implement causal broadcast and causal dependency tracking. Causal TreeKEM can then reuse this to determine the causal predecessors of state change messages, without consuming additional bandwidth.

2.3 Key Schedule

In contrast to TreeKEM, in Causal TreeKEM, the root private keys in a given state depend on all causally prior key updates. Thus I propose to use root private keys directly instead of generating separate init secrets. Application keys can be derived similarly to in TreeKEM, using a key derivation function applied to the current root private key and information about the group state.

More precisely, suppose a user A wishes to encrypt a group communication. Let S be the set of state change messages that A has processed so far. Then A derives an application key by applying some key derivation

function to the root private key for S and some deterministic description of the group state, e.g., a lexicographic membership list. A then encrypts their communication using this key or one derived from it. A must also attach some description of S to their communication, necessarily as plaintext, so that recipients know which version of the root private key and group state to use.

A user B receiving a communication tagged with S computes the root private key of S and uses this to derive the encryption key.

As mentioned in Section 2.2, it is likely that applications will already implement causal broadcast and causal dependency tracking. Causal TreeKEM can then reuse this to determine the configurations S corresponding to encrypted communications, without consuming additional bandwidth.

Remark 2.4. Observe that the root public key is actually unnecessary, as in TreeKEM, so we can omit the key pair and instead store only a secret bit string at the root node, which we use to seed the application key in place of the root private key. In this case, we must replace $\star$ with some associative and commutative operation on bit strings, such as exclusive-or. For ease of presentation, we will continue assuming the root has a key pair that can be operated on by $\star$, with the understanding that this may be replaced by a secret bit string operated on by exclusive-or.

3 Dynamic Groups

3.1 Blank Nodes

As in TreeKEM, we allow blank nodes. The private and public key of a blank node are undefined.

We adopt the following rules for blank nodes:

- A state change message that sets a node to blank overrides all concurrent updates to that node's key pair, i.e., any state change message that is not a causal predecessor or successor.
- When computing the ratchet tree of a configuration that includes a state change message involving blanks, we treat the total contribution, of that message and all of its causal predecessors, towards the key pair at any node where it is blank, to be some fixed constant value. In other words, a blank overwrites prior keys with a constant value instead of combining with them. A natural choice for the constant value is the identity of $\star$, if it exists.

These rules have essentially the same effect as the rule, in the totally ordered update case, to always process concurrent Update messages *before* messages that introduce blank nodes, namely, Adds and Removes. Such a rule is mentioned in the Messaging Layer Security working draft [2, §8].

3.2 Removing Users

We define a Remove message to contain:

- A list of the removed users.
- A new secret value s, encrypted under various node public keys such that it can be decrypted by the remaining users only.

For example, if a single user is being removed, then s is sent encrypted under the public keys in (the resolutions of) the removed user's copath nodes, similar to ordinary TreeKEM's approach but without hashing up the tree. At the opposite extreme, if users at alternating leaves are being removed, then s is sent encrypted under the leaf public keys of the remaining users.

Each user processes a Remove message as follows:

- Decrypt s and combine $DKP(s)$ with the root key pair using $\star$.
- Delete the removed users' leaves and set all of their direct path key pairs to blank.

Note that after concurrent Remove messages, the resulting root private key is unknown to any removed user, but two users removed in different messages can work together to determine the root private key. This issue is noted by Bhargavan et al. but not addressed by them [3, §5].

To resolve this issue, we require: if a user wishes to "use" their current group state, either to encrypt a communication or to generate a state change message, after processing concurrent Remove messages, then they must first send a Remove message that removes the union of the concurrent messages' removed sets. This has the effect of sending a new secret to the remaining users only. (The same approach could be used with ordinary TreeKEM.)

Update messages concurrent to a Remove message are processed as usual, noting that their non-root private key contributions might be overwritten by blanks. Because they still contribute to the root private key, we intuitively expect to get the usual post-compromise security guarantees.

To prevent a removed user from altering the group state, we adopt the rule that no user will process an Update message from a removed user, unless another user generates a state change message that causally depends on it, presumably because they had not yet received the Remove message. Note that any Update messages that do get processed for this reason are harmless, since they cannot undo the Remove message's contribution to the group state.

