A preliminary version of this paper appears in the proceedings of the 45th Annual International Conference on theTheory and Applications of Cryptographic Techniques (EUROCRYPT 2026). This is the full version.

ETK: External-Operations TreeKEM and the Security of MLS in RFC 9420

Cas Cremers[1][0000−0003−0322−2293], Esra Günsay[1,2][0000−0001−6672−4253], Vera Wesselkamp[3][0009−0002−8481−9103], and Mang Zhao[4(✉)][0009−0002−4720−6551]

[1] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany
{cremers, esra.guensay}@cispa.de
[3] Hasso Plattner Institute, Potsdam, Germany
vera.wesselkamp@t-online.de
[4] School of Cyber Science and Engineering, Wuhan University, Wuhan, P.R.China
mang.zhao@whu.edu.cn

Version 2.0, April 11, 2026

Abstract. The Messaging Layer Security protocol MLS is standardized in IETF's RFC 9420 and allows a group of parties to securely establish and evolve group keys even if the servers are malicious. The core design of MLS is based on the TreeKEM protocol, which was significantly modified and extended during the standard's development. Over the last years, several partial security analyses have appeared of incomplete drafts of the standard. One of the major additions to MLS RFC 9420 (the final version of the standard) are the *external operations*, i.e., external commits and proposals. These additional operations have not been considered in any previous security analysis, while they can have a significant impact on the standard's security.

In this work, we prove the consistency, confidentiality and authentication of MLS in RFC 9420. To this end, we formalize ETK: External-Operations TreeKEM, which models RFC 9420 and includes the external commits and proposals. We propose a corresponding ideal functionality $\mathcal{F}_{\text{ECGKA}}$ and prove that ETK realizes it. Our work is the first cryptographic analysis that considers both the final changes to the standard, and the first approach overall to cover external proposals and external commits. Compared to previous works that considered MLS drafts, our ETK protocol is by far the closest to the final MLS RFC 9420 standard.

Our analysis implies that the core of MLS in RFC 9420 is an ETK protocol that realizes $\mathcal{F}_{\text{ECGKA}}$. Notably, we show that when external proposals and commits are allowed, MLS achieves a weaker form of security than was suggested by previous analyses, because the external operations can be exploited to violate Post-Compromise Security guarantees.

We show that the security of the protocol can be further strengthened by leveraging the standard's optional PSK mechanism, allowing another form of healing, and give a corresponding construction $\mathsf{ETK}^{\mathrm{PSK}}$ and ideal functionality $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$.

Keywords: Message Layer Security · MLS · Secure Messaging · Group Messaging.

1 Introduction

Messaging Layer Security (MLS) is a messaging protocol standardized by the Internet Engineering Task Force (IETF) in [12]. MLS was designed as a highly-secure asynchronous group messaging protocol that scales efficiently for large groups. MLS allows for dynamic membership and provides strong security guarantees such as Forward Secrecy (FS) and Post-Compromise Security (PCS). Informally, FS states that an attacker cannot decrypt messages exchanged before a compromise, while PCS [27] ensures that after a compromise, messages are secure after a *healing period*, in which clients exchange a few secure messages.

The MLS protocol is set to become the backbone of many globally-deployed messaging applications. MLS is already used in Cisco WebEx [7], Discord [33], RingCentral [40], XMTP [45], and Wire [44]. Google and Apple are implementing end-to-end encryption between Apple and Google devices using MLS (as part of RCS) [35]. Matrix [36] and Wickr [38] will also move to MLS. Finally, MLS provides the base of an IETF protocol for interoperable messaging services called More Instant Messaging Interoperability (MIMI) [39]. Thus, getting the security of MLS right is of critical importance.

The early development of the standard included rigorous security analyses by the scientific community, e.g., [1, 2, 3, 4, 23, 30, 42]. Most of these works focused on two MLS sub-protocols: *TreeKEM*, which defines how to compute a shared group secret via a tree structure, and the subsequent *key schedule* for message encryption. These two components are considered the core of MLS, providing PCS through regular updates to clients' key material, and FS through ratcheting the message encryption keys. While there is no computational analysis of the final version of the standard, the final version of the standard's basic operations were analyzed in the symbolic DY* model [43].

However, the full MLS protocol specifies operations beyond the basic operations analyzed in previous works. One of the main categories of additional operations are the so-called *external* operations, such as external commit and external proposal. For example, applications can allow group members who lost access to the group secrets to *resync* with the group by rejoining it. Further use cases of external proposals are automated services to remove inactive members, or to propose adding new staff members. Moreover, some groups may be open, and allow external members join directly by themselves. Although not considered in previous academic work, these, and further operations are used in real-world implementations of MLS [26, 38].

In this work, we analyze and prove formal security guarantees for the core TreeKEM protocol in the final MLS RFC 9420 including external operations. We build on a previous state-of-the-art analysis of MLS Draft 12 [4].

Contributions. This work presents the first computational security analysis of the final version of MLS RFC 9420, and the first work in general to consider MLS's external proposals and external commits. Our main contributions are:

1. We introduce ETK: Extended-Operations TreeKEM, which models the core TreeKEM protocol of the final version of the MLS standard in RFC 9420. Whereas previous cryptographic analyses analyzed Draft 12 (October 2021) or earlier, we consider the final version [12] (July 2023), which is based on Draft 20 [34]. Our ETK protocol captures the latest changes to the standard, such as encrypting `groupInfo`, separating keys for leaf nodes, and allowing multiple welcome messages. Notably, ETK is the first TreeKEM variant to include external operations, including proposals, commits, and resyncs/rejoins.
   We formalize the security of the ETK protocol in the ideal functionality $\mathcal{F}_{\text{ECGKA}}$, and prove that ETK realizes $\mathcal{F}_{\text{ECGKA}}$ with respect to active adversaries and potentially malicious insiders, yielding fine-grained guarantees. Our results imply that MLS as in RFC 9420, when instantiated with an SUF-CMA-secure signature scheme, provides strong security guarantees.
   However, we show that because of the external operations, MLS can only offer PCS security against session state compromise rather than full state compromise. This is notably weaker than the security guarantees proven in [4]. The reason is that MLS path updates can heal compromised session states, but not long-term secrets. An adversary who compromises the long-term secret of an honest party can at any time replace that party's representation in the group through external operations resync/rejoin. We expand on this in Section 5.2. Our results provide the first proof for the authentication concern incurred by external operations and addressed in RFC 9420 [12, Section 16.5], and further clarify the impact on confidentiality.
2. We propose ETK^{PSK}, an extension of ETK, in Section 6, and prove that it can improve PCS in the presence of external operation resync/rejoin, meeting a stronger security notion called $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$. Our ETK^{PSK} design is lightweight: we leverage the optional "resumption PSK" mechanism from RFC 9420 [12] whenever a member rejoins the group.

As a minor point, we show in Section 5.2 that contrary to previous claims, MLS can only realize $\mathcal{F}_{\text{CGKA}}$ [4] if the signature scheme is SUF-CMA-secure. RFC 9420 requires implementation of the IETF version of Ed25519, which satisfies SUF-CMA [22], but also allows ECDSA, which only satisfies EUF-CMA [41].

Outline. We recall background on MLS and related work in Section 2. We define our protocol syntax in Section 3 and our security functionality in Section 4. We introduce MLS RFC 9420 and its core operations in Section 5 and prove that it realizes our functionality in Section 5.2, where we also discuss the implications. We formally show the security improvements of PSK injections in Section 6.

We give detailed functionalities, protocol descriptions, and all proofs in the supplementary material in the appendix.

2 Background

We first recall some major changes in the evolution of the Messaging Layer Security (MLS) protocol in Section 2.1. In Section 2.2 we recall the latest existing formal security analysis for MLS. In Section 2.3 we introduce our notation.

2.1 The Evolution of the MLS Protocol

MLS is an asynchronous group key exchange protocol that aims to provide both high efficiency and strong security guarantees. MLS was initiated by the Internet Engineering Task Force (IETF) in February 2018 [17], based on the *Asynchronous Ratchet Tree* (ART) protocol [28]. ART allows a static group of n parties, each corresponding to a leaf node in a tree, to generate a shared group key, corresponding to the root in a tree, by recursively computing Diffie-Hellman Exchanges (DHE) from leaf nodes to the tree root. Compared to conventional group communication via pairwise channels, ART reduces the computational and communication efforts for every group member from $O(n)$ to $O(\log(n))$. Moreover, every group member can refresh their DHE keys along the path to the root, which allows ART to achieve several strong security guarantees, e.g., *forward secrecy* (FS) and *post-compromise security* (PCS).

Since Draft 2 [18], MLS has replaced ART with TreeKEM. TreeKEM has a similar tree structure to ART, but relies on a generic *Key Encapsulation Mechanism* (KEM) instead of DHE. This enables TreeKEM to achieve post-quantum security by selecting a suitable KEM instantiation. Every group member may rekey the secrets on the path from their leaf node to the root. The secret at the tree root is then used for message encryption. However, TreeKEM cannot deal with conflicts caused by concurrent message sending or group operations (e.g., add/remove members, key update), and impractically assumes that no two group members execute group operations at the same time.

To better handle concurrency, MLS added two mechanisms. First, with Draft 7 [8] MLS decreased the frequency of the key update from every message encryption to every "epoch", and passes the root key to a *key schedule* to initialize a *secret tree* (which was originally named "Tree-based Application Key Schedule" but renamed in the next draft) for continuous message encryption within every epoch. Second, with Draft 8 [9] MLS decomposed group operations into *Proposals* and *Commits* (also known as "P&C" mechanism). Whenever a group member wants to execute a group operation (e.g., add/remove members or key update), they first send a corresponding proposal. Any group member can then collect and process the received proposals in a commit. The committer is responsible for ordering the proposals and dealing with potential conflicts. Since Draft 8 until the final RFC 9420 standard, the ratchet tree (i.e., TreeKEM), key schedule,

	Version	Framework	Scope	Adversary	Operations
[28]	Draft 01	symbolic/game-based	static groups	active	-
[2]	Draft 06	game-based	integrity mechanisms	passive	O_1
[21]	Draft 07	symbolic	messaging	insider	O_1
[23]	Draft 11	state separating proofs	key derivation schedule	passive	-
[1]	Draft 11	game-based	rTreeKEM +group splitting attack	passive+	O_1
[30]	Draft 11	-	cross-group PCS analysis + concurrent group attacks	active	update-only
[4]	Draft 12	UC	integrity mechanisms + 3 insider attacks	insider	O_1
[42]	Draft 16	symbolic	integrity mechanisms + signature confusion attack	active	O_1
[43]	RFC 9420	symbolic	integrity mechanisms + signature compromise attack	active	O_1
This work	RFC 9420 (Draft 20)	UC	integrity mechanisms + randomness leakage attack	insider	$O_1 \cup O_2$

Table 1: Overview of previous formal analyses of the MLS protocol. *passive* adversaries can only eavesdrop the network. *passive+* adversaries can inject messages, provided these messages are ultimately rejected by a client. *active* adversaries can inject arbitrary messages that parties can process, and *insider* adversaries can additionally be group members. O_1 represents add, update, and remove proposals with regular commits and O_2 represents the external proposals and commits.

and secret tree form the core of MLS. The security of drafts of these has been formally analyzed in, e.g., [1, 4, 20, 23, 28, 30, 31, 42].

After in total twenty drafts, MLS was standardized as RFC 9420 in July 2023 [12]. Compared to basic TreeKEM, RFC 9420 incorporates several new operations and advanced features. These include group re-initialization, subgroup branching, joining via external commit, and optional PSK injection since Draft 10 [11]; separate keys for leaf nodes in TreeKEM from identity key package since Draft 13 [14]; external self-add proposal since Draft 15 [15]; and allowing multiple welcomes per commit since Draft 17 [16]. To the best of our knowledge, no existing work covers the formal security analysis for any of the above novel operations. We give an overview over previous formal analysis in Table 1. Beyond studies analyzing TreeKEM, some designs aim to enhance its features. Tainted-TreeKEM [37] improves efficiency and security by using a proxy to update encryption keys without blanking direct paths. Quarantined-TreeKEM [25] (QTK) introduces a "quarantine" for inactive users, who randomly blank paths and update keys via a distributed secret. QTK strengthens PCS if out-of-band authentication is possible and is compatible with RFC 9420. MLS-Cutoff [6] improves the communication complexity of MLS, in the case that the proposers and committers are chosen at random, from around $\sqrt{N}$ to $\Theta(\log N)$, where N denotes the size of groups.

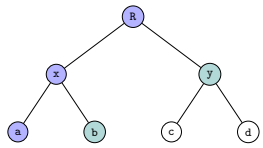
2.2 Formal Security Analysis of MLS Draft 12

The latest formal computational analysis for MLS was provided by Alwen, Jost, and Mularczyk [4] for Draft 12. The paper [4] isolated a core component in MLS Draft 12 called "*Insider Secure TreeKEM*" (ITK), and proved that it realizes a functionality $\mathcal{F}_{\text{CGKA}}$ in the Universal Composability (UC) framework. ITK assumes MLS Draft 12 to be equipped with two abstract services: an Authentication Service (AS), which manages long-term identity keys, and a Key Service (KS), which allows parties to upload single-use key packages, used by group members to non-interactively add them to the group. Below, we recall ITK and $\mathcal{F}_{\text{CGKA}}$.

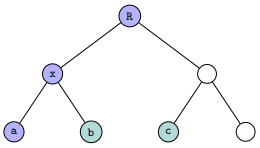
TreeKEM. Each member in TreeKEM is represented as a leaf node in the tree. The leaf node contains, among other things, the member's encryption key, public signature key, and a credential that links the member's identity to the signature key. When using, e.g., an X.509 credential, the public signature key of the leaf node is contained in the `subjectPublicKeyInfo` field. The AS validates that the public signature key matches an accepted identity (called a reference ID). An AS might be implemented as a service that signs the members' credentials, allowing other members to verify the credential by verifying the AS's signature. Members must be able to recognize that two credentials belong to the same member according to the application's policy, e.g., because they use identical signature keys. An update can replace the credentials used by a client. What constitutes a valid successor to a credential is application-defined.

Insider Secure TreeKEM ITK. ITK models core TreeKEM, extended with message authentication, tree-signing, computation of tags, and parts of the key schedule. Members suggest changes to the group by sending a *proposal* to all group members. The change is applied upon a member *committing* one or more proposals. A commit moves the group to a new epoch by updating the node secrets along the path of the committer, including the root secret. ITK covers TreeKEM with selected operations that can be invoked by group members, including group members' regular proposals and commits for adding new members, updating leaf node secrets, and removing existing members.

- Add Proposal: To add a new member, the proposer fetches a key package containing a fresh leaf node for the client from the KS and adds it to the proposal. With the commit, the new member's leaf node is added to the tree. The committer sends to the new member a *welcome* message that contains all information necessary for to join the group: the current group secret, encrypted with the new member's public encryption key, as well as the secret key of the lowest node in the ratchet tree that the committer and the new member share. The information in the welcome message is encrypted by the *init key* from the key package. The add proposal adds members to the leftmost free node.
- Update Proposal: An update proposal allows a member to rotate its encryption and optionally its signature keys. The committer of the proposal replaces the updating member's node with the new leaf node contained in the update proposal. It then "blanks" (i.e., deletes the content of) all intermediate nodes



(a) rekey path of `a`



(b) rekey path after removing `d`

Fig. 1: MLS group with four members: `a`, `b`, `c`, and `d`. The blue nodes ● are the secrets known to member `a` that are replaced upon performing a rekey path. The new node secrets are encrypted to the public key of green nodes ●, i.e., the resolution of their sibling nodes. The right figure shows the removal of node `d`, where the right child node of root is blanked and its resolution is `c`.

on the path from that leaf to the root (the blue path in Figure 1a). Afterwards, the committer updates its own path and encrypts the path secrets to other group members, in particular to the updated party using the updated leaf node's key. This replaces all knowledge of the updated party about the tree.

- Remove Proposal: A remove proposal specifies a member that should be removed from the group. Upon committing, the committer removes the leaf node of the member and blanks all nodes on the path of the removed member to the root. This guarantees that the removed member does not have access to any secrets in the public ratchet tree anymore. The committer updates its path secrets (including the new secret for the root) and encrypts the path secrets only to the existing group members. Figure 1b visualizes an example of the process.
- Regular Commit: A regular commit initializes a new epoch, validates and applies the collected proposals, and optionally rekeys the committer's path if the proposals are neither empty nor add-only. The committer rekeys the path from their leaf node to the tree root: refreshes their key pairs of their leaf node by simply generating a new key pair, and of the nodes on their path to root in the following four steps: (1) generate an initial path secret for the first parent node of the leaf, (2) recursively expand it to the next path secret for the next parent node, along the path to the root, (3) derive key pairs for all intermediate nodes on the path to the root from their path secrets, and (4) encapsulate the path secrets at every node on the path under the KEM public keys of the resolution of their sibling nodes. The resolution of a node is an ordered list of non-blank nodes that collectively cover all non-blank descendants of the node. After the optional rekey path execution, the committer runs "tree-signing", signs the content of the commit, executes the key schedule to derive the secrets for the new epoch, and compute tags. A regular commit message includes the information about the current epoch, identifier of the committer, content of the commit, and above signatures and tags, is output together with a new welcome message for new parties to join the group if available.

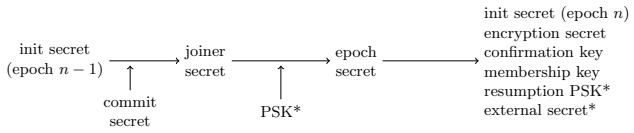


Fig. 2: Key derivation schedule for epoch n. *: PSK injection, resumption PSK, and external secret are not in ITK [4] but are part of MLS since Draft 10 [11].

Tree-Signing. The tree-signing guarantees newly joining parties that each internal node has been sampled by one of the parties contained in its sub-tree. Upon committing, the committer computes a so-called *parent hash* value that recursively binds every node with new key pairs on the committer's path (from the leaf to the root) to their sibling nodes and parent nodes.

Key Schedule. TreeKEM's ratchet tree provides an efficient way to share a new secret with all group members. The root secret of the ratchet tree is called the commit secret, as it is derived newly in each commit. The commit secret of the new epoch is then combined with a secret from the last epoch, called the *init secret*. In MLS Draft 12, another secret called a "pre-shared key" (PSK) can also be injected into the key derivation process, but these are not modeled in ITK. The derivation of the above secrets is called the *epoch secret*. As the name indicates, ITK derives from the epoch secret a number of group secrets for the computation within this epoch. We depict the key derivation schedule in Figure 2, where arrows denote various combinations of two HKDF operations: HKDF.Extract and HKDF.Expand. The most important group secrets are:

- init secret: used for key schedule in the next epoch.
- encryption secret: used to derive this epoch's secret tree.
- confirmation key: used for computing confirmation tags.
- membership key: used for computing membership tags.
- resumption PSK: used to ensure the membership consistency when the group is branched or reinitialized.
- external secret: used to derive a key pair whose private key is held by the entire group and whose public key can be shared with outsiders to enable them to join the group via external commit.

Computation of Tags. Commits include a confirmation tag and an optional membership tag if the commit includes a remove proposal. Confirmation tags guarantee the succession of the ratchet tree in every new epoch from the one in the past epoch, and authenticate the group's history. The ratchet tree is equipped with two hashes on the communication transcript: *confirmed transcript hash* and *interim transcript hash*. The confirmed transcript hash is computed by hashing the previous epoch's interim transcript hash, the content of the commit message, and its signature. The interim transcript hash is computed by hashing the confirmed transcript hash with the confirmation tag. Upon committing, the committer uses the confirmation key (from the key schedule) to MAC the confirmed transcript

hash, for a confirmation tag, which allows the receiving members to immediately verify whether they agree on the new epoch's key-schedule.

Note that the committer will not encrypt new path secrets to any party that is included in any remove proposal in the commit. Thus, the removed members cannot verify the confirmation tag, because they cannot execute the key schedule to derive the new epoch's confirmation key. In this case, the committer additionally MACs the content of the commit under the current epoch's membership key. Every group member can verify the commit under the same membership key to determine whether they will be removed in the next epoch.

UC Security and Functionality $\mathcal{F}_{\text{CGKA}}$. We briefly recall the UC framework [24] and the functionality $\mathcal{F}_{\text{CGKA}}$ in [4].

Universal Composability (UC). In the UC framework there exists an environment $\mathcal{Z}$ which can generate the inputs to, read all outputs from, and interact with two worlds: the *real world* and the *ideal world*. In the real world, an instance of a protocol P interacts with an adversary $\mathcal{A}$. In the ideal world, an ideal functionality $\mathcal{F}$, which perfectly simulates the behaviors of P without involving any cryptography, interacts with a simulator $\mathcal{S}$ (also called *ideal adversary*), which mimics every attack of $\mathcal{A}$. We say that the protocol P securely realizes the functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the real world (i.e., the instance of P and the adversary $\mathcal{A}$) or with the ideal world (i.e., the ideal functionality $\mathcal{F}$ and the simulator $\mathcal{S}$).

Functionality $\mathcal{F}_{\text{CGKA}}$ in [4]. The functionality $\mathcal{F}_{\text{CGKA}}$ in [4] captures a very strong adversary, which can (1) fully control the network of message and key packages distribution (except for the long-time identity keys on the authentication service `AS`), (2) adaptively corrupt parties' states, (3) adaptively corrupt and manipulate parties' randomness, and (4) potentially collude with malicious group members. In order to avoid the so-called commitment problem caused by adaptive corruptions in simulation-based frameworks, $\mathcal{F}_{\text{CGKA}}$ restricts the environment not to corrupt parties at certain times to prevent "trivial attacks".

$\mathcal{F}_{\text{CGKA}}$ captures three core security properties of ITK: consistency, confidentiality, and authenticity. Informally, consistency means that all parties in the same epoch agree on the same group state, including the history of the group's evolution. Confidentiality means that the adversary learns nothing about the important secrets in any epoch that is marked "safe", evaluated by a predicate **safe**. The secrets in the "safe" epochs will be derived from a random and independent secret, while the others will be chosen by the simulator $\mathcal{S}$. Finally, authenticity means that the environment cannot forge any message on behalf of a group member as long as the evaluation on a predicate **inj-allowed** is false.

The core theorem of [4] is that ITK, when used with an EUF-CMA signature, realizes $\mathcal{F}_{\text{CGKA}}$ in the random oracle model under the assumption of a modified *Generalized Selective Decryption* (GSD) security of the underlying PKE, which can be further reduced to the standard IND-CCA security. We will show in Section 5.2 that EUF-CMA is not sufficient for their main theorem.

Text color	Meaning
●	Updates from Draft 12 (as in [4]) to the final RFC 9420
●	External commit / resync
●	External self-add proposal
●	(Resumption) PSK proposals

Table 2: We use color coding to highlight the main new elements and functions.

2.3 Notation

Our notation follows [4]. Let $v \leftarrow x$ denote assigning the value x to v while $v \xleftarrow{\$} S$ denote sampling a random element from a set S. For a set V, $V \xleftarrow{+} x$ and $V \xleftarrow{-} x$ denotes adding and removing an element x. Let $A[i] \leftarrow x$ and $y \leftarrow A[i]$ respectively denote the assignment and retrieval of i-th element from an array A, while $A[*] \leftarrow v$ denote setting all array positions to the value v. For brevity, we sometimes omit set or list brackets when there is only one element and the type is clear from the context. Concatenation is denoted by $x \mathbin{+\!\!+} y$. We denote invoking an algorithm A with explicitly supplied randomness r by $A(.; r)$.

We use three keywords:

- **req** ⟨**condition**⟩: if the condition following the **req** keyword fails, the function returns $\perp$ and reverts all changes that were applied so far. The keyword **req** is used to control the allowed function inputs.
- **try** $y \leftarrow f(x)$ is equivalent to $y \leftarrow f(x)$ followed by **req** $y \neq \perp$.
- **assert** ⟨**condition**⟩: if the condition fails, the functionality permanently halts, making the ideal and real-world trivially distinguishable. The keyword **assert** encodes security properties of the protocol.

We use pattern matching to simplify the definitions of algorithms in our protocol and security model. For example, in $p \leftarrow (\texttt{Propose}, \texttt{rem-id}_t)$, `rem` is a keyword, while $\texttt{id}_t$ is the actual identity signifier of the removed party. In our algorithms and interfaces, we adopt a Python-style function definition with named parameters and default values.

We highlight the main new elements using the color coding in Table 2.

3 Extended Continuous Group Key Agreement: Syntax

In this section, we define our syntax for *Extended Continuous Group Key Agreement* (ECGKA) protocols. Our definition extends the definition in [4, Section 3.3] to include the additional operations: *external commit/resync*, *external self-add proposal*, and *PSK proposal*, which we describe in more detail in Section 5.1. We assume each party in ECGKA has a unique identifier `id`. For simplicity, we assume that every algorithm includes the `id` of the invoker (i.e., the party who runs this algorithm) as an implicit input.

Definition 1. *An ECGKA protocol includes the following algorithms:*

Group Creation: $(\texttt{Create}, \texttt{spk})$ *creates a new group with signature public key* $\texttt{spk}$ *and* $\texttt{id}$ *as the only member.*

Add Proposals: $p \leftarrow (\texttt{Propose}, \texttt{add-id}_t)$ *proposes to add the party* $\texttt{id}_t$*. The output is a proposal message* p *if* $\texttt{id}_t$ *is not yet a group member or* $\perp$ *if it is.*

Remove Proposals: $p \leftarrow (\texttt{Propose}, \texttt{rem-id}_t)$ *proposes to remove the party* $\texttt{id}_t$*. The output is a proposal message* p *if* $\texttt{id}_t$ *is a group member or* $\perp$ *if it is not.*

Update Proposals: $p \leftarrow (\texttt{Propose}, \texttt{up-spk})$ *proposes to update the member's leaf key and (if different from the current) signature public key* $\texttt{spk}$*. The output is a proposal* p *or* $\perp$ *if* $\texttt{id}$ *is not a member.*

Commit: $(c, \texttt{vec\_w}, g) \leftarrow (\texttt{Commit}, \boldsymbol{p}, \texttt{spk}, \texttt{force-rekey})$ *commits the vector of proposals* $\boldsymbol{p}$ *(that were previously generated) and outputs the commit message* c*. If* $\boldsymbol{p}$ *contains an add proposal, it outputs a vector of welcome messages* $\texttt{vec\_w}$*. It additionally outputs the group information (groupInfo)* g *for external joiners.* $\texttt{force-rekey}$ *determines if the committer's keying material is enforced to be replaced. Optionally, the committer's signature public key is updated to* $\texttt{spk}$*.*

Process: $(\texttt{id}_c, \texttt{propSem}) \leftarrow (\texttt{Process}, c, \boldsymbol{p})$ *processes the commit* c *that includes proposals* $\boldsymbol{p}$ *and advances the epoch to the next one. The output is the committer* $\texttt{id}_c$ *and* $\texttt{propSem}$*, a vector of the proposers and actions of* $\boldsymbol{p}$*. If* $\boldsymbol{p} = \perp$*, it assumes that* c *is an external commit and the proposals are directly included in the commit message.*

Join: $(\texttt{roster}, \texttt{id}_c) \leftarrow (\texttt{Join}, \texttt{vec\_w})$ *allows a new member* $\texttt{id}$ *to use a vector of welcome messages* $\texttt{vec\_w}$ *to join the group. The output is* $\texttt{roster}$*, the set of all group members' identities and signature public keys, and* $\texttt{id}_c$*, the identity of the committer of the add proposal.*

Key: $\texttt{K} \leftarrow \texttt{Key}$ *outputs the current application secret, i.e., the secret that* $\texttt{id}$ *derives for the current epoch and is expected to be used by a higher-level cryptographic application*[5]*.*

External Self-Add Proposal: $p \leftarrow (\texttt{Propose}, \texttt{extAdd-spk}, \texttt{epoch})$ *proposes to add the proposing member* $\texttt{id}$ *with the signature key* $\texttt{spk}$ *to the group. It outputs an external proposal* p*, or* $\perp$ *if the member is already in the group.*

PSK Proposals: $p \leftarrow (\texttt{Propose}, \texttt{psk-epoch-use})$ *proposes to inject the (internal) resumption PSK from an epoch into the key derivation schedule. It outputs a PSK proposal message* p*, or* $\perp$ *if the PSK is invalid.*

External Commit: $(c, g') \leftarrow (\texttt{Commit}, g, \texttt{spk}, \boldsymbol{e}_{psk}, \texttt{resync})$ *creates an external commit using groupInfo* g *and injecting PSKs from a vector of epochs* $\boldsymbol{e}_{psk}$ *that adds the committer to the group and (optionally) removes an old representation of the member (if* $\texttt{resync} = \texttt{true}$*). The operations return the commit* c *and the group info* g' *of the next epoch for external joiners.*

[5] For example, the encryption secret in Figure 2 that is used to derive a secret tree for real message encryption, see Section 2.1.

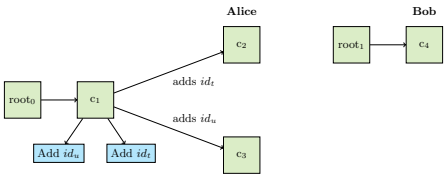


Fig. 3: An example history graph similar to [4]. The green and blue boxes respectively denote commit and proposal nodes. In the epoch created by the commit c_1, two proposals that respectively add parties $\mathtt{id}_u$ and $\mathtt{id}_t$ are generated and respectively committed by c_2 and c_3. Now the group splits and Alice chooses to process the commit c_2. Bob is invited to join the group via a welcome injected by the adversary. The history graph hence creates a new detached node root_1, and appends a new node c_4 to this root, to which Bob then joins.

4 Security Model and Functionality

We first define our extended notion of *history graph* in Section 4.1, and use this to present our extended generic functionality $\mathcal{F}_{\text{ECGKA}}$ in Section 4.2.

4.1 History Graph

A history graph is a labeled directed graph that can symbolically represent a group's evolution. The concept was introduced in [1] and adapted in [4]. A history graph consists of two types of nodes: *commit* and *proposal nodes*, which respectively represent all executed commit and proposal operations. We show an example history graph in Figure 3. For our work, we define an extended notion of history graphs, in which every node stores the following variables:

- orig: the creator `id` (i.e., the proposal or commit sender) of this node.
- `stat` $\in \{\mathtt{good}, \mathtt{bad}, \mathtt{adv}\}$: a status flag indicating whether the secrets in this node are secure (`good`), leaked (`bad`), or injected by the adversary (`adv`).
- pars: the list of parent commit nodes, representing the node's evolution history.[6]

Every *proposal* node further stores the following variables:

- `act` $\in \{\mathtt{up\text{-}spk}, \mathtt{add\text{-}id}_t\text{-}\mathtt{spk}_t, \mathtt{rem\text{-}id}_t, \mathtt{extAdd\text{-}spk}, \mathtt{psk\text{-}epoch\text{-}use}\}$: the proposed action with related information. We further explain these actions in Section 4.2.
- c_{psk}: specifies the commit, whose PSK secret this PSK proposal refers to.[7] This variable is only defined for PSK proposal `psk`.

[6] Our work lets all regular proposals and commits have a unique parent as in [4], while allowing the new external add and psk proposals to have multiple parents, as they can be applied to different commit nodes of the same epoch.

[7] If the adversary injects a PSK proposal on behalf of any party, then this PSK proposal might point to a node in a detached tree, where the commit itself and the resumption PSK value have not been fixed. In this case, we use the variable c_{psk} to denote the commit message that the PSK proposal refers to.

– psk $\in$ {good, bad, adv}: a status flag indicating whether the resumption PSK secrets in this node is secure (good), leaked to the adversary (bad), or injected by the adversary (adv). This variable is only defined for PSK proposal nodes.

Every *commit* node further stores the following values:

– pro: the ordered list of committed proposals. This variable is only defined for regular commits.
– mem: the list of group members and their signature public keys.
– sender_type $\in$ {'member', 'new_member'}: the role of sender that distinguishes regular (i.e., 'member') and external (i.e., 'new_member') commit nodes.
– ExtCommitProps: the ordered list of committed proposals. This variable is only defined for external commits.
– epoch: the sender's current epoch.
– has_psk: the list of group members who have derived the resumption PSK from the key schedule, see Figure 2.
– key: the group key.
– chall $\in$ {true, false}: a flag indicating whether the group key has been challenged (i.e., random, if chall = true) or not (i.e., chosen by the adversary or not generated yet, if chall = false).
– exp: a set that includes the corrupted parties in this node.

We compare with the history graph in [4] in Appendix H.

4.2 Functionality $\mathcal{F}_{\text{ECGKA}}$

We give a simplified version of our functionality $\mathcal{F}_{\text{ECGKA}}$ in Figures 4 and 5 with high-level versions of helper functions in Figure 6, due to the page limit. We give a detailed specification of $\mathcal{F}_{\text{ECGKA}}$ and all helper functions in Appendix D. Note that we define two functionalities in these figures: while we will present $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ in Section 6, in this section we focus on $\mathcal{F}_{\text{ECGKA}}$, which is defined in the figures by omitting the orange (●) code. Similar to [4], our $\mathcal{F}_{\text{ECGKA}}$ is also equipped with two additional functionalities $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$ and $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$ that capture the behaviors of authentication and key services (See Section 2.2).

Threat Model. We consider an adversary with the following strong capabilities:

(1) Fully control the network of message and key packages distribution, except for the long-time identity keys on the authentication service AS. This is captured by allowing the adversary (which is controlled by the environment $\mathcal{Z}$) to invoke each interface and to provide their inputs in $\mathcal{F}_{\text{ECGKA}}$, $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$, $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$, except for having the interface verify-cert directly return to the invoking party id.
(2) Adaptively corrupt the authentication and key service, i.e., obtain the secrets of any party id's identity or single-use key packages. This is captured by the expose interface with any id as input in $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$ and $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$.
(3) Adaptively corrupt any party's state through the Expose interface in $\mathcal{F}_{\text{ECGKA}}$.

(4) Adaptively control any party's randomness. This is captured by `CorRand` in $\mathcal{F}_{\text{ECGKA}}$ and `corRand` in $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$ and $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$ with $b = \texttt{bad}$ as input.
(5) Potentially collude with malicious group members and malicious servers. This is captured in all the above points.

$\mathcal{F}_{\text{ECGKA}}$ restricts the corruption in the following scenarios: once an epoch is "created" securely, and hence the group secret is generated randomly by $\mathcal{F}_{\text{ECGKA}}$, a corruption of this epoch is no longer possible, similar to [4].

Security Guarantees. $\mathcal{F}_{\text{ECGKA}}$ captures three core security guarantees: *consistency (and correctness)*, *confidentiality*, and *authenticity*.

Consistency. Consistency ensures that all parties in the same epoch agree on the group state, including the group membership and the history of group evolution (i.e., proposals and commits). We capture this by checking **assert *cons-invariant** at the end of every interface that causes an incremented epoch, i.e., `Commit`, `Process`, `Join`, and `ExtCommit`. We provide the high-level description of our ***cons-invariant** in Figure 6. Consistency holds (i.e., ***cons-invariant** = `true`) if and only if all the following conditions hold:

a) every (non-detached) commit c and all included proposals must be generated at the same parent commit node. This ensures that all committed proposals must be applied to the same group state.
b) every party `id` that can enter a new epoch created by any commit c only if `id` belongs to the membership specified by c. This ensures that all group members at the same epoch must agree on the same group membership.
c) the history graph contains no cycles. This ensures that the epoch always moves forward and never reverts.
d) every external self-add proposal must have the same epoch as all its parent commit nodes. This ensures that external self-add proposals will be only applied to a group with the epoch that this proposal aims at.

Confidentiality. Confidentiality means that the environment must have no information about the important secrets, i.e., *application secrets*, in any "safe" epoch, evaluated by a predicate **safe**. We capture this by deriving the application secrets in the "safe" epochs randomly, while having the simulator $\mathcal{S}$ choose them in other "unsafe" epochs. We show the **safe** predicate in Figure 7. We say that a commit c's epoch is safe if and only if the secrets of the epoch can neither be directly exposed, i.e., $(*, \texttt{true}) \in \texttt{Node}[c].\texttt{exp}$, nor indirectly leaked through other secrets, i.e., ***can-traverse**(c). Our ***can-traverse**(c) considers five cases:

Case a) regular commit node c is an orphan root with a corrupted signature key. An orphan root indicates a commit node that is injected by the adversary. Thus, a regular commit c is secure unless the adversary can inject any commit or welcome message with any group member's corrupted signature key.
Case b) regular commit c adds a corrupted member. The only difference to [4] is that our definition also considers a corrupted member added by an external self-add proposal. Thus, a regular commit c is secure unless no added signer in this epoch is corrupted.

Case c) the secrets in the welcome message are exposed by corrupting an added member in the future. The difference to [4] is that our definition additionally considers the welcome message for adding group members via external self-add proposal. Thus, a regular commit c is secure unless a member `id` is added in this epoch and corrupted in the future, without doing any update inbetween.

Case d) the adversary has corrupted sufficient state secrets from members' states to recover the secrets in this epoch.

Case e) the secret included in the external commit is insecure, i.e., either leaked to or injected by the adversary.

While our cases a) - d) are similar to the definitions in [4], with the main difference being the additional coverage of external self-add proposals, our case e) is newly introduced to capture the external commit operation. Notably, the distinction between cases d) and e) highlights that an epoch created by an external commit is expected to achieve weaker guarantees than one created by a regular commit: Once the secret included in the external commit is leaked (e.g., due to randomness leakage) or adversarially injected (e.g., via forgery), then the secrets in this epoch can be directly exposed—even if the group members' session states remain uncompromised.

Authenticity. We model authenticity as the guarantee that the environment cannot forge any message on behalf of any group member if the predicate **inj-allowed** is false. We consider the following four cases.

a) Every regular commit c can be injected by the adversary on behalf of any group member `id` with its valid leaf key `spk` only if both `spk` and the secrets of the epoch at the parent node $c_p = \texttt{Node}[c].\texttt{pars}$ has been exposed.
b) Every regular proposal p can be injected by the adversary on behalf of any group member `id` with its valid leaf key `spk` only if both `spk` and the secrets of the epoch at the parent node $c_p = \texttt{Prop}[p].\texttt{pars}$ has been exposed.
c) Every external commit c can be injected by the adversary on behalf of any group member `id` with its valid leaf key `spk` only if `spk` has been exposed.
d) Every external proposal p can be injected by the adversary on behalf of any group member `id` with its valid leaf key `spk` only if `spk` has been exposed.

