TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security

Théophile Wallez
Inria Paris

Jonathan Protzenko
Microsoft Azure Research

Karthikeyan Bhargavan
Cryspen

Abstract—**The Messaging Layer Security (MLS) protocol standard proposes a novel tree-based protocol that enables efficient end-to-end encrypted messaging over large groups with thousands of members. Its functionality can be divided into three components: TreeSync for authenticating and synchronizing group state, TreeKEM for the core group key agreement, and TreeDEM for group message encryption. While previous works have analyzed the security of abstract models of TreeKEM, they do not account for the precise low-level details of the protocol standard. This work presents the first machine-checked security proof for TreeKEM. Our proof is in the symbolic Dolev-Yao model and applies to a bit-level precise, executable, interoperable specification of the protocol. Furthermore, our security theorem for TreeKEM composes naturally with a previous result for TreeSync to provide a strong modular security guarantee for the published MLS standard.**

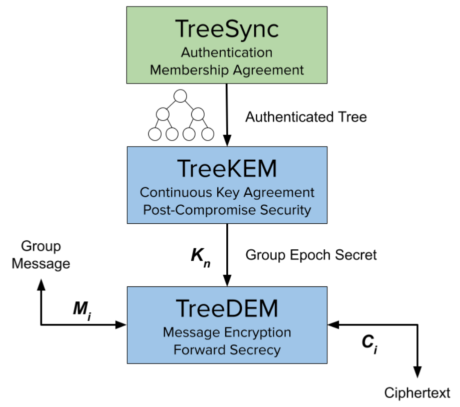


Figure 1: A Modular Treatment of Messaging Layer Security: TreeSync, TreeKEM, and TreeDEM

1. Introduction

The Messaging Layer Security (MLS) standard [9], published in 2023, is the first and till-date only Internet standard for secure end-to-end encrypted messaging. It is currently implemented by multiple messaging software vendors, including Cisco, AWS, Wire, and XMTP, and several vendors have announced their intention to support it in the future. With the advent of new regulations that require messaging interoperability, like the EU Digital Markets Act, an open standard like MLS is seen by many as the basis for the next generation of secure messaging applications.

Compared to popular and widely-deployed messaging protocols like Signal [2] and its many variants, the design of MLS distinguishes itself in two important ways.

First, MLS puts group messaging front and center and seeks to scale up to groups with thousands of members. To achieve this, MLS is built around a new tree-based protocol that scales logarithmically with the group size (in the ideal case) and linearly in the worst case. In contrast, protocols that build group messaging using two-party channels, such as Signal Sender Keys [7], scale linearly with group size in the best case and quadratically in the worst. Furthermore, these protocols do not provide important properties like membership agreement and post-compromise security for group conversations. With the growth in popularity of group messaging, and with the increase in message sizes entailed by post-quantum cryptography, the improved security and scalability of MLS is increasingly desirable.

Second, inspired by the experience of the Transport Layer Security (TLS) working group in the standardization of TLS 1.3, the design of MLS was structured as a collaboration between protocol designers and cryptographic experts with the goal of developing security proofs of the protocol alongside standardization. This process resulted in a number of formal security analyses of MLS (and its variants) using a variety of security models and techniques [3], [4], [6], [13], [15], [18], [21]. The current work is also a result of this long-term collaboration, and it contributes a new machine-checked security proof for TreeKEM, the core key agreement component of MLS.

MLS: TreeSync, TreeKEM, and TreeDEM. In previous work, Wallez et al. [21] identified a modular decomposition of MLS into three sub-protocols, as depicted in Figure 1:

- **TreeSync**: a protocol that synchronizes the shared group state across group members. The shared state includes the current group membership and is structured as a tree, with each occupied leaf corresponding to a member, and each internal node representing a subgroup. TreeSync uses signatures and Merkle-tree style hash computations to authenticate the initial group state provided to a member and all subsequent changes to the state. It also ensures that the tree data structure maintains an internal integrity invariant. This authenticated, synchronized state is then passed to TreeKEM.

- **TreeKEM**: a protocol that allows each member to use its private keys and the sequence of authenticated states provided by TreeSync to derive a sequence of group keys, called *epoch secrets* (K_n). TreeKEM uses the tree structure to efficiently update the epoch secret; in the best case, this requires only a logarithmic number of public key encryptions and a single decryption at each recipient. Furthermore, TreeKEM provides post-compromise security, and in particular, security against members that have been removed.
- **TreeDEM**: a protocol that takes the epoch secret K_n computed by TreeKEM and uses it to derive message encryption keys for each group member. These keys are then used to encrypt and decrypt group messages so that only the current members can send or receive them. After each message, the message encryption keys are ratcheted forward to provide forward secrecy.

When compared with a two-party secure channel protocol like TLS, TreeKEM corresponds to the handshake protocol, and TreeDEM corresponds to the record layer. Of course, the complexity of MLS is in handling the dynamic group setting where the list of participants can grow and change. While all three of these protocols are novel and deserve close scrutiny via formal security analyses, in this paper, we will focus on modeling and analyzing TreeKEM.

Security Analyses for Abstract Models of MLS. A key challenge when analyzing a protocol standard is in finding the right level of abstraction. The MLS standard is 132 pages long; it defines the high-level cryptographic constructions and algorithms of TreeSync, TreeKEM, and TreeDEM, but also defines the concrete tree data structure and operations on it, the precise low-level formats of all messages and cryptographic inputs, and handles the negotiation of versions and ciphersuites. Most prior works on analyzing MLS ignore most of these low-level details and instead model MLS as an abstract group key agreement protocol so that its specification can fit in a few pages and a formal proof of its security can be feasible.

Some works have analyzed the core key agreement of MLS with pen-and-paper proofs: [5] defines a new security definition called *continuous group key agreement* (CGKA) for protocols like TreeKEM; [3] presents a proof that TreeKEM as specified in MLS draft 7 is a CGKA; [4] presents a modular proof of MLS draft 11, by decomposing it into a CGKA protocol (essentially TreeSync+TreeKEM) and a stateful group AEAD protocol (i.e. TreeDEM); [6] analyzes the security of MLS draft 12 against malicious group members, by focusing on the integrity mechanisms within TreeSync. Other pen-and-paper proofs focus on aspects of MLS outside the core TreeKEM component: [15] analyzes the MLS key schedule, [17] studies post-compromise security for group messaging protocols like MLS.

There have also been some attempts at using (semi-)automated tools to obtain machine-checked symbolic security proofs for abstract models of TreeKEM: [13] analyzes the original version of TreeKEM [12] using a symbolic model in F* [20] and compares its security with alternate designs; [18] shows how a simplified version of TreeKEM can be analyzed in the Tamarin prover for forward secrecy (but not post-compromise or post-remove security).

Several of these works suggest improvements to MLS, some of which were incorporated into the MLS standard before publication. Still other works present and analyze new group messaging protocols inspired by MLS, but we do not consider these works here. However, none of the proofs in these works applies to the published MLS standard, since they analyze abstract models that leave out many of the details and options that make MLS complex.

Verifying an Executable Specification of TreeKEM. In contrast to the above works, our work is directly inspired by the work of [21], which presents a proof for a bit-level precise, executable, testable specification of the TreeSync component of MLS. The advantage of working on such a specification is that one can run it against protocol test vectors, or test it for interoperability with other implementations, to gain confidence that the model we are proving security for is not missing any important protocol details.

Handling low-level details can be crucial for security. For example, none of the papers on MLS cited above precisely model the signatures used in MLS. Even the pen-and-paper security proof for MLS in [6] abstracts away from the formats of the signature inputs, which would have been tedious to handle in a manual proof, and assumes that these signatures cannot be confused for each other. As a consequence, this proof misses an important signature ambiguity attack on MLS, which was subsequently found in the machine-checked proof of TreeSync [21] which did model all the low-level signature formats.

In this paper, we present an executable, testable, interoperable model of TreeKEM and a security proof for this model using a symbolic proof framework called DY* (the same methodology as [21]). Consequently, our proof accounts for all the low-level details of the MLS standard, and our confidentiality theorem for TreeKEM composes with the authentication theorem for TreeSync in [21].

Contributions. We present the first machine-checked proof for the TreeKEM component of the published MLS standard. Ours is also the first proof for a bit-level precise, executable, interoperable specification of TreeKEM, which can be seen as a reference implementation. Our proof shows how to modularly compose the guarantees of TreeKEM and TreeSync, and provides some important insights on key management and erasure for MLS implementations and deployments. Finally, ours is likely the first machine-checked symbolic security proof for group key exchange in dynamic groups (supporting add, remove, and update), and the first to establishing properties like post-compromise security in the group setting.

Outline. We start with an informal, accessible description of TreeKEM (§2); next, we show how to capture TreeKEM in formal language, encoding its specification using the F* proof assistant (§3). Then, we state the security properties we proved (§4), with a precise and extensive discussion of potential paths to compromise, followed by insights about our proof techniques (§5). Finally, we discuss our results and conclude (§6).