Remark 3.1. There remain issues with adjudicating concurrent membership changes, e.g., handling two users who remove each other, allegedly concurrently (but possibly actually one in response to the other). These seem orthogonal to the topic at hand.

3.3 Adding Users

A user A adds a user B as follows. First, A generates an Update message u but does not send it. Next, A sends a Welcome message to B containing:

(i) The current set of group members.

(ii) The root private key and public key tree for the ratchet tree resulting after A processes u.

(iii) A's direct path private keys in the ratchet tree of (A, S), for any configuration S such that there may exist a state change message, concurrent to the Add message (below), whose set of causal predecessors is S. Note that the direct path private keys exclude the root private key.

This Welcome message is sent encrypted under a public key which B has designated for use in encrypting Welcome messages to them. This public key could be a signed prekey that B previously uploaded to an untrusted server, as in Signal [7].

To process the Welcome message, B uses (i) and (ii) to set their initial state. B also stores (iii) so that they can decrypt the root private key contribution of any state change messages concurrent to their Add message (below), acting as if they were A, and combine those contributions with their root private key using $\star_1$. This is necessary because concurrent root private key contributions will be combined into future root private keys using $\star_1$ in the usual way, but they are not included in the root private key in (ii).

A then sends an Add message to the group, including B, containing u and a public key which B has designated for use as an initial leaf public key. This public key could be a second signed prekey. Every user except B processes u as a normal update. Every user, including B, then:

- Adjusts the tree structure to accomodate a new leaf for B while keeping it left-balanced.
- Sets all key pairs on B's direct path to blank, then sets B's leaf's public key to that in the Add message. Note that the root is not set to blank.

By our rules for blank nodes, the effect of any concurrent Update message is to adjust nodes outside of B's direct path as usual, including the root node, while changes to nodes on B's direct path will be overridden by the Add message's blank nodes. Thus we intuitively expect to get the usual post-compromise security guarantee for such an Update message. Note that B can decrypt the Update message's contribution to the root key pair using (iii) from the Welcome message.

An exceptional case is when B's Add message changes a concurrently updating user's location in the tree, either because they convert the updating

user's old leaf into their new parent or because of concurrent Add messages (see below). In this case, we adopt the rule that the updating user's originally intended contribution to the root stands. That is, if the updating user generated a new secret s and they were originally at depth n in the tree, then the root key pair contribution is $DPK(H^n(s))$. We do this because all users can compute $H^n(s)$ in the usual way, while they may not be able to compute $H^{n-1}(s)$ if the updating user's depth decreases to $n-1$.

Concurrent Add messages work essentially the same as an Update message concurrent to an Add message. The exception is if concurrent Add messages attempt to insert multiple users at the same leaf node. We adopt the rule:

- Users process concurrent Add messages in some fixed order, e.g., lexicographically by name.

This works because neither the Add nor Welcome messages for a new user include any information that is specific to the leaf where they are added. Update messages which depend on an Add message may become nonsensical due to concurrent Add messages, in that they do not send secrets to the correct users, but any non-root key pairs affected by this will be overridden by blanks from the concurrent Add messages. Note that every user can still compute the root private key contribution from such an update.

4 Security Properties

Of course, Causal TreeKEM is useless if it is not secure. In this section, I'll try to informally motivate some of the security properties we want, although these arguments should not be taken too seriously. We consider passive network adversaries who also have the ability to compromise users, but who cannot actively interfere except to delay message receipt. We assume that users process state change messages in causal order regardless of the adversary's actions.

4.1 Post-Compromise Security

The post-compromise guarantee that we want is as follows. Let U be the set of users who have been compromised before a given point in time. Suppose there is a series of Update messages $m_1 < m_2 < \cdots < m_{|U|}$ such that:

- Each m_i was generated after the moment in which its author's most recent compromise occurred.
- The set of authors of $m_1, \ldots, m_{|U|}$ equals U.

There may be any number of intervening or concurrent updates. Then the private keys in the ratchet tree of any user who has processed $m_{|U|}$ are unknown to the adversary.

Let us informally define some games to make our guarantee a bit more precise, inspired by the proof of ART's security [4]. In these games, we assume that the root public key is never sent on the network, so that we can reason about the root private key's secrecy (see also Remark 2.4).

Let Adv_i denote the maximum over all polynomial-time adversaries $\mathcal{A}$ of the *advantage* of $\mathcal{A}$ in Game i, i.e., the probability that $\mathcal{A}$ wins Game i minus 1/2.