Our cases 3) and 4) are newly defined for external proposals and commits, whose security requirements are expected to be lower than the ones of regular proposals and commits, defined in cases 1) and 2): Once a party's identity signature public key has been exposed, its external proposals and commits can always be injected by the adversary, even if the secrets of previous epochs remain secure. Combing with the above case e) requirement on confidentiality, the exposure of a party's identity signature public key can further yield the leakage of all secrets of following epochs.

Interfaces in $\mathcal{F}_{\text{ECGKA}}$. The functionality $\mathcal{F}_{\text{ECGKA}}$ is initialized with an empty history graph, i.e., all proposal and commit nodes $\texttt{Prop}[p]$ and $\texttt{Node}[c]$ are set to $\perp$ and the root is identified by the label $\texttt{root}_0$. For each party `id`, each vector

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ - Part 1

Parameters: The functionality expects as part of the instance's session identifier sid the group creator's identity $\texttt{id}_{\text{creator}}$. It is parameterized in the predicates **safe**(c), specifying which keys are confidential, and inj-allowed, specifying which injecting allegedly from id in c allowed.

Initialization

// Pointers, commit and prop nodes, welcome vectors
Ptr[$*$], Node[$*$], Prop[$*$], Wel[$*$] $\leftarrow \perp$
// Randomness and application secret status specifiers
RndCor[$*$] $\leftarrow$ good; HasKey[$*$] $\leftarrow$ false
rootCtr $\leftarrow 0$
GroupInfo[$*$] $\leftarrow \perp$ // groupInfo object for external commits, indexed by g
ExtProps[$*$] $\leftarrow \perp$ // External self-add, indexed by p

Inputs from $\texttt{id}_{\text{creator}}$

Input (Create, spk)
// The group can be created only once
req Node[root_0] $= \perp \wedge$ ***valid-spk**($\texttt{id}_{\text{creator}}$, spk)
// Transition of $\texttt{id}_{\text{creator}}$ as root creator
mem $\leftarrow \{(\texttt{id}_{\text{creator}}, \text{spk})\}$
// Create the root
Node[root_0] $\leftarrow$ new node with pars $= \perp$, orig $= \texttt{id}_{\text{creator}}$, pro $= \perp$, mem $=$ mem, stat $=$ RndCor[$\texttt{id}_{\text{creator}}$], epoch $= 0$
HasKey[$\texttt{id}_{\text{creator}}$] $\leftarrow$ true; Ptr[$\texttt{id}_{\text{creator}}$] $\leftarrow \text{root}_0$

Inputs from a party id

Input (Propose, act), act $\in$ {up-spk, add-id_t, rem-id_t, psk-epoch-use}
Send id and all inputs to the adv. and receive *ack*
// Adv. can reject invalid inputs
if $\neg$***req-correctness**('prop', id, act) **then req** *ack*
// Compute the proposal node
$P \leftarrow$ proposal node with pars $=$ Ptr[id], orig $=$ id, act $=$ act, stat $=$ RndCor[id]
if act $=$ add-id_t **then**
  // Adv. can choose the key package for adds
  Receive spk_t from the adversary
  act $\leftarrow$ add-id_t-spk_t
// Insert P into HG
Receive p from the adversary.
if Prop[p] $= \perp$ **then**
  Prop[p] $\leftarrow P$ // Create a new proposal node
else // Re-computing existing p
  assert ***consistent-node**(p, id, act)
if RndCor[id] $=$ bad **then** // Signed with bad randomness
  Notify $\mathcal{F}_{\text{AS}}^{\text{IW}}$ that id's spk is Exposed.
return p

Input (Propose, extAdd-id-spk, epoch)
req Ptr[id] $= \perp$
Send id, spk, epoch and all inputs to the adv. and receive *ack*
// Adv. can reject invalid inputs
if $\neg$***req-correctness**('ext-prop', id, extAdd-spk) **then req** *ack*
assert ***usable-spk**(id, spk)
act $\leftarrow$ extAdd-id-spk
Receive p from the adversary
if Prop[p] $= \perp$ **then**
  // Set c_{par} as all existing commit nodes in same epoch
  $c_{par} \leftarrow \{\forall c$ s.t. Node[c].epoch $=$ epoch $c\}$
  // Create a new proposal node
  $P \leftarrow$ external proposal node with pars $= c_{par}$
  orig $=$ id, act $=$ act, stat $=$ RndCor[id], epoch $=$ epoch
  ExtProps[p] $\leftarrow$ new node with epoch $\leftarrow$ epoch
else
  // Re-computing existing p
  assert ***consistent-node**(p, id, act, epoch)
if RndCor[id] $=$ bad **then** // Signed with bad randomness
  Notify $\mathcal{F}_{\text{AS}}^{\text{IW}}$ that id's spk is Exposed.
return p

Commit by Reference

Input (Commit, $\boldsymbol{p}$, spk, force-rekey, wel_type)
Send id and all inputs to the adv. and receive $(ack, c, \texttt{vec\_w}, g, rt, c_{psk})$.
// Adv. can reject invalid inputs
if $\neg$***req-correctness**('comm', id, $\boldsymbol{p}$, spk, force-rekey) **then req** *ack*
// Adv. interprets injected proposals.
for $p \in \boldsymbol{p}$ s.t. Prop[p] $= \perp$ **do**
  Prop[p] $\leftarrow$ proposal node with related pars, stat $=$ adv, and orig and act chosen by the adversary.
// Check if PSK proposal refers which of the epoch commits
res $=$ ***fixate-psk-refs**((Prop[p] : $p \in \boldsymbol{p}$), c_{psk})
assert res $\neq \perp$
// If add/psk-only proposals, no need to update path
if ***keep-spk**($\boldsymbol{p}$, force-rekey) **then**
  spk $\leftarrow$ Node[Ptr[id]].mem[id] // Set the signature key
// Check if id has PSK and is able to calculate this commit
for $p \in \boldsymbol{p}$ s.t. Prop[p].act $=$ psk-$*$ **do**
  req id $\in$ Node[Prop[p].c_{psk}].has_psk
// Expand the history graph with this commit node
if Node[c] $= \perp \wedge rt = \perp$ **then** // Commit is previously unseen
  if ***keep-spk**($\boldsymbol{p}$, force-rekey) **then**
    // If no rekeying path executed or randomness leaked
    stat $\leftarrow$ bad
  else stat $\leftarrow$ RndCor[id]
  Node[c] $\leftarrow$ commit node with pars $=$ Ptr[id], orig $=$ id, stat $=$ stat pro $= \boldsymbol{p}$, and mem $=$ ***members**(Ptr[id], id, $\boldsymbol{p}$, spk)
else
  if $rt \neq \perp$ **then** // There is no detached root
    $c' \leftarrow \text{root}_{rt}$
    assert RndCor[id] $\neq$ good
  else
    $c' \leftarrow c$
    if $\neg$***keep-spk**($\boldsymbol{p}$, force-rekey) **then**
      assert RndCor[id] $\neq$ good
  Replace each occurrence of root_{rt} in the HG by c
  assert ***consistent-node**(c, id, $\boldsymbol{p}$, mem)
  if $c \neq c'$ **then**
    Attach detached root c' as successor of id's node
assert vec_w $\neq \perp$ **iff** $\exists p \in \boldsymbol{p}$: Node[p].act $\in$ {add-$*$, extAdd-$*$}
if vec_w $\neq \perp$ **then**
  $\boldsymbol{c_{psk}} =$ Prop[p].c_{psk} for all $p \in \boldsymbol{p}$ s.t. Prop[p].act $=$ psk-$*$
  assert Wel[vec_w] $\in \{\perp, (c, \boldsymbol{c_{psk}})\}$
  Wel[vec_w] $\leftarrow (c, \boldsymbol{c_{psk}})$
assert $g \neq \perp$
assert groupInfo[g] $\in \{\perp, c\}$
groupInfo[g] $\leftarrow c$
// Check that inserting C does not violate authenticity and HG-consistency
assert ***cons-invariant** $\wedge$ ***auth-invariant**
if RndCor[id] $=$ bad **then** // Commit signed with bad rand.
  Notify $\mathcal{F}_{\text{AS}}^{\text{IW}}$ that id's spk is Exposed.
return $(c, \texttt{vec\_w}, g)$

Input Key
req Ptr[id] $\neq \perp \wedge$ HasKey[id] // Only possible if id has the key.
// Set the key if id is the first party fetching it in its node.
if Node[Ptr[id]].key $= \perp$ **then**
  if **safe**(Ptr[id]) **then**
    Set key to a fresh random key and chall to true.
  else
    Let the adversary choose key and set chall to false.
// id should remove the key from his state
HasKey[id] $\leftarrow$ false
return Node[Ptr[id]].key

Fig. 4: Part 1 of our simplified $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, using the color coding from Table 2. The orange code (●) is included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, but not in $\mathcal{F}_{\text{ECGKA}}$.

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ - Part 2

Commit by Value

Input (ExtCommit, g, spk, e_{psk}, resync)
- **if** resync = true **then req** Ptr[id] $\neq \perp$ // id is a group member and want to resync, i.e., leave and rejoin
- **else req** Ptr[id] $= \perp$ // id is not a group member
- Send id, e_{psk} and all inputs to the adv. and receive $(ack, c, g', p_{exi}, p_{rem}, p_{psk}, c_{gm}, \text{orig}_g, \text{mem}_g, \text{epoch}, c_{psk})$.
- // Adv. can reject invalid inputs
- **if** $\neg$*req-correctness('ext-comm', id, g, spk, resync = true) **then req** ack
- **assert** *usable-spk(id, spk)
- // Check if the groupInfo maps to an existing commit.
- $c_g \leftarrow$ groupInfo[g]
- **if** $c_g = \perp$ **then** // Injected commit so create detached root
  - **if** Node[c_{gm}] $\neq \perp$ **then**
    - $c_g \leftarrow c_{gm}$
  - **else**
    - rootCtr + +; $c \leftarrow \text{root}_{\text{rootCtr}}$
    - Node[c_g] $\leftarrow$ create a new node with orig $\leftarrow$ orig_g, mem $\leftarrow \text{mem}_g$, stat $\leftarrow$ adv, epoch $\leftarrow$ epoch, psk $\leftarrow$ adv
  - groupInfo[g] $\leftarrow c_g$
- // Create sub-node of self-add proposals
- ExtCommitProps[p_{exi}] += new proposal with groupInfo[g], orig $\leftarrow$ id, act $\leftarrow$ 'exi-id'
- // Create sub-node of self-remove proposals, if exists
- **if** $p_{rem} \neq \perp$ **then**
  - ExtCommitProps[p_{rem}] += new proposal with groupInfo[g], orig $\leftarrow$ id, act $\leftarrow$ 'rem-id'
- **for** e, p in zip(e_{psk}, p_{psk}) **do**
  - ExtCommitProps[p_{psk}] += new proposal with groupInfo[g], orig $\leftarrow$ id, act $\leftarrow$ 'psk-e-application'
- **try** ExtCommitProps = *fixate-psk-refs(ExtCommitProps, c_{psk})
- **for** prop $\in$ ExtCommitProps **s.t.** prop.act = psk-$*$ **do**
  - **req** id $\in$ Node[prop.c_{psk}].has_psk
- $\boldsymbol{p} = [p_{exi}, p_{rem}, p_{psk}]$
- **if** Node[c] $= \perp$ **then** // Correct behavior
  - // Create a node by wrapping all above sub-nodes
  - Node[c] $\leftarrow$ create a node with pars $\leftarrow$ c, orig $\leftarrow$ id, pro $\leftarrow$ $\boldsymbol{p}$, stat $\leftarrow$ RndCor[id], psk $\leftarrow$ RndCor[id], sender_type $\leftarrow$ 'new_member', epoch $\leftarrow$ Node[c].epoch, mem $\leftarrow$ *members(groupInfo[g], id, ExtCommitProps, spk)
  - // External commit must have a correct member list
- **else**
  - **assert** *consistent-node(c, g, id, p, mem, ExtCommitProps)
- **assert** $g' \neq \perp$
- **assert** groupInfo[g'] $\in \{\perp, c\}$
- groupInfo[g'] $\leftarrow c$
- **assert** *cons-invariant $\wedge$ *auth-invariant
- **if** RndCor[id] = bad **then** // Commit signed with bad rand.
  - Notify $\mathcal{F}_{\text{AS}}^{\text{TW}}$ that id's spk is Exposed.
- **return** (c, g')

Corruptions

Input (Expose, id)
- **if** Ptr[id] $\neq \perp$ **then** // If id is in the group, its state contains:
  - // Secrets needed to process other parties' messages and potentially the group key
  - Node[Ptr[id]].exp +$\leftarrow$ (id, HasKey[id])
  - // Secrets needed to process id's own messages
  - For each commit or update-proposal node with orig = id and Ptr[id] $\in$ pars, set stat $\leftarrow$ bad.
  - Notify $\mathcal{F}_{\text{AS}}^{\text{TW}}$ that id's current spk is compromised.
- // Whether id is in the group or not, its state contains secrets needed to process welcome messages.
- **for** c **s.t.** *can-join(Node[c], id) **do**
  - Node[c].exp +$\leftarrow$ (id, true)
- // Disallow adaptive corruptions in some cases.
- This input is disallowed if $\exists c$: Node[c].chall $\wedge \neg$safe(c)

Input (CorrRand, id, b), $b \in \{\text{good}, \text{bad}\}$
- RndCor[id] $\leftarrow b$

Input (Process, $c, \boldsymbol{p}$)
- **if** $\boldsymbol{p} = \perp$ **then return** *process-ec(c) // The case of external proposal process
- Send id and all inputs to the adv. and receive $(ack, rt, \text{orig}', \text{spk}', c_{psk})$
- // Adv. can reject invalid inputs
- **if** $\neg$*req-correctness('proc', id, $c, \boldsymbol{p}$) **then req** ack
- **for** $p \in \boldsymbol{p}$ s.t. Prop[p] $= \perp$ **do**
  - Prop[p] $\leftarrow$ proposal node with stat = adv, , and related pars, and adv chosen orig, act
- // Which of the epoch commits the PSK proposal refers to
- **try** res = *fixate-psk-refs(Prop[p] : $p \in \boldsymbol{p}$, c_{psk})
- **if** Node[c] $= \perp \wedge rt = \perp$ **then**
  - // Check if id has the PSK and is able to process this commit
  - **for** $p \in \boldsymbol{p}$ **s.t.** Prop[p].act = psk-$*$ **do**
    - **req** id $\in$ Node[Prop[p].c_{psk}].has_psk $\vee \nexists$(id, $*$) $\in$ mem
  - Node[c] $\leftarrow$ commit node with pars = Ptr[id], orig = orig', stat = adv, pro = $\boldsymbol{p}$, and mem = *members(Ptr[id], id, $\boldsymbol{p}$, spk')
- **else**
  - // If c explains detached root, let adv specify it
  - **if** Node[c] $= \perp$ **then** $c' \leftarrow \text{root}_{rt}$
  - **else** $c' \leftarrow c$
  - $\text{id}_{c'} \leftarrow$ Node[c'].orig; $\text{spk}_{c'} \leftarrow$ Node[c'].mem[$\text{id}_{c'}$]
  - mem $\leftarrow$ *members(Ptr[id], $\text{id}_{c'}$, $\boldsymbol{p}$, $\text{spk}_{c'}$)
  - **assert** mem $\neq \perp$
  - // Check if id has PSK and is able to process commit
  - **for** $p \in \boldsymbol{p}$ **s.t.** Prop[p].act = psk-$*$ **do**
    - **req** id $\in$ Node[Prop[p].c_{psk}].has_psk $\vee \nexists$(id, $*$) $\in$ mem
  - **if** $c \neq c'$ **then**
    - Attach detached root c' as successor of id's node
- // Check if any input proposal indicates to remove the invoker
- **if** $\exists p \in \boldsymbol{p}$: Prop[p].act = rem-id **then**
  - Ptr[id] $\leftarrow \perp$
- **else**
  - **assert** id $\in$ Node[c].mem // id can be challenged
  - Ptr[id] $\leftarrow c$; HasKey[id] $\leftarrow$ true
  - Node[c].has_psk +$\leftarrow$ id // Add id to the list of members that derived the PSK
- *cons-invariant $\wedge$ *auth-invariant
- **return** *output-proc(c)

Input (Join, vec_w)
- Send id and all inputs to the adv. and receive related values including $(ack, \text{epoch}, c_{psk}')$.
- **req** ack
- $(c, c_{psk}) \leftarrow$ Wel[vec_w]
- // If vec_w is injected, then assign a commit node to it.
- **if** $c = \perp$ **then**
  - Receive c' from the adv.
  - **if** Node[c'] $\neq \perp$ **then**
    - $c \leftarrow c'$; $c_{psk} \leftarrow c_{psk}'$
  - **else** // Create a detached root
    - rootCtr++; $c \leftarrow \text{root}_{\text{rootCtr}}$; $c_{psk} \leftarrow c_{psk}'$
    - Node[c] $\leftarrow$ new node with orig = orig', mem = mem',
    - stat = adv, epoch = epoch, psk $\leftarrow$ adv,
  - Wel[vec_w] $\leftarrow (c, c_{psk})$
- // Every psk proposals correctly refers to a valid c_{psk}
- **for** $p \in$ Node[c].pro **s.t.** Prop[p].act = psk-$*$ **do**
  - **req** id $\in$ Node[Prop[p].c_{psk}].has_psk
- Ptr[id] $\leftarrow c$; HasKey[id] $\leftarrow$ true
- Node[c].has_psk +$\leftarrow$ id // Add id to the list of members that derived the PSK
- **assert** $g \neq \perp$
- **assert** groupInfo[g] $\in \{\perp, c\}$
- groupInfo[g] $\leftarrow c$
- // Check that joining id does not violate authenticity and HG-consistency
- **assert** id $\in$ Node[c].mem$\wedge$*cons-invariant$\wedge$*auth-invariant
- **return** *output-join(c)

Fig. 5: Part 2 of our simplified $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, using the color coding from Table 2. The orange code (●) is included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, but not in $\mathcal{F}_{\text{ECGKA}}$.

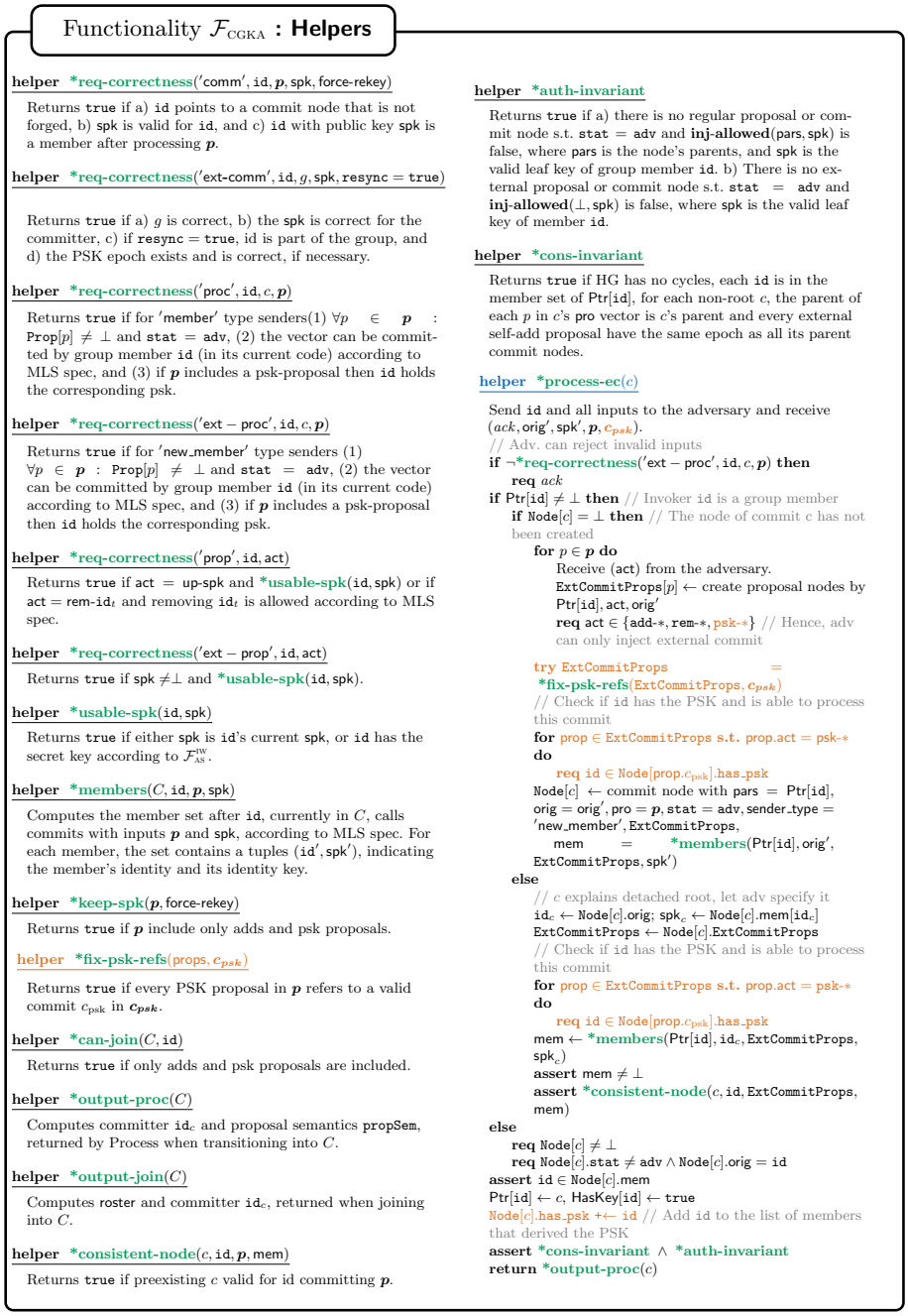


Fig. 6: The simplified helper functions and invariant predicates for the $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ functionalities using the color coding from Table 2. The codes highlighted with ● color are only included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

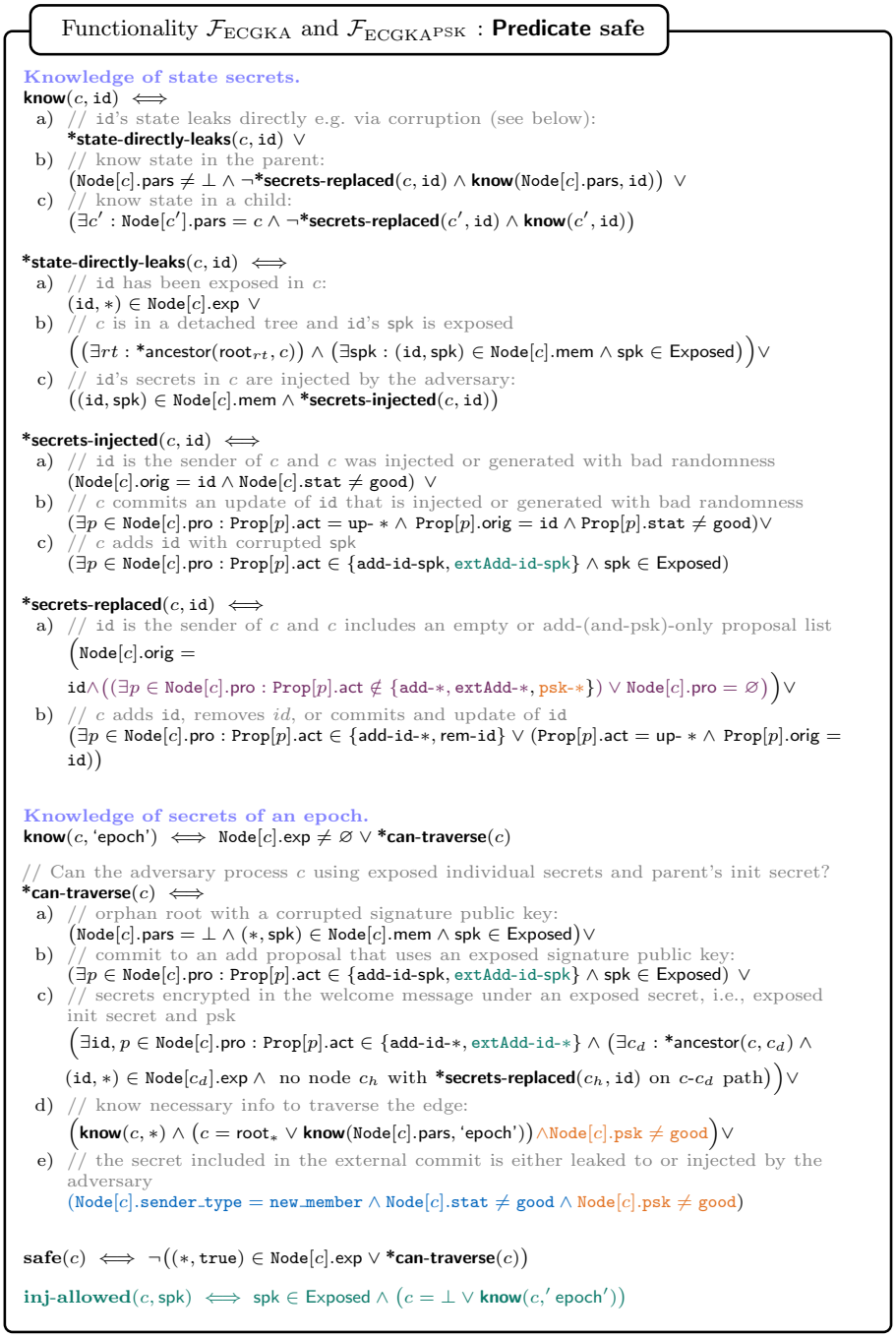


Fig. 7: The safe predicate for the $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ functionalities using the color coding from Table 2. The codes highlighted with ● color are only included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$. We define *ancestor$(t, c)$, for a root or commit node t and a commit code c, to be `true` if and only if t is an ancestor of c in the tree.

of welcome messages $\texttt{vec\_w}$, and each groupInfo g, $\mathcal{F}_{\text{ECGKA}}$ respectively stores a pointer $\mathsf{Ptr}[\texttt{id}]$, $\texttt{Wel}[\texttt{vec\_w}]$, $\texttt{GroupInfo}[g]$, to its corresponding commit node in the history graph, initialized with $\perp$. Moreover, for each external self-add proposal p, $\mathcal{F}_{\text{ECGKA}}$ also uses a helper node $\texttt{ExtProps}[p]$ to store the epoch that p will apply to. Furthermore, $\mathcal{F}_{\text{ECGKA}}$ uses $\text{RndCor}[\texttt{id}]$ to record whether a party $\texttt{id}$'s randomness source is indeed random (i.e., $\texttt{good}$) or chosen by the adversary (i.e., $\texttt{bad}$), and $\mathsf{HasKey}[\texttt{id}]$ to specify whether the current application secret of $\texttt{id}$ can be challenged (i.e., $\texttt{true}$) or not (i.e., $\texttt{false}$).

Below, we briefly explain the external self-add proposal and external commit related interfaces of $\mathcal{F}_{\text{ECGKA}}$. We detail the remaining interfaces in Appendix D.

Interface $\texttt{Create}$*:* This interface models group creation with an invoker that has a valid spk. The initial root of the history graph is initialized with the only member $\texttt{id}_{\mathsf{creator}}$. The pointer $\mathsf{Ptr}[\texttt{id}_{\mathsf{creator}}]$ is set to the initial root and $\texttt{id}_{\mathsf{creator}}$'s current application key is set to a challengeable state.

Interface $\texttt{Propose}$ *(External):* This interface models the new external self-add proposal operation with input epoch that specifies the epoch that this proposal is intended for. Since the proposal originates externally, it is only signed by $\texttt{id}$, who is not yet a member, has no internal group secret, and does not specify the commit node of the history graph p to which it pertains. Different from a regular add, for non-existing $\texttt{Prop}[p]$, $\mathcal{F}_{\text{ECGKA}}$ sets the list of parent nodes, $\texttt{Prop}[p].\mathsf{pars}$, to all existing commit nodes at the same epoch. Recall that an external self-add proposal can be applied to all commit nodes at the epoch. Moreover, $\mathcal{F}_{\text{ECGKA}}$ stores the input epoch in the helper node $\texttt{ExtProps}[p]$. All parent commit nodes of $\texttt{Prop}[p]$ must have the same epoch as the one included in the query input. If the randomness in this query is chosen by the adversary, then $\mathcal{F}_{\text{ECGKA}}$ exposes secret of the invoker $\texttt{id}$'s public key. This captures the fact that the ECDSA signature underlying MLS is vulnerable against randomness leakage attacks [5].

Interface $\texttt{ExtCommit}$*:* This models the external commit operation, which has two use cases: (i) an existing group member ($\mathsf{Ptr}[\texttt{id}] \neq \perp$) wants to resync, i.e., leave and rejoin the group; or (ii) a party $\texttt{id}$ that is not a group member ($\mathsf{Ptr}[\texttt{id}] = \perp$) wants to externally add itself. $\mathcal{F}_{\text{ECGKA}}$, upon receiving variables including a new groupInfo g', a self-add proposal p_{exi}[8], and a self-remove proposal p_{rem}, checks if this query is honestly generated or injected. If injected, it creates a detached root. Within this interface, $\mathcal{F}_{\text{ECGKA}}$ does not create independent proposal nodes for every included proposal. Instead, it creates sub-nodes for the self-add and self-remove proposals. If the node of the external commit c has not been created, $\mathcal{F}_{\text{ECGKA}}$ creates it, which wraps all above sub-nodes. Otherwise, $\mathcal{F}_{\text{ECGKA}}$ checks whether the existing node $\texttt{Node}[c]$ is consistent with this external commit and the included proposals. Note that the external commit only adds the invoker $\texttt{id}$ to the group. This query will not yield any welcome message.

Interface $\texttt{Process}$ *(regular commit):* This interface processes a regular commit c and a vector of proposals $\boldsymbol{p}$. After checking that the commit and proposals

[8] *exi* is named after the "**Ex**ternal**I**nit" operation in [12, Section 12.1.6], which "is used by new members that want to join a group by using an external commit".

are correct, or acknowledged by the adversary, $\mathcal{F}_{\text{ECGKA}}$ creates proposal nodes for all injected proposals. If c injected by the adversary, $\mathcal{F}_{\text{ECGKA}}$ creates the node of c, attaches the commit node c to them. If any of the input proposals indicate removing the invoker `id` from the group, then the pointer is set to $\perp$. Otherwise, $\mathcal{F}_{\text{ECGKA}}$ verifies that `id` appears in the member list of the node of the c, updates `id`'s pointer to c, and marks `id`'s key as challengeable. Finally, returns the information about the processed commit and its included proposals.

Interface `Process` *(external commit):* This interface processes an external commit c. If the invoker `id` is a group member, $\mathsf{Ptr}[\texttt{id}] \neq \perp$, and the node of commit c has not been created, then this commit must be injected by the adversary. Similarly, $\mathcal{F}_{\text{ECGKA}}$ creates the "sub-node" of the proposals and checks whether their actions are valid, i.e., only add and remove proposals are allowed and the added or removed party must be the origin of the commit. Then, $\mathcal{F}_{\text{ECGKA}}$ creates the corresponding commit node. If the node of commit c has already been created, it checks whether with a valid member list and whether the existing commit node is consistent with c and its included proposals $\boldsymbol{p}$. If $\mathsf{Ptr}[\texttt{id}] = \perp$, then the invoker `id` must be outside the group. Note that a non-member party can only process external commits that were created by itself. $\mathcal{F}_{\text{ECGKA}}$ simply checks if the node of commit c has been honestly generated by the invoker `id`, i.e., not injected by the adversary. Finally, $\mathcal{F}_{\text{ECGKA}}$ marks `id`'s key as challengeable.

We compare $\mathcal{F}_{\text{CGKA}}$ [4] and our $\mathcal{F}_{\text{ECGKA}}$ in Appendix I.

5 RFC 9420 and ETK: Extended Operations TreeKEM

In this section, we introduce our ETK protocol, which is a simplified version of MLS RFC 9420 [12], as an instance of the ECGKA protocol. Our ETK protocol extends ITK [4], which simplifies MLS Draft 12 [13], with the novel external proposal and commit operations as well as some selected updates from Draft 10 to RFC 9420 that have not been formally studied in the literature to the best of our knowledge. In Section 5.1, we present the differences of our ETK with ITK [4]. In Section 5.2, we prove that our ETK protocol securely realizes $\mathcal{F}_{\text{ECGKA}}$. Due to the page limit, we provide the complete details of our ETK protocol in Appendix E, and summarize our simplifications in Appendix C.

5.1 The ETK Protocol

Similar to [4], we abstract the associated authentication and key services of our ETK as functionalities $\mathcal{F}_{\mathcal{AS}}$ and $\mathcal{F}_{\mathcal{KS}}$, which we depict in Appendix E of our full version [29]. Compared with ITK [4] that abstracts Draft 12 [13], our ETK includes the following updates.

Updated GroupInfo. While ITK outputs `groupInfo` only included in the welcome messages for parties that have already been invited via regular add proposal, our ETK outputs `groupInfo` whenever regular or external commits are generated, in order to allow non-member party to join the group by itself. This

modification has been included in MLS since Draft 10 [11]. Moreover, while ITK includes both confTransHash and interimTransHash in the `groupInfo`, our ETK only includes confTransHash and excludes interimTransHash, since confTransHash itself is sufficient for the verification on the history of group evolution. This modification has also been included in MLS since Draft 10 [11].

Leaf Nodes and Key Packages. While in ITK the leaf nodes and key packages had the same data structure, in our ETK they have different data structures. More concretely, leaf nodes are now a data structure that can be either contained within a tree, or within a key package. In our ETK, we use a variable `leafNode_source` within leaf node to denote where it is included: key package, commit, or update. This modification has been included in MLS since Draft 13 [14]. Moreover, while ITK encrypts the group secrets in the welcome messages with the public key of the leaf node, our ETK includes a new "init key" in the key packages that is different from the key in the leaf node and uses this "init key" to encrypt the group secrets in the welcome messages. This modification has been included in MLS since Draft 13 [14]. Finally, while ITK refers to every leaf node via an allotLeaf function inputting the hash of their owner's key package, our ETK refers to leaf nodes simply via their leaf index in the tree. Consequently, our allotLeaf function now takes no reference anymore, but assigns the leftmost leaf, and returns the index of this leaf node. This modification is in MLS since Draft 15 [15].

Updated Parent Hash Computation. In ITK the parent hash of every node is computed by hashing its parent node and the resolution of its sibling node, as did MLS versions before Draft 13. Our ETK instead follows RFC 9420, and computes the parent hash of a node by hashing its public encryption key, its parent's parent hash, and its *original sibling tree hash*, which is a modified tree hash of its sibling that excludes all unmerged leaves in the sibling's sub-tree.

Updated Welcome Messages. While in ITK the `groupInfo` was unencrypted in welcome messages, our ETK encrypts it with the key enclosed in the encrypted group secrets. This modification has been included in MLS since MLS Draft 9 [10]. Moreover, while ITK outputs every commit along with a single welcome messages for all added parties, our ETK allows the committer to freely decide how many welcome messages to construct, indicated by `wel_type`, and outputs a vector `vec_w` that includes one or more welcome messages, each of which adds either all, single, or batch parties. This is included in MLS since Draft 17 [16].

New Operation: External Self-Add Proposal. Since Draft 15 [15], MLS includes the external self-add proposal, which allows a party to propose to add itself to a group. The proposer `id` must not be in the group. To generate the external self-add proposal, `id` first generates a key package that wraps two data structures: a "leaf node" and an "init key", followed by signing the key package using identity ssk. Next, `id` generates the proposal message that includes the intended groupId, the intended epoch, proposal type "`new_member_proposal`", and the signed key package, followed by signing them together. In the end, the signed proposal message is output.

New Operation: External Commit / Resync. Since Draft 10 [11], MLS includes the external commit operation, which serves two purposes. First, it enables an external user to join an "open" group asynchronously without asking any group member to add them. In this case, we have `resync` = `false`. Second, an existing group member can use an external commit to replace (resync) its prior representation in the group with a new one. In this case, we have `resync` = `true`.

In both cases, the committer `id` first parses the latest group information and a signature from the input groupInfo object g. The latest group information includes (1) the identifier of the group groupId, (2) the current epoch number epoch, (3) the hash of (the public part) of the binary ratchet tree τ, (4) the transcript hash of confTransHash, (5) a labeled left-balanced binary ratchet tree τ, (6) a confirmation tag `confTag`, (7) the groupInfo g's sender index `senderIdx`, and (8) an HPKE public key external-pub, the secret key of which is shared by all existing group members. The committer `id` verifies whether the signature can be verified using the sender `senderIdx`'s public key spk encoded in the tree τ. Additionally, `id` checks whether the public ratchet tree is valid. Then, the committer `id` derives the initial secret initSecret for the key schedule using the HPKE export method with external-pub and initializes a new group state for the next epoch. The committer `id` generates an `extInit` proposal that is tailored for external commit to add itself. If `resync` = `true`, then the committer additionally generates a self-removal proposal. Afterwards, similar to the regular commit, the committer `id` adds itself to the ratchet tree at the leftmost available leaf, optionally removes itself from the current group, rekeys path, signs the commit content C using its signature key γ.ssk, computes new transcript hash, and runs key schedule. The key schedule derives a confirmation key `confKey` and a joiner secret `joinerSec`. In the end, the committer `id` generates a confirmation tag `confTag` for the new group state, an external commit message ec, and an associated external groupInfo g. A mapping associating the new group state with each pending commit issued by `id` is added to the mapping γ.pendCom. The external commit message ec and the new groupInfo g are returned.

5.2 Security Results for ETK and $\mathcal{F}_{\text{ECGKA}}$

Theorem 1. *Assume that* PKE *is* IND-CCA *secure and that* Sig *is* SUF-CMA *secure. The* ETK *protocol securely realizes* $(\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}, \mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}, \mathcal{F}_{\text{ECGKA}})$ *in the* $(\mathcal{F}_{\mathcal{AS}}, \mathcal{F}_{\mathcal{KS}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where* $\mathcal{F}_{\text{ECGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Figure 7 and calls to* HKDF.Expand, HKDF.Extract, *and* MAC *functions are replaced by calls to the global random oracle* $\mathcal{G}_{\text{RO}}$.