2. The MLS TreeKEM Protocol

We now describe TreeKEM as specified by the MLS standard [9]. We start by describing the overall goals of TreeKEM (§2.1), then define the two main mechanisms of TreeKEM: the use of a tree to produce a fresh *commit secret* shared by the group, guaranteeing post-compromise security and remove-security (§2.2), and key schedule that provides forward secrecy and add-security (§2.3).

2.1. Goals of TreeKEM

The goal of TreeKEM is, at each epoch, to establish an *epoch secret* that is known to exactly the participants currently in the group. This epoch secret is then used in TreeDEM to derive the same message encryption keys at each participant. The functionality provided by TreeKEM is sometimes called *continuous group key agreement* [5].

Design constraints. The initial design of group key establishment in MLS was based around Asynchronous Ratcheting Trees [16] which used a tree of Diffie-Hellman operations to enable efficient asymmetric ratcheting for groups. TreeKEM [12] was proposed as a KEM-based more efficient alternative to ART. The current version of TreeKEM in the MLS standard is the culmination of multiple revisions and extensions since these early designs. It aims to satisfy several constraints; in particular, the protocol must i) handle dynamic groups (i.e. participants can join and leave the group over time), ii) be asynchronous, (i.e. participants are not required to always be online), iii) be efficient (i.e. scale better than linearly on the number of participants) and iv) provide security properties like key confidentiality, forward secrecy and post-compromise security. Goal iv) is the main topic of study in this paper.

Additionally, TreeKEM makes certain assumptions about the design of the overall system, which are captured in the MLS architecture document [11]. Notably, TreeKEM relies on an untrusted *Delivery Service*, which is tasked with receiving messages from individual group participants, and broadcasting them back to all other group participants. In other words, MLS is *not* a peer-to-peer service where messages are sent directly from one participant to another.

TreeKEM Terminology. MLS is an asynchronous, distributed protocol. The TreeKEM specification therefore distinguishes the construction (locally, by a participant) of operations over the group (e.g. addition and removal of participants), known as *proposals*; the bundling of possibly many such operations into a *commit*; the application of this commit to the group to reach the next epoch.

The matter of how competing concurrent commits (by two different participants) are dealt with falls outside the scope of the MLS protocol; this is a matter handled by the untrusted delivery service, which we do not cover in the present paper. MLS assumes that each participant receives a sequence of commits from the delivery service and attempts to process them in order.

Upon processing a commit, the group enters a new *epoch* and TreeKEM outputs an *epoch secret* for the group to use. In other words, each commit does a round of key derivation which produces a fresh epoch secret: intuitively, this means that should anything change with the group (the membership, a member's public key, etc.), then a new epoch secret will be derived for the group.

Crucially, a commit may apply a *path update* operation on the internal state of TreeKEM (explained in §2.2) and output a *commit secret*, which is used in the computation of the next epoch secret (explained in §2.3). Such path updates are mandatory except in *add-only commits*, a special flavor of commit that does not contain a path update operation (we leave the description of add-only commits to §2.3). We explain path updates and commit secrets in detail in the remainder of this section.

An additional element that flows into the construction of the epoch secret is the *group context*, which summarizes information about the current state of the group – as we will see later, this is important for the security proofs.

Security properties, informally. TreeKEM aims to offer several security guarantees on the epoch secret:

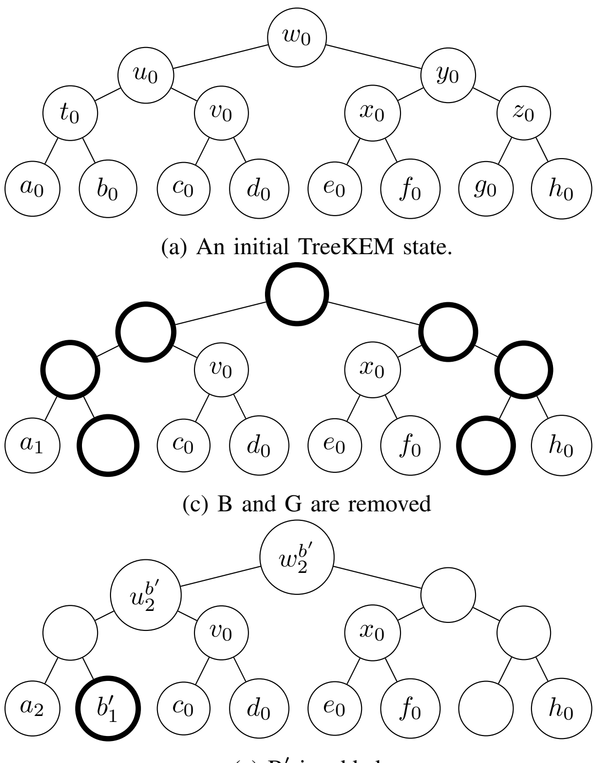
- add security (i.e. new participants must not know epoch secrets that predate their joining the group),
- remove security (i.e. removed participants must not know epoch secrets after they have left the group),
- forward secrecy (i.e. the compromise of a participant by an attacker must not reveal past epoch secrets) and
- post-compromise security (i.e. the epoch secret can eventually heal from a past compromise).

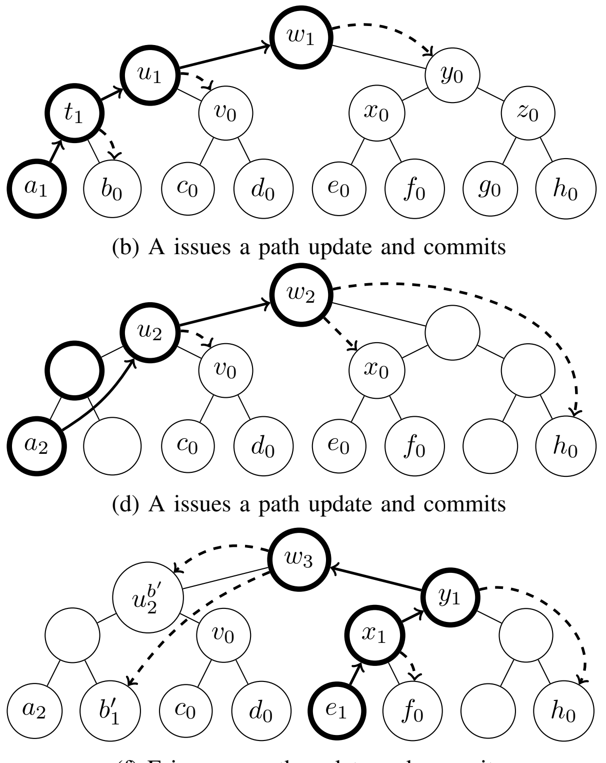
These properties will be more formally studied in §4.4. Suffices to say, for now, that forward secrecy and add security are achieved by judicious *key erasure* and that post-compromise security and remove security are achieved with by sharing *fresh randomness* (through the path update operation). In the rest of this section, we describe the state and mechanisms of the TreeKEM protocol and informally explain how it achieves these desired security guarantees.

2.2. A Tree for Group Key Agreement

Throughout this section, we will use uppercase letters to denote nodes of TreeKEM's tree (A to H for leaves and T to Z for internal nodes) and lowercase letters to denote the content stored in these nodes during the lifetime of the group (e.g. a_0, a_1, etc).

In TreeKEM, group participants are arranged in the leaves of a complete binary tree, as depicted in Figure 2a (nodes A to H). Each node contains a public-key encryption keypair, whose secret key is known by (and only by) the participants in the subtree rooted at that node (e.g. the secret key of v_0 is known to c_0 and d_0, and the secret key of the





w_1
u_1
t_1
a_1
(b) A issues a path update and commits
w_2
u_2
a_2
(d) A issues a path update and commits
w_3
y_1
x_1
e_1
b'_1
(f) E issues a path update and commits

Figure 2: Evolution of a group's tree in TreeKEM. Nodes in **bold** are the nodes updated by the current operation, plain arrow ($\longrightarrow$) indicates hashing the path secret and dashed arrow ($\dashrightarrow$) indicates encrypting the path secret (the cryptographic operations are detailed in Figure 3).

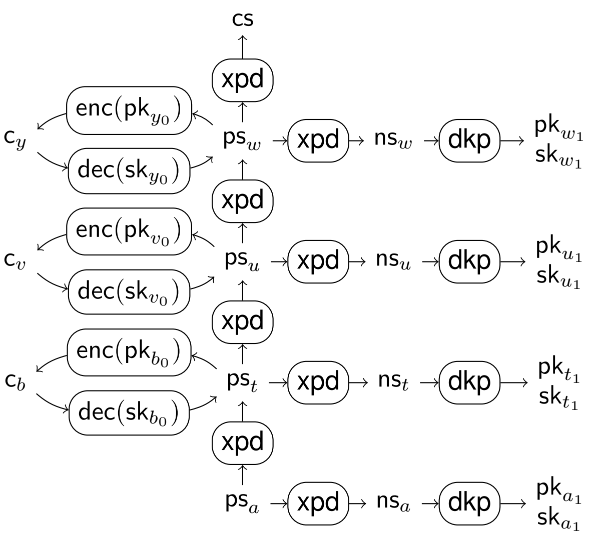


Figure 3: Cryptographic operations performed during A's path update in Figure 2b. xpd is HKDF.Expand, dkp is HPKE.DeriveKeyPair, enc is HPKE.Seal, dec is HPKE.Open, ps is "path secret", ns is "node secret", cs is "commit secret", c is "ciphertext".

root w_0 is known to every participant in the group). This property is called the *tree invariant* in the MLS standard [9].