Definition 4.1. Let $\mathcal{C}$ be a set of user configurations, and let T be a configuration. A set of users U is *fresh at T w.r.t. $\mathcal{C}$* if there is a sequence of Update messages $m_1 < m_2 < \cdots < m_k$ in T such that for every $A \in U$ and every $(A, S) \in \mathcal{C}$, there is an m_i authored by A with $m_i \notin S$. A ratchet tree node N is *fresh at T w.r.t. $\mathcal{C}$* if the set of users with a leaf descended from N is fresh w.r.t. $\mathcal{C}$.

Game 1 (Security Game for Causal TreeKEM with Static Groups)**.** Fix a number of users n and a maximum number of Update messages M. The challenger initializes a group of size n with random key pairs at each node in the ratchet tree. The adversary makes a series of queries of the form:

- $Send(A)$, where A is a user: Instruct A to generate an Update message, apply it to their own state, and give it to the adversary.
- $Recv(A, m)$, where A is a user, m is the output of some call to $Send$, A has processed all of m's causal dependencies, and A has not yet processed m: Instruct A to process the Update message m.
- $Reveal(A, S)$, where A is a user, S is a set of outputs from calls to $Send$ that form a configuration, and A has previously processed all messages in S: Reveal the ratchet tree for the user configuration (A, S).
- $Test(T)$, where T is a set of outputs from calls to $Send$ that form a configuration: the challenger sets k_0 to be the root private key of T, sets k_1 to be a uniformly randomly sampled private key, chooses $b \in \{0, 1\}$ uniformly at random, and returns k_b.
- $Guess(b')$ where $b' \in \{0, 1\}$: This terminates the game.

The adversary wins if they call $Test$ exactly once, say with input T, they guess $b' = b$, the root node is fresh at T w.r.t. the set of all user configurations revealed by the adversary, and they call $Send$ at most M times. Otherwise, they lose.

Remark 4.2. The adversary should also have the ability to compromise the secrets contained in state change messages. From Causal TreeKEM's perspective, compromising such a secret is at most as bad as compromising some user's state immediately before and after sending the message, so we will be satisfied with just *Reveal* queries.

Game 2. Same as Game 1, but the adversary must specify at the start of the game the order in which they intend to make queries to *Send* and *Recv*, the user configurations on which they will call *Reveal*, and the configuration on which they will call *Test*. The inputs to *Recv*, *Reveal*, and *Test* are expressed in terms of the calls to *Send* that will generate them. If the adversary violates this specification, they lose.

In other words, the adversary must specify at the start of the game the causal dependency graph of Update messages that they will construct, plus the user configurations in the graph that they will reveal and the configuration in the graph that they will guess. We refer to the messages in this specification as *abstract messages*, since they represent Update messages but have not yet had their random values filled in.

Lemma 4.3. *There is a non-tight reduction from Game 1 to Game 2, proving that Adv_1 is at most Adv_2 multiplied by a (large) function of n and M.*

Proof Sketch. Let us be given an adversary $\mathcal{A}$ for Game 1 with parameters n and M. We define an adversary $\mathcal{A}'$ for Game 2 which first guesses the required information, then plays as $\mathcal{A}$. If their guess turns out be wrong, they play a trivial strategy that wins with probability 1/2. The advantage of $\mathcal{A}'$ is that of $\mathcal{A}$ times the probability that their guess is correct, which is the inverse of some (large) function of n and M. □

Definition 4.4. Define a *partial ratchet tree* to be a tree in which every node is labelled with a key pair, a public key, or null. Given two partial ratchet trees $\mathcal{T}, \mathcal{U}$ with the same shape, define their *product* $\mathcal{T} \star \mathcal{U}$ to be the partial ratchet tree whose label at a node N is given by

$$(\mathcal{T} \star \mathcal{U})_N := \begin{cases} (k_T, K_T) \star (k_U, K_U) & \text{if } (k_t, K_T) = \mathcal{T}_N,\ (k_U, K_U) = \mathcal{U}_N \text{ are both key pairs} \\ K_T \star_2 K_U & \text{if one of } \mathcal{T}_N, \mathcal{U}_N \text{ is a public key,} \\ & \quad \text{the other is a public key or key pair,} \\ & \quad \text{and } K_T, K_U \text{ are their public keys} \\ \mathcal{T}_N & \text{if } \mathcal{U}_N \text{ is null } (\mathcal{T}_N \text{ may also be null}) \\ \mathcal{U}_N & \text{if } \mathcal{T}_N \text{ is null.} \end{cases}$$

Given an Update message m, define the *partial ratchet tree of* m to be the partial ratchet tree that has every key pair that m contributes (i.e., the key pairs on its author's direct path plus the root) and is null elsewhere in the tree.