We provide a proof sketch in Appendix F and our full proof in Appendix J.

External operations and PCS An adversary who compromises the state of a group member can still rejoin the group and eavesdrop group messages after healing, contrary to what is implied by previous results. If there are no external operations (which were not considered by previous works), after a group member's compromise, the committer's key package can be regenerated by `updatePath`.

That is, exposed member keys' healing is captured via ***secrets-replaced**$(c, \texttt{id})$, as in case c) of ***can-traverse**(c) of Figure 7. This ensures new per-session state can be recovered by using good randomness, and hence our proof implies PCS when there are no external operations.

However, in reality, applications will have to implement resync (in order to reliably communicate, as shown for two-party messaging in [32]). When accounting for external operations, an adversary that has compromised a member and subsequently lost the current group state can still rejoin by leveraging external operations. Specifically, if an adversary compromises a member's full state, the adversary can always inject an external commit, i.e., effectively rejoining. By signing the commit content with its retained signature key, the adversary can rejoin the group impersonate the honest member. This violates the original definition of PCS [27] as well as the PCS notion implied by the security definition in [4], which is described as the ability to return to a fully secure state after a state compromise. Here, however, even though a member's state can be healed via `updatePath`, group communication remains exposed to the adversary. In our security definitions, this case is excluded by the additional case e) in ***can-traverse**(c) in Figure 7.

EUF- vs SUF-CMA Our Theorem 1 requires the signature scheme to be SUF-CMA-secure, notably to provide commit message integrity against malicious parties. In reality, MLS RFC 9420 allows for the use of ECDSA (for FIPS compliance), which only satisfies EUF-CMA: it is possible to mangle an ECDSA signature such that it still verifies as expected [41].

We give a concrete example for a malicious insider. Suppose the signature scheme is ECDSA, and that Alice sends a commit message c to group members Bob and Charlie. A malicious Bob can create a new commit message c' by mangling the signature within the commit message c without knowing Alice's secret signing key, computing a MAC if present, and send it to Charlie. Although Charlie can successfully process c' and agree on the same content included in the commit with Alice and Bob, their group states diverge, as the signature is included in the transcript hash and the key schedule. Consequently, neither the messages from Charlie nor from Alice can be read by the other.

While this possible behavior seems to run counter to the design of MLS, we currently do not know of any real-world attack based on this, since an adversary with such capabilities can cause groups to split in other ways, e.g., by dropping messages for some recipients. However this enables a subtle behavior in which participants process the exact same payloads, yet their views and epoch keys diverge. Notably, for non-SUF-CMA secure schemes, the so-called `AuthenticatedContent` objects in [12] can be mangled.

From a provable security perspective, this shows up as a failure of an assert in $\mathcal{F}_{\text{ECGKA}}$. However, this behavior should also be captured by $\mathcal{F}_{\text{CGKA}}$ in [4], which reduces to EUF-CMA. We confirmed with an author of [4] that the theorem statement is incorrect and should be changed to require SUF-CMA.

6 Stronger Security for Group Re-Synchronization via External Commit

In this section, we show that the resumption PSK mechanism of MLS RFC 9420 can be used to improve the security of group re-synchronization. We first explain the mechanism and the improved security guarantees. In Section 6.2, we introduce our corresponding stronger $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ functionality. In Section 6.3 we propose our $\mathsf{ETK}^{\text{PSK}}$ protocol, and prove in Section 6.4 that it realizes $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

6.1 Resyncs and Resumption Pre-Shared Keys

A member that cannot process group commits, e.g., because it lost part of its group state, has to re-synchronize with the group. This can be done through a regular remove-then-add P&C mechanism. Alternatively, it can be implemented by the member sending an external commit that removes their old instance while re-adding itself with a new leaf, simply referred to as resync. The resync is the simpler mechanism, but provides less security: as explained in Section 4.2, the adversary can learn the epoch secret by choosing the external committers' randomness. To learn the epoch secret of a regular commit, in addition to choosing the members' randomness, the adversary needs to obtain the group secret. Note that a malicious delivery service - cooperating with an adversary that compromised a members randomness - can manipulate message transmission such that this member falls behind others, and is required to resync its state with the group. Improving the security of this resync becomes a natural and significant question.

We show that the use of the optional *resumption pre-shared key (PSK)* in MLS improves the security of a resync. The mechanism uses three steps:

- PSK derivation and storage: As depicted in Figure 2, the key schedule phase derives a resumption PSK along with the other secrets from the epoch secret. The derived PSKs is stored in the local state by every group member for a certain period, determined by the application.
- PSK proposal: A party (insider or outsider) can generate a PSK proposal that specifies a past epoch, the resumption PSK of which will be injected to the next epoch's key schedule phase. The PSK proposals can be committed in both regular and external commits.
- PSK injection: When receiving a commit that includes PSK proposals, every group member retrieves the PSKs of all epochs specified in these proposals from their local state. They then derive the epoch secret by hashing the PSKs with the joiner secret. Note that only members that possess all PSKs are hence able to process such a commit.

When a member performs a resync (potentially after being maliciously forced to), this member - in possession of a past PSK of that group - can inject that PSK into the external commit to prove previous membership. An adversary controlling the member's randomness, but not knowing its state and hence PSK, cannot process this commit and is hence locked out. Therefore, the adoption of PSK can improve PCS.

6.2 The Stronger Functionality $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$

We propose our functionality $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ that extends our $\mathcal{F}_{\text{ECGKA}}$ with the PSK injection, highlighted with orange color (●) in Figures 4, 5 and 7. Below, we explain the additional execution and security guarantees captured by $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

Additional Executions. Our $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ includes additional executions in the `Propose`, `Commit`, `ExtCommit`, `Process`, and `Join` interfaces.

Interface `Propose`*:* This interface allows an additional `act` = `psk-epoch-use`, with `epoch` specifying which PSK to be injected, and `use` describing its intended usage. In our work, we only consider the usage "application", which can be used to improve the security of regular or external commits as described above. We leave the analysis of the other usages, "reinit" for group re-initialization and "branch" for group branches, for future work. The execution of the PSK proposal (`Propose`, `act`) for `act` = `psk-epoch-use` is identical to the one of other proposals.

Interface `Commit`*:* $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ additionally retrieves a vector $\boldsymbol{c_{psk}}$ that contains all past commits whose PSKs will be injected by this commit's `psk` proposals from the adversary. $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ ensures that every PSK proposal refers to a valid commit c_{psk} in $\boldsymbol{c_{psk}}$, and asserts if no suitable c_{psk} is found or the epoch does not match. $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ does not require re-key path for commits containing only `add` and `psk` proposals, and the committer's keying material is not forced to be replaced if so. For each `psk` in the proposals vector, the committer `id` must be in the list of group members who have derived the resumption PSK from the key schedule for the corresponding c_{psk}. Regardless of whether the commit node c or the detached root exists, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ checks if rekey path is necessary. If the commit c includes any add proposal, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ ensures that the welcome messages differs for each $(c, \boldsymbol{c_{psk}})$.

Interface `ExtCommit`*:* This interface additionally takes an input vector of epochs $\boldsymbol{e}_{\text{psk}}$. Upon forwarding all inputs to the adversary and receives two additional vectors $\boldsymbol{p_{psk}}$ and $\boldsymbol{c_{psk}}$. Similar to the `extAdd` and `rem` proposals, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ does not create independent proposal nodes for each included `psk` proposal. Instead, it zips the `psk` proposals with the corresponding epoch vector $(\boldsymbol{e}_{\text{psk}}, \boldsymbol{p_{psk}})$ and creates sub-nodes for every self-add, self-remove and PSK proposals. Later, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ ensures that every `psk` proposal refers to a valid c_{psk}. Similarly, for each `psk` in `ExtCommitProps`, the committer `id` must be in the list of group members who have derived the resumption PSK from the key schedule.

Interface `Process`*:* In case of a regular commit ($\boldsymbol{p} \neq \perp$), $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ additionally receives $\boldsymbol{c_{psk}}$ that specifies the commit messages that the included `psk` proposals refer to, from the adversary. $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ ensures that every PSK proposal refers to a valid c_{psk} in $\boldsymbol{c_{psk}}$. No matter whether the input commit message c is honestly generated or injected by adversary, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ additionally verifies that at least one PSK for each `psk` is in `id`'s possession. `id` is added to the list of group members who have derived the resumption `psk` of the new epoch.

In case of an external commit ($\boldsymbol{p} = \perp$), similarly $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ receives an additional $\boldsymbol{c_{psk}}$. After creating the sub-nodes of the proposals, the functionality

$\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ verifies whether their actions are valid, i.e., either `add`, `rem`, or `psk`. The functionality $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ verifies that each `psk` in `ExtCommitProps` refers to a valid c_{psk}. `id` is added to the list of group members who have derived the resumption PSK of the new epoch.

Interface `Join`*:* The interface receives an additional $\boldsymbol{c_{psk}}'$ from the adversary and looks for a commit c and its $\boldsymbol{c_{psk}}$ that `vec_w` maps to. If no such commit exists, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ utilizes the adversary-provided $\boldsymbol{c_{psk}}'$ to create a node and maps `vec_w` to the newly generated pair $(c, \boldsymbol{c_{psk}})$. In all cases, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ verifies that every `psk` proposal in $\texttt{Node}[c].\texttt{pro}$ refers to a valid c_{psk}. `id` is added to the list of group members who have derived the resumption PSK of the new epoch.

Additional Security Guarantees. $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ captures the stronger confidentiality provided by the PSK mechanism in the **safe** predicate in Figure 7 for both regular and external commits.

Regular Commit: Compared to $\mathcal{F}_{\text{ECGKA}}$, there are two differences. First, in the predicate ***secrets-replaced**, in addition to the regular commit with an add-only proposal list, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ captures the rekey path except for a regular commit that is empty or includes only (regular) add, external self-add, or PSK proposals. This reflects the suggestion in MLS RFC 9420 [12, Section 12.4] that the rekey path can be omitted for a regular commit with an add-or-PSK only proposal list.

Second, in case d) of the helper predicate ***can-traverse**, where $\mathcal{F}_{\text{ECGKA}}$ only takes the leakage of the commit secret (covered by $\textbf{know}(c, *)$) and the initSecret (indicated by $c = \text{root}_* \vee \textbf{know}(\texttt{Node}[c].\text{pars}, \text{'epoch'})$) into account, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ additionally considers leakage of the `psk-secret` (indicated by $\texttt{Node}[c].\text{psk} \neq \texttt{good}$). In this case, a regular commit c is secure if at least one of the commit-, the init-, and the optional psk-secret is not leaked.

External Commit: In case e) of the predicate ***can-traverse**, where $\mathcal{F}_{\text{ECGKA}}$ considers an external commit insecure if it involves bad randomness, $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ relaxes the requirement on randomness as long as the included psk-secret is not leaked. This provides the external commit stronger resilience against randomness leakage attacks by using PSKs from the local states.

6.3 The ETK^{PSK} Protocol

Our ETK^{PSK} protocol uses the resumption PSK mechanism from RFC 9420 as described in Section 6.1 to extend ETK in Section 5 with the following differences.

New State Variable. Member's group state include an additional `psk_store` that contains the resumption `psk`s and their associated groupId and epoch.

New Operation: PSK Proposal. To perform a PSK proposal with `usage` = "application", the proposer `id` must be in the group. To generate the `psk` proposal, `id` first generates the `pskId` by using groupId, the intended epoch and random nonce variables, then signs the proposal and outputs the proposal message p.

Extended Operation: Regular Commit. If the proposal list only includes add-or-PSK proposals, then no re-key path is executed. Once the commit is signed and

confTransHash is set, the committer retrieves all PSK values, identified by `pskIds` in the applied `psk` proposals, from its `psk_store`, and derives `psk-secret`, then hashes both `joinerSec` and `psk-secret` to derive the epoch secrets. If the commit adds new members, the vector of welcome messages additionally includes `pskIds` to let joiners know which resumption PSKs to use. Then, the `welcome_secret` is derived from both `joinerSec` and `psk-secret` to encrypt groupInfo. The commit message, the welcome vector, and the encrypted groupInfo object are output.

Extended Operation: External Commit. $\mathsf{ETK}^{\mathrm{PSK}}$ inputs an additional vector $\boldsymbol{e}_{\mathrm{psk}}$ that identifies the epochs, the PSKs of which are injected in this external commit. For each `epoch` in the vector $\boldsymbol{e}_{\mathrm{psk}}$, a `pskId` with `usage` "application" and a `psk` proposal are generated. As in a regular commit execution, the external committer derives a `psk-secret` and injects it into the key schedule phase to derive epoch secrets.

Extended Operation: Process. The extended execution is the same for processing both regular and external commits. As above, if `id` is not removed by the commit, `id` derives the `psk-secret` from all included PSK proposals to inject it into the key schedule phase. In the end, `id` outputs PSK proposals along with the id of the committer and all other proposals.

Extended Operation: Join. The joiner additionally extracts a vector `pskIds` from the decryption of `encGroupSec` from the vector of welcome messages. As described above, the joiner derives `psk-secret` from all local PSK values, which is then used to compute the welcome secret to recover the encrypted groupInfo and to inject it into the key schedule.

6.4 Security Results for $\mathsf{ETK}^{\mathrm{PSK}}$ and $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$

Theorem 2. *Assume that* PKE *is* IND-CCA *secure and that* Sig *is* SUF-CMA *secure. The* $\mathsf{ETK}^{\mathrm{PSK}}$ *protocol securely realizes* $(\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}, \mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}, \mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}})$ *in the* $(\mathcal{F}_{\mathcal{AS}}, \mathcal{F}_{\mathcal{KS}}, \mathcal{G}_{\mathrm{RO}})$*-hybrid model, where* $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Figure 7 and calls to* HKDF.Expand, HKDF.Extract, *and* MAC *functions are replaced by calls to the global random oracle* $\mathcal{G}_{\mathrm{RO}}$.

We provide a proof sketch in Appendix G and our full proof in Appendix K.

Acknowledgements. The corresponding author's work was supported by National Natural Science Foundation of China under Grant U25A20435.

References

[1] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Modular Design of Secure Group Messaging Protocols and the Security of MLS". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Nov. 2021.

[2] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *Advances in Cryptology – CRYPTO 2020*. Cham: Springer International Publishing, 2020.

[3] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. "Continuous Group Key Agreement with Active Security". In: *Theory of Cryptography*. Vol. 12551. Cham: Springer International Publishing, 2020.

[4] Joël Alwen, Daniel Jost, and Marta Mularczyk. "On the Insider Security of MLS". In: *Advances in Cryptology - CRYPTO 2022*. Cham: Springer Nature Switzerland, 2022.

[5] Diego F Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. "LadderLeak: Breaking ECDSA with less than one bit of nonce leakage". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020.

[6] Benedikt Auerbach, Miguel Cueto Noval, Boran Erol, and Krzysztof Pietrzak. "Continuous Group-Key Agreement: Concurrent Updates Without Pruning". In: *Advances in Cryptology - CRYPTO 2025*. Springer-Verlag, 2025.

[7] Richard Barnes. *How Messaging Layer Security Enables Scalable End-to-End Security in Webex*. https://blog.webex.com/hybrid-work/scalable-end-to-end-security-in-webex/ (Accessed October 1, 2025). WebEx Blog.

[8] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 07)*. Internet-Draft. Internet Engineering Task Force.

[9] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 08)*. Internet-Draft. Internet Engineering Task Force, Nov. 2019.

[10] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 09)*. Internet-Draft. Internet Engineering Task Force, Mar. 2020.

[11] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 10)*. Internet-Draft. Internet Engineering Task Force, Oct. 2020.

[12] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol*. https://www.rfc-editor.org/info/rfc9420 (RFC 9420). July 2023.

[13] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol (Draft 12)*. Internet Engineering Task Force, Oct. 11, 2021.

[14] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol (Draft 13)*. Internet-Draft. Internet Engineering Task Force, Mar. 2022.

[15] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol (Draft 15)*. Internet-Draft. Internet Engineering Task Force, June 2022.

[16] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. *The Messaging Layer Security (MLS) Protocol (Draft 17)*. Internet-Draft. Internet Engineering Task Force, Dec. 2022.

[17] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 00)*. Tech. rep. Internet Engineering Task Force, Aug. 2018.

[18] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. *The Messaging Layer Security (MLS) Protocol (Draft 02)*. Tech. rep. Internet Engineering Task Force, Oct. 2018.

[19] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: Association for Computing Machinery, 1993.

[20] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. "Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS". In: *Doctoral dissertation, Inria Paris* (2019).

[21] 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, ffhal-02425229f https://inria.hal.science/hal-02425229v1/file/mls-treekem.pdf. 2019.

[22] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. *The Provable Security of Ed25519: Theory and Practice*. Cryptology ePrint Archive, Paper 2020/823. 2020.

[23] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. "Security Analysis of the MLS Key Derivation". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022 IEEE Symposium on Security and Privacy (SP). May 2022.

[24] R. Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 2001.

[25] Céline Chevalier, Guirec Lebrun, Ange Martinelli, and Abdul Rahman Taleb. "Quarantined-TreeKEM: A Continuous Group Key Agreement for MLS, Secure in Presence of Inactive Users". In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. 2024.

[26] Cisco. *Cisco/MLSPP*. https://github.com/cisco/mlspp (Accessed Octorber 1, 2025). Cisco Systems.

[27] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. "On Post-compromise Security". In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*.

[28] 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*. Oct. 15, 2018.

[29] Cas Cremers, Esra Günsay, Vera Wesselkamp, and Mang Zhao. *ETK: External-Operations TreeKEM and the Security of MLS in RFC 9420*. Cryptology ePrint Archive, Paper 2025/229. 2025.

[30] Cas Cremers, Britta Hale, and Konrad Kohbrok. "The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter". In: *Proceedings of the 30th USENIX Security Symposium*. 2021.

[31] 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)*. 2023.

[32] Cas Cremers, Niklas Medinger, and Aurora Naska. "Impossibility Results for Post-Compromise Security in Real-World Communication Systems". In: *46th IEEE Symposium on Security and Privacy* (2025).

[33] Stephen Birarda Discord. *Meet DAVE: Discord's New End-to-End Encryption for Audio & Video*. https://discord.com/blog/meet-dave-e2ee-for-audio-video (Accessed October 1, 2025). 2025.

[34] Internet Engineering Task Force. *MLS RFC 9420 Datatracker*. https://datatracker.ietf.org/doc/rfc9420/. 2025.

[35] Tom Van Pelt (GSMA). *RCS Encryption: A Leap Towards Secure and Interoperable Messaging*. https://www.gsma.com/newsroom/article/rcs-encryption-a-leap-towards-secure-and-interoperable-messaging/ (Accessed October 1, 2025). Mar. 2025.

[36] Matthew Hodgson and Chathi. *A Giant Leap Forwards for Encryption with MLS*. https://matrix.org/blog/2023/07/a-giant-leap-with-mls/ (Accessed October 1, 2025). [matrix]. July 18, 2023.

[37] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. "Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021.

[38] AWS Labs. *Awslabs/MLS-RS: An Implementation of Messaging Layer Security (RFC)*. https://github.com/awslabs/mls-rs/tree/main (Accessed October 1, 2025). GitHub. 2023.

[39] Rohan Mahy. *More Instant Messaging Interoperability (MIMI) Message Content (Draft 06)*. Internet Draft. https://datatracker.ietf.org/doc/draft-ietf-mimi-content. Internet Engineering Task Force, 2024.

[40] RingCentral. *Security Features*. https://www.ringcentral.com/trust-center/secure-features.html (Accessed October 1, 2025). 2025.

[41] Jacques Stern, David Pointcheval, John Malone-Lee, and Nigel P. Smart. "Flaws in Applying Proof Methodologies to Signature Schemes". In: *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference*. Vol. 2442. Lecture Notes in Computer Science. Springer, 2002.

[42] Théophile Wallez, Jonathan, Benjamin Beurdouche, and Karthikeyan Bhargavan. "TreeSync: Authenticated Group Management for Messaging Layer Security". In: *32nd USENIX Security Symposium*. (USENIX Security 23). 2023.

[43] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. *TreeKEM: A Modular Machine-Checked Symbolic Security Analysis of Group Key Agreement in Messaging Layer Security*. Cryptology ePrint Archive, Paper 2025/410. 2025.

[44] Wire. *Messaging Layer Security (MLS)*. https://support.wire.com/hc/en-us/articles/12434725011485-Messaging-Layer-Security-MLS (Accessed October 1, 2025). 2025.

[45] XMTP. *Messaging security properties with XMTP*. https://docs.xmtp.org/protocol/security (Accessed October 1, 2025). 2025.

Appendix: Supplementary Material

A Overview of Supplementary Material

Our supplementary material is grouped into several sections.

We first tackle general background-related topics: In Appendix B, we give the cryptographic preliminaries, and in Appendix C, we describe our simplifications compared to the MLS RFC.

We then introduce the detailed versions of the functionalities in Appendix D and provide the protocol descriptions together with the helper functions in Appendix E. These are the technical details of our protocol definitions and security notions that did we abstracted away in the main parts of the paper.

We then give two very high-level proof sketches to describe our general proof approach in Appendices F and G, where we give proof sketches for Theorems 1 and 2, respectively.

We then compare our notions to [4] in more detail. We first compare our history graph with [4] in Appendix H and then our functionality $\mathcal{F}_{\text{ECGKA}}$ with $\mathcal{F}_{\text{CGKA}}$ from [4] in Appendix I.

Finally, we conclude with the full proof details: We provide the full proof of Theorem 1 in Appendix J and that of Theorem 2 in Appendix K.

B Preliminaries

B.1 Public Key Encryption

Definition 2 (Public Key Encryption). *A* public key encryption *scheme over message space $\mathcal{M}$ is a triple of algorithms, where* $\mathsf{PKE} = \{\mathsf{PKE.Kg}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec}\}$ *as defined below.*

- ***Key Generation*** $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{PKE.Kg}()$ *inputs the public parameters* pp *and outputs a public encryption key and a private decryption key pair* $(\mathsf{pk}, \mathsf{sk})$.
- ***Encryption*** $c \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk}, m)$ *inputs an encryption key* pk *and a message* $m \in \mathcal{M}$ *and outputs a ciphertext* c.
- ***Decryption*** $m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c)$ *inputs a private decryption key* sk *and a ciphertext* c. *It deterministically outputs a message* m.

We call a public key encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Kg}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *correct* if for all $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{PKE.Kg}()$ and all messages $m \in \mathcal{M}$ it holds that

$$m = \mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{PKE.Enc}(\mathsf{pk}, m)).$$

Definition 3. *We say a public key encryption scheme* $\mathsf{PKE} = (\mathsf{PKE.Kg}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *is* ϵ-$\mathsf{IND\text{-}CCA}$ *secure, if the advantage of all adversaries* $\mathcal{A}$ *that win* $\mathsf{Expt}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}$ *experiment in Figure 8 is bounded by*

$$|\Pr[\mathsf{Expt}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}(\mathcal{A}) = 1] - \frac{1}{2}| \leq \epsilon.$$

The Generalized Selective Decryption *(GSD) Security.* The GSD game constructs a hypergraph of vertices, where, in the context of ETK, each vertex is referred to as a node u, each storing a seed s_u. For each node, a public-private key pair can be derived using the seed as randomness, or the seed can be used as a symmetric key. Edges connect the nodes to represent dependencies between seeds. Briefly, the GSD game includes the following oracles and functions:

- **Enc**(u, v): An *encryption edge* is created upon calling the **Enc**(u, v) oracle, encrypting the seed of v under the public key of u.
- **Hash**(u, v, lbl): A *key derivation edge* is created when calling **Hash**(u, v, lbl), hashing the seed s_u with the label lbl to derive the seed s_v.
- **Join-Hash**(u, u', v, lbl): Calling **Join-Hash**(u, u', v, lbl) hashes both s_u and $s_{u'}$ together to derive the seed s_v.
- **Dec**: Analogous to the IND-CCA game, this oracle decrypts the ciphertext using the private key of u, provided that the seed of u is not valid or u is not a sink node.
- **Corr**: Outputs the seed of a node and records it in the Corr set.
- $*\mathsf{get\text{-}pk}(u)$: In the GSD experiment, the $*\mathsf{get\text{-}pk}(u)$ function allows $\mathcal{A}$ to obtain the corresponding public key of vertex u by calling the oracle **Enc**$(u, 0)$.

$\underline{\mathsf{Expt}^{\mathsf{IND\text{-}CCA}}_{\mathsf{PKE}}(\mathcal{A})}$:

1 $\mathsf{b} \xleftarrow{\$} \{0,1\}$
2 $(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{PKE.Kg}()$
3 $(m_0, m_1) \xleftarrow{\$} \mathcal{A}^{\mathbf{Dec}}(\mathsf{pk})$
4 $c \xleftarrow{\$} \mathsf{PKE.Enc}(\mathsf{pk}, m_{\mathsf{b}})$
5 $\mathsf{b}' \xleftarrow{\$} \mathcal{A}(c)$
6 **return** $[\![\mathsf{b} = \mathsf{b}']\!]$

$\underline{\mathbf{Dec}(c)}$:

7 $m \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, c)$
8 **return** m

Fig. 8: The IND-CCA security game for a PKE scheme.

Game $\mathrm{GSD}_{\mathsf{PKE},\mathcal{A}}$

The game is parameterized by the number of vertices N, the security parameter κ and a hash function Hash.

$(V, E) \leftarrow ([N], \varnothing)$ // GSD graph
$\mathsf{Corr}, \mathsf{Ctxt} \leftarrow \varnothing$ // corrupted vertices, ciphertexts
$s_u, \mathsf{pk}_u, \mathsf{sk}_u \leftarrow \bot$ for each $u \in [N]$ // keys for vertex u
$u \leftarrow \bot$ // challenge vertex
$b \leftarrow\$ \{0,1\}$
$s' \leftarrow\$ \{0,1\}^\kappa$
$b' \leftarrow \mathcal{A}^{\mathbf{Enc,Dec,Corr,Chal,Hash,Join\text{-}Hash}}_{\mathsf{PKE}}$
if (V, E) acyclic $\wedge$ u is a sink $\wedge$ $\neg$gsd-exp(u) **then**
  return $b = b'$
else return `false`

Oracle Chal(u)
  req $u = \bot$
  $u \leftarrow u$
  if $b = 0$ **then return** s_u
  else return s'

Oracle Hash(u, v, lbl)
  req $s_v = \bot \wedge (u, *, \mathsf{h\text{-}lbl}) \notin E$ // hash is deterministic
  gen-key-if-nec(u)
  $s_v \leftarrow \mathsf{Hash}(s_u, \mathsf{lbl})$
  gen-key-if-nec(v)
  E +← $(u, v, \mathsf{h\text{-}lbl})$
  return pk_u

Oracle Corr(u)
  req $s_u \neq \bot$
  Corr +← u
  return s_u

Oracle Join-Hash(u, u', v, lbl)
  req $s_v = \bot \wedge ((u, u', \mathsf{lbl}), *, \mathsf{h\text{-}lbl}) \notin E$
  gen-key-if-nec(u); gen-key-if-nec(u')
  $s_v \leftarrow \mathsf{Hash}(s_u, s_{u'}, \mathsf{lbl})$
  gen-key-if-nec(v)
  E +← $((u, u'), v, \mathsf{h\text{-}lbl})$
  return $(\mathsf{pk}_u, \mathsf{pk}_{u'})$

Oracle Enc(u, v)
  gen-key-if-nec(u); gen-key-if-nec(v)
  E +← (u, v, e)
  $c \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_u, s_v)$
  Ctxt +← (u, c)
  return (pk_u, c)

Oracle Dec(u, c)
  req $s_u \neq \bot \wedge u$ not a sink
  req $(u, c) \notin \mathsf{Ctxt}$
  return $\mathsf{PKE.Dec}(\mathsf{sk}_u, c)$

gen-key-if-nec(u)
  if $s_u = \bot$ **then** $s_u \leftarrow\$ \{0,1\}^\kappa$
  $(\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{PKE.Kg}(\mathsf{Hash}(s_u, \mathtt{node}))$
  // in ETK and $\mathsf{ETK}^{\mathrm{PSK}}$, the label "node" is used for key generation

gsd-exp(u)
  return $u \in \mathsf{Corr}$
    $\vee\ \exists (v, u, *) \in E$: gsd-exp(v)
    $\vee\ \exists ((v, v'), u, *) \in E$: gsd-exp$(v) \wedge$ gsd-exp(v')

Fig. 9: The GSD game, modified to explain ETK and $\mathsf{ETK}^{\mathrm{PSK}}$ executions. Taken from [4].

The gsd-exp(u) function indicates whether the seed s_u of the node u of commit c is exposed. The core component of the proof shows that gsd-exp(u) is analogous to predicate **safe**(c). For further details, we refer to [4]. The full GSD game is

provided in Figure 9. Figure 28 provides a visualization of the GSD graph for a regular commit, where Figure 30 visualizes the GSD graph of an external commit.

Definition 4. *We say a public key encryption scheme* $\mathsf{PKE} = (\mathsf{PKE.Kg}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *is* ϵ-GSD *secure, if the advantage of all adversaries* $\mathcal{A}$ *that win* $\mathrm{GSD}_{\mathsf{PKE},\mathcal{A}}$ *game in Figure 9 is bounded by*

$$|\Pr[\mathrm{GSD}_{\mathsf{PKE},\mathcal{A}} = 1] - \frac{1}{2}| \leq \epsilon.$$

Theorem 3 ([4]). *Let* PKE *denotes a public key encryption scheme. If* PKE *is* IND-CCA *secure, then* PKE *is* GSD *secure.*

B.2 Digital Signature

Definition 5 (Digital Signature). *A* digital signature scheme *over message space* $\mathcal{M}$ *is a triple of algorithms, where* $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Vrfy})$ *as defined below.*

- ***Key Generation*** $(pk, sk) \xleftarrow{\$} \mathsf{DS.KGen}()$ *outputs a public verification and private signing key pair* (pk, sk)*.*
- ***Signing*** $\sigma \xleftarrow{\$} \mathsf{DS.Sign}(sk, m)$ *inputs a secret key* sk *as well as the message* $m \in \mathcal{M}$ *to be signed and outputs a signature* σ*.*
- ***Verification*** $\texttt{true}/\texttt{false} \leftarrow \mathsf{DS.Vrfy}(pk, \sigma, m)$ *inputs a public verification key* pk*, a signature* σ *and message* $m \in \mathcal{M}$*. It deterministically outputs* `true` *if* σ *is a valid signature over* m *wrt.* pk*, else it outputs* `false`*.*

We call a signature scheme $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Vrfy})$ *correct* if for all $(pk, sk) \xleftarrow{\$} \mathsf{DS.KGen}()$ and all messages $m \in \mathcal{M}$, it always holds that

$$\texttt{true} = \mathsf{DS.Vrfy}(pk, \mathsf{DS.Sign}(sk, m), m).$$

Definition 6 (EUF-CMA and SUF-CMA security). *Let* $\mathsf{DS} = (\mathsf{DS.KGen}, \mathsf{DS.Sign}, \mathsf{DS.Vrfy})$ *be a digital signature scheme with message space* $\mathcal{M}$*. We say that* DS *is* ϵ-EUF-CMA*-secure if the adversary of all adversaries that win in the* $\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{DS}}$ *experiment is bounded by*

$$\Pr[\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{A}) = 1] \leq \epsilon.$$

Analogously, we say that DS *is* ϵ-SUF-CMA*-secure if the adversary of all adversaries that win in the* $\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}$ *experiment is bounded by*

$$\Pr[\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{A}) = 1] \leq \epsilon,$$

where the $\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{DS}}$ *and* $\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}$ *experiments are defined in Figure 10.*

$\underline{\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{A})}$:
1 $\mathcal{L}_{\mathsf{Sign}} \leftarrow \varnothing$
2 $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{DS.KGen}(\mathsf{pp})$
3 $(m*, \sigma*) \xleftarrow{\$} \mathcal{A}^{\mathbf{Sign}}(\mathsf{pk})$
4 **return** $[\![\mathsf{DS.Vrfy}(\mathsf{pk}, \sigma*, m*) \wedge m* \notin \mathcal{L}_{\mathsf{Sign}}]\!]$

$\underline{\mathbf{Sign}(m)}$:
5 $\sigma \xleftarrow{\$} \mathsf{DS.Sign}(\mathsf{sk}, m)$
6 $\mathcal{L}_{\mathsf{Sign}} \leftarrow \mathcal{L}_{\mathsf{Sign}} \cup \{m\}$
7 **return** σ

$\underline{\mathsf{Expt}^{\mathsf{SUF\text{-}CMA}}_{\mathsf{DS}}(\mathcal{A})}$:
1 $\mathcal{L}_{\mathsf{Sign}} \leftarrow \varnothing$
2 $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{DS.KGen}(\mathsf{pp})$
3 $(m*, \sigma*) \xleftarrow{\$} \mathcal{A}^{\mathbf{Sign}}(\mathsf{pk})$
4 **return** $[\![\mathsf{DS.Vrfy}(\mathsf{pk}, \sigma*, m*) \wedge (m*, \sigma*) \notin \mathcal{L}_{\mathsf{Sign}}]\!]$

$\underline{\mathbf{Sign}(m)}$:
5 $\sigma \xleftarrow{\$} \mathsf{DS.Sign}(\mathsf{sk}, m)$
6 $\mathcal{L}_{\mathsf{Sign}} \leftarrow \mathcal{L}_{\mathsf{Sign}} \cup \{(m, \sigma)\}$
7 **return** σ

Fig. 10: The EUF-CMA and SUF-CMA security games for a DS scheme.

$\underline{\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A})}$:
1 $\mathcal{L}_{\mathsf{MAC}} \leftarrow \varnothing$
2 $(k \xleftarrow{\$} \mathsf{KGen}(1^{\lambda})$
3 $(m*, t*) \xleftarrow{\$} \mathcal{A}^{\mathbf{MAC}}(.)$
4 **return** $[\![\mathsf{Vrfy}(k, m*, t*) \wedge m* \notin \mathcal{L}_{\mathsf{MAC}}]\!]$

$\underline{\mathbf{MAC}(m)}$:
5 $(m, t) \xleftarrow{\$} \mathsf{MAC}(k, m)$
6 $\mathcal{L}_{\mathsf{MAC}} \leftarrow \mathcal{L}_{\mathsf{MAC}} \cup \{m, t\}$
7 **return** t

Fig. 11: The EUF-CMA security game for a MAC scheme.

B.3 Message Authentication Code

Definition 7 (Message Authentication Code). *A* message authentication code scheme *over message space $\mathcal{M}$ and key space $\mathcal{K}$ is a tuple of algorithms, where* $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Vrfy})$ *as defined below.*

Signing $t \leftarrow \mathsf{MAC.Sign}(k, m)$ *inputs a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$ and outputs a tag t.*

Verification $\texttt{true}/\texttt{false} \leftarrow \mathsf{MAC.Vrfy}(k, m, t)$ *inputs a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$ and a tag t. It deterministically outputs* `true` *if t is a valid tag of m w.r.t the key k, else it outputs* `false`*.*

We say a $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Vrfy})$ is *correct* if for every key $k \xleftarrow{\$} \mathcal{K}$ and every $m \in \mathcal{M}$, it holds that

$$\texttt{true} = \mathsf{MAC.Vrfy}(k, m, \mathsf{MAC.Sign}(k, m)).$$

Definition 8 (EUF-CMA security of MAC). *We say a message authentication code* $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Vrfy})$ *is ϵ-EUF-CMA secure, if the advantage of all adversaries that break* $\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}$ *experiment in Figure 11 is bounded by*

$$\Pr[\mathsf{Expt}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A}) = 1] \leq \epsilon.$$

B.4 Random Oracle

The Random Oracle Model (ROM), introduced by Bellare and Rogaway [19], conceptualizes an idealized random function. Our work models hash function as

an idealized random function, i.e., the random oracle. More concretely, for an adversary aiming to compute the hash function H on input x, it must query the random oracle with x. This process shapes adversarial behavior by granting the ability to choose adversarial inputs to the hash function, which is referred to as extractability. For each query, the random oracle returns uniformly random answers drawn from the predefined range of the hash function. Regardless of how many times the random oracle is queried, the function H always produces the same output.