By relying on the tree invariant, we can efficiently encrypt data to specific sub-groups in the tree. For example, we can perform one encryption to y_0 instead of separate encryptions to e_0, f_0, g_0 and h_0. This optimization is the essence of TreeKEM; we will see how it permits the efficient creation of path updates, i.e. refreshing secrets from a leaf node to the root without modifying every node in the tree.

For encryption, TreeKEM relies on the Hybrid Public Key Encryption (HPKE) construction [10], which uses a key encapsulation mechanism (KEM), a key derivation function (HKDF), and an authenticated encryption (AEAD) algorithm to build a public-key encryption scheme that provides integrity for the plaintext and for additional data.

We now describe how the group evolves through a series of updates depicted in Figure 2, and explain how each tree modification preserves the tree invariant.

Path update. In Figure 2b (and more precisely in Figure 3), A wants to recover from a potential compromise and benefit from post-compromise security properties of TreeKEM. Hence, it updates any secret (potentially compromised) it knows in the tree. To do so, A updates the HPKE keypairs of the nodes between its leaf and the root (i.e. of the nodes A, T, U, W, shown in **bold** in Figure 2b), while ensuring the new secret keys are known by participants in the corresponding subtree (e.g. the secret key of u_1 is transmitted to b_0, c_0 and d_0), and issues a new commit secret to the group.

Here is how the new secrets are generated. A generates an initial *path secret* ps_a (at the bottom of Figure 3), from which the new commit secret and all new keypairs will be derived. Here is how it happens: each updated node is associated with a path secret ($\mathsf{ps}_\square$ in Figure 3), from which two secrets are derived: the *node secret* for the same node ($\mathsf{ns}_\square$ in Figure 3), and the path secret for the node directly above (depicted as $\longrightarrow$ in Figure 2b). The node secret is used to derive a new HPKE keypair ($\mathsf{pk}_\square$, $\mathsf{sk}_\square$ in Figure 3). The commit secret (cs in Figure 3) is the path secret corresponding to the node that would be above the root.

Here is how the new secrets are transmitted to the participants that should know them (e.g. b_0 must learn the new secret key of t_1). The path secret of a node can be used to compute the secret keys of this node and all the nodes between them and the root, and hence can be used to compute the commit secret. Therefore, it is sufficient for every participant to obtain the path secret of the least common ancestor between them and A (the updater). We could use this criterion to encrypt one path secret to each group participant: this requires a linear number of encryptions in the size of the group. We can instead do a logarithmic number of encryptions, by relying on the fact that group participants that obtain a given path secret are arranged in a subtree, and benefit from the tree invariant to do only one encryption to this subtree root. Hence it is sufficient to encrypt ps_t to b_0, ps_u to v_0 and ps_w to y_0 (depicted as $\dashrightarrow$ in Figure 2b). The ciphertexts obtained ($\mathsf{c}_\square$ in Figure 3) and the new public keys are then sent to every group participant through the Delivery Service.

Removing participants. In Figure 2c, B and G are removed from the group. This action is performed using the concept of *blank node*. To remove B and G, the contents of their leaf nodes are erased: their leaf nodes are now blank. Without further action, this breaks the tree invariant: for example, B knows t_1 although it is not in its subtree anymore. As a drastic solution to restore the tree invariant, any node whose secret value is known by B (such as t_1) is also blanked, and the same is done for G (the nodes that were blanked are shown in **bold** in Figure 2c).

In Figure 2d, A issues a path update to create a commit secret known neither by B nor G, thereby obtaining security after their removal. Blank nodes make this operation more complex than in Figure 2b, we now describe how a path update is performed in this situation and introduce two new concepts: *filtered nodes* and *resolution*.

Path update & filtered nodes. In Figure 2d, A issues a path update to obtain security back after the removal of B and G. It happens similarly as in Figure 2b and Figure 3, with one difference about the path secret of T. In Figure 2b, the path secret of T is encrypted to B, but in Figure 2d, B is not here because it was removed in Figure 2c. There is no c_b as in Figure 3, hence computing a path secret for T does not achieve any purpose. The node T is therefore *filtered*: it stays blank, and what should have been its path secret is now the path secret of U, or graphically, the path-secret arrow ($\longrightarrow$) goes directly from A to U.

This optimization ensures that from the viewpoint of the tree invariant, there are no redundant non-blank nodes in the tree. Indeed, if during a path update, a bigger subtree (e.g. rooted at T) covers the same set of participants as a smaller subtree (e.g. rooted at A), because of the tree invariant their secret keys will be known by the same set of participants (e.g. $\{\text{A}\}$) hence the bigger subtree (e.g. rooted T) is redundant and its root (e.g. T) is filtered.

This filtering happens each time a bigger subtree covers the same set of participants as a smaller subtree. For example, if C and D were blanked, then the node U would also be filtered, and what should have been the path secret of T would become the path secret of W.

Path update & resolution. In Figure 2d (again), A issues a path update to obtain security back after the removal of B and G. It happens similarly as in Figure 2b and Figure 3, with one difference about the encryption of the path secret of W. In Figure 2b, the path secret of W is encrypted to Y, but in Figure 2d, Y is blank after the removal of G in Figure 2c. Instead, we encrypt ps_w with the smallest set of public keys such that all participants in the subtree rooted at Y can decrypt, here, x_0 and h_0. This set of public keys is called the *resolution* of the node Y. In the simplest case (as it happened in Figure 2b), the resolution of a node is the public key at that node (when it is not blank). In the other cases, the resolution of a node is computed by descending in the tree until encountering a non-blank node and collecting the public keys of all these non-blank nodes. This is how we find in Figure 2d that the resolution of Y is the set of public keys $\{x_0, h_0\}$.

Adding participants. In Figure 2e, B$'$ is added to the group. To perform this operation, we place the keypair of B$'$ in the left-most blank leaf. If there were no such blank leaf, we would extend the tree to the right, adding a new root whose left child is the current tree and right child is an all-blank tree, thereby doubling the number of leaf nodes and creating new blank leaves.

This operation breaks the tree invariant, which specifies that the private key of u_2 is known by (and only by) a_2, b'_1, c_0 and d_0. This is not true, because b'_1 doesn't know the private key of u_2 which was generated by the path update in Figure 2d when b'_1 was not in the tree at that time.

To account for this fact, we keep track that b'_1 doesn't know the private key of u_2: we say that b'_1 is *unmerged* for u_2. We do this bookkeeping for all nodes above b'_1, and note that b'_1 is also unmerged for w_2. With this new concept, we now reveal the complete formulation of the tree invariant: the secret key of each *non-blank* node is known by (and only by) the *merged* participants in the subtree rooted at that node.

Path update & unmerged leaves. In Figure 2f, E issues a path update. It happens similarly to Figure 2b and Figure 3, with one difference about the encryption of the path secret

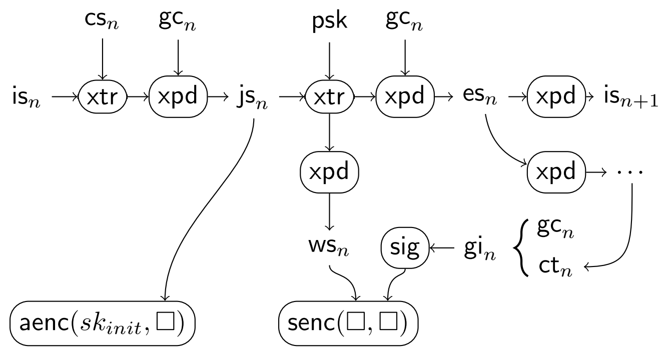


Figure 4: Cryptographic operations performed in the key schedule of TreeKEM (§2.3) and Welcome (§2.4). xtr is HKDF.Extract, xpd is HKDF.Expand, aenc is HPKE.Seal, senc is AEAD.Seal, sig is signature, is is init secret, cs is commit secret (see §2.2), gc is group context (appears 3 times), js is joiner secret, psk is pre-shared key, es is epoch secret (the output of TreeKEM), ws is welcome secret. gi is group info (see §2.4), ct is confirmation tag. In "…" are the various keys derived from TreeKEM. Decryption functions of the Welcome process (§2.4) are left implicit to simplify the diagram.

of W: it is encrypted to the resolution of u_2 which is the set $\{u_2, b'_1\}$. Indeed, recall that the resolution of u_2 is the smallest set of keys to cover all participants in the subtree of U, and b'_1 is unmerged for u_2, meaning that it does not know the private key at u_2. Hence, the resolution of u_2 must include the public key of all its unmerged leaves.