Game 3. Game 2 with two modifications:

- The order that the adversary commits to must be a total order on Update messages that all users agree on. That is, whenever the adversary calls $Send(A)$ and receives an Update message m, they must call $Recv(B, m)$ for all other users B before they call $Send$ again.

- The adversary has an additional oracle:

  - $StarIn(\mathcal{T})$, where $\mathcal{T}$ is a partial ratchet tree with the same shape as the users' ratchet trees and every node in $\mathcal{T}$ is labelled with a key pair or null (not a public key): Instruct every user to update their ratchet tree $\mathcal{U}$ by setting $\mathcal{U} \leftarrow \mathcal{U} \star \mathcal{T}$.

For any choice of starting information I for Game 2, let *Game 2.I* denote a modified version of Game 2 in which the adversary must specify starting information I or lose, and let $Adv_{2.I}$ denote its advantage. We similarly define *Game 3.J* and $Adv_{3.J}$.

Note that any maximum-advantage adversary for Games 2 or 3 can be converted into an adversary who deterministically chooses their starting information, with the same advantage. Thus it suffices to reason about the advantages of Games 2.I and 3.J as I and J vary.

Lemma 4.5. *For any I with n users and $\leq M$ Update messages, there is a J with n users and $\leq M$ Update messages such that there is a tight reduction from Game 2.I to Game 3.J, proving that $Adv_{2.I} = Adv_{3.J}$.*

Proof Sketch. Let us be given an adversary $\mathcal{A}$ for Game 2.I. We assume $\mathcal{A}$ and I are such that $\mathcal{A}$ never trivially fails by violating rules such as freshness.

Let $(P, <)$ be the partial order on abstract messages determined by I, i.e., the causal dependency graph that $\mathcal{A}$ plans to construct. From I, we can find a sequence of abstract messages $a_1 < a_2 < \cdots < a_k$ in P that witnesses the freshness of the configuration T on which $\mathcal{A}$ will call $Test$. Let $C = \{a_1, \ldots, a_k\} \subset T$. From the total order C, we can derive a sequence of $Send$ and $Recv$ queries meeting the requirements of Game 3. Let J be given by: the order of $Send$ and $Recv$ queries is that given by C; the $Reveal$ queries are those of I but with their configurations intersected with C; and the $Test$ configuration is $C = C \cap T$.

We define an adversary $\mathcal{A}'$ for Game 3.J. $\mathcal{A}'$ poses as a challenger for Game 2.I and plays against $\mathcal{A}$, processing the queries of $\mathcal{A}'$ as follows.

- $Send(A)$: Let P' be the set of (abstract) messages generated by previous calls to $Send$. Let a be the unique abstract message authored by A such that $a \notin P'$ but all of a's causal predecessors are in P. (Uniqueness holds because A always includes all of their prior messages as causal predecessors of their next message.)

  If $a \notin C$, compute the tree $\mathcal{T}$ of public keys in the ratchet tree of the configuration $\{m' \mid a' \in P, a' < a,\ a' \text{ is the abstract message for } m'\}$.

Then generate an Update message m with author A using public keys from $\mathcal{T}$, using a random secret, and give this message to the adversary.

If $a \in C$, let $R \subset P'$ be the set of messages m' for which $\mathcal{A}$ has called $Recv(A, m')$. Let a' be the predecessor of a in C, and let $Q = \{a'' \in R \mid a'' \notin C, a'' \not< a'\}$. (In case a is the minimum element of C, we set $Q = R \setminus C$.) Note that by the previous case, $\mathcal{A}'$ knows the private keys in every key pair that appears in an Update message in Q, since $Q \cap C = \emptyset$. Let $\mathcal{T}_2$ be the product of all partial ratchet trees for Update messages in Q. In Game 3.J, query $StarIn(\mathcal{T}_2)$, then query $Send(A)$ and give its result to $\mathcal{A}$.