C Simplifications

Although our ETK and $\mathsf{ETK}^{\mathrm{PSK}}$ protocols are by far, to the best of our knowledge, the closest to the MLS RFC 9420 standard, our study still has the following simplifications:

- **External Proposals:** Our ETK includes the external self-add proposal `extAdd`, while omitting other external proposals, e.g., external removal and external group_context_extension.
- **PSK Usages:** Our $\mathsf{ETK}^{\mathrm{PSK}}$ reuses the resumption PSK mechanisms underlying MLS RFC 9420 standard to improve the security of regular and external commits. However, our $\mathsf{ETK}^{\mathrm{PSK}}$ omits the original applications of the resumption PSK mechanism, e.g., group re-initialization and branch operations.
- **Exporter Secrets:** In the key schedule of MLS RFC 9420, an exporter secret, which can be used by an application to derive new secrets for use outside of MLS, is derived together with other secrets of the epoch. Since our work focuses only on the evolution of the continuous group key agreement, rather than its external application outside MLS, we omit the exporter secret and leave it as future work.
- **External PSK:** MLS RFC 9420 employs two flavors of pre-shared keys: The resumption PSK and external PSK. The resumption key is derived together with secrets of the epoch in the key schedule and covered by our $\mathsf{ETK}^{\mathrm{PSK}}$. The external PSK is distributed among group members over some out-of-band channels. For simplicity, our work omits the external PSK mechanism.
- **Private Handshake Messages:** In MLS RFC 9420, the handshake messages, i.e., a message carrying an MLS Proposal or Commit object, can be either public or private. In our work, we consider public handshake message, which means that the handshake messages are signed by its sender and authenticated as coming from a member of the group in a particular epoch, but not encrypted. Note that the private handshake messages are signed by its sender, authenticated as coming from a member of the group in a particular epoch, and encrypted so that it is confidential to the members of the group in that epoch. Intuitively, the security achieved by using public handshake messages can also be achieved by using private handshake messages. For simplicity, we omit the private handshake messages and leave its detailed security analysis as future work.
- **Ciphersuites and Protocol Version:** MLS RFC 9420 introduces several available ciphersuits and their extensions in [12, Section 5.1, 13.1, and 17.1]. Moreover, every group member should specify the MLS protocol version and ciphersuit this member supports in its key packages. For simplicity, we omit this in the key package and simply assume that the MLS protocol version and ciphersuit every party supports are publicly known.
- **Cryptographic Primitive Contexts:** MLS RFC 9420 uses cryptographic primitives often with some labels, which are constant and indicates the computation context. For simplicity, our work continues to use primitives

such as Hash, Enc, Sig, and KDF without explicity introducing additional context and labels.
- **Key Expiration:** Our work follows [4] and omits the consideration of expiration of key packages or certificates.
- **Order of Proposal Applications:** Our work follows [4] and omits the formalization of the order that a list of proposal should apply, see [12, Section 12.3]. For simplicity, our ETK and $\mathsf{ETK}^{\mathrm{PSK}}$ protocols enforce a list of proposals to be applied in the order as each proposal appears in the list.
- **Membership Tag:** Our work follows [4] and omits the computation of membership tag on commits that do not include removal proposal.
- **Proposal Reference by Value:** [4] only contains proposals by reference, meaning that a proposal needs to be sent separately from a commit, and is included only via a ProposalRef. External commits require proposals by value in the commit, as the sender is not yet part of the group. For simplicity, we restrict the use of proposals by value to external commits.
- **PSK Life Time:** MLS RFC 9420 standard suggests that "the application SHOULD specify an upper limit on the number of past epochs for which the resumption PSK may be stored" [12, Section 8.6]. In our work, we do not model the life time of resumption PSKs, and simply assume that PSKs are never removed once generated. Consequently, all PSKs in possession of the member are exposed upon state corruption.

Additionally, upon a compromise, $\mathcal{F}_{\mathrm{ECGKA}}$ marks *all past* key packages as exposed and all commits that add such a key package. $\mathcal{F}_{\mathrm{ECGKA}}$ introduces the new init key to key packages, which enhances forward secrecy in practice: While the leaf node key can be compromised in one of the early epochs after adding, the init key can be deleted immediately after a join. However, We keep the simplification of [4] and also expose all past key packages upon compromise. $\mathcal{F}_{\mathrm{ECGKA}}$ can hence not capture this increased forward secrecy.

D Details on $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ Functionalities

Here, we describe the interfaces not described in the body of the paper. Furthermore, we give the formal description of the Authentication Service and Key Service functionalities in Figures 12 and 13. We define the helper functions in Figures 17 to 19. As before, we mark changes and additions from the model of [4] using colors according to Table 2. For details on the security model of $\mathcal{F}_{\text{CGKA}}$, we refer the reader to [4].

Interface `Commit`*:* This interface, of the form $(\texttt{Commit}, \boldsymbol{p}, \mathsf{spk}, \mathsf{force\text{-}rekey}, \texttt{wel\_type})$, considers the regular commit operation and is invoked only if the invoker `id` is in the group. $\mathcal{F}_{\text{ECGKA}}$ forwards this query to the adversary and receives an acknowledgment *ack*, a commit message c, welcome messages `vec_w`, groupInfo g, and an index rt of a detached root. Similar to the interface `Propose`, $\mathcal{F}_{\text{ECGKA}}$ first checks whether the commit is correct, or injected by the adversary. Next, $\mathcal{F}_{\text{ECGKA}}$ creates proposal nodes for all injected proposals. Then, $\mathcal{F}_{\text{ECGKA}}$ checks whether the public key spk of the invoker `id` is valid. If neither the node of the commit c nor the detached root root_{rt} exists, then $\mathcal{F}_{\text{ECGKA}}$ creates the node of c. The secret of this commit node is marked to `Bad` if no rekey-path is executed or the randomness is leaked. Otherwise, either the node of the commit c or the detached root root_{rt} exists. In this case, $\mathcal{F}_{\text{ECGKA}}$ attaches the commit node c to them if the attachment is valid. If this commit c includes any proposal that adds new members, the welcome messages must not be $\perp$, and point to c. The groupInfo g must not be $\perp$ and point to c. If the consistency and authenticity checks are not violated, then the commit c, welcome messages `vec_w`, and groupInfo g are returned.

Interface `Propose` *(Regular):* This interface, of the form $(\texttt{Propose}, \mathsf{act}), \mathsf{act} \in \{\mathsf{up\text{-}spk}, \mathsf{add\text{-}id}_t, \mathsf{rem\text{-}id}_t\}$, considers three conventional proposal operations: (1) update public key spk, (2) add a party $\mathtt{id}_t$, and (3) remove a party $\mathtt{id}_t$. The invoker `id` must be within the group and $\mathcal{F}_{\text{ECGKA}}$ forwards this query to the adversary $\mathcal{A}$. Upon receiving the proposal p, public key spk_t, and an acknowledgment *ack*, $\mathcal{F}_{\text{ECGKA}}$ checks whether the proposal is correct. Otherwise, it must be maliciously injected by the adversary (i.e., $ack = \texttt{true}$). If the node of proposal p has not been created, then $\mathcal{F}_{\text{ECGKA}}$ creates it using the information in this query (i.e., the query input and the reply from the adversary). Otherwise, $\mathcal{F}_{\text{ECGKA}}$ checks whether the information in this query is consistent with the existing node of p. If the randomness in this query is chosen by the adversary, then $\mathcal{F}_{\text{ECGKA}}$ exposes the secret key corresponding to the invoker `id`'s public key. This models the case in which ETK is used with a signature scheme (such as ECDSA) that is vulnerable to a randomness leakage attack [5]. In the end, the proposal p is returned.

Interface `Join`*:* The interface, of the form $(\texttt{Join}, \texttt{vec\_w})$, joins a group via a vector of welcome messages `vec_w`. $\mathcal{F}_{\text{ECGKA}}$ forwards this query to the adversary and receives an acknowledgment *ack*, a commit c', a groupInfo g, an origin orig', and a member list mem'. $\mathcal{F}_{\text{ECGKA}}$ checks whether the adversary has acknowledged

this query and looks for a commit $c = \texttt{Wel}[\texttt{vec\_w}]$ that this vector of welcome messages maps to. In the case that such a commit c does not exist, $\mathcal{F}_{\text{ECGKA}}$ creates and maps $\texttt{vec\_w}$ to it by using the input provided by the adversary: either simply use c' as c if the node of c' exists, or a new detached node for c with the origin orig', the member list mem', and status $\texttt{adv}$. In all cases, $\mathcal{F}_{\text{ECGKA}}$ moves the pointer of $\texttt{id}$ to this commit c that either previously existed or was newly created and marks $\texttt{id}$'s key to $\texttt{true}$ so that it can be challenged. The groupInfo g must be non-$\perp$ and point to c. If $\texttt{id}$ is included in the member list of this commit c and the consistency and authenticity checks are not violated, then the member list and origin of this commit c are returned.

Interface $\texttt{Key}$*:* This interface sets or challenges the application secret of the current node of the invoker $\texttt{id}$. $\mathcal{F}_{\text{ECGKA}}$ checks whether $\texttt{id}$ is a group member and whether the current application secret can be challenged (i.e., $\mathsf{HasKey}[\texttt{id}] = \texttt{true}$). If either check fails, then $\mathcal{F}_{\text{ECGKA}}$ exits. Otherwise, if the current application secret of the node c that $\texttt{id}$ points to has not been set, $\mathcal{F}_{\text{ECGKA}}$ further checks whether the corresponding safe predicate $\textbf{safe}(c)$ holds. If $\textbf{safe}(c) = \texttt{true}$, then the application secret is sampled uniformly at random and marked as challenged. Otherwise, the application secret is chosen by the adversary and marked as not challenged. In both cases, $\mathsf{HasKey}[\texttt{id}]$ is set to $\texttt{false}$, indicating that the application secret of every epoch can be challenged only once, and the new application secret is returned.

Interface $\texttt{Expose}$*:* The interface of the form $(\texttt{Expose}, \texttt{id})$ enables the adversary to corrupt $\texttt{id}$'s states and all included secrets. If the party $\texttt{id}$ is outside the group, i.e., the pointer of $\texttt{id}$ equals $\perp$, then $\mathcal{F}_{\text{ECGKA}}$ exits. Otherwise, $\texttt{id}$ is a member in the group. In this case, $\mathcal{F}_{\text{ECGKA}}$ marks $\texttt{id}$ as a corrupted member in the current commit node that $\texttt{id}$ points to. Next, $\mathcal{F}_{\text{ECGKA}}$ updates all status flags in all existing proposal and commit nodes. Then, $\mathcal{F}_{\text{ECGKA}}$ notifies the authentication and key services $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$ and $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$ that the current identity and ephemeral keys are exposed. $\mathcal{F}_{\text{ECGKA}}$ marks $\texttt{id}$ as a corrupted member in all commit nodes that $\texttt{id}$ joins using the corrupted ephemeral keys. Note that this corruption must not infect any commit node that has been challenged. If there exists any commit c that has been challenged[9], i.e., $\texttt{Node}[c].\mathsf{chall} = \texttt{true}$, and its safe predicate $\textbf{safe}(c)$ is changed to $\texttt{false}$, then all executions in this query will be undone.

Interface $\texttt{CorrRand}$*:* The interface ,of the form $(\texttt{CorrRand}, \texttt{id}, b)$, enables the adversary to manipulate or recover the randomness of the party $\texttt{id}$. If $b = \texttt{true}$, then $\mathcal{F}_{\text{ECGKA}}$ sets $\text{RndCor}[\texttt{id}]$ to $\texttt{true}$, indicating that the adversary now manipulates the randomness of $\texttt{id}$. If $b = \texttt{false}$, then $\mathcal{F}_{\text{ECGKA}}$ sets $\text{RndCor}[\texttt{id}]$ to $\texttt{false}$, indicating that the adversary loses control of the random number generator of the party $\texttt{id}$, i.e., all further values sampled by $\texttt{id}$ are again uniformly at random.

[9] Recall that a commit c can be challenged only if its safe predicate $\textbf{safe}(c) = \texttt{true}$.

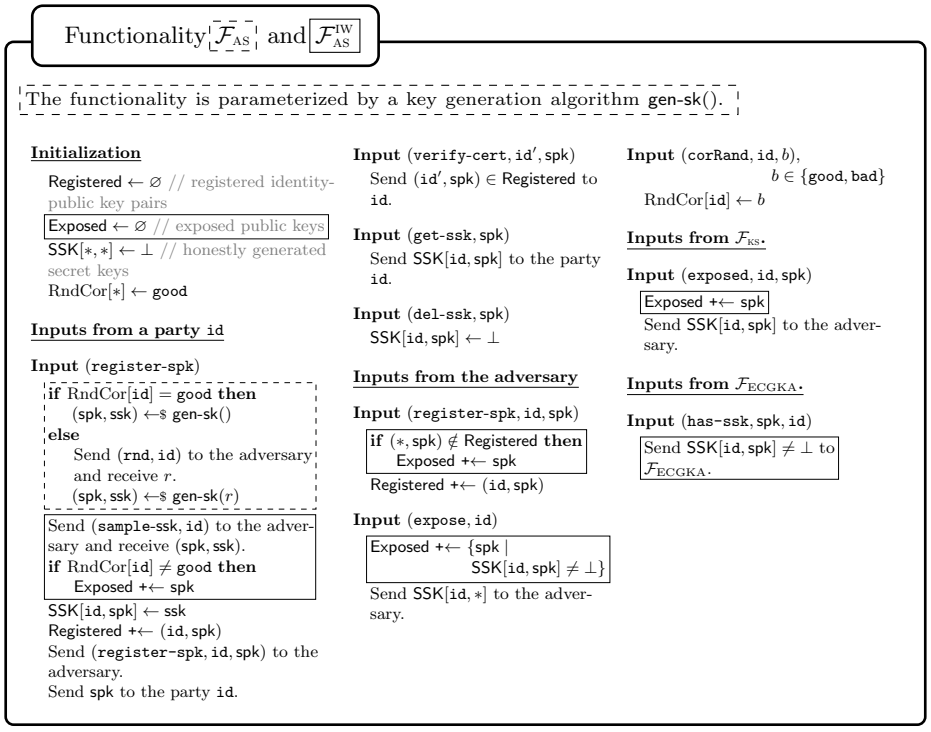


Fig. 12: The Authentication functionalities $\mathcal{F}_{\text{AS}}$, and their ideal-world counterparts $\mathcal{F}_{\text{AS}}^{\text{IW}}$. Code marked by solid or dashed boxes are executed only by the respective version, whereas code outside those boxes is shared by both variants. Taken from [4].

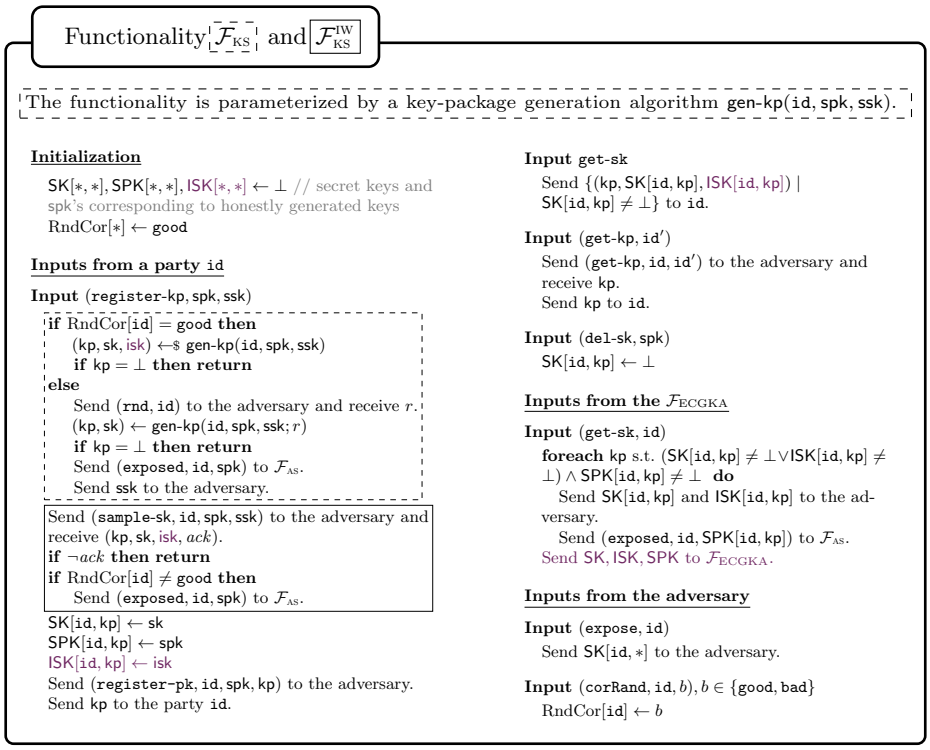


Fig. 13: The Key Service functionalities $\mathcal{F}_{\text{KS}}$, and their ideal-world counterparts $\mathcal{F}_{\text{KS}}^{\text{IW}}$. Code marked by solid or dashed boxes is executed only by the respective version, whereas code outside those boxes is shared by both variants. Reproduced from [4].

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ - Part 1

The functionality expects as part of the instance's session identifier sid the group creator's identity $\texttt{id}_{\text{creator}}$. It is parameterized in the predicates **safe**(c), specifying which keys are confidential.

Initialization

$\texttt{Ptr}[*], \texttt{Node}[*], \texttt{Prop}[*], \texttt{Wel}[*] \leftarrow \bot$
$\text{RndCor}[*] \leftarrow \texttt{good}$; $\text{HasKey}[*] \leftarrow \texttt{false}$
$\text{rootCtr} \leftarrow 0$
$\texttt{GroupInfo}[*] \leftarrow \bot$ // groupInfo object for external commits, indexed by groupInfo g
$\texttt{ExtProps}[*] \leftarrow \bot$ // external self-add, indexed by p

Inputs from $\texttt{id}_{\text{creator}}$

Input $(\texttt{Create}, \text{spk})$
req $\texttt{Node}[\text{root}_0] = \bot \wedge$ ***valid-spk**$(\texttt{id}_{\text{creator}}, \text{spk})$
$\text{mem} \leftarrow \{(\texttt{id}_{\text{creator}}, \text{spk})\}$
$\texttt{Node}[\text{root}_0] \leftarrow$ ***create-root**$(\texttt{id}_{\text{creator}}, \text{mem}, \text{RndCor}[\texttt{id}_{\text{creator}}], \text{epoch} = 0)$
$\text{HasKey}[\texttt{id}_{\text{creator}}] \leftarrow \texttt{true}$; $\texttt{Ptr}[\texttt{id}_{\text{creator}}] \leftarrow \text{root}_0$

Inputs from a party id

Input $(\texttt{Propose}, \text{act}), \text{act} \in \{\text{up-spk}, \text{add-id}_t, \text{rem-id}_t, \texttt{psk-epoch-use}\}$
req $\texttt{Ptr}[\texttt{id}] \neq \bot$
Send $(\texttt{Propose}, \texttt{id}, \text{act})$ to the adversary and receive (p, spk_t, ack).
if $\neg$***req-correctness**$(\text{'prop'}, \texttt{id}, \text{act})$ **then req** ack
if $\text{act} = \text{up-spk}$ **then assert** ***valid-spk**$(\texttt{id}, \text{spk})$
if $\text{act} = \text{add-id}_t$ **then** $\text{act} \leftarrow \text{add-id}_t\text{-spk}_t$
if $\texttt{Prop}[p] = \bot$ **then**
  $\texttt{Prop}[p] \leftarrow$ ***create-prop**$(\texttt{Ptr}[\texttt{id}], \texttt{id}, \text{act}, \text{RndCor}[\texttt{id}])$
else
  ***consistent-prop**$(p, \texttt{id}, \text{act})$
if $\text{RndCor}[\texttt{id}] = \texttt{bad}$ **then**
  Send $(\texttt{exposed}, \texttt{id}, \text{spk})$ to $\mathcal{F}_{\text{AS}}$.
return p

Input $(\texttt{Propose}, \texttt{extAdd-id-spk}, \text{epoch})$
req $\texttt{Ptr}[\texttt{id}] = \bot$
Send $(\texttt{Propose}, \texttt{id}, \texttt{extAdd-spk}, \text{epoch})$ to the adversary and receive (p, ack).
if $\neg$***req-correctness**$(\text{'ext-prop'}, \texttt{id}, \texttt{extAdd-spk})$ **then req** ack
assert ***valid-spk**$(\texttt{id}, \text{spk})$
$\text{act} \leftarrow \texttt{extAdd-id-spk}$
if $\texttt{Prop}[p] = \bot$ **then**
  $\texttt{Prop}[p] \leftarrow$ ***create-ext-prop**$(\texttt{id}, \text{act}, \text{epoch}, \text{RndCor}[\texttt{id}])$
  $\texttt{ExtProps}[p] \leftarrow$ new node with epoch $\leftarrow$ epoch
else
  ***consistent-ext-prop**$(p, \texttt{id}, \text{act}, \text{epoch})$
if $\text{RndCor}[\texttt{id}] = \texttt{bad}$ **then**
  Send $(\texttt{exposed}, \texttt{id}, \text{spk})$ to $\mathcal{F}_{\text{AS}}$.
return p

Input $(\texttt{Commit}, \boldsymbol{p}, \text{spk}, \text{force-rekey}, \texttt{wel\_type})$
req $\texttt{Ptr}[\texttt{id}] \neq \bot$
Send $(\texttt{Commit}, \texttt{id}, \boldsymbol{p}, \text{spk}, \text{force-rekey}, \texttt{wel\_type})$ to the adversary and receive $(ack, c, \texttt{vec\_w}, g, rt, \boldsymbol{c_{psk}})$.
if $\neg$***req-correctness**$(\text{'comm'}, \texttt{id}, \boldsymbol{p}, \text{spk}, \text{force-rekey})$ **then req** ack
***fill-props**$(\texttt{id}, \boldsymbol{p})$
$\text{res} =$ ***fixate-psk-refs**$((\texttt{Prop}[p] : p \in \boldsymbol{p}), \boldsymbol{c_{psk}})$
assert $\text{res} \neq \bot$
if ***keep-spk**$(\boldsymbol{p}, \text{force-rekey})$ **then**
  $\text{spk} \leftarrow \texttt{Node}[\texttt{Ptr}[\texttt{id}]].\text{mem}[\texttt{id}]$
assert ***valid-spk**$(\texttt{id}, \text{spk})$
$\text{mem} \leftarrow$ ***members**$(\texttt{Ptr}[\texttt{id}], \texttt{id}, [\texttt{Prop}[p] : p \in \boldsymbol{p}], \text{spk})$
assert $\text{mem} \neq \bot \wedge (\texttt{id}, \text{spk}) \in \text{mem}$
for $p \in \boldsymbol{p}$ **s.t.** $\texttt{Prop}[p].\text{act} = \text{psk-}*$ **do**
  req $\texttt{id} \in \texttt{Node}[\texttt{Prop}[p].c_{\text{psk}}].\texttt{has\_psk}$
if $\texttt{Node}[c] = \bot \wedge rt = \bot$ **then**
  if ***keep-spk**$(\boldsymbol{p}, \text{force-rekey})$ **then**
    $\text{stat} \leftarrow \texttt{bad}$
  else $\text{stat} \leftarrow \text{RndCor}[\texttt{id}]$
  $\texttt{Node}[c] \leftarrow$ ***create-child**$(\texttt{Ptr}[\texttt{id}], \texttt{id}, \boldsymbol{p}, \text{mem}, \text{stat})$
else
  if $rt \neq \bot$ **then**
    $c' \leftarrow \text{root}_{rt}$
    assert $\text{RndCor}[\texttt{id}] \neq \text{good}$
  else
    $c' \leftarrow c$
    if $\neg$***keep-spk**$(\boldsymbol{p}, \text{force-rekey})$ **then**
      assert $\text{RndCor}[\texttt{id}] \neq \text{good}$
  assert $\texttt{Node}[c'].*\texttt{origRChild} = \texttt{id}$
  assert ***valid-successor**$(c', \texttt{id}, \boldsymbol{p}, \text{mem})$
  if $c \neq c'$ **then** ***attach**$(c, c', \texttt{id}, \boldsymbol{p})$
assert $\texttt{vec\_w} \neq \bot$ **iff** $\exists p \in \boldsymbol{p} : \texttt{Node}[p].\text{act} \in \{\text{add-}*, \texttt{extAdd-}*\}$
if $\texttt{vec\_w} \neq \bot$ **then**
  $\boldsymbol{c_{psk}} = \texttt{Prop}[p].c_{\text{psk}}$ for all $p \in \boldsymbol{p}$ s.t. $\texttt{Prop}[p].\text{act} = \text{psk-}*$
  assert $\texttt{Wel}[\texttt{vec\_w}] \in \{\bot, (c, \boldsymbol{c_{psk}})\}$
  $\texttt{Wel}[\texttt{vec\_w}] \leftarrow (c, \boldsymbol{c_{psk}})$
assert $g \neq \bot$
assert $\texttt{groupInfo}[g] \in \{\bot, c\}$
$\texttt{groupInfo}[g] \leftarrow c$
assert **cons-invariant** $\wedge$ **auth-invariant**
if $\text{RndCor}[\texttt{id}] = \texttt{bad}$ **then**
  Send $(\texttt{exposed}, \texttt{id}, \texttt{Node}[\texttt{Ptr}[\texttt{id}]].\text{mem}[\texttt{id}])$ to $\mathcal{F}_{\text{AS}}$.
return $(c, \texttt{vec\_w}, g)$

Input Key
req $\texttt{Ptr}[\texttt{id}] \neq \bot \wedge \text{HasKey}[\texttt{id}]$
if $\texttt{Node}[\texttt{Ptr}[\texttt{id}]].\text{key} = \bot$ **then** ***set-key**$(\texttt{Ptr}[\texttt{id}])$
$\text{HasKey}[\texttt{id}] \leftarrow \texttt{false}$
return $\texttt{Node}[\texttt{Ptr}[\texttt{id}]].\text{key}$

Fig. 14: Part 1 of our $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ definition, using the color coding from Table 2. The orange code (●) is included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, but not in $\mathcal{F}_{\text{ECGKA}}$.

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ - Part 2

Input (ExtCommit, g, spk, e_{psk}, resync)
  if resync = true **then req** Ptr[id] $\neq \bot$
  else req Ptr[id] $= \bot$
  Send (ExtCommit, id, g, spk, e_{psk}, resync) to the adversary and receive ($ack, c, g', p_{exi}, p_{rem}, p_{psk}$, c_{gm}, orig_g, mem_g, epoch, c_{psk}).
  if ¬*req-correctness('ext-comm', id, g, spk resync = true) **then req** ack
  assert *valid-spk(id, spk)
  $c_g \leftarrow$ groupInfo[g]
  if $c_g = \bot$ **then**
    if Node[c_{gm}] $\neq \bot$ **then**
      $c_g \leftarrow c_{gm}$
    else
      rootCtr++; $c \leftarrow \text{root}_{\text{rootCtr}}$
      Node[c_g] $\leftarrow$ *create-root(orig_g, mem_g, adv, epoch)
    groupInfo[g] $\leftarrow c_g$
  ExtCommitProps[p_{exi}] += *create-ext-commit-props(groupInfo[g], 'exi-id', id)
  if $p_{rem} \neq \bot$ **then**
    ExtCommitProps[p_{rem}] += *create-ext-commit-props(groupInfo[g], 'rem-id', id)
  for e, p in zip(e_{psk}, p_{psk}) **do**
    ExtCommitProps[p_{psk}] += *create-ext-commit-props(groupInfo[g], 'psk-e-application', id)
  try ExtCommitProps = *fixate-psk-refs(ExtCommitProps, c_{psk})
  for prop ∈ ExtCommitProps **s.t.** prop.act = psk-∗ **do**
    req id ∈ Node[prop.c_{psk}].has_psk
  $\boldsymbol{p} = [p_{exi}, p_{rem}, p_{psk}]$
  mem $\leftarrow$ *members(groupInfo[g], id, ExtCommitProps, spk)
  assert mem $\neq \bot \wedge$ (id, spk) ∈ mem
  if Node[c] $= \bot$ **then**
    stat $\leftarrow$ RndCor[id]
    Node[c] $\leftarrow$ *create-child(groupInfo[g], id, $\boldsymbol{p}$, mem, stat, 'new_member', ExtCommitProps)
  else
    *consistent-ext-comm(c, g, id, p, mem, ExtCommitProps)
  assert $g' \neq \bot$
  assert groupInfo[g'] $\in \{\bot, c\}$
  groupInfo[g'] $\leftarrow c$
  assert cons-invariant $\wedge$ **auth-invariant**
  if RndCor[id] = bad **then**
    Send (exposed, id, Node[Ptr[id]].mem[id]) to $\mathcal{F}_{\text{AS}}$.
  return (c, g')

Corruptions

Input (Expose, id)
  if Ptr[id] $\neq \bot$ **then**
    Node[Ptr[id]].exp +← (id, HasKey[id])
    *update-stat-after-exp(id)
    Send (exposed, id, Node[Ptr[id]].mem[id]) to $\mathcal{F}_{\text{AS}}$.
  Send (get-sk, id) to $\mathcal{F}_{\text{KS}}$ and receive SK, ISK and SPK.
  for each kp **s.t.** (SK[id, kp] $\neq \bot \vee$ ISK[id, kp] $\neq \bot$) $\wedge$ SPK[id, kp] = spk **do**
    for each c **s.t.** $\exists p \in$ Node[c].pro : Prop[p].act = add-id-spk **do**
      Node[c].exp +← (id, true)
  This input is disallowed if $\exists c$: Node[c].chall $\wedge$ ¬**safe**(c)

Input (CorrRand, id, b), $b \in$ {good, bad}
  RndCor[id] $\leftarrow b$

Input (Process, $c, \boldsymbol{p}$)
  if $\boldsymbol{p} = \bot$ **then return** *process-ec(c)
  Send (Process, id, $c, \boldsymbol{p}$) to the adversary and receive (ack, rt, orig', spk', c_{psk}).
  if ¬*req-correctness('proc', id, $c, \boldsymbol{p}$) **then req** ack
  *fill-props(id, $\boldsymbol{p}$)
  try res = *fixate-psk-refs(Prop[p] : $p \in \boldsymbol{p}, c_{psk}$)
  if Node[c] $= \bot \wedge rt = \bot$ **then**
    mem $\leftarrow$ *members(Ptr[id], orig', $\boldsymbol{p}$, spk')
    assert mem $\neq \bot$
    for $p \in \boldsymbol{p}$ **s.t.** Prop[p].act = psk-∗ **do**
      req id ∈ Node[Prop[p].c_{psk}].has_psk $\vee$ ∄(id, ∗) ∈ mem
    Node[c] $\leftarrow$ *create-child(Ptr[id], orig', $\boldsymbol{p}$, mem, adv)
  else
    if Node[c] $= \bot$ **then** $c' \leftarrow \text{root}_{rt}$
    else $c' \leftarrow c$
    $\text{id}_c \leftarrow$ Node[c'].orig; $\text{spk}_c \leftarrow$ Node[c'].mem[id_c]
    mem $\leftarrow$ *members(Ptr[id], $\text{id}_c, \boldsymbol{p}, \text{spk}_c$)
    assert mem $\neq \bot$
    for $p \in \boldsymbol{p}$ **s.t.** Prop[p].act = psk-∗ **do**
      req id ∈ Node[Prop[p].c_{psk}].has_psk $\vee$ ∄(id, ∗) ∈ mem
    *valid-successor(c', id, $\boldsymbol{p}$, mem)
    if $c \neq c'$ **then** *attach(c, c', id, $\boldsymbol{p}$)
  if $\exists p \in \boldsymbol{p}$: Prop[p].act = rem-id **then**
    Ptr[id] $\leftarrow \bot$
  else
    assert id ∈ Node[c].mem
    Ptr[id] $\leftarrow c$; HasKey[id] $\leftarrow$ true
    Node[c].has_psk +← id
  assert cons-invariant $\wedge$ **auth-invariant**
  return *output-proc(c)

Input (Join, vec_w)
  Send (Join, id, vec_w) to the adversary and receive (ack, c', g, orig', mem', epoch, c_{psk}').
  req ack
  (c, c_{psk}) $\leftarrow$ Wel[vec_w]
  if $c = \bot$ **then**
    if Node[c'] $\neq \bot$ **then**
      $c \leftarrow c'$; $c_{psk} \leftarrow c_{psk}$'
    else
      rootCtr++; $c \leftarrow \text{root}_{\text{rootCtr}}$; $c_{psk} \leftarrow c_{psk}$'
      Node[c] $\leftarrow$ *create-root(orig', mem', adv, epoch)
    Wel[vec_w] $\leftarrow$ (c, c_{psk})
  for $p \in$ Node[c].pro **s.t.** Prop[p].act = psk-∗ **do**
    req id ∈ Node[Prop[p].c_{psk}].has_psk
  Ptr[id] $\leftarrow c$
  HasKey[id] $\leftarrow$ true
  Node[c].has_psk +← id
  assert $g \neq \bot$
  assert groupInfo[g] $\in \{\bot, c\}$
  groupInfo[g] $\leftarrow c$
  assert id ∈ Node[c].mem $\wedge$ **cons-invariant** $\wedge$ **auth-invariant**
  return *output-join(c)

Fig. 15: Part 2 of our $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ definition, using the color coding from Table 2. The orange code (●) is included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$, but not in $\mathcal{F}_{\text{ECGKA}}$.

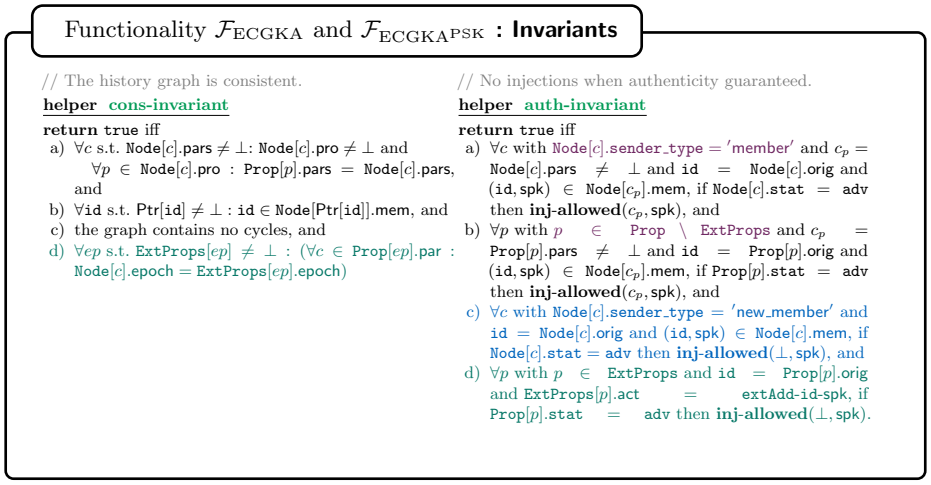


Fig. 16: The cons- and auth-invariant predicates for the $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ functionalities, adapted from [4] using the color coding from Table 2. The codes highlighted with ● color are only included in $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$: Bookkeeping Helpers

```
// Creating nodes
helper *create-child(c, id, p, mem, stat,
        sender_type = 'member', ExtCommitProps = ⊥)

  if sender_type = 'new_member' then
     req ExtCommitProps ≠ ⊥
  c' = new node with pars ← c, orig ← id, pro ← p,
          mem ← mem, stat ← stat, psk ← stat,
          epoch ← Ptr[id] + 1,
          sender_type ← sender_type,
          ExtCommitProps ← ExtCommitProps,
  epoch ← Node[c].epoch
  for eP ∈ ExtProps do
     if ExtProps[eP].epoch = epoch then
     Prop[eP].pars +← c
  return c

helper *create-root(id, mem, stat, epoch,
          sender_type = 'member')

  return new node with pars ← ⊥, orig ← id,
          pro ← ⊥, mem ← mem, stat ← stat,
          epoch ← epoch,
          sender_type ← sender_type, psk ← stat,

helper *create-prop(c, id, act, stat, c_psk = ⊥)

  return new proposal with pars ← c, orig ← id,
          act ← act, stat ← stat,
          psk ← stat, c_psk ← c_psk

helper *create-ext-prop(id, act, epoch, stat)

  c_par ← {∀c s.t. Node[c].epoch = epoch c}
  return new proposal with pars ← c_par, orig ← id,
          act ← act, stat ← stat, psk ← stat,

helper *fill-props(id, p)

  for p ∈ p s.t. Prop[p] = ⊥ do
     Send (Proposal, p) to the adversary and receive
     (orig, act, epoch).

     if epoch ≠ ⊥ then

        Prop[p] ← *create-ext-prop(orig, act,
        epoch, adv)
     else
        Prop[p] ← *create-prop(Ptr[id], orig, act, adv)

// Does the vector of proposals create an add-only
commit?
helper *only-adds-and-psk(p)

  return p ≠ () ∧ ∀p ∈ p : Prop[p] ≠ ⊥ ∧ Prop[p].act =
  (add-∗ ∨ psk-∗)

// Does the commit require an update path?
helper *keep-spk(p, force-rekey)

  return *only-adds-and-psk(p) ∧ ¬force-rekey∗

// Output of process and join
helper *output-proc(c)

  id_c ← Node[c].orig
  (∗, propSem) ← *apply-props(c, id_c, Node[c].pro,
  Node[c].mem[id_c])
  return (Node[c].orig, propSem)

helper *output-join(c)

  return (Node[c].mem, Node[c].orig)

// Is the (new) spk' valid for update or commit?
helper *valid-spk(id, spk')

  spk ← Node[Ptr[id]].mem[id]
  if spk ≠ ⊥ ∧ spk' = spk then return true
  Send (has-ssk, spk', id) to F_AS and receive ack
  return ack

// Generating the group key (secure or insecure)
helper *set-key(c)

  if ¬safe(c) then
     Send (Key, id) to the adversary and receive I.
     Node[c].key ← I
     Node[c].chall ← false
  else
     Node[c].key ←$ I
     Node[c].chall ← true

// Corruptions
helper *update-stat-after-exp(id)

  for each p s.t. Prop[p] ≠ ⊥ and
      (a) Ptr[id] ∈ Prop[p].pars and
      (b) Prop[p].orig = id and
      (c) Prop[p].act = up
  do Prop[p].stat ← bad
  for each c s.t. Node[c] ≠ ⊥ and
      (a) Node[c].pars = Ptr[id] and
      (b) Node[c].orig = id
  do Node[c].stat ← bad, Node[c].psk ← bad

helper *create-ext-commit-props(parent, act, orig)

  return new proposal with pars ← parent, orig ←
  orig, act ← act

helper *fixate-psk-refs(props, c_psk)

  pskProps = [prop ∈ props s.t. prop ≠ ⊥ and
  prop.act = psk-epoch-∗]
  for prop, c_psk ∈ zip(pskProps, c_psk) do
     if prop.c_psk = ⊥ then
        req Node[c_psk] ≠ ⊥
        req Node[c_psk].epoch = epoch
        prop.c_psk = c_psk
  return props
```