In Figure 2e, b'_1 was unmerged for w_2. Now, the path secret of w_3 has been encrypted to b'_1, hence b'_1 is *not* unmerged for w_3: a path update clears unmerged leaves on the updated nodes.

2.3. The MLS Key Schedule

Intuitively, the tree component of TreeKEM provides post-compromise security (because secrets are refreshed upon a path update), and remove security (because a new commit secret is derived after removing a participant). We leave a precise characterization of these security guarantees to §4, and continue our tour of TreeKEM, now describing the second component of TreeKEM: its key schedule.

Recall that the path secret above the root of the tree is the commit secret (§2.2). This secret has add-security, remove-security and post-compromise security. We can deduce this from the tree invariant: indeed, the commit secret is as secret as the root node secret key, hence is known by (and only by) participants in the tree, because after a path update the root has no unmerged leaves. However, this ensures only a weak form of forward secrecy: for example, compromising h_0 in Figure 2f would allow the attacker to decrypt the path secret of w_2 in Figure 2d (because it is encrypted with a key now known by the attacker and we suppose the attacker knows the ciphertexts), hence compute the commit secret of this previous epoch.

Strong forward secrecy. Therefore, the commit secret cannot be used directly for our purposes. We now explain how to derive the epoch secret (§2.1) from the commit secret; the epoch secret has the guarantees we desire, such as strong forward secrecy: a compromise of a participant should not reveal past epoch secrets. To obtain strong forward secrecy, the commit secret is injected into a *key schedule* from which the epoch secret is computed. The key schedule inherits all security properties of the commit secret, and further provides add-security and strong forward secrecy (independently of the commit secret security) because previous secrets can be erased upon key derivation. The key schedule is depicted in Figure 4 and is explained below. It is structured as a loop, we present the keys in the order they are derived, starting with the epoch secret and ending with the epoch secret of the next epoch.

Key schedule. The **epoch secret** (es$_n$ in Figure 4) is the main key established by TreeKEM, which is used to derive the keys used by TreeDEM (§2.1). It is also used to derive the next *init secret*, which serves to initialize the next epoch. The **init secret** (is$_n$ and is$_{n+1}$ in Figure 4) is combined with the commit secret and the *group context* to obtain the *joiner secret*. The group context is a summary of the current group state, in particular, it contains (in hashed form) the *initialization keys* (see §2.4) with which the joiner secret is encrypted (see below and §2.4), this will be crucial in the security proof in §5.3. The **joiner secret** (js$_n$ in Figure 4) is encrypted with the initialization key of new participants (or "joiners") in the group. (We explain initialization keys in §2.4.) It is then combined with the pre-shared keys and the group context to obtain the epoch secret (thereby closing the keyschedule loop), and is also used to derive the *welcome secret*. The **welcome secret** (ws$_n$ in Figure 4) produces a symmetric key that is used to encrypt the group context to new participants (as we will see in §2.4). Only a minimal amount of information is encrypted with the initialization key; the information that is the same for every joiner (such as the group context) is encrypted symmetrically via the welcome secret.

Add-only commits. Because the key schedule provides forward secrecy and add-security, when the set of group proposals since the last epoch only contains participant additions (hence contains no participant removal), it is not necessary to issue a path update to obtain a commit secret: instead, we can move to the next epoch using an empty commit secret. Doing such "add-only commits" still provides forward secrecy and add-security (because we do a round of key schedule) and remove-security (because we only do that when there were no removals). However, this doesn't provide post-compromise security (because no new randomness was injected in the key schedule), hence shouldn't be used when we want to recover from a potential compromise.

2.4. Welcoming New Group Members

We briefly mentioned how new participants join a group in §2.2 and §2.3, we now describe in depth how it happens. These explanations support the description of our security proofs in §5.

Key packages. Because participants are added asynchronously, they publish *key packages* on the Delivery Service, which can be used by any group member to add them to an MLS group. A key package contains a leaf node that is added to the tree (§2.2), and an *initialization key* (sk_{init} in Figure 4) that is used to encrypt the joiner secret (js_n in Figure 4), which bootstraps the key schedule. Notice that as we have described things, two asymmetric encryptions are required to add a new participant to the group: the joiner secret is encrypted with the initialization key (in §2.3), and the path secret is encrypted with the leaf node key (described in §2.2, but omitted from Figure 4). In reality, TreeKEM features an optimization and performs only one asymmetric encryption: both the joiner secret and the path secret are encrypted with the initialization key, in a package called *encrypted group secret* – this is the aenc node in Figure 4. Our TreeKEM API (§3.2) is designed to support this behavior, and the fact that the path secret is encrypted with the initialization key and not the leaf node key will also need to be taken into account in the security proof in §5.4.

Group info. To join a messaging group, it is not sufficient to know the group secrets. For example, the TreeKEM protocol (§2.2) requires each participant to know the tree of public keys, and the key schedule requires each participant to know the group context which summarizes the group state (gc_n in Figure 4). The tree may come from an untrusted source (e.g. the Delivery Service), and the group context is packaged in the *group info* (gi_n in Figure 4), which also contains a value derived from the current epoch secret, called *confirmation tag* (ct_n in Figure 4). The group info is further signed by a group participant, this will be crucial in the security proofs for the Welcome process (§5.2). Although the group info contains in principle only public data, it is opportunistically encrypted with the welcome secret (ws_n in Figure 4, see §2.3).

3. An executable specification of TreeKEM

In §2 we have explained at a high-level the inner workings of the TreeKEM protocol. We now describe how we specify TreeKEM in F* [20], a dependently-typed functional programming language. The specification is byte-level precise, passes the published test-vectors [1], and is used in a broader MLS specification that interoperates with other MLS implementations. Although the explanations in §2 are from a global viewpoint, we here specify the local computations performed by one TreeKEM participant.

3.1. TreeKEM's Tree in F*

Earlier work [21] modularizes MLS into three sub-protocols (TreeSync, TreeKEM and TreeDEM) and proves that the TreeSync sub-protocol authenticates all of the TreeKEM state. Our specification is based on their work, and in particular, we reuse their definition of trees to define TreeKEM trees as follows:

```
 1 type treekem_leaf = {
 2   public_key: bytes; }
 3
 4 type treekem_node = {
 5   public_key: bytes;
 6   unmerged_leaves: list nat; }
 7
 8 type treekem_public =
 9   tree (option treekem_leaf) (option treekem_node)
10
11 type treekem_private = path (bytes) (option bytes)
```

To each node is associated an HPKE keypair (§2.2); since we are implementing TreeKEM from the (local) point of view of a participant, we know the public HPKE keys of all participants; but we only know the private keys on the path from the leaf (us) to the root. Therefore, tree nodes and leaves contain public keys only (lines 2 and 5), and internal nodes additionally contain the list of unmerged leaves (line 6). An additional data structure, named treekem_private (line 11) contains the private keys known to us. Because nodes can be blank, we use the option type whose empty value represent blank nodes, except for the private HPKE key for leaf nodes (second argument of path line 11), because it points to our leaf that is non-blank.

We give an example of code that decrypts the path secret in Figure 5. This function searches for the least common ancestor between the updater and us (e.g. node U for participant C in Figure 2b), finds which ciphertext we must decrypt depending on our position in the resolution (e.g. second ciphertext for participant H in Figure 2d) and find for which private key it was encrypted (e.g. private key of u_2 for participant A in Figure 2f, but private key of b'_1 for B' because it is unmerged for U). We remark that this function exhibits many different behaviors depending on the participant executing it. This level of complexity, combined with the asymmetry between the sets of operations performed by different participants, is exactly why a mechanized proof of security is, in our opinion, necessary to trust that MLS provides the expected security guarantees.

3.2. TreeKEM API

Users of TreeKEM are not expected to use low-level functions as shown in Figure 5. Instead, they use a high-level API that handles modifications to both the public state (the tree of public keys) and the private state (our path of private keys from our leaf to the root). We structure our F* code to implement a high-level API that only exposes functions to process proposals and commits, and to generate commits. Note that the group management functions (add, remove,

```
val decrypt_path_secret:
  my_li:leaf_index → upd_li:leaf_index {my_li ≠ upd_li} →
  treekem_public → treekem_private → update_path →
  bytes
let rec decrypt_path_secret my_li upd_li t p_priv p_upd =
  if leaf_index_same_side t my_li upd_li then (
    // The update path and the path to our leaf are on the same
    // side of the tree. Recurse in that subtree.
    let (child, _) = get_child_sibling t upd_li in
    decrypt_path_secret child (next p_priv) (next p_upd)
  ) else (
    // We are at the least common ancestor between us and the
    // updater. Obtain the path secret by decryption.
    let ciphertext_list = get_data p_upd in
    let (_, sibling) = get_child_sibling t upd_li in
    // Find our ciphertext by descending in the tree until we find
    // a non-blank node, and recover the index in the resolution.
    let my_index = find_resolution_index sibling my_li in
    let my_ciphertext = ciphertext_list[my_index] in
    // Find the corresponding decryption key. This involves
    // checking whether we were encrypted to as an unmerged leaf.
    let private_key = find_private_key sibling (next p_priv) in
    // With all this data gathered, we can now decrypt.
    decrypt private_key my_ciphertext
  )
```

Figure 5: Implementation of the decrypt_path_secret function, simplified.

update) are part of the TreeSync API and are orthogonal to TreeKEM. We focus on the functions for TreeKEM commits as they are the most interesting.