- $Recv(A, m)$, where A is a user, m is the output of some call to $Send$, A has processed all of m's causal dependencies, and A has not yet processed m: If m was generated by a call to $Send$ that reached the second case above, query $Recv(A, m)$ in Game 3.J.

- $Reveal(A, S)$, where A is a user, S is a set of outputs from calls to $Send$, and A has previously processed all messages in S: Query $Reveal(A, S \cap C)$ in Game 3.J. Interpret the answer as a partial ratchet tree $\mathcal{T}$. Let $\mathcal{T}'$ be the $\star$-product of $\mathcal{T}$ with every partial ratchet tree of an Update message in $S \setminus C$. Give $\mathcal{T}'$ to $\mathcal{A}$.

- $Test(T)$, where T is a configuration: Query $Test(C)$ in Game 3.J; let k be the result. Let $\mathcal{T}$ be the product of every partial ratchet tree of an Update message in $T \setminus C$, and let l be its root private key. Give $k \star_1 l$ to $\mathcal{A}$.

- $Guess(b')$ where $b' \in \{0, 1\}$: Query $Guess(b')$ in Game 3.J.

Essentially, we are using $StarIn$ to simulate the effects of the messages in $P \setminus C$. Note that because $\star_1$ is cancellative, the guess by $\mathcal{A}$ will be correct if and only if the guess by $\mathcal{A}'$ is correct. To fill in the proof, we would need to carefully check that C, the input to $Test$, is fresh in Game 3.J because T is fresh in Game 2.I, plus various other details. □

Corollary 4.6. *There is a tight reduction from Game 2 to Game 3, proving that* $Adv_2 \leq Adv_3$.

Game 4. Game 2 with two modifications:

- The order that the adversary commits to must be a total order on Update messages that all users agree on. That is, whenever the adversary calls $Send(A)$ and receives an Update message m, they must call $Recv(B, m)$ for all other users B before they call $Send$ again.

- We use ordinary TreeKEM in place of Causal TreeKEM.

Game 4 should be essentially the "security game for ordinary TreeKEM", but with the adversary constrained to choose their Update message order, *Reveal* queries, and *Test* configuration at the beginning of the game. I expect that any reasonable proof of TreeKEM's security would imply that Adv_4 is negligible.

The only difference between Games 3 and 4 is that at various points in time, each user replaces their ratchet tree with the $\star$-product of that ratchet tree and some other partial ratchet tree. Sometimes this partial ratchet tree is provided by an adversary through $StarIn$; sometimes it comes from past state, since Causal TreeKEM processes Update messages using $\star$ instead of overwriting. Either way, because $\star_1$ is cancellative, these products should not affect which private keys are "secret" to an adversary. Thus I expect that we could modify a proof that Adv_4 is negligible to prove that Adv_3 is negligible, so that Causal TreeKEM is secure if ordinary TreeKEM is secure.

Remark 4.7. Since internal nodes have their public keys revealed, we cannot reason about indistinguishability of their private keys from random. We may need to instead make some computational hardness assumption about $\star_1$, e.g., if it is hard for an adversary to compute y, then it is also hard to compute x given $x \star_1 y$. This follows from the cancellative property of $\star_1$ if it is easy to invert, since one can use x, $x \star_1 y$, and an inversion algorithm to compute y.

4.2 Forward Secrecy

In the discussion so far, we have implicitly assumed that users keep around all of the state change messages they have received forever, so that they can process arbitrarily delayed state change messages. In practice, users should delete these messages after some amount of time and keep only the aggregate state, i.e., the ratchet tree corresponding to some configuration that includes all deleted messages. I expect that deleting old messages in this way will provide forward secrecy, similar to how we achieve post-compromise security.

Note that there is a trade-off here between the degree of forward secrecy and the amount of delay that we can tolerate. In applications with a single central server, we can generally assume a delay of at most a few seconds, but in applications that allow the group of users to partition and re-merge, state change messages may be delayed for much longer.

4.3 Adding Users

The procedure for adding users is rather suspicious because it requires the adder to give away a good deal of private key material to the new user. In this section, we give informal evidence that this is not a serious concern.

First, we address "forward secrecy" with respect to the added user. This means that the added user should not be able to read any communications

that do not causally depend on their Add message.