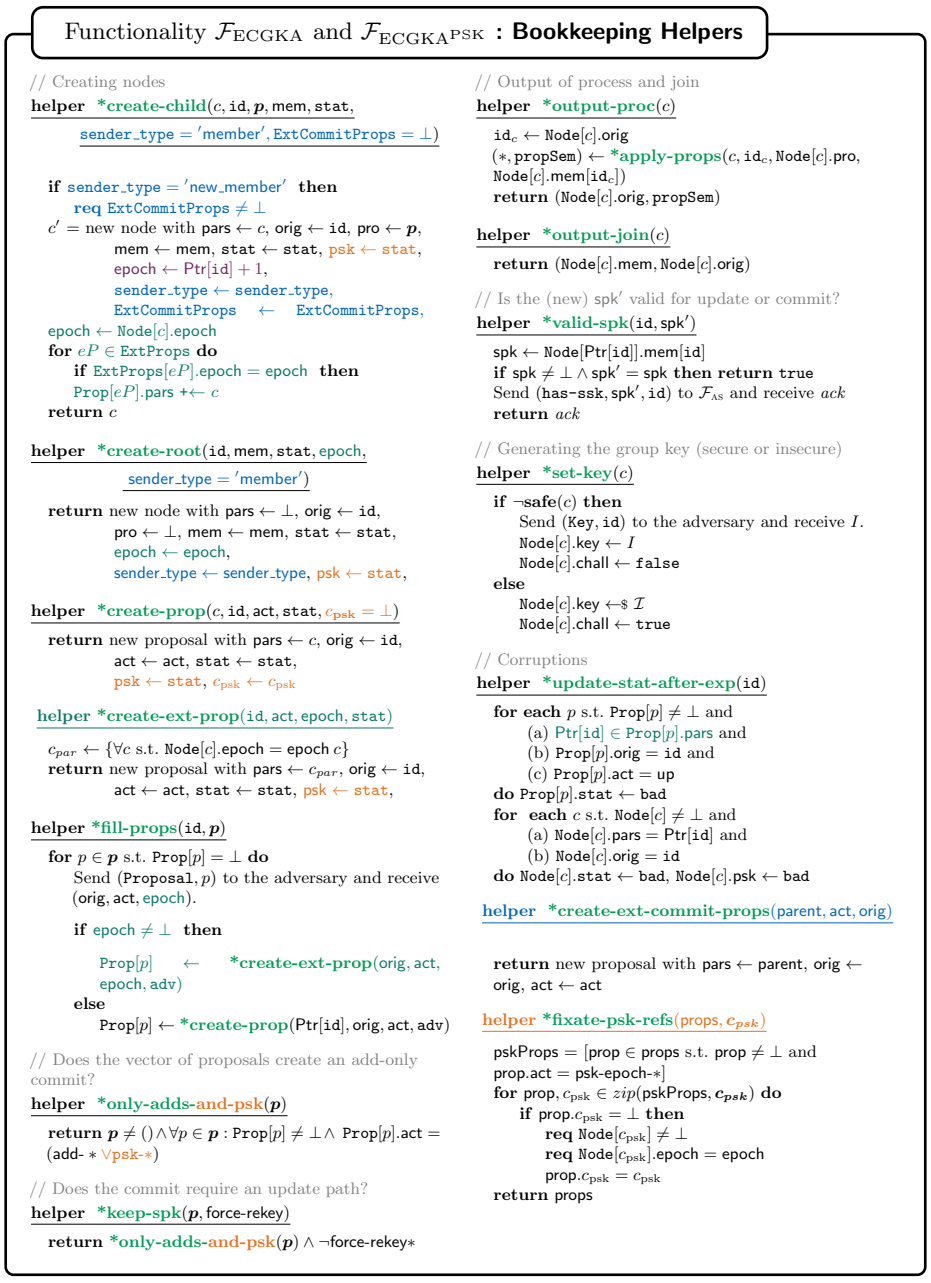


Fig. 17: Bookkeeping helper functions for $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

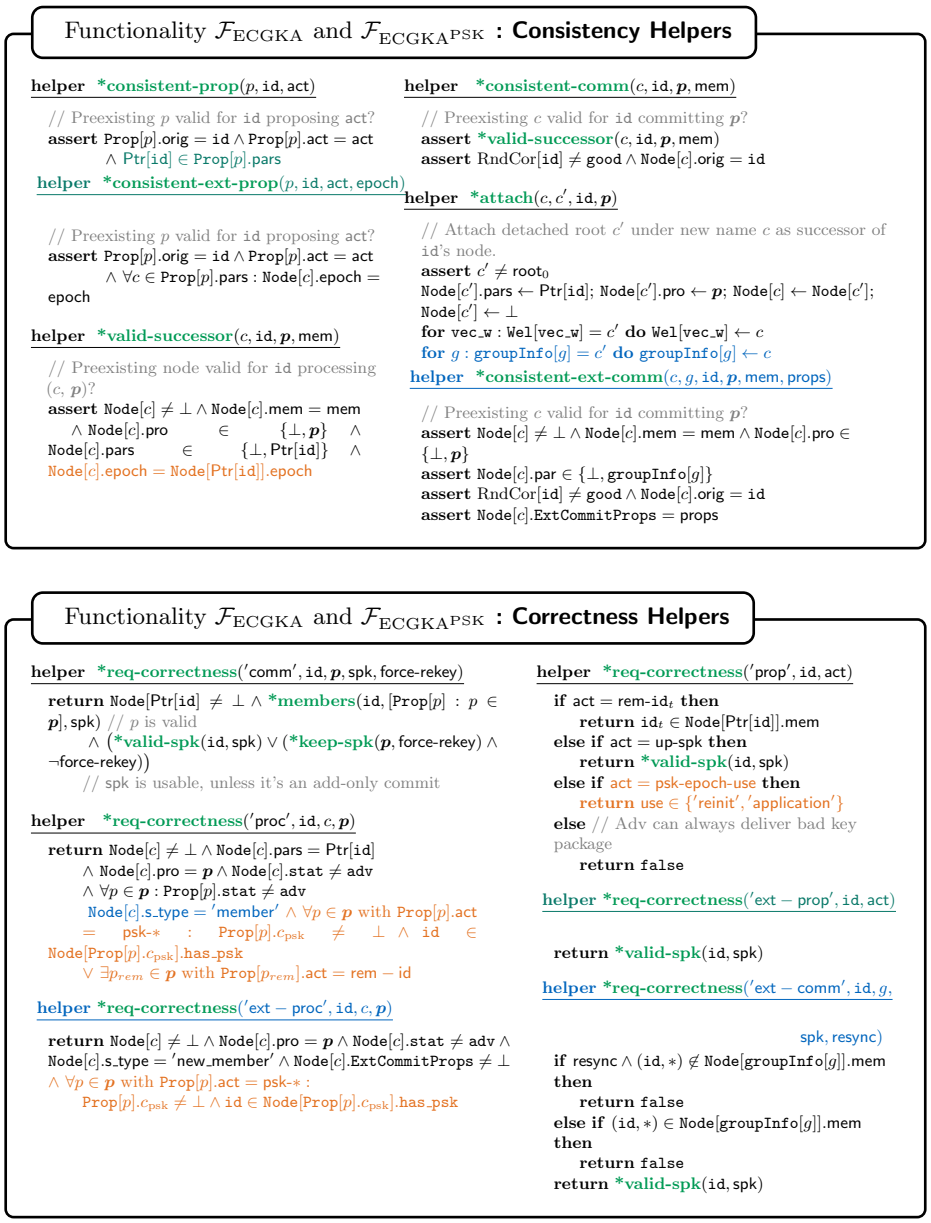


Fig. 18: Consistency and correctness helper functions for $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

Functionality $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$: **Group State Helpers**

helper ***members**$(c, \mathtt{id}_c, \text{props}, \mathsf{spk}_c)$

$(G, *) \leftarrow$ ***apply-props**$(c, \mathtt{id}_c, \text{props}, \mathsf{spk}_c)$
if $(G, *) = \bot$ **then return** $\bot$
else return G

helper ***apply-props**$(c, \mathtt{id}_c, \boldsymbol{p}, \mathsf{spk}_c)$

// Returns group members G and proposal semantics P resulting from applying $\boldsymbol{p}$ to state $\mathtt{Node}[c]$, or $\bot$ if $\boldsymbol{p}$ is invalid.
req $\mathtt{Node}[c] \neq \bot$
req $\forall \text{prop} \in \text{props} : \text{prop} \neq \bot \wedge c \in \text{prop.pars}$
req $\boldsymbol{p} = \boldsymbol{p}_{\text{up}} +\!\!+ \boldsymbol{p}_{\text{rem}} +\!\!+ \boldsymbol{p}_{\text{add}} +\!\!+ p_{psk}$ for some $\boldsymbol{p}_{\text{up}}$, $\boldsymbol{p}_{\text{rem}}$, $\boldsymbol{p}_{\text{add}}$, $+\!\!+ \boldsymbol{p_{psk}}$
$\vee \boldsymbol{p} = p_{exi} +\!\!+ p_{rem} +\!\!+ \boldsymbol{p_{psk}}$ for some p_{exi} with $\text{len}(p_{exi})=1$ and p_{rem} with $\text{len}(p_{rem})=1$
with $\forall \text{act} \forall p \in \boldsymbol{p}_{\text{act}} : \mathtt{Node}[p].\text{act} = \text{act-}*$
$G \leftarrow \mathtt{Node}[c].\text{mem}$;
if $p_{exi} \in p \wedge \neg p_{rem} \in p$ **then**
  req $(id_c, *) \notin G$
else
  req $(id_c, *) \in G$
  G -$\leftarrow$ $(\mathtt{id}_c, *)$;
G +$\leftarrow$ $(\mathtt{id}_c, \mathsf{spk}_c)$
$L \leftarrow \{\mathtt{id}_c\}$ // set of updated parties
for $p \in \boldsymbol{p}_{\text{up}}$ **do**
  $(\mathtt{id}_s, \text{up-spk}) \leftarrow (\text{prop.orig}, \text{prop.act})$
  req $\mathtt{id}_s \in G \setminus L$
  G -$\leftarrow$ $(\mathtt{id}_s, *)$; G +$\leftarrow$ $(\mathtt{id}_s, \text{spk})$
  L +$\leftarrow$ $\mathtt{id}_s$
for $p \in \boldsymbol{p}_{\text{rem}}$ **do**
  $(\mathtt{id}_s, \text{rem-id}_t) \leftarrow (\text{prop.orig}, \text{prop.act})$
  if $p_{exi} \notin p$ **then**
    req $\mathtt{id}_s \in G \wedge \mathtt{id}_t \in G \setminus L$
    G -$\leftarrow$ $(\mathtt{id}_t, *)$
  else
    req $id_s = id_t = id_c \wedge id_s \in G$
for $p \in \boldsymbol{p}_{\text{add}}$ **do**
  $(\mathtt{id}_s, \text{add-id}_t\text{-spk}_t) \leftarrow (\text{prop.orig}, \text{prop.act})$
  req $\mathtt{id}_s \in G \wedge \mathtt{id}_t \notin G \vee (id_s = id_t \wedge id_t \notin G)$
  G +$\leftarrow$ $(\mathtt{id}_t, \mathsf{spk}_t)$
for $p \in \boldsymbol{p}_{psk}$ **do**
  $id_s, \text{psk-epoch-use}, c_{\text{psk}} \leftarrow (\text{prop.orig}, \text{prop.act}, \text{prop.}c_{\text{psk}})$
  req $(\exists \mathtt{Node}[c] \text{ s.t. } \mathtt{Node}[c].\text{epoch} = \text{epoch}) \wedge (\mathtt{id} \in \mathtt{Node}[c].\mathtt{has\_psk} \vee \neg\exists(\mathtt{id}, *) \in G)$
$P \leftarrow ((\text{prop.orig}, \text{prop.act}) : p \in \boldsymbol{p})$
return (G, P)

helper ***process-ec**(c)

Send $(\mathtt{Process}, \mathtt{id}, c)$ to the adversary and receive $(ack, \mathsf{orig}', \mathsf{spk}', \boldsymbol{p}, \boldsymbol{c_{psk}})$.
if $\neg$***req-correctness**$(\text{'ext} - \text{proc}', \mathtt{id}, c, \boldsymbol{p})$ **then**
  req ack
if $\text{Ptr}[id] \neq \bot$ **then**
  if $\mathtt{Node}[c] = \bot$ **then**
    for $p \in \boldsymbol{p}$ **do**
      Send $(\mathtt{Proposal}, p)$ to the adversary and receive $(\mathtt{act})$.
      $\mathtt{ExtCommitProps}[p] \leftarrow$ ***create-ext-commit-props**$(\text{Ptr}[id], \text{act}, \text{orig}')$
      req $\text{act} \in \{\text{add-}*, \text{rem-}*, \text{psk-}*\}$
      if $\text{act} \in \{\text{add-id}_t \text{or} \text{rem-id}_t\}$ **then**
        req $\mathtt{id}_t = \text{orig}'$
    try $\mathtt{ExtCommitProps} =$ ***fixate-psk-refs**$(\mathtt{ExtCommitProps}, \boldsymbol{c_{psk}})$
    for $\text{prop} \in \mathtt{ExtCommitProps}$ **s.t.** $\text{prop.act} = \text{psk-}*$ **do**
      req $\mathtt{id} \in \mathtt{Node}[\text{prop.}c_{\text{psk}}].\mathtt{has\_psk}$
    $\text{mem} \leftarrow$ ***members**$(\text{Ptr}[\mathtt{id}], \text{orig}', \mathtt{ExtCommitProps}, \mathsf{spk}')$
    assert $\text{mem} \neq \bot$
    $\mathtt{Node}[c] \leftarrow$ ***create-child**$(\text{Ptr}[\mathtt{id}], \text{orig}', \boldsymbol{p}, \text{mem}, \text{adv}, \text{'new\_member'}, \mathtt{ExtCommitProps})$
  else
    $\mathtt{id}_c \leftarrow \mathtt{Node}[c].\text{orig}$
    $\mathsf{spk}_c \leftarrow \mathtt{Node}[c].\text{mem}[\mathtt{id}_c]$
    $\mathtt{ExtCommitProps} \leftarrow \mathtt{Node}[c].\mathtt{ExtCommitProps}$
    for $\text{prop} \in \mathtt{ExtCommitProps}$ **s.t.** $\text{prop.act} = \text{psk-}*$ **do**
      req $\mathtt{id} \in \mathtt{Node}[\text{prop.}c_{\text{psk}}].\mathtt{has\_psk}$
    $\text{mem} \leftarrow$ ***members**$(\text{Ptr}[\mathtt{id}], \mathtt{id}_c, \mathtt{ExtCommitProps}, \mathsf{spk}_c)$
    assert $\text{mem} \neq \bot$
    ***valid-successor**$(c, \mathtt{id}, \mathtt{ExtCommitProps}, \text{mem})$
else
  req $\mathtt{Node}[c] \neq \bot$
  req $\mathtt{Node}[c].\mathtt{stat} \neq \mathtt{adv} \wedge \mathtt{Node}[c].\text{orig} = \mathtt{id}$
assert $\mathtt{id} \in \mathtt{Node}[c].\text{mem}$
$\text{Ptr}[\mathtt{id}] \leftarrow c$
$\text{HasKey}[\mathtt{id}] \leftarrow \mathtt{true}$
$\mathtt{Node}[c].\mathtt{has\_psk}$ +$\leftarrow$ $\mathtt{id}$
assert **cons-invariant** $\wedge$ **auth-invariant**
return ***output-proc**(c)

Fig. 19: Group state helper functions for $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$.

E Details on ETK and ETK^{PSK} Protocols

In the following sections, we provide a detailed description of our ETK and ETK^{PSK} protocols, highlighting modifications for adapting to external commits, external self-add proposals, and PSK proposals. These changes are color-coded as indicated in Table 2. In this section, we outline the necessary modifications to transition from Draft 12 (partially analyzed in [4]) to the final RFC draft. Additionally, we describe the simplifications and operations that were excluded from ETK and ETK^{PSK}.

We provide the notation for the public ratchet tree and the protocol state in Tables 3 to 7. We depict the ETK protocol in Figures 20 and 21. We define helper functions in Figures 22 to 27. Changes and additions from the model of [4] are color-coded as indicated in Table 2.

τ.root	Returns the root.
τ.nodes	Returns the set of all nodes in the tree.
v.isroot	Returns true iff $v = \tau$.root.
v.isleaf	Returns true iff v has no children.
v.par	Returns the parent node of v (or $\perp$ if v.isroot).
v.lchild	Returns the left child of v (or $\perp$ if v.isleaf).
v.rchild	Returns the right child of v (or $\perp$ if v.isleaf).
v.sibling	Returns the unique node $u \neq v$ s.t. of u.par $= v$.par.
v.nodeIdx	Returns the node index of v.

Table 3: Notation for the public ratchet tree, based on [4].

v.pk	The public key of a public-key encryption scheme.
v.sk	The corresponding secret key.
v.parentHash	A hash value binding the node to all of its ancestors.
v.unmergedLvs	The set of leaf indices rooted below v, for which the corresponding party does not know v.sk.
v.id	If v.isleaf: the identity associated with that leaf.
v.leafIdx	If v.isleaf: a unique identifier of the leaf, counted from the leftmost to the rightmost leaf.
v.spk	If v.isleaf: an associated verification key of a signature scheme.
v.sig	If v.isleaf: A signature of the leaf's labels under the signing key corresponding to v.spk.

Table 4: The node labels of the ratchet tree.

Protocol ETK and ETK^{PSK} - Part 1

Input (Create, spk)
  req $\gamma = \bot \wedge \texttt{id} = \texttt{id}_{\text{creator}}$
  γ.groupId, γ.initSecret $\leftarrow\$$ $\{0,1\}^\kappa$
  γ.epoch $\leftarrow 0$; γ.interimTransHash $\leftarrow \epsilon$
  γ.certSpks[$*$], γ.pendUp[$*$], γ.pendCom[$*$] $\leftarrow \bot$
  γ.psk_store $\leftarrow ()$
  $\gamma.\tau \leftarrow$ new LBBT$_1$
  try γ.ssk $\leftarrow$ *fetch-ssk-if-nec(γ, spk)
  (ln, sk) $\leftarrow\$$ gen-leaf-node(id, spk, ssk, ϵ, 'commit',
      (γ.groupId, $\gamma.\tau$.leafIdx))
  $\gamma.\tau$.leaves[0].leafIdx $\leftarrow 0$
  $\gamma.\tau$.leaves[0].assignKp(ln)
  $\gamma.\tau$.leaves[0].sk $\leftarrow$ sk

Input (Propose, up-spk)
  req $\gamma \neq \bot$
  try ssk $\leftarrow$ *fetch-ssk-if-nec(γ, spk)
  (ln, sk) $\leftarrow\$$ gen-leaf-node(id, spk, ssk, ϵ, 'update',
      (γ.groupId, $\gamma.\tau$.leafIdx))
  $P \leftarrow$ ('upd', ln); $p \leftarrow$ *frameProp(γ, P)
  γ.pendUp[p] $\leftarrow$ (ssk, sk)
  return p

Input (Propose, add-id$_t$)
  req $\gamma \neq \bot \wedge \texttt{id}_t \notin \gamma.\tau$.roster()
  kp$_t \leftarrow$ **query** (get-kp, id$_t$) to $\mathcal{F}_{\mathcal{KS}}$
  try $\gamma \leftarrow$ *validate-kp(γ, kp$_t$, id_t, 'key_package', ϵ)
  $P \leftarrow$ ('add', kp$_t$); $p \leftarrow$ *frameProp(γ, P)
  return p

Input (Commit, $\boldsymbol{p}$, spk, force-rekey, wel_type)
  req $\gamma \neq \bot$
  $\gamma' \leftarrow$ *init-epoch(γ)
  try (γ', upd, rem, add, psk) $\leftarrow$
   *apply-props(γ, γ', $\boldsymbol{p}$)
  req ($*$, 'rem'-id) $\notin rem \wedge$ (id, $*$) $\notin upd$
  if force-rekey $\vee\ \boldsymbol{p} = () \vee upd \neq () \vee rem \neq ()$ **then**
    try (γ', commitSec, updatePath, pathSecs)
      $\leftarrow$ *rekey-path(γ', id, spk)
  else
    commitSec $\leftarrow 0$; updatePath $\leftarrow \epsilon$
    pathSecs[$*$] $\leftarrow \epsilon$
  propIDs $\leftarrow ()$
  for $p \in \boldsymbol{p}$ **do**
    propIDs $\mathbin{+\!\!\leftarrow}$ Hash(p)
  $C \leftarrow$ (propIDs, updatePath)
  sig $\leftarrow$ *signCommit(γ, C)
  $\gamma' \leftarrow$ *set-conf-trans-hash(γ, γ', γ.leafIdx(), C, sig)
  pskIds $=$ *get-pskIds($\boldsymbol{p}$)
  try psk_secret $\leftarrow$ *derive-psk-secret(pskIds)
  (γ', confKey, joinerSec)
    $\leftarrow$ *derive-keys(γ, γ', commitSec, psk-secret)
  confTag $\leftarrow$ *conf-tag(γ', confKey)
  if $rem \neq ()$ **then**
    membTag $\leftarrow$ MAC.tag(γ.membKey, C)
  else membTag $\leftarrow \bot$
  $c \leftarrow$ *frameCommit(γ, C, confTag, sig, membTag)
  if $add \neq ()$ **then**
    (γ', vec_w) $\leftarrow$ *welcome-msg(γ', add, joinerSec,
      pathSecs, confTag, pskIds)
  else vec_w $\leftarrow \bot$
  $g \leftarrow$ *external-GroupInfo(γ', confTag)
  $\gamma' \leftarrow$ *set-interim-trans-hash(γ', confTag)
  γ.pendCom[c] $\leftarrow$ (γ', $\boldsymbol{p}$, upd, rem, add, psk)
  return (c, vec_w, g)

Input (Propose, rem-id$_t$)
  req $\gamma \neq \bot \wedge \texttt{id}_t \in \gamma.\tau$.roster()
  leafIdx$_t \leftarrow \gamma.\tau$.leafof($it_t$)
  $P \leftarrow$ ('rem', leafIdx$_t$); $p \leftarrow$ *frameProp(γ, P)
  return p

Input (Propose, extAdd-id-spk, epoch)
  req $\gamma = \bot$
  try ssk $\leftarrow$ **query** (get-ssk, spk) to $\mathcal{F}_{\mathcal{AS}}$
  (kp, sk, isk) $\leftarrow\$$ gen-kp(id, spk, ssk)
  $P \leftarrow$ ('add', kp)
  $p \leftarrow$ *frameExtProp(groupId, epoch, ssk, P)
  γ.pendAdd[epoch] $\leftarrow$ (kp, sk, isk)
  return p

Input (Propose, psk-epoch-use)
  req $\gamma \neq \bot$
  preSharedKeyId $\leftarrow$ *gen-pskId(epoch, use)
  $P \leftarrow$ ('psk', preSharedKeyId)
  $p \leftarrow$ *frameProp(γ, P)
  return p

Input (Process, c, $\boldsymbol{p}$)
  req $\gamma \neq \bot$
  if $\boldsymbol{p} = \bot$ **then**
    (id_c, (exi, rem)) $\leftarrow$ *process-ec(c, p)
    ordered_proposals $= exi \mathbin{+\!\!+} rem$
    return (id_c, ordered_proposals)
  (senderIdx, C, confTag, sig, membTag)
    $\leftarrow$ *unframeCommit(γ, c, sig)
  id$_c \leftarrow \gamma.\tau$.leaves[senderIdx].$ID$
  if senderIdx $= \gamma$.leafIdx() **then**
    parse (γ', $\boldsymbol{p}'$, upd, rem, add, psk) $\leftarrow \gamma$.pendCom[c]
    req $\boldsymbol{p} = \boldsymbol{p}'$
    $\gamma = \gamma'$
    return (id$_c$, $upd \mathbin{+\!\!+} rem \mathbin{+\!\!+} add \mathbin{+\!\!+} psk$)
  parse (propIDs, updatePath) $\leftarrow C$
  req $\forall i \in [|\boldsymbol{p}|] :$ Hash($\boldsymbol{p}[i]$) $=$ propIDs[i]
  $\gamma' \leftarrow$ *init-epoch(γ)
  try (γ', upd, rem, add, psk) $\leftarrow$ *apply-props(γ, γ', $\boldsymbol{p}$)
  req ($*$, id$_c$) $\notin rem \wedge$ (id$_c$, $*$) $\notin upd$
  if ($*$, 'rem'-id) $\in rem$ **then**
    req MAC.vrf(γ.membKey, c)
    $\gamma \leftarrow \bot$
  else
    if updatePath $\neq \epsilon$ **then**
      (γ', commitSec)
        $\leftarrow$ *apply-rekey(γ', senderIdx, updatePath)
    else
      req $\boldsymbol{p} \neq () \wedge upd = () \wedge rem = ()$
      commitSec $\leftarrow 0$
    $\gamma' \leftarrow$ *set-conf-trans-hash(γ, γ', senderIdx, C, sig)
    pskIds $=$ *get-pskIds($\boldsymbol{p}$)
    req psk-secret $\leftarrow$ *derive-psk-secret(pskIds)
    (γ', $*$) $\leftarrow$ *derive-keys(γ, γ',
      commitSec, psk-secret)
    req *vrf-conf-tag(γ', confKey, confTag)
    $\gamma \leftarrow$ *set-interim-trans-hash(γ', confTag)
  return (id$_c$, $upd \mathbin{+\!\!+} rem \mathbin{+\!\!+} add \mathbin{+\!\!+} psk$)

Fig. 20: The UC model ETK and ETK^{PSK} as run by party id. The group creator's identity id$_{\text{creator}}$ is encoded a part of the instance's session identifier. As elsewhere we use the color coding from Table 2. Note that code highlighted in orange (●) is only included in ETK^{PSK}, and not in ETK.

Protocol ETK and ETK^{PSK} - Part 2

```
Input (Join, vec_w)
  req γ = ⊥∨γ.pendAdd[epoch] ≠ ⊥
  parse (encGroupSecs, encGroupInfo, wel_type) ←
  vec_w
  γ.certSpks[∗], γ.pendUp[∗], γ.pendCom[∗] ← ⊥
  kbs ← query get-sk to F_ks
  kbs ++ γ.pendAdd
  joinerSec, pathSec ← ⊥
  for e ∈ encGroupSecs do
    parse (hash, cipher) ← e
    for (kp, sk, isk) ∈ kbs do
      if hash = Hash(kp) then
        v.sk ← sk
        req v.kp() = kp
        parse (joinerSec, pathSec, pskIds) ←
        PKE.Dec(isk, cipher)
  req joinerSec ≠ ⊥
  try psk-secret ← *derive-psk-secret(pskIds)
  s ← HKDF.Extract(joinerSec, psk-secret)
  welcome_secret = HKDF.Expand(s, 'welcome')
  welcome_nonce = HKDF.Expand(welcome_secret, 'nonce')
  welcome_key = HKDF.Expand(welcome_secret, 'key')
  groupInfo ← PKE.Dec(welcome_key, welcome_nonce,
              encGroupInfo)
  parse (groupInfoTBS, sig) ← groupInfo
  parse (groupCtxt, γ.τ, confTag, senderIdx) ←
              groupInfoTBS
  parse (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash)
              ← groupCtxt
  req Sig.Vrf(γ.τ.leaves[senderIdx].spk, sig, groupInfoTBS)
  if ∃epoch s.t. v.kp() = γ.pendAdd[epoch] then
    req γ.epoch = epoch
    γ.pendAdd[epoch]epoch⊥
  try γ ← *vrf-tree-state(γ)
  v ← γ.τ.leaves[γ.leafIdx()]
  try γ.ssk ← *fetch-ssk-if-nec(γ, v.spk)
  kbs ← query get-sk to F_ks
  if pathSec ≠ ϵ then
    v ← γ.τ.lca(γ.leafIdx(), senderIdx)
    while v ≠ ⊥ do
      nodeSec ← HKDF.Expand(pathSec, 'node')
      (sk, v.sk) ← PKE.Kg(nodeSec)
      req v.sk = sk
      pathSec ← HKDF.Expand(pathSec, 'path')
      v ← v.par
  try psk-secret ← *derive-psk-secret(pskIds)
  joiner_psk ← HKDF.Extract(joinerSec, psk-secret)
  (γ, confKey) ← *derive-epoch-keys(γ, joinerSec,
              psk-secret)
  req *vrf-conf-tag(γ, confKey, confTag)
  return (γ.τ.roster(), γ.τ.leaves[senderIdx].id)

Input Key
  req γ ≠ ⊥
  (k, γ.appSecret) ← (γ.appSecret, ⊥)
  return k
```

```
Input (ExtCommit, g, spk, e_psk, resync)
  if resync then
    req γ ≠ ⊥
  else
    req γ = ⊥
    γ.certSpks[∗], γ.pendUp[∗], γ.pendCom[∗] ← ⊥
  parse (extGroupInfoTBS, sig) ← g
  parse (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash,
        γ.τ.confTag, senderIdx, external-pub)
        ← extGroupInfoTBS
  req Sig.Vrf(y.τ.leaves[senderIdx].spk, sig,
        extGroupInfoTBS)
  try γ ← *vrf-tree-state(γ)
  (kem_output, context) ← setupBaseS(external-pub,
  "")
  γ.initSecret =context.export("MLS 1.0 external
        init secret", KDF.Nh)
  γ′ ← *init-epoch(γ)
  p = ()
  p ++← ('extInit', kem_output)
  if resync then
    req id ∈ γ.τ.roster()
    p ++← ('rem', γ.leafInd)
  for epoch ∈ e_psk do
    preSharedKeyId ←
    *gen-pskId(epoch, 'application')
    p ++← ('psk', preSharedKeyId)
  try (γ′, exi, rem, psk) ←
  *apply-ec-props(γ, γ′, p, id)
  try (γ′, commitSec, updatePath, pathSecs) ←
  *rekey-path-upon-join(γ′, id, spk)
  C ← (propIDs, updatePath)
  sig ← *signExtCommit(γ, C, ssk)
  γ′ ← *set-conf-trans-hash(γ, γ′, γ.leafIdx(), C, sig)
  pskIds = *get-pskIds(p)
  try psk-secret ← *derive-psk-secret(pskIds)
  (γ′, confKey, joinerSec)
    ← *derive-keys(γ, γ′, commitSec, psk-secret,
        kem_output)
  confTag ← *conf-tag(γ′, confKey)
  ec ← *frameExtCommit(γ, C, γ′.ssk, sig, confTag)
  g ← *external-GroupInfo(γ′, confTag)
  γ′ ← *set-interim-trans-hash(γ′, confTag)
  γ.pendCom[ec] ← (γ′, p, upd, rem, add, psk)
  return (ec, g)
```

Fig. 21: The UC model ETK and ETK^{PSK} as run by party id. The group creator's identity $\texttt{id}_{\text{creator}}$ is encoded as part of the instance's session identifier.

τ.clone()	Returns and (independent) copy of τ.
τ.public()	Returns a copy of τ for which all private labels (v.sk) are set to $\perp$.
τ.roster()	Returns the identities of all parties in the tree.
τ.leaves[leafIdx]	Returns the leaf with identifier leafIdx.
τ.leafof(id)	Returns the leaf identifier of the v for which v.id = id.
τ.allotLeaf()	Finds the leaf v with the lowest nodeIdx for which $\neg v$.inuse(), or adds a new leaf v using addLeaf. Returns leafIdx.
τ.directPath(leafIdx)	Returns the direct path, excluding the leaf, as an ordered list from the leaf to root.
τ.coPath(leafIdx)	Returns the co-path to τ.directPath(leafIdx) as an ordered list.
τ.lca(leafIdx$_1$, leafIdx$_2$)	Returns the least common ancestor of the two leafs.
τ.blankPath(leafIdx)	Calls v.blank() on all $v \in \tau$.directPath(leafIdx).
τ.mergeLeaves(leafIdx)	Sets v.unmergedLvs $\leftarrow \varnothing$ for all $v \in \tau$.directPath(leafIdx)
τ.unmergeLeaf(leafIdx)	Sets v.unmergedLvs +$\leftarrow$ leafIdx for all v returned by τ.directPath(leafIdx)
v.kp()	Returns (v.id, v.pk, v.spk, v.parentHash, v.sig) (undefined if $\neg v$.isleaf).
v.assignKp(kp)	Sets (v.id, v.pk, v.spk, v.parentHash, v.sig) from kp (only allowed if v.isleaf).
v.inuse()	Returns false iff all labels except parentHash are $\perp$.
v.blank()	Sets all labels except parentHash to $\perp$.
v.resolution()	Return $\begin{cases} (v) +\!\!+ v.\text{unmergedLvs} & \text{if } v.\text{inuse}() \\ v.\text{lchild.resolution}() & \\ \qquad +\!\!+ v.\text{lchild.resolution}() & \text{else if } \neg v.\text{isleaf} \\ () & \text{else.} \end{cases}$
v.resolvent(u)	For a descendant u of v, returns the (unique) node in v.resolution() which is an ancestor of u.

Table 5: Helper methods defined on the ratchet tree.

Protocol ETK and ETK^{PSK} : Setup Algorithms

Algorithm gen-sk
  (spk, ssk) ← Sig.Kg()
  return(spk, ssk)

Algorithm gen-kp(id, spk, ssk, parentHash $= \epsilon$, source $= \epsilon; r$)
  (leafNode, sk) ← gen-leaf-node(id, spk, ssk, parentHash, 'key_package', source; r)
  (init-key, isk) ← PKE.Kg()
  kpTBS ← (leafNode, initKey)
  sig ← Sig.Sign(ssk, kpTBS)
  kp ← (kpTBS, sig)
  return (kp, sk, isk)

Algorithm gen-leaf-node(id, spk, ssk, parentHash $= \epsilon$, ln_source, source $= \epsilon; r$)
  (pk, sk) ← PKE.Kg(r)
  req ln_source$in\{$'key_package', 'update', 'commit'$\}$
  if ln_source $\neq$ 'key_package' **then req** source $\neq \epsilon$
  leafNodeTBS ← (id, pk, spk, parentHash, ln_source, source)
  sig ← Sig.Sign(ssk, leafNodeTBS)
  leafNode ← (id, pk, spk, parentHash, ln_source, sig)
  return (leafNode, sk)

Fig. 22: The algorithms gen-sk and gen-kp, used by $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$, respectively.

γ.groupId	An identifier of the group.
γ.epoch	The current epoch number.
$\gamma.\tau$	The labeled left-balanced binary tree.
γ.treeHash	A hash of (the public part) of τ.
γ.confTransHash	The confirmed transcript hash.
γ.interimTransHash	The interim transcript hash for the next epoch.
γ.ssk	The current signing key.
γ.certSpks[$*$]	A mapping associating the set of validated signature verification keys to each party $\mathtt{id}'$.
γ.pendUp[$*$]	A mapping associating the secret keys for each pending update proposal issued by $\mathtt{id}$.
γ.pendCom[$*$]	A mapping associating the new group state for each pending commit issued by $\mathtt{id}$.
γ.appSecret	The current epoch's ECGKA key.
γ.membKey	The key used to MAC packages.
γ.initSecret	The next epoch's init secret.

Table 6: The protocol state.

γ.leafIdx()	Returns $\gamma.\tau$.leafof($\mathtt{id}$).leafIdx.
γ.groupCtxt()	Returns (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash).

Table 7: Helper methods on the protocol state.

Protocol ETK and ETK^{PSK} : Commit. Process, and Join Helpers - Part 1

```
helper *init-epoch(γ)
  γ' ← γ.clone()
  γ'.epoch ← γ'.epoch + 1
  γ'.pendUp[∗], γ'.pendCom[∗] ← ⊥
  return γ'

helper *rekey-path(γ', id, spk)
  leafIdx = γ'.leafIdx()
  (updatePathNodes, pathSecs, commitSec,
              leafNodeSec)              ←
  *derive-rekey-values(γ', id, spk, leafIdx)
  try ssk ← *fetch-ssk-if-nec(γ', spk)
  v ← γ'.τ.leaves[γ'.leafIdx()]
  r ← leafNodeSec
  (ln, sk) ← gen-leaf-node(id, spk, ssk, v.parentHash,
              ('commit', γ'.groupId, γ'.τ.leafIdx()); r)
  v.assignKp(ln)
  v.sk ← sk
  γ' ← *set-tree-hash(γ')
  updatePath ← (ln, updatePathNodes)
  return (γ', commitSec, updatePath, pathSecs)

helper *derive-rekey-values(γ', id, spk, leafIdx)
  directPath ← γ'.τ.directPath(γ'.leafIdx())
  coPath ← γ'.τ.coPath(γ'.leafIdx())
  updatePathNodes ← ()
  pathSecs[∗] ← ⊥
  leafSec ←$ {0,1}^κ
  leafNodeSec                ←
  HKDF.Expand(leafSec, 'node')
  pathSec ← HKDF.Expand(leafSec, 'path')
  for (v, c) ∈ zip(directPath, coPath) do
    pathSecs[v] ← pathSec
    nodeSec                  ←
    HKDF.Expand(pathSec, 'node')
    (v.pk, v.sk) ← PKE.Kg(nodeSec)
    encPathSecs ← ()
    for t ← c.resolution() do
      encPathSecs ∥← PKE.Enc(t.pk, pathSec)
    updatePathNodes ∥← (v.pk, encPathSecs)
    pathSec ← HKDF.Expand(pathSec, 'path')
  commitSec ← pathSec
  γ'.τ.mergeLeaves(γ'.leafIdx())
  γ' ← *set-parent-hash(γ', γ'.leafIdx())
  return (updatePathNodes, pathSecs, commitSec,
              leafNodeSec)

helper *rekey-path-upon-join(γ', id, spk)
  try ssk ← *fetch-ssk-if-nec(γ', spk)
  v ← γ'.τ.leaves[γ'.leafIdx()]
  r ← leafNodeSec
  (ln, sk) ← gen-kp(id, spk, ssk, ϵ, 'commit', (γ'.groupId,
              γ'.τ.leafIdx()); r)
  γ'.τ.leaves[γ'.leafIdx()].assignKp(ln)
  γ'.τ.leaves[γ'.leafIdx()].sk ← sk
  (updatePathNodes, pathSecs, commitSec, leafNodeSec) ←
              *derive-rekey-values(γ', id, spk, leafIdx)
  γ' ← *set-tree-hash(γ')
  updatePath ← (ln, updatePathNodes)
  return (γ', commitSec, updatePath, pathSecs)

helper *apply-rekey(γ', senderIdx, updatePath)
  parse (ln, updatePathNodes) ← updatePath
  (γ', commitSec) ← *derive-apply-rekey-values(γ',
              updatePathNodes, senderIdx)
  commitSec ← pathSec
  v ← γ'.τ.leaves[senderIdx]
  try γ' ← *validate-ln(γ', ln, v.id, 'commit', v.parentHash)
  v.assignKp(kp)
  γ' ← *set-tree-hash(γ')
  return (γ', commitSec)

helper *derive-apply-rekey-values(γ', updatePathNodes,
              senderIdx)
  directPath ← γ'.τ.directPath(senderIdx)
  coPath ← γ'.τ.coPath(senderIdx)
  lca ← γ'.τ.lca(γ'.leafIdx(), senderIdx)
  for (v, c, updatePathNode) ∈ zip(directPath,
                    coPath, updatePathNodes) do
    parse (v.pk, encPathSecs) ← updatePathNode
    if v = lca then
      r ← c.resolvent(γ'.τ.leaves[γ'.leafIdx()])
      i ← c.resolution().indexof r
      pathSec ← PKE.Dec(r.sk, encPathSecs[i])
    if pathSec ≠ ⊥ then
      nodeSec ← HKDF.Expand(pathSec, 'node')
      (pk, v.sk) ← PKE.Kg(nodeSec)
      req v.pk = pk
      pathSec ← HKDF.Expand(pathSec, 'path')
  commitSec ← pathSec
  γ'.τ.mergeLeaves(senderIdx)
  γ' ← *set-parent-hash(γ', senderIdx)
  return (γ', commitSec)

helper *apply-ec-rekey(γ', id_c, updatePath)
  parse (ln, updatePathNodes) ← updatePath
  try leafIdx ← γ.τ.allotLeaf()
  try γ' ← *validate-ln(γ', ln, id_c, 'commit', ϵ)
  γ'.τ.leaves[senderIdx].assignKp(ln)
  (γ', commitSec) ← *derive-apply-rekey-values(γ',
              updatePathNodes, senderIdx)
  γ' ← *set-tree-hash(γ')
  return (γ', commitSec)

helper *external-GroupInfo(γ', confTag)
  externalGroupInfoTBS                    ←
  (γ'.groupId, γ'.epoch, γ'.treeHash,
              γ'.confTransHash, γ'.τ.public(),
              confTag, γ'.leafIdx(), γ'.external_pub)
  sig ← Sig.Sign(γ'.ssk, externalGroupInfoTBS)
  g ← (externalGroupInfoTBS, sig)
  return (γ', g)
```

Fig. 23: Helper methods related to the commit and welcome messages.