Processing a commit. Each participant needs to process two kinds of commits: add-only commits (without path update), and full commits (§2.3). For this reason, we process commits in two steps: first, we update our state and compute the commit secret, second, we perform a round of key schedule. Furthermore, the first step comes in two flavors, one for each commit type.

```
val prepare_process_full_commit:
  treekem_state → path_update → group_context →
  result pending_process_commit

val prepare_process_add_only_commit:
  treekem_state →
  result pending_process_commit

val finalize_process_commit:
  pending_process_commit →
  pre_shared_keys → group_context →
  result (treekem_state & bytes)
```

These functions might fail, as indicated by the fact that return values are wrapped in a result. Reasons for failure include failed decryptions, or malformed path updates – the error case of result describes the nature of the error so that the client can act accordingly.

Creating a commit. Just like processing commits, creating new commits happens in two steps. In the case of a full commit, the first step, which handles the refresh of the tree and the commit secret, must itself be decomposed into two sub-steps, below.

```
val prepare_create_commit:
  treekem_state → entropy →
  result (pending_create_commit & pre_path)

val continue_create_commit:
  pending_create_commit →
  added_leaves:list nat → group_context →
  entropy →
  result (pending_create_commit_2 & path & list bytes)
```

The first function generates fresh path secrets and outputs the new public keys (in pre_path), for nodes along the affected path. The user can feed these new public keys into TreeSync to compute the new signature of our leaf node (that authenticates these new public keys) and compute a hash of the new tree – as mentioned earlier, we treat TreeSync as a signature primitive for the tree itself. This new tree hash is used within the group context, which is itself used when encrypting path secrets: this is what the second function does. It returns a pending commit creation object, an updated path, and the path secret that will be sent over to the joiners along with their welcome package to invite them into the group.

3.3. Execution model

One detail we omitted from our presentation (for conciseness and readability) is that all of those specification-level functions are actually parametric over the type of *bytes*, and over operations that operate on such bytes. We do so efficiently using the type class mechanism of F*.

This allows us to instantiate the specification either with concrete bytes (i.e. bitstrings) or with abstract symbolic bytes that are used in DY* [14] proofs (see §4.1). The former allows us to show that we are byte-for-byte conformant with the MLS standard, by running our specification (via F*'s extraction mechanism to OCaml) against test vectors and other implementations for interoperability testing. The latter, naturally, allows us to conduct our proof of security, in the next section.

Furthermore, we point out that our specification is free of any side effects: there is no memory (we never use a pointer or reference type), meaning the functions take, and return, a state, rather than modifying a global memory. Should some IO action (or, effect) need to happen, it suffices for the function to return, e.g., a list of messages to be effectively sent on the network.

To execute our specification and test it for interoperability, we wrote some glue code to allow our pure specification to interact with the effectful libraries such as networking. For proofs, we embed our specification in the trace-based semantics of DY*, as explained next.

4. A security theorem for TreeKEM

4.1. Background on DY*

DY* [14] is an F* [20] framework to state and prove security properties of cryptographic protocols. DY* uses a symbolic trace-based runtime model, where various participants can participate in a cryptographic protocol by calling cryptographic functions, generating random bytestrings, storing local state, logging events that indicate progress in the protocol execution, and sending messages on the network.

Threat model. DY* considers an active attacker that controls the network (hence can intercept, replay, or modify messages) and can dynamically compromise participants to learn the content of their private state.

Cryptographic assumptions. DY* abstracts cryptographic functions using the Dolev-Yao (or symbolic) model [19]. The symbolic model treats cryptographic functions as being perfect: for example, when sending a ciphertext on the network, the attacker learns nothing about the associated plaintext unless the attacker knows the corresponding decryption key (e.g. by compromising a participant), in which case they also learn the content of the plaintext.

Security theorems. DY* users can express security properties as *reachability properties*, meaning that all traces reachable through protocol execution satisfy some security property (specific to each protocol). An example of trace property that encodes confidentiality would be: if a participant finishes the key exchange protocol and the attacker knows the exchanged key, then the attacker must have compromised one of the participants involved in the key exchange.

Security proofs. DY* relies on its user to provide a *trace invariant*, then prove that each protocol step preserves the invariant (hence any reachable trace satisfies the trace invariant) and prove that the trace invariant implies the desired security properties. Note that the trace invariants are not trusted, they are only a proof technique to prove properties on all reachable traces. To define the trace invariant, DY* provides two tools. The first tool, related to confidentiality, are security *labels*, which encode an over-approximation of the compromises for an attacker to know some given bytestring. Hence, if the attacker knows some bytestring (e.g. the private signature key of participant p) then this bytestring's label ensures that the attacker must have compromised some particular state (e.g. the state where participant p stores their private signature keys). Some labels are more secure than others, in which case we say the less secure label *flows* to the more secure one (which we note $l_1 \gtrsim l_2$). Security labels will be the main workhorse of security proofs for TreeKEM (§5). The second tool, related to authenticity, are cryptographic predicates: for example, every participant will only sign messages that satisfy the (protocol-specific) signature predicate. When a signature verifies, we can then deduce that it was either computed by an honest participant, in which case the signature predicate holds on the message, or that it was computed by the attacker, in which case they know the signature key, hence must have performed some compromise depicted by the signature key security label. Signature predicates will also be a workhorse for security proofs in TreeKEM, mostly through the work of TreeSync [21].

4.2. Preliminaries

History of a group. In our security theorem, we consider everything from the viewpoint of a participant p belonging to a TreeKEM group G. This participant has seen the group evolve over time, resulting in several epoch secrets being established throughout the group's lifetime. At each epoch, participant p logs an event containing the information on their local group state at this epoch: the epoch secret K_n, the group roster $\mathrm{p}_1^{(n)}, \dots, \mathrm{p}_m^{(n)}$ (containing p), the joiners $\mathrm{j}_1^{(n)}, \dots, \mathrm{j}_q^{(n)}$ (contained in the group roster), the tree T_n (whose leaves form the group roster), and whether the commit is add-only. For example in Figure 2, assuming we start at epoch 0, we have $\mathrm{p}_1^{(0)} = a_0$ and $\mathrm{p}_1^{(1)} = a_1$, however, $\mathrm{p}_5^{(0)} = \mathrm{p}_5^{(1)} = \mathrm{p}_5^{(2)} = e_0$ and $\mathrm{p}_5^{(3)} = e_1$. Participants that didn't update since they joined have a special tag in T_n. When that is the case, we write $\mathrm{stale}_n(p')$; for example in Figure 2f, $\mathrm{p}_2^{(3)} = b'_1$ and $\mathrm{stale}_n(\mathrm{p}_2^{(3)})$. Naturally, joiners of this epoch didn't update since they joined, so $\mathrm{stale}_n(\mathrm{j}_i^{(n)})$. If we did not create the group, we have been invited in it by participant p_{inv} at epoch n_0. Furthermore participant p records the time at which they verify signatures: we write $\mathrm{T}(\mathrm{p}_i^{(n)})$ the time of verification of the signature of leaf node of $\mathrm{p}_i^{(n)}$, and $\mathrm{T}(\mathrm{j}_i^{(n)})$ the time of verification of the signature of key package of $\mathrm{j}_i^{(n)}$.

State storage. We consider a fine-grained model, where different parts of the state may be compromised independently. For instance, to account for a deployment that may use higher-security storage (e.g. HSMs) to store (long-term) signature keys, we can let the private node keys stored by a participant be compromised, without necessarily compromising the signature keys. Furthermore, we consider that different signature keys can themselves be compromised independently, just like initialization keys and epoch secrets. However, we consider that all nodes private keys are compromised together, since compromising one reveals all node private keys on the path from the compromised participant to the root.

State identifiers. We write $K_n@p$ to identify the state of participant p that stores the epoch secret K_n of group G at epoch n. We write $\mathrm{Sig}(\mathrm{p}_i^{(n)})$ to identify the state that stores the signature key of $\mathrm{p}_i^{(n)}$. We write $\mathrm{Init}(\mathrm{j}_i^{(n)})$ to identify the state that stores the initialization key of $\mathrm{j}_i^{(n)}$. We write $\mathrm{Node}(\mathrm{p}_i^{(n)})$ to identify the state that stores the node keys of $\mathrm{p}_i^{(n)}$ for the current version of $\mathrm{p}_i^{(n)}$. For example in Figure 2, $\mathrm{p}_5^{(2)} = e_0$ hence $\mathrm{Node}(\mathrm{p}_5^{(2)})$ corresponds to the node keys of $\{e_0, x_0, y_0, w_0, w_1, w_2\}$, while $\text{Node}(\text{p}_5^{(3)})$ corresponds to the node keys of $\{e_1, x_1, y_1, w_3\}$.