Proposition 4.8. *Suppose a user A adds a user B. Let S be a configuration that does not include the Add message, and let s be the root private key of S. Assume that the root private key contributions of the messages in S and the group's initial root private key are all independent and uniform random. Then as a random variable, s is independent of the private keys in parts (ii) and (iii) of B's Welcome message.*

Proof Sketch. First, write $s = v \star_1 w$, where v is the group's initial root private key and w is the combined contribution of all messages in S. By our independence assumption, v is independent of w and of part (iii) of B's Welcome message, since (iii) does not include any root private keys. Since v is uniform random and $\star_1$ is cancellative, regardless of w, s is independent of (iii).

Next, let x be the root private key in part (ii) of the Welcome message. Write $x = y \star_1 z$, where y is the root private key contribution of the Update message u used to generate the Welcome message, and z is the root private key of the preceding configuration. By our independence assumption and the assumption that S does not include the Add message, y is independent of z and s. Then x is independent of s, i.e., s is independent of the private key in (ii).

Finally, by the same reasoning as in the first paragraph, the private key x in part (ii) is independent of part (iii). Hence s, (ii), and (iii) are mutually independent, so s is independent of the private keys in (ii) and (iii) considered together. □

Thus informally, B can only determine s by breaking Causal TreeKEM. Note that our assumption on independent uniform randomness may fail if H is imperfect or there are hash collisions. Also, this proposition does not help us if B compromised the group's initial root private key at some point; in principle, we should be able to prove a similar security result even if B has previously compromised some private keys.

A second concern with our protocol for adding users is its interaction with forward secrecy and post-compromise security: what happens if an adversary who already possesses some secret information gets hold of a Welcome message, either by compromising B or colluding with them? Note that such a compromise is at most as bad as if the adversary compromised A. In most update schedules, I expect that all users will store roughly the same amount of old state, for the purpose of decrypting late messages. Thus an adversary who compromises B learns at most as much as they would from compromising any other user. This seems acceptable.

In addition, compromising B does not give the adversary any private keys from the Add message's key update u, except the resulting root private key. Hence it appears that if B is compromised, revealing the Welcome

message, then only B has to do a key update, not A. Thus we still get intuitive post-compromise security guarantees.

Finally, there may be issues if prekeys are reused in different conversations. Prekey reuse should be avoided if possible, but we cannot assure it asynchronously. If A gives away their initial leaf private key in an Add message in one conversation, and they are still using that leaf private key in another conversation, then they should send an Update message in the second conversation. Also, it is important that users designate which prekeys are for initial leaves and which are for Welcome messages, as we have described. If A gives away an initial leaf private key in one conversation that doubles as a Welcome encryption key in a second conversation, then the recipient automatically compromises A's initial state in that second conversation, allowing them to read early messages.

5 Miscellaneous

5.1 Elliptic Curve Diffie-Hellman Issues

The $\star$ operator for Diffie-Hellman keypairs described in Section 2 does not directly work with well-known elliptic curve DH groups, such as EC25519, due to issues with cofactors and clamping. Richard Barnes pointed this out on the MLS list originally [1] and would know more about it, but my impression is that it can be worked around (perhaps with a different, less well-studied curve).

6 Acknowledgements

Alastair Beresford and Martin Kleppmann provided helpful discussions and feedback on an initial draft. Any mistakes or issues are the author's alone.

References

[1] Richard Barnes. Delta-Remove may not be feasible. `https://mailarchive.ietf.org/arch/msg/mls/JQIhiT2fU_wXHPA_R6qpnKNucmo`, 2018.

[2] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-03.txt, IETF Secretariat, January 2019. `https://www.ietf.org/id/draft-ietf-mls-protocol-03.txt`.

[3] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous decentralized key management for large

dynamic groups. https://mailarchive.ietf.org/arch/msg/mls/v1CY0jFAOVOHokB4DtNqS__tX1o, 2018.

[4] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1802–1819, New York, NY, USA, 2018. ACM.

[5] P. Eugster, G. A. Marson, and B. Poettering. A cryptographic look at multi-party channels. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 31–45, July 2018.

[6] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12:1–12:38, December 2011.

[7] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol. Technical Report Revision 1, November 2016. https://signal.org/docs/specifications/x3dh/.