Protocol ETK and ETK^{PSK} : Commit. Process, and Join Helpers - Part 2

```
helper *truncate-tree(γ')
  v ← γ'.τ.leaves[|γ'.τ.leaves| − 1]
  if ¬v.inuse()
      ∧ (¬v.par.inuse() ∨ v.par = γ'.τ.root) then
      γ'.τ ← τ.pruneRightmost()
      γ' ← *truncate-tree(γ')
  return γ'
helper *apply-ec-props(γ, γ', p, id_s)
  exi, rem, psk ← ()
  pskIds ← ∅
  for (type, val) ∈ p do
      if type = 'rem' then
          id_t ← γ.τ.leaves[val].id
          req γ'.τ.leaves[val] ≠ ⊥
          req γ'.τ.leaves[val].inuse() ∧ id_s = id_t
          γ'.τ.leaves[val].blank()
          γ'.τ.leaves[val].blankPath(val)
          γ' ← *truncate-tree(γ')
          rem +← (id_s, 'rem'-id_t)
      else if type = 'psk' then
          parse (∗, psk_data, ∗) ← val
          req γ.psk_store.last((psk_data.groupId,
                  psk_data.epoch)) ≠ ⊥
          usage = psk_data.usage
          req usage = 'application'
          psk +← (id_s, 'psk'-epoch-usage)
      else if type = 'exi' then
          exi +← (id_s, val)
      else return ⊥
  return (γ', exi, rem, psk)
helper *gen-pskId(epoch, usage)
  psk_data ← (usage, γ.groupId, epoch)
  nonce ←$ {0,1}^κ
  return ('resumption', psk_data, nonce)
helper *get-pskIds(p)
  pskIds ← ()
  for p ∈ p do
      (sender, P) ← *unframeProp(γ, p)
      parse (∗, id) ← P
      pskIds ← id
  return pskIds
helper *derive-psk-secret(pskIds)
  pskSec[0] = 0
  for (i, pskId) ∈ pskIds do
      (∗, pskData, ∗) ← pskId
      psk_[i] = γ.psk_store.last((pskData.groupId,
                  pskData.epoch))
      req psk_[i] ≠ ⊥
      psk_extracted[i] = HKDF.Extract(0, psk_[i])
      PSKLabel = (pskId, i, len(pskIds))
      psk_input[i] =
      HKDF.Expand(psk_extracted[i], 'derivedpsk')
      pskSec[i + 1] =
      HKDF.Extract(psk_input[i], pskSec[i])
  return pskSec[−1]
helper *apply-props(γ, γ', p)
  upd, rem, add, psk ← ()
  pskIds ← ∅
  for p ∈ p do
      if p.sender = 'member' then
          try (senderIdx, P) ← *unframeProp(γ, p)
          id_s ← γ.τ.leaves[senderIdx].id
      else if p.sender = 'new_member_proposal' then
          try (id_s, P) ← *unframeExtProp(γ, p)
      parse (type, val) ← P
      if type = 'upd' then
          req (id_s, ∗) ∉ upd ∧ rem = () ∧ add = ()
          try γ' ← *validate-ln(γ', val, id_s, ϵ)
          γ'.τ.leaves[senderIdx].assignKp(val)
          γ'.τ.blankPath(senderIdx)
          if senderIdx = γ.leafIdx() then
              parse (ssk, sk) ← γ.pendUp[p]
              γ'.τ.leaves[senderIdx].sk ← sk
              γ'.ssk ← ssk
          spk ← γ'.τ.leaves[senderIdx].spk
          upd +← (id_s, 'upd'-spk)
      else if type = 'rem' then
          id_t ← γ.τ.leaves[val].id
          req γ'.τ.leaves[val] ≠ ⊥
          req γ'.τ.leaves[val].inuse() ∧ (id_t, ∗) ∉ upd ∧ add = ()
          γ'.τ.leaves[val].blank()
          γ'.τ.leaves[val].blankPath(val)
          γ' ← *truncate-tree(γ')
          rem +← (id_s, 'rem'-id_t)
      else if type = 'add' then
          parse (id_t, ∗, spk, ∗, ∗) ← val
          req id_t ∉ γ'.τ.roster()
          try γ' ← *validate-kp(γ', val, id_t, 'key_package'ϵ)
          try γ'.τ.allotLeaf()
          γ'.τ.leaves[leafIdx()].assignKp(val)
          γ'.τ.unmergeLeaf(leafIdx())
          add +← (id_s, 'add'-id_t-spk)
      else if type = 'psk' then
          req val ∉ PSKIds
          parse(∗, psk_data, ∗) ← val
          req γ.psk_store.last((psk_data.groupID,
                  psk_data.epoch)) ≠ ⊥
          usage = psk_data.usage
          psk +← (id_s, 'psk'-epoch-usage)
          PSKIds ++= val
      else return ⊥
  return (γ', upd, rem, add, psk)
```

Fig. 24: The helper methods related to creating and processing the commit and welcome messages.

Protocol ETK and ETK^{PSK} : Commit. Process, and Join Helpers - Part 3

```
helper *welcome-msg(γ, γ', add, joinerSec,
pathSecs, confTag, pskIds, wel_type)
  groupCtxt ← (γ'.groupId, γ'.epoch, γ'.treeHash,
      γ'.confTransHash)
  groupInfoTBS                                  ←
  (groupCtxt, γ'.τ.public(), confTag, γ'.leafIdx())
  sig ← Sig.Sign(γ'.ssk, groupInfoTBS)
  groupInfo ← (groupInfoTBS, sig)
  try psk-secret                ←
   *derive-psk-secret(pskIds)
  s ← HKDF.Extract(joinerSec, psk-secret)
  welcome_secret = HKDF.Expand(s,
  'welcome')
  welcome_nonce                          =
  HKDF.Expand(welcome_secret, 'nonce')
  welcome_key                            =
  HKDF.Expand(welcome_secret, 'key')
  encGroupInfo ← PKE.Enc(welcome_key,
  welcome_nonce, groupInfo)
  encGroupSecs ← ()
  for (∗, 'add'-id_t-spk_t, ipk-id_t) ∈ add do
      leafIdx_t ← γ'.τ.leafof(id_t)
      v_t ← γ'.τ.leaves[leafIdx_t]
      lca ← γ'.τ.lca(γ'.leafIdx(), leafIdx_t)
      encGroupSec  ←  PKE.Enc(ipk-id_t,
      (joinerSec, pathSecs[lca], pskIds))
      encGroupSecs #← (Hash(v_t.kp), encGroupSec)
  vec_w ← *parse-wel(encGroupSecs,
  encGroupInfo, wel_type)
  return (γ', vec_w)

helper *vrf-tree-state(γ')
  req γ'.treeHash = *tree-hash(γ'.τ.root)
  for v ∈ γ'.τ.nodes : v.inuse() ∧ ¬v.isleaf do
      lchild ← v.lchild
      rchild ← *origRChild(v.rchild)
      phr ← *parent-hash(v, v.rchild)
      phl ← *parent-hash(v, v.lchild)
      req (lchild.inuse ∧ lchild.parentHash = phr)
            ∨ (rchild.inuse ∧ rchild.parentHash  =
      phl)
  mem ← ∅
  for v ∈ γ'.τ.nodes : v.inuse() ∧ v.isleaf do
      req v.id ∉ mem
      mem +← v.id
      try γ' ← *validate-kp(γ', v.kp(), v.id, v.parentHash)
  return γ'

helper *origRChild(v)
  if v.inuse ∨ v.isleaf then return  v
  else return  *origRChild(v.lchild)

helper *parse-wel(encGroupSecs, encGroupInfo, wel_type)
  if wel_type = all then
      vec_w +← (encGroupSecs, encGroupInfo)
  else if wel_type = one then
      for all encGroupSec ∈ encGroupSecs do
          vec_w +← (encGroupSec, encGroupInfo)
  else if wel_type = batch then
      for all encGroupSec stated as batched
      encGroupSecs do
          vec_w +← (encGroupSec, encGroupInfo)
  return  vec_w

helper *process-ec(p)
  (id_c, C, confTag, sig, membTag) ←
              *unframeExtCommit(γ, ec, sig)
  (p, updatePath) ← C
  if id_c = id then
      parse (γ', p, ext − init, rem, psk) ← γ.pendCom[c]
      req p = p
      return(id_c, ext − init ++ rem ++ psk)
  γ' ← *init-epoch(γ)
  try (γ', ext − init, rem) ← *apply-ec-props(γ, γ', p)
  req len(rem) = 1 ∧ len(ext − init) = 1
  (∗, kem_output) ← parse ext − init
  context = SetupBaseR(kem_output, γ.external_priv, "")
  initSecret = context.export("MLS 1.0 external init se-
  cret", KDF.Nh)
  req updatePath ≠ ⊥
  (γ', senderIdx, commitSec)            ←
   *apply-ec-rekey(γ', id_c, updatePath)
  γ' ← *set-conf-trans-hash(γ, γ', senderIdx, C, sig)
  pskIds = *get-pskIds(p)
  req psk-secret ← *derive-psk-secret(pskIds)
  (γ', ∗) ← *derive-keys(γ, confKey, γ', commitSec,
              psk-secret)
  req *vrf-conf-tag(γ', confKey, confTag)
  γ ← *set-interim-trans-hash(γ', confTag)
  return (id_c, upd ++ rem ++ add ++ psk)
```

Fig. 25: Helper methods related to commit and welcome messages.

Protocol ETK and ETK^{PSK} : Confirmation-Tag

```
helper *conf-tag(γ', confKey)
  return MAC.tag(confKey, γ'.confTransHash)

helper *vrf-conf-tag(γ', confKey, confTag)
  return MAC.vrf(confKey, confTag, γ'.confTransHash)
```

Protocol ETK and ETK^{PSK} : Tree-Hash

```
helper *set-parent-hash(γ', leafIdx)
  path ← γ'.τ.directPath(leafIdx)
  path ← path.reverse()
  path +← γ'.τ.leaves[leafIdx]
  for v ∈ path do
      if v.isroot then
          v.parentHash ← ϵ
      else
          v.parentHash ← *parent-hash(v.par, v.sibling)
  return γ'

helper *parent-hash(v, u)
  for all w ∈ u.unmergedLvs do
      w.blank()
      for all c ∈ u.roster do
          par.unmergedLvs ← par.unmergedLvs \ {w}
  osth ← *tree-hash(u)
  return Hash(v.pk, v.parentHash, osth)

helper *set-tree-hash(γ')
  γ'.treeHash ← *tree-hash(γ'.τ.root)
  return γ'

helper *tree-hash(v)
  if v.isleaf then
      return Hash(v.nodeIdx, v.kp())
  else
      leftHash ← *tree-hash(v.lchild)
      rightHash ← *tree-hash(v.rchild)
      return Hash(v.parentHash, leftHash, rightHash)
```

Protocol ETK and ETK^{PSK} : Key-Schedule

```
helper *derive-keys(γ, γ', commitSec, psk-secret,
kem_output)
  if kem_output ≠ ⊥ then
      context = SetupBaseR(kem_output,
              γ.external_priv, "")
      initSecret = context.export("MLS
              1.0externalinitsecret", KDF.Nh)
  else
      initSecret = γ.initSecret
  s ← HKDF.Extract(γ.initSecret, commitSec)
  joinerSec ← HKDF.Expand(s, γ'.groupCtxt())
  joiner_psk ← HKDF.Extract(joinerSec, psk-secret)
  (γ', confKey) ← *derive-epoch-keys(γ', joiner_psk)
  return (γ', confKey, joinerSec)

helper *derive-epoch-keys(γ', joinerSec)
  s ← HKDF.Expand(γ.joinerSec, 'member')
  memberSec ← HKDF.Extract(s, 0)
  e ← HKDF.Expand(memberSec, 'epoch')
  epochSec ← HKDF.Extract(e, γ'.groupCtxt())
  confKey ← HKDF.Expand(epochSec, 'confirm')
  γ'.appSecret ← HKDF.Expand(epochSec, 'app')
  γ'.membKey ← HKDF.Expand(epochSec, 'membership')
  γ'.initSecret ← HKDF.Expand(epochSec, 'init')
  extSecret ← HKDF.Expand(epochSec, 'external')
  γ'.external_priv, γ'.external_pub ←
  PKE.Kg(extSecret)
  resumptionPsk ←
          HKDF.Expand(epochSec, 'resumption')
  γ'.resumptionPsk ← resumptionPsk
  γ'.psk_store((γ'.groupId, γ'.epoch)) ++
  resumptionPsk
  return (γ', confKey)
```

Protocol ETK and ETK^{PSK} : Setup Interaction

```
helper *fetch-ssk-if-nec(γ, spk)
  if γ.τ.leaves[γ.leafIdx()].spk ≠ spk then
      ssk ← query (get-ssk, spk) to F_AS
  else
      ssk ← γ.ssk
  return ssk

helper *validate-kp(γ, kp, id, kp_source, parentHash)
  parse (sig, ln, init-key) ← kp
  req Sig.Vrf(spk, sig, (ln, init-key))
  γ ← *validate-ln(γ, ln, id, kp_source, parentHash)
  return γ

helper *validate-ln(γ, ln, id, ln_source, parentHash)
  parse (id, pk, spk, parentHash', ln_source', source,
          sig) ← ln
  req ln_source = ln_source'
  if spk ∉ γ.certSpks[id] then
      succ ← query (verify-cert, id', spk) to F_AS
      req succ
      γ.certSpks[id] +← spk
  req Sig.Vrf(spk, sig, (id, pk, spk, parentHash))
  return γ
```

Fig. 26: Various helper methods for the protocol ETK and ETK^{PSK}.

Protocol ETK and ETK^{PSK} : Message-Framing

helper *signCommit(γ, C)
  tbs ← (γ.groupCtxt(), γ.groupId, γ.epoch, γ.leafIdx(), 'commit', C)
  sig ← Sig.Sign(γ.ssk, tbs)
  return sig

helper *frameCommit(γ, C, confTag, sig, membTag)
  return (γ.groupId, γ.epoch, γ.leafIdx(), 'commit', C, confTag, sig, membTag)

helper *unframeCommit(γ, c)
  parse (groupId, epoch, senderIdx, contentType, C, confTag, sig, membTag) ← c
  req contentType = 'commit' ∧ groupId = γ.groupId
  reqepoch = γ.epoch
  tbs ← (γ.groupCtxt(), groupId, epoch, senderIdx, 'commit', C)
  req $\gamma.\tau$.leaves[senderIdx] ≠ ⊥
    ∧ $\gamma.\tau$.leaves[senderIdx].inuse()
    ∧ Sig.Vrf($\gamma.\tau$.leaves[senderIdx].spk, sig, tbs)
  return (senderIdx, C, confTag, sig, membTag)

helper *signExtCommit(γ, C, ssk)
  framedContent ← (γ.groupId, γ.epoch, 'new_member_commit','commit', C)
  sig ← Sig.Sign(γ.ssk, framedContent, γ.groupCtxt())
  return sig

helper *frameExtCommit(γ, C, confTag, sig, ssk)
  framedContent ← (γ.groupId, γ.epoch, 'new_member_commit','commit', C)
  framedContentAuth ← (sig, confTag)
  return (framedContent, framedContentAuth)

helper *unframeExtCommit(γ, ec, sig)
  parse(framedContent, framedContentAuth) ← ec
  parse (γ.groupId, γ.epoch, type, contentType, C) ← framedContent
  req contentType = 'commit' ∧ type = 'new_member_commit'
  reqepoch = γ.epoch ∧ groupId = γ.groupId
  parse (sig, confTag) ← framedContentAuth
  req Sig.Vrf(kp.spk, sig, framedContent)
  return (kp.id, Hash(kp), confTag, sig)

helper *frameProp(γ, P)
  tbs ← (γ.groupCtxt(), γ.groupId, γ.epoch, γ.leafIdx(), 'proposal', P)
  sig ← Sig.Sign(γ.ssk, tbs)
  tbm ← (tbs, sig)
  membTag ← MAC.tag(γ.membKey, tbm)
  return (γ.groupId, , γ.epoch, γ.leafIdx(), 'proposal', P, sig, membTag)

helper *unframeProp(γ, p)
  parse (groupId, epoch, senderIdx, contentType, P, sig, membTag) ← p
  req contentType = 'proposal' ∧ groupId = γ.groupId
    ∧ epoch = γ.epoch
  tbs ← (γ.groupCtxt(), groupId, epoch, senderIdx, 'proposal', P)
  tbm ← (tbs, sig)
  req $\gamma.\tau$.leaves[senderIdx] ≠ ⊥
    ∧ $\gamma.\tau$.leaves[senderIdx].inuse()
    ∧ Sig.Vrf($\gamma.\tau$.leaves[senderIdx].spk, sig, tbs)
    ∧ MAC.vrf(γ.membKey, membTag, tbm)
  return (senderIdx, P)

helper *frameExtProp(groupId, epoch, ssk, P)
  framedContent ← (groupId, epoch, 'new_member_proposal', 'proposal', P)
  sig ← Sig.Sign(ssk, framedContent)
  return (framedContent, sig)

helper *unframeExtProp(γ, ep)
  parse (framedContent, sig) ← ep
  parse (groupId, epoch, sender, contentType, P) ← framedContent
  req contentType = 'proposal' ∧ sender = 'new_member_proposal' ∧ γ.groupId = groupId
    ∧ epoch = γ.epoch
  parse(proposal, kp) ← P
  req proposal = 'add'
  req Sig.Vrf(kp.spk, sig, framedContent)
    ∧ kp.id ∉ roster
  return (kp.id, P)

Protocol ETK and ETK^{PSK} : Transcript-Hash

helper *set-conf-trans-hash(γ, γ', senderIdx, C, sig)
  commitContent ← (γ.groupId, γ.epoch, senderIdx, 'commit', C, sig)
  γ'.confTransHash ← Hash(γ.interimTransHash, commitContent)
  return γ'

helper *set-interim-trans-hash(γ', confTag)
  γ'.interimTransHash ← Hash(γ'.confTransHash, confTag)
  return γ'

Fig. 27: The helper methods related to message framing and transcript hashes.

F Proof Sketch of Theorem 1

Proof (Sketch). Our proof follows [4] and defines four consecutive hybrid games from *H1* to *H4*. We prove that the real world (i.e., *H1*) and the ideal world (i.e., *H4*) are indistinguishable for all environments $\mathcal{Z}$ via intermediate steps.

In hybrid *H1*, a dummy functionality $\mathcal{F}_{dummy}$ forwards all in- and outputs through the simulator $\mathcal{S}$, which executes ETK. Hence, $\mathcal{F}_{dummy}$ encodes no security guarantees and *H1* is identical to the real world.

In *H2*, $\mathcal{F}_{dummy}$ is replaced by a functionality consisting of $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$, $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$ and a modified version of $\mathcal{F}_{\text{ECGKA}}$, where $\textbf{safe}(c) = \texttt{false}$ and $\textbf{inj-allowed}(c, \mathsf{spk}) = \texttt{true}$ for all commits c and leaf nodes' spk corresponding some identity id. The functionality interacts with a trivial simulator that chooses all application secrets according to the protocol. For every input operation, we show that the outputs in ETK (i.e., $\mathcal{F}_{dummy}$) and $\mathcal{F}_{\text{ECGKA}}$ of *H2* is the same. This particularly includes that if *H2* halts due to an **assert** statement, so would ETK. The indistinguishability between *H1* and *H2* proves consistency and correctness of ETK.

The hybrid *H3* is identical to *H2* except that the application secrets in safe epochs are random, i.e., the original **safe** is restored. Moreover, we also create a sequence of sub-hybrid games Hs_i, where the simulator $\mathcal{S}$ sets the secrets for the first i epochs even if they are safe, and all secrets of the epoch after that are set according to the modified version of $\mathcal{F}_{\text{ECGKA}}$ of *H2*. It is easy to observe that Hs_0 is identical to *H3* and the final sub-hybrid Hs_n for some polynomial n is identical to *H2*. We prove that every two adjacent sub-hybrid games Hs_{i-1} and Hs_i are indistinguishable, and further *H2* and *H3*. The indistinguishability between *H2* and *H3* proves the confidentiality of ETK.

The hybrid *H4* is identical to *H3* except that the messages (proposals and commits) forgery on behalf of uncorrupted parties are excluded, i.e. the original **inj-allowed** is restored. Now, *H4* is identical to the ideal world. We prove that the environment $\mathcal{Z}$ cannot forge any Sig signature or MAC on behalf of any uncorrupted party in *H3* whenever the evaluation of corresponding **inj-allowed** is true except for negligible probability. The indistinguishability between *H3* and *H4* will prove the authenticity of ETK.

G Proof Sketch of Theorem 2

Proof (Sketch). This proof is similar to the proof of Theorem 1, except for the analysis of the execution related to the resumption `psk`. The $\mathsf{ETK}^{\mathrm{PSK}}$ protocol additionally allows to inject a resumption `psk` from a past epoch to the key derivation schedule at the current epoch by a PSK proposal $p \leftarrow (\mathsf{Propose}, \mathsf{psk\text{-}epoch\text{-}use})$. The `psk`s are stored locally and can be exposed later. Their status `stat` tracks the adversary's application key knowledge. If a member `id` is exposed, the `psk` in its possession are determined by `has_psk` and `stat` of all its `psk` is set to `bad`. $\mathcal{F}_{\mathrm{ECGKA}}$ uses $\texttt{Prop}[p].\texttt{c-psk}$ to track the `psk`'s origin commit. Future processors use this to ensure that all members process commits with consistent `psk`s. We additionally show that this approach maintains consistency in $\mathcal{F}_{\mathrm{ECGKA}}$.

Its possible that the processing client does not know the injected `psk`. $\mathcal{F}_{\mathrm{ECGKA}}$ performs the necessary checks for possession for confidentiality to be maintained. Even if the latest epoch state and the secrets included in a commit are corrupted, the environment still cannot distinguish the epoch secret from random as long as the injected PSKs are not leaked. This is because to derive new epoch secret, the PSK values are injected into hashes, which are modeled as random oracles.

H Comparison with the History Graph in [4]

Our history graph definition differs in three main ways from [4]. First, our graph incorporates external proposals that can be applied to all commit nodes with the same epoch. To this end, the pars variable stores a list of parent commit nodes rather than unique one, and every commit node additionally stores a new epoch variable.

Second, our history graph incorporates the external commit operation. To distinguish these node from the regular commits, our history graph includes a `sender_type` variable. While regular proposals can be generated and sent independently before they are committed, external commits and the included proposals are closely connected and sent together. To capture this, our external commit nodes have an additional `ExtProps` variable for all included proposals.

Third, our history graph can also be used to incorporate the resumption PSK proposal operation, which we will introduce in Section 5. To this end, our history graph includes a new PSK proposal node that has two additional variables: c_{psk} and `psk`, which respectively specify the commit whose PSK this proposal refers to, and the security of this PSK. Moreover, every commit node has an additional `has_psk` variable that stores the list of group members who have processed this commit and derived its resumption PSK.

I Comparison between $\mathcal{F}_{\text{CGKA}}$ [4] and our $\mathcal{F}_{\text{ECGKA}}$

Compared to $\mathcal{F}_{\text{CGKA}}$ in [4], the main difference is that our $\mathcal{F}_{\text{ECGKA}}$ captures "External Joins" by including the external operations described in [12, Section 3.3]. External proposals allow a non-group member to apply for joining the group. Recall that the external proposal node will be appended to all commit nodes of the same epoch. To capture this, our $\mathcal{F}_{\text{ECGKA}}$ includes the epoch information in the commit and external proposal nodes. The external commit allows non-members to directly join an "open" group but also allows group members to resync with the group. To capture this, every commit (regular and external) must be generated together with groupInfo g that can be used by outsiders to generate external commits. Moreover, $\mathcal{F}_{\text{ECGKA}}$ offers different security guarantees in three aspects.

Consistency. The consistency predicate ***cons-invariant** in our $\mathcal{F}_{\text{ECGKA}}$ additionally captures the consistency of external proposal nodes, i.e., whether every external proposal has the same epoch as all its parent commit nodes.

Confidentiality. The confidentiality predicate **safe** in our $\mathcal{F}_{\text{ECGKA}}$ has the following three core differences: First, $\mathcal{F}_{\text{ECGKA}}$ encodes the confidentiality requirement for external self-add proposals, similarly to regular add proposals. Second, $\mathcal{F}_{\text{ECGKA}}$ encodes that a rekey path is not issued if the proposals included in a commit are either empty or add-only, see case a) in ***secrets-replaced**. This captures the modifications in MLS since Draft 10 [11]. Third, $\mathcal{F}_{\text{ECGKA}}$ encodes the confidentiality requirement for external commits, identical to normal commits.

Authenticity. The authentication predicate ***auth-invariant** in our $\mathcal{F}_{\text{ECGKA}}$ has two core differences: First, we redefine the **inj-allowed** helper function. While **inj-allowed** in $\mathcal{F}_{\text{CGKA}}$ in [4] checks whether the adversary can inject a message on behalf of an `id` in the group, our **inj-allowed** checks whether an injection can be verified on `id`'s public key `spk`, as the sender of an external proposal or commit might not be in a group. Second, our $\mathcal{F}_{\text{ECGKA}}$ encodes the authenticity requirement for the external proposals and commit operations, respectively in cases c) and d).

J Proof of Theorem 1

Theorem 1. Assume that PKE is IND-CCA secure and that Sig is SUF-CMA secure. The ETK protocol securely realizes $(\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}, \mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}, \mathcal{F}_{\text{ECGKA}})$ in the $(\mathcal{F}_{\mathcal{AS}}, \mathcal{F}_{\mathcal{KS}}, \mathcal{G}_{\text{RO}})$-hybrid model, where $\mathcal{F}_{\text{ECGKA}}$ uses the predicates **safe** and **inj-allowed** from Figure 7 and calls to HKDF.Expand, HKDF.Extract, and MAC functions are replaced by calls to the global random oracle $\mathcal{G}_{\text{RO}}$.

Proof. At a high level, our proof has a similar structure to that of [4]. First, we define four consecutive hybrids as follows:

1. **Hybrid 1 (*H1*):** In this first hybrid, a dummy functionality $\mathcal{F}_{dummy}$ forwards all in- and outputs through the simulator $\mathcal{S}_1$. $\mathcal{S}_1$ executes ETK. $\mathcal{F}_{dummy}$ hence encodes no security guarantees.
2. **Hybrid 2 (*H2*):** In the second hybrid, the dummy functionality $\mathcal{F}_{dummy}$ is replaced by a functionality consisting of $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$, $\mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}$ and a modified version of $\mathcal{F}_{\text{ECGKA}}$, where the underlying **safe**$(c) =$ `false` and **inj-allowed**$(c, \mathsf{spk}) =$ `true` for all commits c and leaf nodes' spk corresponding some identity `id`. The functionality interacts with a trivial simulator that chooses all application secrets according to the protocol. The indistinguishability between *H1* and *H2* will prove the consistency of ETK.
3. **Hybrid 3 (*H3*):** The third hybrid is identical to *H2* except that the application secrets in safe epochs are random, i.e. the original **safe** is restored. The indistinguishability between *H2* and *H3* will prove the confidentiality of ETK.
4. **Hybrid 4 (*H4*):** The fourth hybrid is identical to *H3* except that forgery of messages (for proposals and commits) on behalf of uncorrupted members is excluded, i.e., the original **inj-allowed** predicate is restored, such that functionality exactly matches $\mathcal{F}_{\text{ECGKA}}$. The indistinguishability between *H3* and *H4* will prove the authenticity of ETK.

We show that for functionality interacting with corresponding simulator, no environment can distinguish this ideal world interaction with the real world interaction, i.e., the instance of ETK and $\mathcal{A}$. To this end, we respectively prove the indistinguishability between every two adjacent hybrids by Lemma 1 (in Section J.1), Lemma 2 (in Section J.2), and Lemma 3 (in Section J.3). Note that the first hybrid *H1* and the last hybrid *H4* are respectively equivalent to the real world and the ideal world. Hence, the protocol ETK securely realizes the functionality $\mathcal{F}_{\text{ECGKA}}$. This concludes the proof.

J.1 Consistency and Correctness

Intuition. We prove that *H1* and *H2* are indistinguishable by proving for every input operation that the outputs in ETK and $\mathcal{F}_{\text{ECGKA}}$ of *H2* are the same. We especially show that if *H2* halts due to an **assert** statement, so would ETK. The **assert** statements in $\mathcal{F}_{\text{ECGKA}}$ of *H2* capture that if one party has processed a

proposal, (external) commit, or welcome, all parties that process it in the future end up in the same shared group state. Thus, the indistinguishability between *H1* and *H2* indicates that $\mathcal{F}_{\text{ECGKA}}$ guarantees consistency.

Lemma 1. *Hybrids H1 and H2 are indistinguishable, that is,* ETK *guarantees consistency.*

Proof. We first define the simulator $\mathcal{S}_2$ and examine that (1) $\mathcal{S}_2$ will not trigger any **assert** statement in the $\mathcal{F}_{\text{ECGKA}}$ in hybrid *H2*, and (2) the outputs of $\mathcal{F}_{dummy}$ (i.e., ETK) and the $\mathcal{F}_{\text{ECGKA}}$ in hybrid *H2* are same. Recall that the simulator can monitor the whole history graph. We examine every possible commit node in the history graph. Recall also that every commit node in the history graph includes the application secrets and a confTransHash value, which hashes the history of commits until c. Since we model the hash function as random oracle, the same confTransHash values in different commit nodes must indicate that they have the same history of commits leading to them. Thus, every commit node in the history graph must be unique.

A repeating argument in the proof is that a value in ETK matches the attribute that $\mathcal{F}_{\text{ECGKA}}$ stores for consistency. The proof will use the phrasing *the attributes in* ETK *and* $\mathcal{F}_{\text{ECGKA}}$ *correspond directly* for this relationship. The argument is the same in all cases: The attribute value is either set by $\mathcal{F}_{\text{ECGKA}}$ and then passed to the simulator, which returns the identifier encoding exactly these attributes, or the value is read directly out of an (adversarially injected) identifier by the simulator and passed back to $\mathcal{F}_{\text{ECGKA}}$.

We now define $\mathcal{S}_2$ in more detail. Concretely, we define $\mathcal{S}_2$ in several steps: regular internal proposals, external proposals, commits, process, join, and external commit.

Regular (internal) Proposals. When $\mathcal{F}_{\text{ECGKA}}$ sends $(\texttt{Propose}, \mathsf{act})$, where $\mathsf{act} \in \{\mathsf{up\text{-}spk}, \mathsf{add\text{-}id}_t, \mathsf{rem\text{-}id}_t\}$, then the simulator $\mathcal{S}_2$ executes the ETK protocol to obtain the packet p. We then distinguish two cases:

1. If $\mathsf{act} = \mathsf{add\text{-}id}_t$, then ETK fetches the key package $\mathtt{kp}_t$ for $\mathtt{id}_t$ from $\mathcal{F}_{\mathcal{KS}}$, which obtains $\mathtt{kp}_t$ by asking the environment $\mathcal{Z}$. Note that the simulator $\mathcal{S}_2$ executes the code of both ETK and $\mathcal{F}_{\mathcal{KS}}$. Thus, $\mathcal{S}_2$ simply uses $\mathtt{kp}_t$ provided by the environment $\mathcal{Z}$. If $p = \bot$, the simulator $\mathcal{S}_2$ sends $ack = \texttt{false}$ to $\mathcal{F}_{\text{ECGKA}}$. Otherwise, $\mathcal{S}_2$ sends $(p, \mathsf{spk}_t, \texttt{true})$ to $\mathcal{F}_{\text{ECGKA}}$, where spk_t is taken from $\mathtt{kp}_t$.
2. Otherwise, $\mathsf{act} \in \{\mathsf{up\text{-}spk}, \mathsf{rem\text{-}id}_t\}$. If $p = \bot$, the simulator $\mathcal{S}_2$ sends $ack = \texttt{false}$ to $\mathcal{F}_{\text{ECGKA}}$. Otherwise, $\mathcal{S}_2$ simply sends $(p, \mathsf{spk}_t, \texttt{true})$ to $\mathcal{F}_{\text{ECGKA}}$, where $\mathsf{spk}_t = \bot$.

Indistinguishability of Output: Both $\mathcal{F}_{\text{ECGKA}}$ and ETK return the same p or $\bot$. $\bot$ is returned if $\mathsf{Ptr}[\mathtt{id}] = \bot$, meaning that $\mathtt{id}$ has not yet successfully parsed a welcome message or processed its own external commit. In this case, γ in ETK must also be $\bot$.

Assert Statements: There are two **assert** statements.

1. The **assert** in update proposals holds: ***valid-spk** queries `has-ssk` to $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$, which means it fails only if the ssk that belongs to spk is not in the secret key store. ETK on the other hand queries `get-ssk` within a **try** in `*fetch-ssk-if-nec`, which means the protocol returns $\perp$. In this case, the simulator $\mathcal{S}_2$ has passed $ack = \texttt{false}$ to $\mathcal{F}_{\text{ECGKA}}$, which also has returned $\perp$. The **assert** statement hence never fails.
2. The **assert** in ***consistent-prop** holds: Note that the proposals in ETK include a `membTag`, which is computed by using MAC over confTransHash. Recall that we model MAC as a random oracle. Recall also that confTransHash that identifies the history of commits is unique, as we have introduced. Thus, the proposals computed by every party `id` in node c must be different from those computed in any other node c'. Thus, the **assert** statement in the ***consistent-prop** holds.

External Proposals. When $\mathcal{F}_{\text{ECGKA}}$ sends $(\texttt{Propose}, \texttt{extAdd-id-spk}, \texttt{epoch})$ to the simulator $\mathcal{S}_2$, $\mathcal{S}_2$ executes the ETK protocol to obtain the packet p. ETK necessarily fetches the secret key from $\mathcal{F}_{\mathcal{AS}}$. If $p = \perp$, then $\mathcal{S}_2$ sends $ack = \texttt{false}$ to $\mathcal{F}_{\text{ECGKA}}$. Otherwise, it sends $(p, \texttt{true})$.

Indistinguishability of Output: Both $\mathcal{F}_{\text{ECGKA}}$ and ETK return the same p or $\perp$. $\perp$ is returned if $\mathsf{Ptr}[\texttt{id}] \neq \perp$, meaning that `id` has successfully parsed a welcome or process its own external commit. In this case, γ in ETK must also not be $\perp$ and ETK must also return $\perp$. Otherwise, $\perp$ is returned if the proposal was performed with an invalid spk for `id`, in which case $\mathcal{S}_2$ has also returned $ack = \texttt{false}$.

Assert Statements: There are two **assert** statements.

1. The **assert** ***valid-spk** never gets triggered for the same reason as in update proposals: if there was no valid entry in SSK, then ETK will have returned $\perp$. Thus, the simulator $\mathcal{S}_2$ will send $ack = \texttt{false}$ to $\mathcal{F}_{\text{ECGKA}}$.
2. The second **assert** statement is contained in ***consistent-ext-prop**, which enforces that proposals adding `id` to an epoch (and hence all nodes of that epoch) with a certain key package are different from those adding `id` to another epoch or using another key package. This is given by including the epoch in p. It also ensures consistency upon a re-calculation, which holds because attributes in ETK and $\mathcal{F}_{\text{ECGKA}}$ correspond directly.

Commits. The simulator $\mathcal{S}_2$ computes the packets c and the vector of welcome messages `vec_w` according to ETK. If $c = \perp$, then $\mathcal{S}_2$ sets $ack = \texttt{false}$. Otherwise, $\mathcal{S}_2$ first checks whether c corresponds to a detached root, indicated by $\texttt{Node}[c] = \perp$, there exists a vector of welcome messages `vec_w` such that $\texttt{Wel}[\texttt{vec\_w}] = \mathsf{root}_{rt}$ and confTransHash in `vec_w` matches the information in c, sends rt to $\mathcal{F}_{\text{ECGKA}}$ (alongside c and w and c's epoch). Then, $\mathcal{F}_{\text{ECGKA}}$ runs ***fill-props**. For every proposal p in the input $\boldsymbol{p}$ without a node, $\mathcal{S}_2$ sets orig and act according to p. This is always doable, as the basic check in ***apply-props** by ETK guarantees that p is well-formed.