Notations. We write $\text{Att}_t(b)$ when the attacker knows the bytestring b at time t. We write $\text{Compromise}_t(S)$ when the attacker has compromised the state identified by S before time t.

4.3. Security properties

We now describe the security properties we have proved on TreeKEM. We state confidentiality as a trace property: if the attacker knows some epoch secret, then some set of states must have been compromised at some time in the past. In turn, we will see in §4.4 that this trace property implies the desired security guarantees of TreeKEM, such as add-security, remove-security, forward secrecy and post-compromise security (see §2.1). For the purpose of stating security goals in this paper, we assume that some state stores the epoch secret, but in our code, this secret is never actually stored since it would break forward secrecy of TreeDEM.

We consider three scenarios: in the first scenario, we consider a participant in a group that has moved into a new epoch, in the second scenario, we consider a participant that has just joined a group, in the third scenario, we consider a participant that has just created a group. These three scenarios cover all that may happen within an MLS group; indeed, advancing an epoch in the first scenario is done via a commit that may contain any number of add, remove, or other operations, and optionally a path update. These three scenarios come with different security guarantees.

Confidentiality theorem for new epochs. Suppose a participant p is in a group G at epoch n with epoch secret K_n, participants $\text{p}_i^{(n)}$, PSKs psks_n and joiners $\text{j}_i^{(n)}$. If $\text{Att}_t(K_n)$, then one of the following cases hold:

(1) $\exists i.\text{Compromise}_t(K_n@\text{p}_i^{(n)})$: the attacker has compromised before t the state containing the epoch secret of one of the participants $\text{p}_i^{(n)}$ in the current group.
(2) $\exists i.\text{Compromise}_t(\text{Init}(\text{j}_i^{(n)}))$ and $\text{Att}_t(\text{psks}_n)$: the attacker has compromised before t the initialization key used to invite the joiner $\text{j}_i^{(n)}$ into the group at epoch n.
(3) $\exists i.\text{Compromise}_{\text{T}(\text{j}_i^{(n)})}(\text{Sig}(\text{j}_i^{(n)}))$ and $\text{Att}_t(\text{psks}_n)$: the attacker has compromised the signature key of one of the joiners $\text{j}_i^{(n)}$ in the group at epoch n, namely the one that signed their initialization key. In that case, the compromise must have happened before we checked their key package signature. This is a variant of case (2) where the attacker is active.
(4) $\text{Att}_t(K_{n-1})$ and add-only_n and $\text{Att}_t(\text{psks}_n)$: the attacker knows the previous epoch secret, and the commit that led to epoch n is an add-only commit (as explained in §2.3).
(5) $\text{Att}_t(K_{n-1})$ and $\exists i.\text{Compromise}_t(\text{Node}(\text{p}_i^{(n)}))$ and $\text{Att}_t(\text{psks}_n)$: the attacker knows the previous epoch secret, and has compromised before t the node keys stored by a participant $\text{p}_i^{(n)}$ of the current group after they last issued a path update.
(6) $\text{Att}_t(K_{n-1})$ and $\exists i.\text{Compromise}_{\text{T}(\text{p}_i^{(n)})}(\text{Sig}(\text{p}_i^{(n)}))$ and $\text{Att}_t(\text{psks}_n)$: the attacker knows the previous epoch secret, and has compromised the signature key of a participant of the current group $\text{p}_i^{(n)}$, namely the one that signs their leaf node is the tree. In that case, the compromise must have happened before we checked their leaf node signature. This is a variant of case (5) where the attacker is active.
(7) $\text{Att}_t(K_{n-1})$ and $\exists i.\text{Compromise}_t(\text{Init}(\text{p}_i^{(n)}))$ and $\text{stale}_n(\text{p}_i^{(n)})$ and $\text{Att}_t(\text{psks}_n)$: the attacker knows the previous epoch secret, and has compromised before t the initialization keys stored by a stale participant $\text{p}_i^{(n)}$ of the current group. This possibility of compromise exists because the path secret is encrypted using the initialization keys of joiners. Note that p might not know what precise initialization key was compromised – it might be that p joined after $\text{p}_i^{(n)}$, meaning p never saw the key package of $\text{p}_i^{(n)}$. However, p knows it is *an* initialization key of $\text{p}_i^{(n)}$ that got compromised (i.e., p knows i).

Confidentiality theorem when joining. Suppose a participant p joined a group G at epoch n with epoch secret K_n, participants $\text{p}_i^{(n)}$, PSKs psks_n. If $\text{Att}_t(K_n)$, one of the following cases hold:

(8) $\text{Compromise}_{\text{T}(p_{inv})}(\text{Sig}(p_{inv}))$: the attacker has compromised the signature key of the participant that invited p. In that case, the signature must have happened before we checked the signature in the GroupInfo part of the Welcome message (as explained in §2.4).
(9) Participant p_{inv} (who invited participant p in the group G) belongs to a group G at epoch n with epoch secret K_n, participants $\text{p}_i^{(n)}$, PSKs psks_n and joiners that are a subset of $\{\text{p}_i^{(n)} \mid \text{stale}_n(\text{p}_i^{(n)})\}$. In that scenario, we have all the hypotheses required to apply this theorem inductively on p_{inv}.

Confidentiality theorem when creating. Suppose a participant p created a group G (hence at epoch 0) with epoch key K_0. If $\text{Att}_t(K_0)$, then:

(10) $\text{Compromise}_t(K_0@p)$: the attacker has compromised before t the state containing the epoch secret of participant p (the only participant in the group).

Malicious participants. In previous MLS drafts, TreeKEM was vulnerable to attacks wherein a malicious participant could break the tree invariant and compute the epoch secrets after they are removed from the tree, hence breaking remove-security (e.g. "double-join attack" in [13, Fig 5 and Fig 8], or "attack on tree-signing" in [6, Fig 8]). In DY*, we model malicious participants as participants whose complete state is fully compromised: this has the same effect as if they were the attacker. Hence our security theorem accounts for malicious participants in its threat model, and we will see in §4.4 that it entails remove-security, making such attacks impossible. Indeed, the "double-join attack" [13, Fig 5] was since fixed by introducing the concept of blank nodes, and the "attack on tree-signing" [6, Fig 8] was since fixed by introducing the concept of parent hash and formally analyzed as part of the TreeSync sub-protocol [21].

4.4. Security corollaries

Using the TreeKEM security theorem in §4.3, we can now prove as corollaries the desired TreeKEM security guarantees stated in §2.1. What follows is manual reasoning: we are auditing our theorem statement to make sure it does indeed provide the security guarantees we want.

Add-security, remove-security. The security theorem implies that necessarily, one of the participants in the current epoch must be compromised. Indeed, each of the cases (1) to (3), (5) to (8) and (10) implies the compromise of a participant of the current group, because $\mathrm{j}_i^{(n)}$ and p_{inv} are participants of the current group. By induction, (4) implies a compromise of a participant in the group at epoch $n-1$ which is a subset of participants at epoch n because the commit is add-only. In case (9), by instantiating the theorem inductively on p_{inv} we deduce that a compromise must have happened in the current group roster. This implies that if no group participant at epoch n is compromised, then compromising any participant that was removed or that is not yet added provides no useful knowledge to the attacker.

Forward secrecy. The security property implies that some compromise of keys must have happened in the past, or not too far in the future. Indeed, in the case of compromise of a signature (cases (3), (6) and (8)), the compromise must have happened before we checked the signature, hence in the past. In cases (1) and (10), because participants delete epoch secrets when moving to the next epoch, the compromise must happen before group participants move to the next epoch. In case (2), we notice that forward-secrecy relies on initialization keys being deleted quickly after processing a Welcome message. Not doing this undermines the forward-secrecy guarantees of MLS. We have found this was not part of the MLS deployment recommendations by the architecture document of MLS and notified the working group. In cases (4), (5) and (7), we do an induction on epoch $n-1$, and in case (9) we do an induction on participant p_{inv}.

Post-compromise security. The security theorem implies that a compromise cannot happen too far in the past. We can do a case analysis again. In cases (1) and (10) the compromise must have happened after $\mathrm{p}_i^{(n)}$ has computed the epoch secret K_n. Similarly, in case (5) the compromise must have happened after $\mathrm{p}_i^{(n)}$ has last issued a path update. In case (4), the group cannot heal from a compromise (unless psks_n is unknown to the attacker), hence we rely the healing of the previous epoch by doing an induction on epoch $n-1$. This means that if the group keeps doing add-only commits, there is no opportunity to recover from compromise (and implementations might need to adopt a policy encouraging updates to avoid this situation). In the cases of signature key compromise (cases (3), (6) and (8)) notice that such a compromise might have happened a while ago if the signature key is not rotated. This highlights that signature keys must be rotated regularly (so that it is changed before the attacker has the chance to forge a signature with it and perform an active attack) or stored securely e.g. in a hardware security module (HSM): this is recommended by the MLS architecture document. In cases (2) and (7), we note that to provide post-compromise security, the initialization key must not have been generated too long ago, otherwise this compromise may have happened far in the past. This highlights that key packages (hence initialization keys) must expire: adding a key package that was created too long ago could undermine post-compromise security. We have communicated to the MLS working group that this recommendation should be added to the MLS architecture document. The last case left to consider is (9), on which we do an induction on p_{inv}.