Indistinguishability of Output: $\mathcal{F}_{\text{ECGKA}}$ returns c, w, and g, which were generated according to ETK. There are three cases where the Commit interface returns $\perp$:

– **req** $\mathsf{Ptr}[\mathtt{id}] \neq \bot$ holds as above.
– **req** *ack* is false if and only if ETK failed and returned $\bot$

Assert Statements: The commit interface includes the following **assert** statements:

1. **assert** ***valid-spk** succeeds because it is either the committers current spk, in which case `true` is returned, or the spk is updated, in which case `*rekey-path` will be failed upon an invalid spk, the simulator $\mathcal{S}_2$ will return $ack = \mathtt{false}$, and $\mathcal{F}_{\text{ECGKA}}$ will return $\bot$ in ***req-correctness**.
2. **assert** $\mathsf{mem} \neq \bot$ will succeed because `*apply-props` executed in ***req-correctness** guarantees that at least the committer is part of the group. It is left to show that if `*apply-props` in $\mathcal{F}_{\text{ECGKA}}$ returns $\bot$ due to the proposals not being well-formed, then so would `*apply-props` in ETK and hence the simulator $\mathcal{S}_2$ will return $ack = \mathtt{false}$.
   – $\mathsf{Node}[c] \neq \bot$ with $c = \mathsf{Ptr}[\mathtt{id}]$: Since $\mathsf{Ptr}[\mathtt{id}]$ is only set after successfully running `Process`, it always points to the commit that produced the secrets available to `id` in ETK. $\mathcal{S}_2$ will return $ack = \mathtt{false}$ if $\mathsf{Ptr}[\mathtt{id}]$ was not a valid c. As $\mathsf{Ptr}[\mathtt{id}]$ in $\mathcal{F}_{\text{ECGKA}}$ is only ever set to c with $\mathsf{Node}[c] \neq \bot$, $\mathsf{Node}[\mathsf{Ptr}[\mathtt{id}]]$ is hence not $\bot$.
   – $(\mathtt{id}_c, *) \in \mathsf{Node}[c].\mathsf{mem}$: Assuming that $\mathsf{Node}[c].\mathsf{mem}$ correctly reflects the roster of τ after c the committer's membership is checked by ETK when confirming the commit signature. This holds due to the indistinguishability of the output of `Process`, which shows that the changes applied by `*apply-props` are equivalent between $\mathcal{F}_{\text{ECGKA}}$ and ETK.
   – $\mathsf{Prop}[p] \neq \bot$ is guaranteed by running ***fill-props**.
   – $c \in \mathsf{Prop}[p].\mathsf{par}$ with $c = \mathsf{Ptr}[\mathtt{id}]$, is either given by ***fill-props**, or the confirmation of the membership MAC ETK guarantees that `id` is member of the epoch p is sent to. For external self-add proposals, $\mathcal{F}_{\text{ECGKA}}$ matches them purely based on epoch, and ETK checks that the epoch is correct and parsable for `id`.
   – $p = p_{up} + p_{rem} + p_{add}$: Because of the direct correspondence of $\mathsf{Prop}[p].\mathsf{act}$ and the action of p, ETK performs the same check and $\mathcal{S}_2$ will return $ack = \mathtt{false}$. $\mathsf{Prop}[p]$ accurately reflects the action because either p is created by $\mathcal{S}_2$ according to act, or act is set by $\mathcal{S}_2$ extracting act from p.
   – $\mathtt{id}_s \in G \setminus L$ (update) is checked by ETK.
   – $\mathtt{id}_s \in G \wedge \mathtt{id}_t \in G \setminus L$ (remove) is checked by ETK.
   – $\mathtt{id}_s \in G \wedge \mathtt{id}_t \notin G$ (internal add) or $\mathtt{id}_s = \mathtt{id}_t \wedge \mathtt{id}_t \notin G$ (external add) is checked during unframing of the proposal and in `*apply-props`.
3. **assert** ***valid-successor**:
   – $\mathsf{Node}[c] \neq \bot$ is given by the **else** branch.
   – $\mathsf{Node}[c].\mathsf{mem} = \mathsf{mem}$ follows from the other requirements: either $\mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[\mathtt{id}]$ and $\mathsf{Node}[c].\mathsf{pro} = \boldsymbol{p}$, in which case $\mathsf{Node}[c].\mathsf{mem}$ has been set to the same results as the passed parameter mem. Else, $\mathsf{Node}[c].\mathsf{par} = \bot$ and $\mathsf{Node}[c].\mathsf{pro} = \bot$, in which case, c is a detached root but has been returned by $\mathcal{S}_2$ due to matching confTransHash, which includes the tree and hence membership.

- $\texttt{Node}[c].\mathsf{pro} \in \{\bot, \boldsymbol{p}\}$ holds because p and $\texttt{Node}[c].\mathsf{pro}$ correspond directly between ETK and $\mathcal{F}_{\text{ECGKA}}$.
- $\texttt{Node}[c].\mathsf{par} \in \{\bot, \mathsf{Ptr}[\texttt{id}]\}$: if $\texttt{Node}[c].\mathsf{par}$ were neither $\bot$ nor $\mathsf{Ptr}[\texttt{id}]$, it would be another c'. As shown before $\mathcal{F}_{\text{ECGKA}}$ and ETK place `id` in the same commit. $\mathcal{S}_2$ would have returned $ack = \texttt{false}$, since `id` cannot process c without the epoch's secrets.

4. **assert** $\texttt{Node}[c].\mathsf{orig} = \texttt{id}$: c as returned by $\mathcal{S}_2$ is unique and includes the committers identity as part of the confTransHash. `id` is encoded in c and $\texttt{Node}[c].\mathsf{orig}$ correspond directly between ETK and $\mathcal{F}_{\text{ECGKA}}$.
5. **assert** $\text{RndCor}[\texttt{id}] \neq \texttt{good}$: If $\texttt{Node}[c] \neq \bot$, then $\mathcal{S}_2$ previously returned the same c. $\mathcal{S}_2$ constructs c according to ETK, which uses randomness to generate the new leaf node on an update path. In order to return the same c twice, the member's randomness must have been corrupted by the adversary in the real world. If the adversary in the real world corrupts a member `id`'s randomness, $\text{RndCor}(\texttt{id}, \texttt{Bad})$ is called in $\mathcal{F}_{\text{ECGKA}}$ and $\text{RndCor}[\texttt{id}]$ would have been set to bad. If $\texttt{Node}[c] = \bot$ and $rt \neq \bot$: then there exists a vector of welcome message `vec_w` whose confTransHash matches that of c. As the hash also includes the new random leaf node, the same argument as above holds.
6. **assert** $\texttt{vec\_w} \neq \bot$: ETK and also $\mathcal{S}_2$ return `vec_w` if there is an add proposal in $\boldsymbol{p}$. If such is the case, the proposal would have received the $\mathsf{act} = \texttt{add-}*$ by either `Propose`, or by $\mathcal{S}_2$ during ***fill-props**.
7. **assert** $\texttt{Wel}[\texttt{vec\_w}] \in \{\bot, c\}$ guarantees that the vector of welcome message is different for each c. This is achieved by including the `confTag` over the transcript hash in the Welcome's GroupInfo In Commit, $\texttt{Wel}[\texttt{vec\_w}]$ can previously only have been set to c. The only other option for setting $\texttt{Wel}[\texttt{vec\_w}]$ is in `Join`. There, c is set to the value provided by $\mathcal{S}_2$, which retrieves c as the only fitting commit by comparing the confTransHash, or it is set to a detached node. In the latter case, c is providing the link to the detached node, hence ***attach** would have set $\texttt{Wel}[\texttt{vec\_w}]$ to c.
8. **assert** $g \neq \bot$ succeeds because ETK and $\mathcal{S}_2$ both return g.
9. **assert** $\texttt{groupInfo}[g] \in \{\bot, c\}$: as for Welcomes, the `confTag` in the `groupInfo` guarantees the uniqueness of g for each c. In `Commit` and `ExtCommit`, $\texttt{groupInfo}[g]$ is only ever set to c as retrieved by $\mathcal{S}_2$ by confirming the matching `confTag`.

Process. Process splits the processing of external commits and regular commits based on id handing in the full proposal vector. If `id` hands over the proposal vector, c is treated as a regular commit and the following proceeds:

The simulator $\mathcal{S}_2$ runs ETK protocol and checks whether the receiver would accept the inputs. If the inputs are rejected, then the simulator $\mathcal{S}_2$ sends $ack = \texttt{false}$. Otherwise, the simulator $\mathcal{S}_2$ checks whether c corresponds to a detached root exactly as dealing with `Commit` algorithm above. If c creates a new node, which means that there was no detached root and $\texttt{Node}[c] = \bot$, the simulator $\mathcal{S}_2$ retrieves c's epoch, orig', spk' from c (by checking the committer's key package that was generated in `updatePath`).

In the case of $\boldsymbol{p} = \bot$, `Process` treats the commit as an external commit. The simulator $\mathcal{S}_2$ executes ETK (which also treats the commit based as an external

commit due to $\boldsymbol{p} = \bot$) and sets $ack = \texttt{false}$ if the processing party does not accept the input. Otherwise, $\mathcal{S}_2$ checks if c creates a new node (an existing detached root is impossible in this case, as an external commit does not produce any welcome messages). In this case, $\mathcal{S}_2$ retrieves c's epoch, orig' and spk' from c. In both cases, $\mathcal{S}_2$ also extracts the proposal vector $\boldsymbol{p}$ from c. If creating a new node, $\mathcal{S}_2$ retrieves act from the proposal p and returns it.

Indistinguishability of Output: We first explain why the outputs of ETK and $\mathcal{F}_{\text{ECGKA}}$ are the same for regular commits. Recall that a regular commit includes the hash of all proposals and we model Hash function as random oracle. Thus, for every commit c there is only one correct $\boldsymbol{p}$ that can be passed by $(\texttt{Process}, c, \boldsymbol{p})$. Recall also that the output of `Process` is determined by $\boldsymbol{p}$ and the member set mem in c's parent node, both in ETK and ***output-proc** in $\mathcal{F}_{\text{ECGKA}}$. Thus, the outputs of ETK and $\mathcal{F}_{\text{ECGKA}}$ are the same for a regular commit. Note that $\texttt{Node}[c].\mathsf{orig}$ is either determined here by $\mathcal{S}_2$ directly according to ETK or it was set honestly following ETK that was run accordingly. The `Process` returns $\bot$ only in the following cases:

- **req** *ack* holds as above.

In the case of external commits, the relation between c and p is even more immediate, as $\mathcal{S}_2$ extracts the latter from the former. In this case, ***output-proc** is meanwhile determined by the same parameters as the regular commit, and the argumentation is analogous.The `Process` returns $\bot$ only in the following case:

- **req** *ack* holds as above.

Assert statements : The `Process` interface includes the following **assert** statements for regular commits:

- **assert** $\mathsf{mem} \neq \bot$ means `*apply-props` failed. In case c already existed and `*apply-props` must have been run on the exact same parameters before, especially on $\mathsf{Ptr}[\texttt{id}]$ (or ***req-correctness** would have failed while checking $\texttt{Node}[c].\mathsf{par} = \mathsf{Ptr}[\texttt{id}]$) and passed. Else, ***req-correctness** would have failed. If $\texttt{Node}[c] = \bot$ or c were malformed according to ETK (i.e. a proposal removing the committer), $\mathcal{S}_2$ would return $ack = \texttt{false}$ and `Process` would unwind. This means that c either corresponds to a correct detached root or a correct new node. It is left to show that every single **req** in `*apply-props` is either fulfilled or $\mathcal{S}_2$ would have returned $ack = \texttt{false}$, unwinding the function. This holds for the same reason as for `Commit`.
- **assert** ***valid-successor**: is equivalent to the `Commit`.
- **assert** $\texttt{id} \in \texttt{Node}[c].\mathsf{mem}$: As shown before, $\mathsf{Ptr}[\texttt{id}]$ points to the same c that `id` has most recently processed in ETK. $\mathcal{S}_2$ guarantees that `id` is a member of the previous epoch by either verifying the membership tag or the confirmation tag, else returns $ack = \texttt{false}$, and ***req-correctness** would fail as $\texttt{Node}[c].\mathsf{par}$ is unequal to $\mathsf{Ptr}[\texttt{id}]$. In both cases, ***members** is executed on $\mathsf{Ptr}[\texttt{id}]$ and the same $\boldsymbol{p}$, and if no proposal removing id is executed, `id` will be part of $\texttt{Node}[c].\mathsf{mem}$.

In the case of an external commit, *process-ec(c) contains the following **assert** statement:

- **assert** mem $\neq \bot$ for both Node[c] $= \bot$ and Node[c] $\neq \bot$: holds if *apply-props runs fully. As ***req-correctness** returned false, any malformed c would have lead to an unwinding of *process-ec:
  - Node[c] $\neq \bot$ with $c =$ Ptr[id]: given by placement in if statement.
  - ($\texttt{id}_c, *$) $\in$ Node[c].mem with $c =$ Ptr[id] (resync): Assuming that Node[c].mem correctly reflects the roster of τ after c,the committer's membership is checked by ETK when confirming the commit signature. ($\texttt{id}_c, *$) $\notin$ Node[c].mem with $c =$ Ptr[id] (external commit): same as above.
  - prop $\neq \bot$ is guaranteed by the fact that an external commit is never created as a detached node, and prop is hence always set.
  - $c \in$ prop.par with $c =$ Ptr[id] is given by prop being directly included, and ExtCommitProps parents being set according to the including commit. In case that Node[c] pre-exists, it hence depends on Ptr[id] $=$ Node[c].par. This is the case as o.w. ***req-correctness** would have failed and assuming that Ptr[id] points to the commit parsed in ETK, S2 would have returned $ack =$ false.
  - $p = p_{exi} + p_{rem}$: (Note: given the only setting in which an external commit node is created is after checking that all included commits are either exi or rem.) Prop[p].act directly corresponds to p's action. Since ETK hence performs the same checks $\mathcal{S}_2$ would have returned $ack =$ false.
  - $\texttt{id}_s = \texttt{id}_t = \texttt{id}_c$ (remove) is given as ExtCommitProps creates all proposals in such a way that orig$'$ ($= \texttt{id}_s$) is equal to the committer ($\texttt{id}_c$) without allowing the adversary to set it separately. $\texttt{id}_t$ being equal is given by the **req** in Process and the way that the proposals are created.
- **assert** ***valid-successor**: holds for the same reason as in Commit.
- **assert** id $\in$ Node[c].mem is given by mem $\neq \bot$ and the fact that an external commit does not remove anyone, and adds the committer.

Join. When a party id joins using a vector of welcome message vec_w, the simulator $\mathcal{S}_2$ runs ETK and sets $ack =$ true if no error occurs. If $ack =$ false or Wel[vec_w] $\neq \bot$, then $\mathcal{S}_2$ simply sends ($ack, \bot, ...$) to $\mathcal{F}_{\text{ECGKA}}$. Note that if Wel[vec_w] $\neq \bot$, $\mathcal{F}_{\text{ECGKA}}$ already knows the information in vec_w and can ignore nay values received from the simulator other than ack. Otherwise, i.e., $ack =$ true and Wel[vec_w] $= \bot$, the simulator $\mathcal{S}_2$ needs to interpret the injected vec_w. Recall that vec_w $=$ (encGroupSecs, encGroupInfo, wel_type) and that $\mathcal{S}_2$ knows all secrets and therefore is able to recover groupInfo from encGroupInfo. In this case, $\mathcal{S}_2$ sends a value c', which either equals an existing node that owns identical confTransHash as the one in groupInfo or $\bot$ if such node does not exist, a value orig$'$ according to groupInfo, a value mem$'$ according groupInfo, and a value vec_w's epoch to $\mathcal{F}_{\text{ECGKA}}$.

Indistinguishability of Output: $\mathcal{F}_{\text{ECGKA}}$ outputs Node[c].mem, Node[c].orig. These are either retrieved from vec_w by $\mathcal{S}_2$ as described above, in which case they are directly equal to the values output by ETK. In the other case, vec_w determines

the c from which this info is taken. Note that if a party `id` can successfully process `vec_w`, then the corresponding commit c must be uniquely determined for `id`. This is because that the `groupInfo` decrypted from `vec_w` includes a confirmation tag that MAC the hash the history of commits and we model MAC as random oracle. For the commit c, either $\mathcal{F}_{\text{ECGKA}}$ sets orig directly and $\mathcal{S}_2$ supplies the corresponding c, or $\mathcal{S}_2$ derives orig' from c and returns it to $\mathcal{F}_{\text{ECGKA}}$.

The `Join` returns $\perp$ only in the following case:

- **req** *ack*: holds as above.

Assert Statements: The `Join` interface includes the following **assert** statements:

- **assert** `id` $\in$ `Node`$[c]$.mem never triggers, as ETK implicitly performs a check on the tree membership of id by calling $\gamma.\tau.\mathsf{leaves}[\gamma.\mathsf{leafId}()]$ and $\mathcal{S}_2$ would hence send $ack =$ `false`. As shown for the indistinguishability, `Node`$[c]$.mem contains the members of the tree, and hence also `id`.
- **assert** $g \neq \perp$ succeeds because ETK and hence $\mathcal{S}_2$ returns g.
- **assert** `groupInfo`$[g] \in \{\perp, c\}$: like for Welcomes, the confTransHash in the `groupInfo` guarantees the uniqueness of g created by $\mathcal{S}_2$ for a c. In `Commit` and `ExtCommit`, `groupInfo`$[g]$ is only ever set to c as retrieved by $\mathcal{S}_2$ by confirming the matching `confTag`.

Consistency: We still need to argue that every joiner `id` that executes `Join` with a vector of welcome message `vec_w` and every group member `id`$'$ that executes `Process` with a commit c' must end up in the same state, where c' is uniquely determined by `vec_w` for `id` as we have explained above. Recall that both `id` and `id`$'$ verify the confirmation tag, for which they derive the key from the joiner secret combined with the group context. This guarantees (assuming that MAC is modeled as random oracle) that they agree on the context, which in particular includes the tree hash and the confirmed transcript hash. The tree hash binds the whole ratchet tree, including its structure, spk of all members, and all public keys. In particular, this implies agreement on the member set mem'. Agreement on the transcript hash implies agreement on the history, including the last committer orig'. The agreement is maintained in descendants of c', since parties agree on the ratchet tree in c'.

External Commit. When an external party sends `ExtCommit`, $\mathcal{S}_2$ constructs the external commit using g according to ETK and sets $ack =$ `false` if $c = \perp$. Else, it generates all required parameters for $\mathcal{F}_{\text{ECGKA}}$. It generates g, $\boldsymbol{p}_{exi}$, and $\boldsymbol{p}_{rem}$ according to ETK. These are direct or inlined proposal messages for the commit. It then checks if g is known to $\mathcal{F}_{\text{ECGKA}}$ ($g \neq \perp$). If it is not, but there is a matching commit c_{gmatch} that has the same
$confirmedTH$ as g, $\mathcal{S}_2$ returns this. If there is no such c_{gmatch}, $\mathcal{S}_2$ extracts the sender orig_g and the members mem_g from g.

Indistinguishability of Output: $\mathcal{F}_{\text{ECGKA}}$ returns c and g' generated according to the ETK protocol. The `ExtCommit` returns $\perp$ only in the following cases:

– **req** $\mathsf{Ptr}[\mathtt{id}] \neq \bot$: hold as above.
– **req** *ack*: holds as above.

Assert Statements: The `ExtCommit` interface includes the following **assert** statements:

– **assert** ***valid-spk**$(\mathtt{id}, \mathsf{spk})$: holds because otherwise, `*rekey-path-upon-join` would have failed in ETK and ***req-correctness** in $\mathcal{F}_{\text{ECGKA}}$, and upon $\mathcal{S}_2$ returning $ack = \mathtt{false}$, the method unwinds.
– **assert** $\mathsf{mem} \neq \bot \wedge (\mathtt{id}, \mathsf{spk}) \in \mathsf{mem}$: is given as `*apply-props` executed on any $\boldsymbol{p}$ contains at least the committer (in this case `id`) when run successfully. It runs successfully because
  • the requirements on `props` of `*apply-props` are fulfilled through the construction in `ExtCommit`.
  • **req** $(\mathtt{id}_c, *) \in G$ (resync) or $(\mathtt{id}_c, *) \notin G$ (external commit): is equally covered by ***req-correctness** in combination with ETK execution.
  • **req** $\mathtt{id}_s = \mathtt{id}_t = \mathtt{id}_c$ and $\mathtt{id}_s \in G$ fulfilled through construction in `ExtCommit` and the previous **req**.
– ***consistent-ext-comm**() holds for a similar reason as for proposals and commits. An external commit does not include a membership tag, but it does include the confirmation tag, which is a MAC over the GroupContext's confirmed transcript hash that provides the same uniqueness guarantee of a commit c. $\mathtt{Node}[c].\mathsf{pro}$ and $\mathtt{Node}[c].\mathsf{par}$ will hence only be set to the proposals included in c and the parent of c included in the confTransHash, or else to $\bot$ in case of a detached root. As a consequence of these two properties, $\mathtt{Node}[c].\mathsf{mem}$ will always be set to the same membership set handed to ***consistent-ext-comm**.
  As `ExtCommitProps` is directly constructed from $\boldsymbol{p}$, as a consequence, all its parameters are also equivalent.
– **assert** $g \neq \bot$ succeeds because ETK and hence $\mathcal{S}_2$ returns g.
– **assert** $\mathtt{groupInfo}[g] \in \{\bot, c\}$: like for Welcomes, the `confTag` in the `groupInfo` guarantees the uniqueness of g created by $\mathcal{S}_2$ for a c. In `Commit` and `ExtCommit`, $\mathtt{groupInfo}[g]$ is only ever set to c as retrieved by $\mathcal{S}_2$ by confirming the matching `confTag`.

Consistency Invariant. The consistency invariant has four clauses, that explicitly defined in **cons-invariant**, and we below show that each of them holds.

1. $\mathtt{Node}[c].\mathsf{pro} \neq \bot$: holds for all regular child nodes. When attaching a detached node with $\mathtt{Node}[c].\mathsf{pro} = \bot$ to a parent node, the proposals are set to $\boldsymbol{p}$. `*apply-props` in $\mathcal{F}_{\text{ECGKA}}$ checks whether the commit $c \in \mathtt{Prop}[p].\mathsf{par}$. In every case a new commit node is created or attached to a proposal list (in `Join` and `Process`), the **assert** mem would have hence failed if that is not the case.

2. $\forall \texttt{id}$ s.t. $\mathsf{Ptr}[\texttt{id}] \neq \bot$: $\texttt{id} \in \texttt{Node}[\mathsf{Ptr}[\texttt{id}]].\mathsf{mem}$: holds because in both locations where $\mathsf{Ptr}[\texttt{id}]$ is set to c (`Process` and `Join`) an explicit check is performed that $\texttt{id} \in \texttt{Node}[\mathsf{Ptr}[\texttt{id}]].\mathsf{mem}$.
3. The graph contains no cycles: $\mathcal{F}_{\text{ECGKA}}$ will only attach a detached node c' to an existing graph node c if $\mathcal{S}_2$ provides the corresponding root. $\mathcal{S}_2$ will only do so if the $\mathsf{confTransHash}$ match. Since this hash contains all previous commits, a circular graph cannot exist.
4. $\forall ep$ s.t. $\texttt{ExtProps}[ep] \neq \bot : (\forall c \in \texttt{Prop}[ep].\mathsf{par} : \texttt{Node}[c].\mathsf{epoch} = \texttt{ExtProps}[ep].\mathsf{epoch})$: holds because the parents of an external proposal ep are only set if the epochs match.

J.2 Confidentiality

Intuition. We prove that *H2* and *H3* are indistinguishable by demonstrating that for all commits c for which the predicate $\textbf{safe}(c)$ in Figure 7 holds, the group keys are indistinguishable from random. The indistinguishability between *H2* and *H3* indicates the confidentiality of ETK.

To this end, we define a sequence of sub-hybrid Hs_i, where i indicates the index of epochs. In the i-th sub-hybrid Hs_i, the simulator sets the first i epoch secrets, and all epoch secrets after that are set according to $\mathcal{F}_{\text{ECGKA}}$, i.e., randomly if $\textbf{safe} = \texttt{true}$ and otherwise by the simulator. Note that the environment $\mathcal{Z}$ is executed in polynomial time and therefore can trigger only polynomial epochs of $\mathcal{F}_{\text{ECGKA}}$. Thus, the number of the sequence of sub-hybrid games Hs_i must be polynomial. It is easy to observe that in the final sub-hybrid all secrets of the epoch are set according to *H2* (and therefore is equivalent to *H2*), and that in the first sub-hybrid Hs_0 all are set according to *H3* (and therefore is equivalent to *H3*). Thus, the indistinguishability between *H2* and *H3* can be reduced to the indistinguishability between every two adjacent sub-hybrids Hs_{i-1} and Hs_i for the sequence of i.

We prove the indistinguishability between every two adjacent sub-hybrids Hs_{i-1} and Hs_i following the strategy in [4], i.e., by introducing the GSD game. Recall in Theorem 3 that the GSD security of any PKE can be reduced to its IND-CCA security. Thus, our proof methodology in this section is to reduce the confidentiality of ETK to the GSD security of the underlying PKE. Since they only behave differently in epoch i, and only if $\textbf{safe}(c_i) = \texttt{true}$, we show that if the environment $\mathcal{Z}$ can distinguish between a random key and a simulator chosen key for c_i, then there exists a reduction $\mathcal{B}$ that can win the GSD game (and hence break the IND-CCA security) of the underlying PKE by invoking $\mathcal{Z}$. We need to argue that whenever the environment $\mathcal{Z}$ wins, which can happen only under the condition that $\textbf{safe}(c) = \texttt{true}$ (implying a valid challenge for $\mathcal{Z}$), the reduction $\mathcal{B}$ can also win, which happens only under the condition that $\neg\mathsf{gsd\text{-}exp}(c, u_a)$ (implying a valid challenge for $\mathcal{B}$ breaking GSD). To this end, we prove that $\mathsf{gsd\text{-}exp}(c) \implies \neg\textbf{safe}(c)$ by reasoning through all possible cases of $\mathsf{gsd\text{-}exp}(c)$.

Important to note that the reduction $\mathcal{B}$ stores one of three values for each key: (1) if the key is unknown to $\mathcal{Z}$ and $\mathcal{B}$ (and hence only known to GSD game), $\mathcal{B}$

stores (gsd, u), with u indicating the GSD node that contains the key value, (2) if the key is known to both $\mathcal{B}$ and $\mathcal{Z}$, $\mathcal{B}$ stores the actual key value, and (3) if the key is known to $\mathcal{Z}$, but not to $\mathcal{B}$ (or GSD game), $\mathcal{B}$ sets it to $\perp$. For bookkeeping, $\mathcal{B}$ maintains a counter u_{ctr} to index new GSD nodes at every point in the game.

Lemma 2. *If* PKE *is* GSD *secure, then Hybrids H2 and H3 are indistinguishable, that is,* ETK *guarantees confidentiality.*

Proof. We show the indistinguishability by hybrid games. More concretely, for any $i > 1$, we define two adjacent sub-hybrids Hs_{i-1} and Hs_i that behave differently only in epoch i: Hs_{i-1} sets the epoch secret in epoch i according to $\mathcal{F}_{\text{ECGKA}}$ (depending on whether $\textbf{safe}(c)$ holds), while Hs_i sets is according to the simulator $\mathcal{S}$. Further, we reduce the indistinguishability between every two adjacent hybrid games to the GSD security of the underlying PKE. More concretely, we show that if the environment $\mathcal{Z}$ can distinguish between a random key and a simulator chosen *application* key for c with $\textbf{safe}(c) = \texttt{true}$, the reduction $\mathcal{B}$ (i.e., the adversary that aims to break GSD game) can use the environment $\mathcal{Z}$ to win the GSD game with $\textsf{gsd-exp}((c, u)) = \texttt{false}$ for some associated secret u.

The individual steps of the confidentiality part are organized as follows:

Proof part 2.b): Confidentiality

Show indistinguishability:

I) when not allowing any injections or corruption of randomness by reasoning through all possible cases of $\textsf{gsd-exp}((c, u_{\text{app}}))$ (Section J.2):
   (a) the corruption of (c, u_{app})
   (b) the corruption of (c, u_{joi})
   (c) if both $\textsf{gsd-exp}$ $((c, u_{\text{com}}))$ and $\textsf{gsd-exp}$ $((\text{Node}[c].\text{par}, u_{\text{ini}}))$ are true.

II) when allowing injections for cases (a) - (c) (Section J.2).

III) when allowing the corruption of a client's randomness for cases (a) - (c) (Section J.2).

We update the proof to include the changes from draft 12 to the final standard. Additionally, We introduce external commits. We mostly note down changes to the content of [4].

Part 2.b.I) No Injection or Corrupted Randomness Allowed

In this part, we consider the simplest case, where neither malicious injection nor randomness corruption is allowed.

In order to prove by reduction that $\textsf{gsd-exp}((c, u)) \implies \neg\textbf{safe}(c)$, we first outline how reduction $\mathcal{B}$ modifies the code of the functionalities and the simulator in the context where no injection and no corrupted randomness are permitted. Following this, we systematically examine each possible cases of $\textsf{gsd-exp}((c, u))$, corresponding to the three conditions specified above, and illustrate how our proof captures the requirements of the safe predicate.

As described in the *gsd* experiment explanation; in our model, for {`add`, `extAdd`}-only commits, no UpdatePath is executed, and all of `id` 's secrets remain the same. This is addressed in ***secrets-replaced**$(c, \texttt{id})$. Throughout the confidentiality proof, we will examine commits in which an `updatePath` is required as part of ***secrets-replaced**$(c, \texttt{id})$, as well as those in which no `updatePath` is necessary. To this end, we also consider the commits including `add`, `extAdd` proposals only as well as those that involve any combination of these proposal types.

From draft 12 to RFC: Different from draft 12, the RFC separates leaf nodes and key packages. While `add` and `extAdd` proposals contain key packages, update proposals and `updatePath` only contain fresh leaf nodes. Both share a similar structure. Therefore, it is necessary for us to change the welcome messages, which now use the additional *init key* of the key package to encrypt the group secrets. Additionally, the committer has the option to create an encrypted welcome message, which can be designed for all members collectively, or for each member individually, or grouped for specific batches of members. Moreover, we need to define how reduction $\mathcal{B}$ processes the `extAdd` proposals. Hence:

- Key Package Registration: When $\mathcal{Z}$ instructs a party `id` to register a key package in the emulated $\mathcal{F}_{\mathcal{KS}}^{\mathcal{TW}}$, the reduction $\mathcal{B}$ creates a new GSD vertex by executing $\mathsf{pk} \leftarrow *\mathsf{get\text{-}pk}(u_{\mathrm{ctr}})$ and $\mathsf{init\text{-}pk} \leftarrow *\mathsf{get\text{-}pk}(u_{\mathrm{ctr}})$, respectively followed by $u_{\mathrm{ctr}}{+}{+}$.[10] The reduction $\mathcal{B}$ uses pk and $\mathsf{init\text{-}pk}$ to generate the public part of the key package `kp`, and sets the secret keys $\mathsf{SK}[\texttt{id}, \texttt{kp}]$ and $\mathsf{ISK}[\texttt{id}, \texttt{kp}]$ to (gsd, u_{ctr}).
- Proposal `extAdd`: Whenever the protocol requests a key package for `id` creating the proposal from $\mathcal{F}_{\mathcal{AS}}^{\mathcal{TW}}$, $\mathcal{Z}$ gets to choose it. Accordingly, the reduction $\mathcal{B}$ stores in the new proposal node the ssk taken from the key package chosen by $\mathcal{Z}$.
- Applying Proposals: Additionally, for each `extAdd` proposal, secret key of `id` is set using the value stored in proposal node.
- Commits by reference: Compared with [4], there are two differences. On one hand, following RFC version, the `*rekey-path` operation is *not* executed if the proposal list in a regular commit satisfies both of the following conditions: (1) it is non-empty, and (2) it contains only `add`, `extAdd` proposals.
  The reduction $\mathcal{B}$ emulates the `*derive-keys` as follows:
  1. *Add vertices to the GSD graph:* The reduction $\mathcal{B}$ generates vertices u_{app}, u_{mem}, u_{conf}, u_{ini}, u_{ext} by querying **Hash** oracle with inputs $(u_{\mathrm{lbl}}, \mathsf{lbl})$ for $\mathsf{lbl} \in \{\mathsf{app}, \mathsf{mem}, \mathsf{conf}, \mathsf{ini}, \mathsf{ext}\}$.
  2. *Create the welcome message:* The welcome message is a component of the welcome vector, consisting of three parts. The final part, `wel_type`, indicates whether the message contains a single welcome message or multiple welcome messages. In the first part, for each new member $\texttt{id}_t$,

[10] Recall that we use $\mathsf{pk} \leftarrow *\mathsf{get\text{-}pk}(u)$ to denote that $\mathcal{A}$ obtains the public key pk for a vertex u by calling the oracle $\mathbf{Enc}(u, 0)$ (the special vertex 0 is only used here), following [4].

the encryptions of $\texttt{joinerSec}$ and $\texttt{id}_t$'s $\texttt{pathSec}$ under the $\mathsf{init\text{-}key}$ in $\texttt{id}_t$'s key package (obtained from KS by the party adding $\texttt{id}_t$). Let u_i be the GSD vertex corresponding to the $\texttt{pathSec}$ sent to $\texttt{id}_t$. If $\texttt{id}_t$'s $\mathsf{init\text{-}key}$ is of the form (gsd, u)[11], $\mathcal{A}$ obtains the encryptions by creating encryption edges from u to u_{joi} and from u to u_i. Otherwise, it corrupts u_i, u_{joi} encrypts the values itself. The second part of the welcome message contains the encryption of $\texttt{groupInfo}$, under the $\texttt{welcome\_secret}$ which is simply the combination of $\texttt{joinerSec}$ and the vector of welcome messages. To compute the second part of the vector of welcome messages, the reduction $\mathcal{B}$ creates vertex u_{wel} by querying **Hash** oracle with input $(u_{\text{wel}}, \mathsf{wel})$. The reduction $\mathcal{B}$ gets the respective secrets by corrupting corresponding vertices.

3. *Create* $\texttt{groupInfo}$*:* Additionally, the reduction $\mathcal{B}$ creates the $\texttt{groupInfo}$: It runs $*\mathsf{get\text{-}pk}(u_{\text{ext}})$ to retrieve the external public key and includes it in the $\texttt{groupInfo}$. Note that at this point, $\mathcal{B}$ does not need to know the external private key.

– Expose: The private init keys are generated along with the key package secrets, which $\mathcal{A}$ is also required to corrupt.

As the basis for the proof, note that, considering the key derivation illustrated in Figure 2, $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ can only hold true in one of the following cases:

2.b.I.a) The reduction $\mathcal{B}$ corrupts the vertex (c, u_{app}). This happens if and only if it corrupts the state of a party involved in commit c. This case is identical to the case in [4] and will immediately implies $\neg\textbf{safe}(c)$.

2.b.I.b) The reduction $\mathcal{B}$ corrupts the vertex (c, u_{joi}) to ultimately gain access to the application secret. This happens if and only if the reduction $\mathcal{B}$ computes a vector of welcome messages for $\texttt{id}_t$ using an exposed key bundle. The mechanism remains as previously described.

2.b.I.c) This case considers the following two elements to be true: $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ and $\mathsf{gsd\text{-}exp}((\texttt{Node}[c].\mathsf{par}, u_{\text{ini}}))$. We need to show that $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ implies $\mathsf{know}(c, *)$. This implication follows by case d) of ***can-traverse**. Here, $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ is only possible in one of the following cases:

1) the reduction $\mathcal{B}$ corrupts path secrets u_i, and use them to compute $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$
2) the reduction $\mathcal{B}$ calls **Enc** oracle to encrypt path secrets under relevant keys
3) after the commit processed, the reduction $\mathcal{B}$ exposes a party $\texttt{id}$ that holds u_i's secret

Firstly, consider case 3) any action requiring $\texttt{updatePath}$ removes u_i's secret from its state. For the actions of $\texttt{id}$ requiring $\texttt{updatePath}$, the proof remains the same. For $\{\texttt{add}, \texttt{extAdd}\}$-only commits, since no $\texttt{updatePath}$ is executed, all of $\texttt{id}$'s secrets remain the same. Thus, if $\texttt{id}$ is corrupted, we have $\mathsf{know}(c, \texttt{id})$ until an $\texttt{updatePath}$ issued.

[11] Following [4], a secret key of the form (gsd, u) means that it is unknown to the environment $\mathcal{Z}$.

We complete the proof by showing that for cases 1) and 2) $\mathsf{gsd\text{-}exp}((\mathtt{Node}[c].\mathsf{par}, u_{\text{ini}}))$ implies $\mathsf{know}(c, \text{'}\mathsf{epoch}\text{'})$. For these cases the proof follows the same pattern as the encapsulating proof for u_{app} as u_{ini} has the exact same lifetime as its epoch's u_{app} is generated from each epoch secret. The proof's condition c) hence recursively proceeds until at some point, $\mathtt{Node}[c].\mathsf{pars} = \mathsf{root}_x$. At that point, case d) of $\textbf{*can-traverse}(c)$ in $\textbf{safe}(c)$ becomes true.

External Commit. The external commits include the following operations:

- Derivation of the init key $\mathsf{init\text{-}key}$: MLS uses an asymmetric interaction to derive the init key of the new epoch from the external secret. The reduction $\mathcal{B}$ creates a new GSD node (gsd, u_{ctr}) representing the seed s_{exini} chosen by the committer. Then $\mathcal{B}$ creates an edge encrypting the seed with the external public key by querying **Enc** oracle. This encrypted output is shared with the other group members via the external init proposal `extInit` included in the external commit. The init key $\mathsf{init\text{-}key}$ is derived from this seed by calling **Hash** on the seed node. Then $\mathcal{B}$ proceeds using the resulting node as the u_{exini} node in a commit.
- Generation of leaf node: same as `*rekey-path` in a regular commit but occurs during the `*rekey-path-upon-join`.
- Other than that, the reduction $\mathcal{B}$ proceeds as during a regular commit, corrupting the GSD vertices u_{mem} and u_{conf} in order to create the new commit and `groupInfo`. Note that there is no welcome following an external commit. The reduction $\mathcal{B}$ also corrupts any u_i (path secret) that is not $(gsd, .)$ meaning it is known to $\mathcal{Z}$ and computes all secrets following from that.