Lack of epoch authentication in Welcome. Note that in case (9) our theorem do not give the guarantee to the invitee (p) that that the inviter (p_{inv}) did invite them at this epoch. Indeed, in the presence of an active attacker, it may be possible that although the invitee successfully joined the group at epoch n, they were actually invited to join the group in a previous epoch (say, $n-1$). As discussed above, this cannot be used to affect the confidentiality guarantees of TreeKEM, hence is not a practical attack. We describe this more thoroughly in §A.

5. Proof methodology

We now discuss the methodology we used to prove the security theorem in §4.3. At a high level, we will use secrecy labels to prove that we only encrypt messages that are less secret than the key they are encrypted with, we will prove how secrecy labels evolve throughout the key derivations, and we will rely on the signature invariant when needed.

We describe our security proofs in the order keys are used in TreeKEM: we start with proofs on initialization keys (§5.1), then move on how they are used to encrypt the joiner secret in the Welcome message (§5.2), then see how the key schedule produces a sequence of forward secret epoch secrets (§5.3), and finally dive into the tree invariant proofs (§5.4).

5.1. Security lemmas for initialization keys

The first key used by a participant in a group is its initialization key (sk_{init} in Figure 4), to process the Welcome message (§2.4). In this section, we present security lemmas for initialization keys that will be crucial in proofs associated with the Welcome message (in §5.2) and with the key schedule (in §5.3). As with the rest of the proofs, we describe security properties from the viewpoint of a participant, at a specific time point. In what follows, a crucial design choice is that we store each key in a separate state, which allows us to talk about the compromise of a particular key, instead of the compromise of the whole state of a participant.

Lemma for a participant's own key. Each participant generates its initialization key from fresh randomness, stores it in its private state and only uses it to decrypt Welcome messages. As such, we expect the only way for the attacker to obtain the key is to compromise the participant's state. We formally prove this fact, by showing any reachable trace falls into two categories: (1) either the initialization private key is currently unknown to the attacker (2) or the attacker has previously compromised the state storing the initialization private key.

Lemma for others' initialization keys. Each participant receives the initialization public key of each other participant in a key package (described in §2.4). To prevent the attacker from tampering with keys, the key package is signed by the corresponding participant. We expect that if the signature was computed by a honest participant, they have honestly computed their initialization key, otherwise the attacker must have compromised the signature key before we verified the key package. We formally prove this fact, by showing any reachable trace falls into three categories: (1) (2) as in the paragraph above or (3) the attacker has compromised the other participant's signature key before we verified the key package.

Using security labels. We encode the trace properties above using security labels, namely we prove that if we have verified a key package, then $\mathrm{Init}(\mathrm{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathrm{j}_i^{(n)}) \gtrsim \mathcal{L}(sk_{init})$ where $\mathrm{j}_i^{(n)}$ is the joiner we are considering and $\mathcal{L}(sk_{init})$ is the label of the initialization private key in their key package. This labeling property allows us to prove that any reachable trace falls into one of the three categories mentioned above, and is composable with rest of the security proofs.

5.2. Security lemmas for Welcome

The Welcome message consists of two parts: first, the GroupInfo object (gi_n in Figure 4) is signed, and the joiner secret (js_n in Figure 4) is encrypted with the initialization keys of joiners (sk_{init} in Figure 4). We now see the security proofs related to these two cryptographic computations.

Encrypting the joiner secret. To prove that it is safe to encrypt the joiner secret with the initialization keys of joiners, we must prove that the joiner secret is *less* secret than the initialization private key (as explained in §4.1). We prove this by combining the theorem on secrecy label of initialization keys in §5.1 and the theorem on key schedule that we will prove in §5.3. Formally, we prove that $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathcal{L}(sk_{init})$ by transitivity using the chain $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathrm{Init}(\mathrm{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathrm{j}_i^{(n)})$ (proved in §5.3) and $\mathrm{Init}(\mathrm{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathrm{j}_i^{(n)}) \gtrsim \mathcal{L}(sk_{init})$ (proved in §5.1).

Signing GroupInfo. We use the epoch secret (es_n in Figure 4) to derive the confirmation tag (ct_n in Figure 4) and combine it with the group context (gc_n in Figure 4) to form the GroupInfo object (gi_n in Figure 4). Further, the GroupInfo is signed by the inviter. In doing so, the inviter attests that they are in a group with group context gc_n (which includes the epoch number, group identifier, a hash of the tree, etc), and with an epoch secret es_n that produces the confirmation tag ct_n.

Verifying GroupInfo. When the joiner verifies the GroupInfo signature, they deduce that either the attacker knew the inviter signature key before they have verified the GroupInfo (and the attacker is doing an active attack), or that the inviter is in a group with the same group context and with an epoch secret that produces the same confirmation tag. Finally, by collision resistance for the hash function, we deduce that we must have the same epoch secret. In doing so, we have proved the cases (8) and (9) of the security theorem in §4.3.

5.3. Security lemmas for the key schedule

The key schedule (Figure 4) derives a stream of secrets through extraction (xtr) and expansion (xpd). We prove two things about the key schedule. First, we prove that the epoch secret (es_n) combines the security of the commit secret (cs_n) and the previous epoch secret (es_{n-1}). Second, we prove that the joiner secret is less secret than the private key of joiners (sk_{init}). The first goal is easily proved using the semantics of extraction in DY*, we therefore focus on the joiner secret.

Labeling of the joiner secret. The joiner secret can be trivially compromised if the attacker knows is_n and cs_n. If not, the attacker can gain access to js_n if they compromise the initialization key sk_{init} of a joiner at epoch n. The joiner secret js_n is thus the first secret in the key schedule that is revealed to the attacker when they compromise sk_{init}. This fact means that the joiner secret is less secret than the secret that directly precedes it in the key schedule, hence that interesting proofs must happen in the expansion with the group context that produced js_n. In DY* the security label of the output of KDF.expand may be weaker than the label of its input, and it may depend on additional inputs of the KDF (here, the group context). Indeed, the group context contains the transcript hash, which contains the proposals adding the key packages of joiners in the group. This allows us to say that the label of the joiner secret is weakened using the label of the joiners' initialization keys. More precisely, we prove that $\mathcal{L}(\mathsf{js}_n) \gtrsim \mathrm{Init}(\mathrm{j}_i^{(n)}) \sqcup \mathrm{Sig}(\mathrm{j}_i^{(n)})$ which is then used in §5.2 to prove that it is safe to encrypt the joiner secret to each new joiner.

Note that the use of group context in the key derivation here is important for security, without it it would be possible that two group participants who don't agree on the key packages added at this epoch still compute the same joiner secret, which would break the security theorem.

5.4. Formally proving the tree invariant

The tree invariant follows the same principle as earlier (§5.1): first, we consider the security invariant from our own point of view, then when receiving an update from another participant, we consider the possibility that their signature key might have been compromised. To establish our security lemma, we rely on a tree invariant.

The tree invariant. The tree invariant captures the security guarantees offered by TreeKEM; we show that this invariant is preserved through every step of the protocol, which ultimately allows us to conclude that the cryptographic tree state of TreeKEM (§2.2) is secure. The tree invariant is a disjunction that captures the two points of view we mentioned earlier, and states that if the private key of a (possibly internal) node n is known by the attacker, then either $\text{Compromise}_t(\text{Node}(p))$ or $\text{Compromise}_{\text{T}(p)}(\text{Sig}(p))$, where p belongs to the subtree rooted at n and is not an unmerged leaf for n. The former disjunct captures the fact that a participant may simply have been compromised; the latter disjunct captures that we may have been the victim of an active attack, in which the attacker injects a malicious path update that is signed with another participant's compromised signature key. Note that the $\text{T}(p)$ in subscript indicates a temporal relation: the signature key must have been compromised *before* we verified that participant's leaf node.

Concretely, we prove this invariant by relying on DY* secrecy labels, described below.

Lemma on the sender side. We use labels to track the usage of path secrets throughout the specification of a commit. Every base cryptographic operation in DY* is annotated with labels in its type; this means that every usage of the path secret forces us to reason about the set of compromises by the attacker that would lead to knowledge of this path secret.

The label of each refreshed path secret (i.e., the output of the KDF) flows towards all of the sub nodes secrets. Looking back at Figure 2b, performing a KDF expansion with the path secret of t_1 produces the path secret of u_1. Because the label of u_1 covers participants a_1, b_0, c_0, d_0, it is weaker than the label of t_1 that covers participants a_1 and b_0 only. The path secret of u_1 is encrypted with v_0's node secret – this is a safe thing to do, because the label of u_1 is weaker than the label of v_0 (that covers c_0 and d_0), which is imposed by encryption in DY*: the label of the key must be stronger than the label of the message.

At this stage, we have almost obtained the tree invariant: if the attacker knows the path secret of u_1, it must have compromised a participant p that is one of a_1, b_0, c_0 or d_0. Because we chose the label of path secrets to be the same as that of node secrets, and combined with the tree invariant that previously held upon entering the function, it means that $\text{Compromise}_t(\text{Node}(p))$ or $\text{Compromise}_{\text{T}(p)}(\text{Sig}(p))$, and the invariant is re-established.

Lemma on the receiver side. We reuse an earlier formalization (and proofs) of TreeSync [21], in order to use TreeSync as a signature mechanism specialized for TreeKEM. This follows the same logic as with signing the initialization key, except this time the committer authenticates every subtree rooted at nodes they have modified (from their leaf up to the root), using TreeSync to do it efficiently with one signature in their leaf node.

The authentication covers the entire subtree, that is, the new node public keys, and all of their intended recipients, as they appear in the tree invariant. Using the semantics of DY*, we know that if the participant is honest, then the tree invariant is guaranteed by the signature (there was no compromise). If there is a compromise, it must be the case that the signature key was compromised before we checked the signature. This is one of the cases accounted for by the tree invariant, meaning that the invariant is re-established.

6. Discussion

We have presented a machine-checked security proof for a bit-level precise, executable, interoperable specification of TreeKEM. The specification is written in 1.5k lines of F* code, and our security proofs are in 4k lines of F*, relying on the DY* framework. The full development is available at the URL below, along with instructions for running the code and verifying the proofs:
https://github.com/Inria-Prosecco/treekem-artifact

Benefits of Machine-Checked Proofs. MLS is a large protocol and even its TreeKEM component is quite complex. It maintains a dynamic tree data structure with some unusual features such as unoccupied leaves, blank nodes, filtered nodes, unmerged leaves, etc. It defines novel cryptographic mechanisms for encapsulating secrets to trees of public keys. It defines new serialization formats for trees, paths, and various messages and cryptographic inputs.

Ensuring that a formal specification of TreeKEM captures all these notions correctly can be hard and is greatly aided by being able to execute and test the specification. Furthermore, when proving properties about the protocol, it is easy to forget various corner cases, but a machine-checked proof keeps us honest and ensures that we account for anything that may arise in an execution of TreeKEM. Indeed, we believe that a pen-and-paper proof for the full TreeKEM protocol at this level of detail would be hard to write and even harder to check for correctness.

Symbolic vs. Computational Proofs. Our proofs in this paper rely on a symbolic (i.e. Dolev-Yao) model of cryptography, where the public key encryption is treated as a perfect black-box that can only be broken if the attacker knows the private key. In contrast, the classic pen-and-paper proofs of TreeKEM in prior work [3], [4], [6] operate in a computational model of cryptography, where public key encryption is modeled in terms of a probabilistic polynomial-time adversary. Both symbolic and computational models have their strengths and weaknesses [8]. Computational cryptographic assumptions are more precise, but symbolic models yield better proof tools and hence can handle more protocol details and finer-grained key compromise.

For example, let us compare our work to the most recent pen-and-paper proof for TreeKEM [6]. This paper proves a computational security theorem for an abstract model of TreeKEM draft 12, expressed as pseudocode. Like us, they consider an active attacker that can dynamically compromise participants, and consider malicious participants. However, they only allow coarse-grained compromise: the attacker can only compromise all keys held by a participant, unlike our work where the attacker can compromise the node secret keys stored by a participant without compromising their signature keys. Another difference is that they consider bad randomness as part of the threat model, which we do not.

We also note that in MLS draft 12, there is no distinction between the initialization key and the leaf node key. This hurts forward-secrecy in their theorem since new participants need to hold on to this medium-term key even after joining a group. The TreeKEM design in the published MLS standard was updated to separate the initialization key and leaf node key, which yields a stronger theorem in our case.

The main proof in [6] shows that the attacker cannot distinguish the epoch secret from fresh randomness when a safety predicate (i.e. a trace invariant) holds. Much of the high-level logic in their proof and ours is the same; the main differences arise in the different treatment of public-key encryption, in our handling of low-level cryptographic formats (ignored in their proofs), and in our proofs being oriented towards being machine checkable.

Guidance for Erasing Keys, Removing Members. Our security theorem clearly specifies which key compromises could affect the confidentiality guarantees of TreeKEM. This provides useful guidance for MLS implementations and deployments. For example, initialization keys must be deleted after a Welcome message is decrypted or when they expire, and this is important for both forward and post-compromise security. Our theorem also includes cases for participants that have joined the group but not yet updated their encryption keys. Such *stale* participants affect the security of the whole group. They could be identified and potentially removed from the group after a period of inactivity. We have proposed to add these recommendations to the MLS architecture document.

Experimenting with Protocol Improvements. TreeKEM is designed with many defense-in-depth mechanisms; some were needed for our proofs (e.g. the use of group context), and some made our proofs simpler (e.g. the authentication of subtrees in TreeSync). Others we did not need (e.g. the use of group context in HPKE encryption), and this may indicate potential future optimizations in the protocol.

Some changes to the protocol would have simplified our proofs. For example, the transcript hash input format is defined as a concatenation, which makes it harder to prove that it is unambiguous. Using a length-prefixed format would have simplified this proof. As another example, our proofs for path secret derivation would have been much simpler if the sibling tree hash were used in the key derivation.

Our executable specification and machine-checked proofs provide a good basis for experimenting with different optimizations and variations of MLS. Running the specification makes it easy to compare the impact of optimizations on message size and computation time. Rerunning the proofs ensures that the new protocol satisfies the same properties as MLS, and maybe provides new security guarantees.

Future Work. Two natural directions for future work would be to develop a machine-checked proof of TreeDEM (and hence complete the verification effort for the MLS standard) and to investigate the post-quantum security of TreeKEM.

Acknowledgments

We are indebted to Franziskus Kiefer and Raphael Robert for proofreading a draft of this paper and providing precious feedback.

This work received funding from the French Government, managed by the ANR under grant agreements ANR-22-PECY-0006 and ANR-19-P3IA-0001.

References

[1] MLS test vectors. https://github.com/mlswg/mls-implementations/blob/main/test-vectors.md.

[2] Signal specifications, 2016. https://signal.org/docs/.

[3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277. Springer, 2020.

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1463–1483. ACM, 2021.

[5] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*, pages 261–290. Springer, 2020.

[6] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Paper 2020/1327, 2020.

[7] David Balbás, Daniel Collins, and Phillip Gajland. Whatsupp with sender keys? analysis, improvements and security proofs. In *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part V*, page 307–341. Springer-Verlag, 2023.

[8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 777–795. IEEE, 2021.

[9] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[10] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. RFC 9180, February 2022.

[11] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-15, Internet Engineering Task Force, August 2024. Work in Progress.

[12] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, May 2018.

[13] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[14] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.

[15] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the mls key derivation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2535–2553, 2022.

[16] 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*, page 1802–1819. Association for Computing Machinery, 2018.

[17] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1847–1864. USENIX Association, 2021.

[18] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-Based Proof Techniques for Improving the Automation and Scope of Security Protocol Analysis . In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 200–213. IEEE Computer Society, July 2023.

[19] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[20] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

[21] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, August 2023.

Appendix A. Lack of epoch authentication in Welcome

In TreeKEM (hence MLS), when an invitee joins a group through the Welcome procedure (§2.4), they do not have the guarantee that they were invited at this epoch: they may have been invited in a previous epoch. Indeed, an attacker can exploit the fact that during the Welcome process, only the GroupInfo is signed, but the encrypted group secrets are not signed, hence not bound to any epoch.

The attacker must be active, hence we suppose they control the network. The group is at epoch n, with a tree similar to Figure 2a, with E and F blanked. The attacker proceeds as follows:

- A invites E in the group and commits to epoch $n+1$. The attacker do not transmit the Welcome message to E.
- A invites F in the group and commits to epoch $n+2$.
- The attacker compromises the initialization key of F, and use it to decrypt the encrypted group secrets (i.e. joiner secret of epoch $n+2$ and path secret of node X).
- The attacker re-encrypts the group secrets with the initialization key of E.
- The attacker sends a Welcome message to E, containing this re-encrypted group secrets and signed GroupInfo for epoch $n+2$ (obtained when inviting F).
- E successfully joins at epoch $n+2$ although it was invited at epoch $n+1$.

This works by compromising an initialization key, but it works the same if F is a malicious participant (recall that we model malicious participants as participants whose state is fully compromised, see last paragraph of §4.3).

As explained in §4.4, this cannot be used to attack confidentiality guarantees of TreeKEM.