Figure 28 visualizes the GSD game accordingly.

We first show that if $\textbf{safe}(c)$ holds on an epoch then $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ is false and the epoch is still a valid challenge. We then show how the reduction $\mathcal{B}$ can win the GSD game by utilizing the $\mathcal{Z}$ that can distinguish this epoch between simulated by functionality and by simulator.

The argument for Case 2.b.I.a) and Case 2.b.I.b) remain the same as in regular commit. For Case 2.b.I.c) if there is an edge between (c, u_{exini}) and (c, u_{joi}), means that c is an external commit. As the `*rekey-path-upon-join` proceeds in the same way as `*rekey-path` in a regular commit, the reasoning for $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ implying $\mathsf{know}(c, *)$ also remains the same. The only change occurs when the combination of $\mathsf{gsd\text{-}exp}((\mathtt{Node}[c].\mathsf{par}, u_{\text{exini}}))$ implies $\mathsf{know}(c, \text{'}\mathsf{epoch}\text{'})$. Similar to the analysis for regular commits, we consider two cases: i) u_{exini} is compromised when a client `id` is exposed while processing the commit, in which case $\textbf{safe}(c)$ does not hold, or ii) the reduction $\mathcal{B}$ calculates u_{exini} itself if the seed s_{exini} and is exposed. The case i) happens if an id's key bundle is exposed during the commit processing, which again implies $\textbf{safe}(c)$ is false, while the case ii) happens if it is encrypted under a node whose private key is compromised. The seed s_{exini} is encrypted under the key from the u_{ext} node of epoch $\mathtt{Node}[c].\mathsf{par}$. A u_{ext} node is exposed in the exact same cases as a u_{app}, and as in a commit, proof continues recursively until c is a root and hence $\textbf{safe}(c)$ is false.

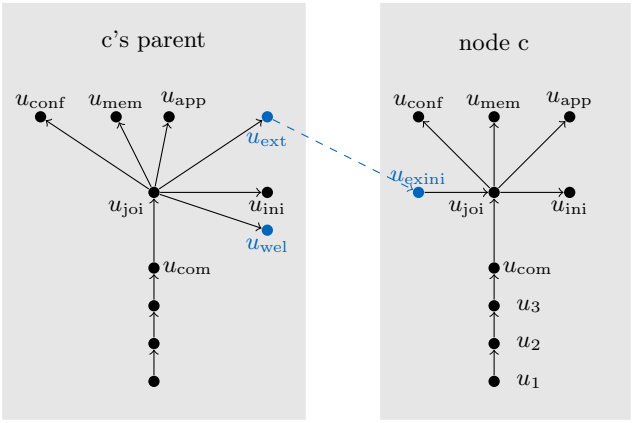


Fig. 28: GSD game graph of an external commit, following an image from [4]. Each circle represents a node in the GSD graph. Dashed arrows mark an encryption of the sink using the key of the source, while bold arrows mark a Hash call on the source. The additionally added nodes of the graph are marked in ●. The external secret encrypts the committer-generated seed s_{exini} that is used as the init secret of epoch c.

The subgraph G continues to correctly describe all cases in which a node secret stays in the tree: all commits that remove a node (c, u) are either (i) commits by an $\texttt{id}_i$ or (ii) removes of an $\texttt{id}_i$, on the subtree of $\tau.v$. This also includes external commits and resyncs.

Part 2.b.II) Allowing Injections In this part of the proof, we begin by outlining the adversarial modifications to the functionalities and the simulator, incorporating the capability to inject malicious messages.

From draft 12 to RFC

- Injected proposals: Additionally, in case of `extAdd` proposal, similar to `add` proposal, reduction $\mathcal{B}$ creates a new node using the private signing key used from the message.
- Commits injected to process: In the case that $\mathcal{Z}$ makes `id` process an injected commit c' from a party $\texttt{id}_c \neq \texttt{id}$, reduction $\mathcal{B}$ attempts to build new commit node's state. For {`add`, `extAdd` }-only commits, the reduction $\mathcal{B}$ only applies the proposals, as these commits do not require rekey operation. For other commits along with the ciphertexts `ctxt` that `id` would decrypt with keys in the ratchet tree, the reduction $\mathcal{B}$ proceeds as follows:

  i-ii) first two items remain the same.

  iii) if the `ctxt` is copied from an honest commit that generated earlier by $\mathcal{B}$, for which the GSD node associated to both `appSecret` is still a valid challenge, we must show that `id` accepting such a commit c would enable $\mathcal{B}$ to succeed in the GSD game. The trick in this step is to reduce the

forgery of the confirmation tag confTag in the commit c to the GSD security of the underlying PKE. Let $\mathcal{B}$ challenge the GSD node u and extract the correct seed from $\mathcal{Z}$'s random oracle calls as follows:

1. Note that $\mathsf{appSecret} = \mathsf{Hash}(\texttt{joinerSec}, .)$ and $\texttt{joinerSec} = \mathsf{Hash}(\mathsf{initSecret}, \texttt{commitSec}, .)$, where HKDF is modeled as RO. Moreover, note also that commitSec must be the same in c and c' for the shared ctxt to be accepted.
2. The reduction $\mathcal{B}$ needs to extract the joinerSec of c'. To this end, the reduction $\mathcal{B}$ proceeds as follows: First, recall that the confTag is a MAC of confKey and confTransHash. The only way for $\mathcal{Z}$ to compute a valid tag is to query RO on (confKey, confTransHash). The reduction $\mathcal{B}$ can search the RO query history and extract confKey. Second, recall that confKey is derived by applying HKDF to joinerSec. The environment $\mathcal{Z}$ needs to query RO on (joinerSec) for deriving confKey. The reduction $\mathcal{B}$ can then extract joinerSec of c' in a similar way.
3. This case also remains the same.

– Injected vector of welcome messages vec_w: with the recent update of RFC, the use of init-key to encrypt group secrets results in a significant change to reduction $\mathcal{B}$'s processing order in [4]; however, this does not have a big impact on the proof. When $\mathcal{Z}$ injects a welcome message $w = (\texttt{encGroupSec}, \texttt{encGroupInfo})$ in the welcome vector vec_w, $\mathcal{B}$ does as follows:

1. Note that wel_type of the corresponding vec_w does not affect the behavior of the reduction. Reduction $\mathcal{B}$ searches for a key package kp such that $\mathsf{SK}[\texttt{id}, \texttt{kp}] \neq \perp$ and $\mathsf{ISK}[\texttt{id}, \texttt{kp}] \neq \perp$ and $\mathsf{H}(\texttt{kp})$ matches an entry $e \in \texttt{encGroupSecs}$ and aborts if no such kp exists.
2. If e is copied from a vector of welcome messages vec_w generated by $\mathcal{B}$ while creating a commit node c and ISK[id, kp] is a GSD node, to win the game, reduction $\mathcal{B}$ proceeds as follows: $\mathcal{B}$ needs to differentiate between the cases, whether the injected welcome message is associated with the commit c that the e stems from. To this end, $\mathcal{B}$ compare encGroupSec. Since e is copied, the joinerSec underlying different welcome messages that include the same e must be the same. Furthermore, an equal encGroupInfo implies an equal confTransHash. In this case, $\mathcal{B}$ moves id to c.
   If the encGroupInfo in the vector of welcome message vec_w is different from the one derived by the reduction $\mathcal{B}$ for c, that means that e was copied but used to create the welcome for a non-existent commit c'. This is not possible as it would allow $\mathcal{B}$ to win the GSD game: In order to construct a encGroupInfo that is decryptable using the secrets stored in the encGroupSecs, the environment $\mathcal{Z}$ must be in possession of the welcome_secret (as $\mathcal{Z}$ cannot copy encGroupInfo from c). In order to derive the welcome_secret, $\mathcal{Z}$ must derive it from the joinerSec using the RO calls. The reduction $\mathcal{B}$ can extract joinerSec by checking queries that have been sent to RO one by one as follows: First, the reduction $\mathcal{B}$ attempts to parse the input of RO into joinerSec candidates. If the parse fails, $\mathcal{B}$ directly omit the following steps and attempts for the next RO input. Second, the reduction $\mathcal{B}$

respective derives `welcome_secret` and `confKey` from the `joinerSec` candidates. Third, the reduction $\mathcal{B}$ derives `welcome_key` and `welcome_nonce` from `welcome_secret`. Fourth, the reduction $\mathcal{B}$ decrypts `encGroupInfo` to recover `groupInfo`. Fifth, the reduction $\mathcal{B}$ parses `groupInfoTBS` from `groupInfo` and further `confTag` from `groupInfoTBS`. Finally, the reduction $\mathcal{B}$ verifies the `confTag` by using the `confKey`. If no error occurs, then reduction $\mathcal{B}$ found the correct `joinerSec`. If any error occurs, then the reduction $\mathcal{B}$ continue to check the next RO query. Note that the environment $\mathcal{Z}$ can send only polynomial queries to RO. The checks that $\mathcal{B}$ needs to execute must also be in polynomial time.

From `joinerSec`, the reduction $\mathcal{B}$ can calculate the appSecret for the valid GSD challenge node c and hence win the game. Copying e is hence not possible.

3. Else, if e has not been copied from any vector of welcome messages generated by $\mathcal{B}$, the reduction $\mathcal{B}$ can simply obtain the encrypted `joinerSec` and `pathSec` either by using the secret in $\mathsf{ISK}[\mathtt{id}, \mathtt{kp}]$, if it has been compromised and therefore known by $\mathcal{B}$, or otherwise by querying $\mathcal{B}$'s GSD decryption oracle **Dec**. By using the `joinerSec`, the reduction $\mathcal{B}$ can decrypt and verify the `encGroupInfo` in the vector of welcome messages. If the verification passes, then $\mathcal{B}$ further checks whether the decrypted `groupInfo` indicates an existing node c (by comparing the $\mathsf{confTransHash} \in$ `groupInfo`) or a new node c'. If `groupInfo` indicates an existing node c, then $\mathcal{B}$ moves `id` to c. Otherwise, `groupInfo` indicates a new node c'. In this case, $\mathcal{B}$ creates c' with labels taken from `groupInfo` and the ratchet tree set to the public part of τ from `groupInfo`. Then, for any node of τ with a public key for which it has a secret key stored (in another ratchet tree or in ISK and SK), $\mathcal{B}$ copies the secrets into τ. Moreover, $\mathcal{B}$ updates ratchet tree secrets to those derived from `pathSec` (if any secret key was set to a GSD node, $\mathcal{B}$ uses `pathSec` to win the game), and computes the secrets of the epoch from `joinerSec`. For all other unknown secrets in τ, $\mathcal{B}$ keeps them to be $\perp$.

Injected `groupInfo`. In case the environment $\mathcal{Z}$ makes `id` process an injected `groupInfo` g to create an external commit, the reduction $\mathcal{B}$ proceeds as follows:

First, the reduction $\mathcal{B}$ checks for a commit c that includes the matching `confTag`, i.e., $\mathsf{confTransHash} \in g$. If such commit c exists, then $\mathcal{B}$ further checks whether all other values included in g are equal to another `groupInfo` g' that was previously generated by $\mathcal{B}$. If so, g indicates an existing node, i.e., environment $\mathcal{Z}$ is re-sending the `groupInfo` g for c from another client. $\mathcal{B}$ proceeds as for an honest g and appends the external commit to c.

If the `confTag` corresponds to a commit c and a previously sent g', but any values other than signer and signature are different from that previous g', the reduction $\mathcal{B}$ treats it like any other injected g: if `id` accepts the message, $\mathcal{B}$ creates the new node with labels taken from `groupInfo` and the ratchet tree set to the public part of τ from `groupInfo`. Then, for any node of τ with a public key for which it has a secret key stored (in another ratchet tree or in ISK and SK), $\mathcal{B}$ copies the secrets into τ. Other unknown secrets remain $\perp$.

The reduction $\mathcal{B}$ then proceeds to create the external commit. If the external key is (gsd, u), which qualifies as a valid secret for GSD, $\mathcal{B}$ utilizes the oracle **Enc** to derive the external seed s. Otherwise, $\mathcal{B}$ performs the encryption itself.

This behavior only affects detached trees, as any injected `groupInfo` that does not correspond to an honest commit immediately creates a new detached root. Consequently, the proof for the main tree remains unaffected, and $\neg$**safe**(c) is addressed by case b) of ***state-directly-leaks**$(c, \texttt{id})$. For the adaptation of the proof concerning detached trees from Section D.5, refer to the explanation below.

External Commit Injected to Process. The reduction $\mathcal{B}$ constructs the new node including the initSecret as follows: On one hand, if the external-key, which encrypts the shared secret s_{exini} in the `extInit` proposal of the external commit, is a GSD node, i.e., neither $\mathcal{B}$ nor $\mathcal{Z}$ has knowledge of it, $\mathcal{B}$ uses the **Dec** oracle of GSD to retrieve the shared secret s_{exini}. On the other hand, if the external-key is a value, meaning that both $\mathcal{B}$ and $\mathcal{Z}$ know it, $\mathcal{B}$ directly decrypts s_{exini}. The external-key cannot be $\perp$, as it is not a key that the adversary can inject. From s_{exini}, $\mathcal{B}$ can then derive the new initSecret.

The only case where $\mathcal{B}$ will not be able to decrypt s_{exini} is if the `kem_output` stems from an honest external commit c' previously generated by $\mathcal{B}$ and copied by $\mathcal{Z}$ and whose appSecret is still a valid challenge. If an `id` accepts such a copied `kem_output`, $\mathcal{B}$ can use it to win the GSD-game similarly to a copied path secrets in commits:

1. Since the `kem_output` is copied, the ext-initSec is identical in both the injected external commit c and the honest external commit c'. At the same time, confTransHash must differ; otherwise, c and c' would be identical.
2. For $\mathcal{Z}$ to compute a valid `confTag` for the injected commit c, it must query the RO on H(confTransHash, `confKey`). From this query, $\mathcal{B}$ can extract the `confKey`. Since `confKey` is derived from (`joinerSec`), $\mathcal{Z}$ must also query RO on (`joinerSec`) to compute a valid tag.
3. For $\mathcal{Z}$ to compute the `joinerSec` for the injected commit c, it needs to query RO on Hash(ext-initSec, `commitSec`). From this query, $\mathcal{B}$ can extract the ext-initSec, which is shared with c', by searching the RO query history.
4. $\mathcal{B}$ can then corrupt the `commitSec` of c', which does not impact the challenge, and compute the `joinerSec` of c'. Using this `joinerSec`, the reduction $\mathcal{B}$ calculates the appSecret.

Thus, this case cannot occur.

The reduction $\mathcal{B}$ handles the `updatePath` of the commit the same way as a regular one, as the argumentation of an injected commit relies on calculating the initSecret of c from the RO calls and corrupting the initSecret of c' directly - regardless of its origin. Note that in the case of an injected external commit, the initSecret always retrieved by $\mathcal{B}$ directly via decryption. This does not affect the argument showing it is impossible to copy `updatePath` secrets from an honest commit.

The final case to consider is when both s_{exini} and ctxt are copied. If they are copied from different commits, the `confTag` can be used to break the challenge

for both commits in the same way as before. Figure 29 illustrates how $\mathcal{B}$ utilizes $\mathcal{Z}$ to win the GSD game in case of a copied ctxt.

If they are copied from the same commit but the confTransHash differs from the original commit (e.g., when the external commit is injected for a different party), the `joinerSec`, retrievable via the `confKey` RO call, are the same between the two commits. The reason behind is that the same ciphertext indicates same path secret and proposals, which further indicate the same `commitSec` . Combing the fact that s_{exini} is copied from the same commit, which yields the same initSecret, the `joinerSec` must also be same. Thus, this allows the reduction $\mathcal{B}$ to compute the appSecret to be computed from it.

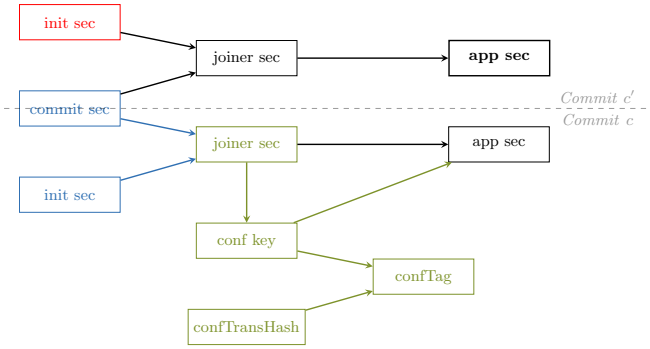


Fig. 29: Visualization of $\mathcal{B}$ using the `confTag` to win the GSD game for a commit c' if the environment $\mathcal{Z}$ successfully copies a ciphertext from a commit c' into an injected commit c. The commit secret is shared between both commits. In the first step (in ●), $\mathcal{Z}$ needs to call the random oracle RO to derive the `confTag` from the `confKey` and the confTransHash, which $\mathcal{B}$ now learns from the query. $\mathcal{Z}$ also needs to query the RO to derive `confKey` from `joinerSec`. $\mathcal{B}$ now knows `joinerSec` of c. In order to derive `joinerSec` for commit$c$, $\mathcal{Z}$ also needs to query Hash(`commitSec`, initSecret), which $\mathcal{B}$ now learns (in ●). $\mathcal{B}$ now knows the `commitSec` of commit c', and corrupts the initSecret of commit c' (in ●). $\mathcal{B}$ can now calculate appSecret of c'.

Proof of Indistinguishability. It remains to show that gsd-exp() $\implies \neg$**safe**(c). As a reminder, this is part **2.b.II** of the main proof. The subproof is structured as follows:

Part 2.b.II): Confidentiality when allowing injections

The reduction $\mathcal{B}$ succeeds in winning the GSD game for commit c' if the environment $\mathcal{Z}$ successfully copies a ciphertext from commit c' into an injected commit c. This can be formulated as follows:

a) Show that if the reduction $\mathcal{B}$ corrupts (c, u_{app}) then $\neg\textbf{safe}(c)$
b) Show that if the reduction $\mathcal{B}$ corrupts (c, u_{joi}) then $\neg\textbf{safe}(c)$
c) Show that if both $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ and $\mathsf{gsd\text{-}exp}((\texttt{Node[c].par}, u_{\text{exini}}))$ are true then $\neg\textbf{safe}(c)$. For this, consider all cases where $\mathsf{gsd\text{-}exp}((c, u_i))$ is true:
   c.1) During an exposure of an `id` that stores u_i
   c.2) When a node $\tau.v$ used in `*rekey-path` is exposed
   c.3) If the secret in $\tau.v$ is $\perp$, which is the case if:
      i) $\mathcal{Z}$ injects a new commit c' on behalf of `id`
      ii) the injected commit c' commits $\tau.v$ as an updated injected leaf
      iii) the injected c' commits $\tau.v$ as an injected key package

The proof is almost the same as in Step 1 (no injections). In the case that c is a regular commit, the proof is extended the same way as in step 2 of the original proof in [4], except for differences are c).

In case c), we still need to prove that $\mathsf{gsd\text{-}exp}((c, u_{\text{com}})) \implies \mathsf{know}(c, *)$, by considering the following three sub-cases, i.e., $\mathsf{gsd\text{-}exp}((c, u_{\text{com}})) = \texttt{true}$ is triggered c.1) during an exposure of an `id` that stores u_i's secret, and c.2) when a node $\tau.v$ used in `*rekey-path` is exposed, and c.3) if the secret in $\tau.v$ is $\perp$.

- c.1) is covered if `id` processed c and has not performed any action *that includes an UpdatePath*, which encompasses any form of external commit but excludes {`add`, `extAdd`}-only commits. Secondly, injecting a node from a previous, honest commit *or external commit* would allow the reduction $\mathcal{B}$ to use the `confTag` to win the GSD game.
- c.2) encompasses the subgraph G that is determined by actions adding and removing the node, *including external commits* and *excluding* {`add`, `extAdd`}*-only commits* that do not perform an `updatePath`.
- c.3) remains the same, as subcase i) $\mathcal{Z}$ injects a new commit c' on behalf of a party `id` into the subtree, encompasses both scenarios, where the injected c' is a regular or an external commit.

In the case that c is an external commit - if there is an edge between $(c, u_{\texttt{exini}})$ and (c, u_{joi}) - the proof proceeds as follows: $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ is only true in three cases

a) $\mathcal{B}$ corrupts (c, u_{app}) - same as in step 1
b) $\mathcal{B}$ corrupts (c, u_{joi}) - same as in step 1
c) if both $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ and $\mathsf{gsd\text{-}exp}((\texttt{Node[c].par}, u_{\text{exini}}))$ are true:

– For $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ the argumentation remains the same as for regular commit. Note that an injected external commit c' (case i)) hence implies $\mathsf{know}(c, *)$.
– In addition, $\mathsf{gsd\text{-}exp}(((\texttt{Node}[c].\mathsf{par}, u_{\text{exini}})))$ can also be true if injected: When the environment $\mathcal{Z}$ injecting the seed s_{exini} in the `extInit` proposal, the reduction $\mathcal{B}$ decrypts it and calculates u_{exini} itself. In the case of an injected external commit, case e) of ***can-traverse** immediately holds, and hence $\mathbf{safe}(c)$ is false.

In detached trees: Extension D.5 in main reference [4] concerning compromise in detached trees needs to be adapted with only minor changes. The proof shows that if an epoch in a detached tree does not include any exposed signing keys, it also does not include any exposed node keys. The first part shows that if $\tau.v$ has an exposed leaf key, then the signature key must also be leaked. This holds because a leaf key is only compromised if either the owning member is compromised, in which case the signature key spk is also marked as exposed, or the leaf is injected as part of a commit or update, in which case $\mathcal{F}_{\text{ECGKA}}$ also marks spk as exposed. A third, not explicitly named case is the use of an adversarially generated ratchet tree in a welcome or `groupInfo`. As each leaf node contains a signature using the nodes spk, the adversary must know the spk by corruption to produce the ratchet tree and spk must have been marked as exposed, otherwise $\mathcal{Z}$ could be used to break SUF-CMA.

The second part shows that for any inner node, assuming no node in its subtree is exposed, the leaf key pk is not exposed. The argument stays the same: 1) all signatures verify because at least one party has accepted the commit or welcome *or external commit or the* `groupInfo` g *to create an external commit.* 2) As the committer of an inner node $\tau.v$, spk cannot have been compromised (by assumption). $\tau.v$ cannot be maliciously generated, as the parentHash of $\tau.v$ is included and signed in the leaf of the committer. The only possible leakage is hence an encryption to an exposed key. However, this is impossible due to the induction hypothesis that states no lower nodes are compromised. The signed parent hash in $\tau.v$ guarantees here that the subtree of $\tau.v$ is the same as when $\tau.v$ was introduced into the tree.

Part 2.b.III) Allowing Bad Randomness *Proposals by Reference and by Value* For the `extAdd` proposal, $\mathcal{B}$ generates the key package using `id`'s spk. Similarly, by using the randomness provided by $\mathcal{Z}$ creates the proposal message p.

Commit by Reference and by Value. Given randomness provided by $\mathcal{Z}$, $\mathcal{B}$ computes the commit and required secrets as:

1. The reduction $\mathcal{B}$ executes `*rekey-path` to obtain the path secrets, `commitSec`, and commitContent. It then generates a valid confTransHash.
2. $\mathcal{B}$ creates a new `joinerSec`, which is a hash of the current initSecret and the newly computed `commitSec`, i.e., `joinerSec` = H (initSecret, `commitSec`, ...). If initSecret stores a GSD node u, $\mathcal{A}$ queries the $\mathsf{H}(u, u_{\text{ctr}}, \texttt{commitSec})$,

increments u_{ctr}, and uses the fresh, uncorrupted node as the `joinerSec`. Otherwise, if initSecret stores a value, $\mathcal{A}$ computes the `joinerSec` itself.
3. Using the secrets obtained, $\mathcal{B}$ executes the key schedule to compute the `confTag`. It then computes the commit message c and the vector of welcome messages `*welcome-msg` accordingly.

Validity of the Challenge. Throughout this study commits are not necessarily calculated using randomness. The reduction $\mathcal{B}$'s actions and the proof remain the same for commits that involve fresh randomness. Recall that the helper predicate ***secrets-replaced** captures the update of the path secret, except for the proposal-list in regular commit c is *non-empty* or only includes regular add or external self-add. Any commit c'' (except only regular add or external self-add proposals) with ***secrets-replaced**$(c'', \texttt{id})$ would hence replace secrets injected by randomness.

Now we will show with above changes, randomness corruption does not affect existing challenges, where **safe**(c) holds. With corrupted randomness the environment $\mathcal{Z}$ cannot derive the correct `epochSec` because the environment $\mathcal{Z}$ needs to query RO on `joinerSec` to compute `epochSec` = HKDF(`joinerSec`).

We adapt the proof that randomness corruption does not affect the validity of a challenge, and gsd-exp$(c) \implies \neg$**safe**(c) still holds. Due to the changes mentioned above, no additional corruption is performed by $\mathcal{B}$ on the `joinerSec`, case b) of gsd-exp holds as before.

The only case that is affected is case c). Here, the proof remains in agreement with [4], showing that gsd-exp$((c, u_{\text{com}})) \implies \neg$know$(c, *)$ also when allowing corruptions. We informed the authors of this bug, and they are in the process of updating the paper accordingly.

Randomness in Detached Trees. In addition to corruption and injection, a leaf or intermediate node $\tau.v$ may also be exposed if it is generated with bad randomness. The proof for detached trees relies on the fact that if any $\tau.v$ is exposed, a member's spk would likewise be revealed, thereby implying $\neg$**safe**(c). However, when committing with corrupted randomness alongside a fresh spk, as described in [4] (corresponding to an `updatePath` in ETK), only the old spk used to sign the commit is leaked, while the fresh spk remains secure. The spk, however, is exposed if it is registered with the AS while the randomness is compromised. The following hence does not contradict **safe**(c) through the case b) of ***state-directly-leaks**$(c, \texttt{id})$:

- The environment $\mathcal{Z}$ creates a detached root by inviting B to a false group, impersonating A
- B adds C and removes A
- Member B registers spk with $\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}$
- The environment $\mathcal{Z}$ corrupts B's randomness
- Member B performs a commit which updates its signature key to spk.

It does, however, contradict **safe**(c) through the case c) of ***state-directly-leaks**$(c,$ `id`$)$, as case a) of ***secrets-injected**$(c, \texttt{id})$ holds for commits with corrupted

randomness. Since the above scenario requires the node in the detached tree to have healed completely, safe is applicable in the same way as it is in the main tree. After such a heal has occurred, case b) of ***state-directly-leaks**$(c, \texttt{id})$ is always "overlayed" by other cases that hold after a compromise.

The proof from [4, Section D.5] that shows that from **safe**(c) follows that no node $\tau.v$ has an exposed public key pk needs to be adapted in the following way: For any leaf node with an exposed pk, either spk must have been exposed, *or the commit c introducing it to the tree was committed with compromised randomness*, both of which contradict **safe**(c). For any intermediate node, since spk is not exposed, so $\tau.v.pk$ is honestly committed (see [4] for details). *Additionally committing with bad randomness would have triggered case a) of* ***secrets-injected**$(c, \texttt{id})$*, contradicting the assumption that* **safe**$(c) = \texttt{true}$*, and in combination with* ***secrets-replaced**$(c, \texttt{id})$ *would have remained so until removed.* This means the secret key of a node is not chosen by $\mathcal{Z}$.

External Commit from `id`. Behaves similarly to a regular commit: Using the randomness provided by $\mathcal{Z}$, reduction $\mathcal{B}$ computes the external commit and the secrets in new commit node, as follows:

1. $\mathcal{B}$ uses the randomness provided by $\mathcal{Z}$ to execute `*rekey-path-upon-join`, obtaining all path secrets, the `commitSec`, and the intermediate commit packet. Then, it signs the commit packet using the `id` 's spk, along with $\mathcal{Z}$'s randomness, and sets the $\mathsf{confTransHash}$ accordingly.
2. A key distinction from a regular commit is that the $\mathsf{initSecret}$ for an external commit is determined by the client. As a result, corrupted randomness can expose this $\mathsf{initSecret}$. Reduction $\mathcal{B}$ computes the $\mathsf{ext\text{-}initSec}$ as follows:
   - If the external secret from the previous epoch stores a GSD node u, $\mathcal{B}$ queries **Enc**(u, u_{ctr}), corrupts u_{ctr}, and sets the external seed s_{exini}, increments u_{ctr}. $\mathcal{B}$ then uses this seed as the $\mathsf{initSecret}$ for the external commit.
   - If the external secret from the previous epoch stores a value (and is therefore known to $\mathcal{B}$), $\mathcal{B}$ computes the `joinerSec` directly.
3. Using the `joinerSec` and the $\mathsf{confTransHash}$ from previous steps, $\mathcal{B}$ runs the key schedule, computes the `confTag`, and finalizes the commit message c.

Proof for External Commit.

Case a) and case b) of the proof remains the same.

Case c): To show $\mathsf{gsd\text{-}exp}((c, u_{\text{com}})) \implies \neg\mathsf{know}(c, *)$ for external commits, we proceed similar as for regular commits. $\mathsf{gsd\text{-}exp}((c, u_i))$ can also be true if the secret in a ratchet tree node $\tau.v$ used in `*rekey-path-upon-join` stores a seed s generated during an action executed with bad randomness. Consider an external commit c' that inserts s into $\tau.v$. Unlike a regular commit, an external commit injects only the secrets it generates during the commit process. It does not, for instance, include an update proposal from another `id` or a corrupted key package. This injection is covered by case a) of ***secrets-injected**$(c, \texttt{id})$. Now consider the case where c itself is the external commit. Since the external committer utilizes the same public ratchet tree as a member performing a regular commit, the proof remains the same to regular commit.

J.3 Authenticity

Intuition. We prove that *H3* and *H4* are indistinguishable by demonstrating that the environment $\mathcal{Z}$ in *H4* cannot inject any message unless the statement **inj-allowed** is `true`. Note that the evaluation of **inj-allowed** only appears in **auth-invariant**. Thus, we need to prove that **auth-invariant** never triggers. We determine all cases where it would trigger as follows and prove that these events do not occur.

1. We show that if a forged signature event occurs, *H4* could be used to win the SUF-CMA game of the underlying digital signature.
2. We show that if a forged MAC event would occur, *H4* could be used to win the EUF-CMA game of the underlying MAC (and that the advantage is upper bounded by the GSD game).

Note that in ETK, messages are authenticated in two ways: First, public handshake messages include a MAC generated with the membership key derived from the epoch secret, confirming the message is from within the group. Private handshake and application messages on the other hand are encrypted with a group-only key and signed with the sender's leaf signature key, ensuring the sender's identity and message authenticity. Thus, the indistinguishability of *H3* and *H4* indicates that ETK guarantees authenticity.

Lemma 3. *If* Sig *is* SUF-CMA *secure,* MAC *is* EUF-CMA *secure, and* PKE *is* GSD *secure, then Hybrids H3 and H4 are indistinguishable, that is,* ETK *guarantees authenticity.*

Proof. Note that the hybrids *H3* and *H4* are identical unless in *H4* there exists an injected history graph node in $\mathcal{F}_{\text{ECGKA}}$ for which **inj-allowed** requires authenticity. More concretely, the hybrids *H3* and *H4* are identical unless the following happens:

– Event `Bad`: There exists a commit or proposal node with `stat` = `adv` and **inj-allowed**$(c, \mathsf{spk}) =$ `false` for its parent c and leaf nodes' spk corresponding some identity `id`.

Note that **inj-allowed**$(c, \mathsf{spk}) =$ `false` if spk in commit c is not exposed or the epoch key in c is not leaked to the reduction. We then can decompose the event `Bad` into the following two sub-events.

– Event $\texttt{Bad}_{\texttt{sig}}$: There exists a commit or proposal node with `stat` = `adv` and $\texttt{Node}[c].\mathsf{mem} \notin \mathsf{Exposed}$ for its parent c and creator `id`.
– Event $\texttt{Bad}_{\texttt{MAC}}$: There exists a commit or proposal node with `stat` = `adv` and $\neg$**know**$(c, '\,\mathsf{epoch}')$ for its parent c.

The proof is concluded by combining Lemma 4 and Lemma 5, where we respectively prove that the events $\texttt{Bad}_{\texttt{sig}}$ and $\texttt{Bad}_{\texttt{MAC}}$ will never happen except for negligible probability.

Lemma 4. *For any environment $\mathcal{Z}$, there exists a reduction $\mathcal{B}_{\mathtt{sig}}$ that succeeds in the* SUF-CMA *game with a probability that is only polynomially smaller than the probability of $\mathcal{Z}$ triggering* $\mathtt{Bad}_{\mathtt{sig}}$.

Proof. For any environment $\mathcal{Z}$, $\mathcal{B}_{\mathtt{sig}}$ emulates the functionalities and simulator by embedding its challenge spk as one of the public keys honestly created during the experiment. To emulate commits signed under ssk, $\mathcal{B}_{\mathtt{sig}}$ calls **Sign** oracle. When $\mathtt{Bad}_{\mathtt{sig}}$ occurs, reduction $\mathcal{B}_{\mathtt{sig}}$ stops the experiment and sends to its challenger to forgery consisting of the sig', $\mathtt{tbs}'$ from the injected node c'.

First assume c' is a commit node. We show that if $\mathtt{Bad}_{\mathtt{sig}}$ occurs and $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}[\mathtt{id}]$, then $\mathcal{B}_{\mathtt{sig}}$ wins.

1. We know sig' is a valid signature over $\mathtt{tbs}'$. The injected node was created when some party accepted c' which means that it verified sig' under $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}$.
2. Simulation differs from the experiment when $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}$ is corrupted, however this does not happen as the event guarantees $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem} \notin \mathsf{Exposed}$.
3. The reduction $\mathcal{B}_{\mathtt{sig}}$ has never queried **Sign** on $\mathtt{tbs}'$ for sig' to its signing oracle. Assuming $(\mathsf{sig}', \mathtt{tbs}')$ is the same as $(\mathsf{sig}^*, \mathtt{tbs}^*)$ queried by $\mathcal{B}_{\mathtt{sig}}$ to the sign oracle for c^*, where c' is injected and c^* is honestly generated by $\mathcal{B}_{\mathtt{sig}}$. By showing our assumption implies $c' = c^*$, we will achieve a contradiction. Now we will divide the proof for two, according to the type of the commit c'.
   - We first consider that c' is a regular commit. c' contains $(\mathsf{groupId}', \mathsf{epoch}', \mathsf{leafIdx}', '\mathtt{Commit}', C', \mathtt{confTag}', \mathsf{sig}', \mathtt{membTag}')$ and $\mathtt{tbs}' = (\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \mathtt{senderIdx}', '\mathtt{Commit}', C')$ and c^* contains the analogous values. Then c' and c^* can only differ on $\mathtt{confTag}' \neq \mathtt{confTag}^*$ or $\mathtt{membTag}' \neq \mathtt{membTag}^*$. Note that $\mathtt{membTag}' = \mathsf{MAC.tag}(\mathsf{membKey}', C')$ and that $\mathtt{membTag}^* = \mathsf{MAC.tag}(\mathsf{membKey}^*, C^*)$. According to our assumption $\mathtt{tbs}' = \mathtt{tbs}^*$ implies $C' = C^*$ and $\mathsf{groupCtxt}' = \mathsf{groupCtxt}^*$. Group context determines the epoch and key schedule. We have $\mathsf{membKey}' = \mathsf{membKey}^*$ and further $\mathtt{confKey}' = \mathtt{confKey}*$. Accordingly $\mathtt{membTag}' = \mathtt{membTag}^*$ and $\mathtt{confTag}' = \mathtt{confTag}^*$, which gives a contradiction.
   - Then we consider that c' is an external commit. c' contains two components: $\mathtt{framedContent} = (\mathsf{groupId}', \mathsf{epoch}', '\mathsf{new\_member\_commit}', '\mathsf{commit}', C', \mathsf{sig}')$ and $\mathtt{framedContentAuth}' = (\mathsf{sig}', \mathtt{confTag}')$ and c^* contains the analogous values. Moreover, $\mathtt{tbs}' = (\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \mathtt{senderIdx}', '\mathtt{Commit}', C')$ and $\mathtt{tbs}^*$ contains the analogous values. Similar to regular commit, c' and c^* can only differ on $\mathtt{confTag}' \neq \mathtt{confTag}^*$. By definition it holds that $\mathtt{confTag}' = \mathsf{Hash}(\mathsf{confTransHash}', \mathtt{confKey}')$ and $\mathtt{confTag}^* = \mathsf{Hash}(\mathsf{confTransHash}^*, \mathtt{confKey}^*)$. Since $\mathtt{confTag}$ is derived on the framed content, we have $\mathtt{confTag}' = \mathtt{confTag}^*$. However this gives us contradiction as c' is injected and c^* is honestly generated.

Lemma 5. *For any environment $\mathcal{Z}$, there exists a reduction $\mathcal{B}_{\mathtt{MAC}}$ and $\mathcal{B}_{\mathtt{PKE}}$ such that the probability that $\mathcal{Z}$ triggering $\mathtt{Bad}_{\mathtt{MAC}}$ is upper bounded by $p(\epsilon_{\mathsf{MAC}} + \epsilon_{\mathsf{PKE}})$, where p is a polynomial, ϵ_{MAC} is the advantage of $\mathcal{B}_{\mathtt{MAC}}$ against the security of MAC and ϵ_{PKE} is the advantage of $\mathcal{B}_{\mathtt{PKE}}$ against the security of* PKE.

Proof. For an injected packet, $\mathcal{Z}$ can trigger $\texttt{Bad}_{\texttt{MAC}}$ in the following cases. First, we consider the case where all receivers verify a MAC tag `confTag` but not `membTag`. The reduction $\mathcal{B}_{\texttt{MAC}}$ runs $\mathcal{Z}$ which emulates the UC experiment exactly, except that it uses its oracles in the EUF-CMA game of MAC, instead of using the MAC key membKey in epoch c. If $\texttt{Bad}_{\texttt{MAC}}$ occurs for parent node c and injected child c', reduction $\mathcal{B}_{\texttt{MAC}}$ outputs forgery $(\texttt{confTag}', \texttt{tbm}')$ where $\texttt{tbm}'$ denotes the content to be MACed.

1. The difference between the probability of $\texttt{Bad}_{\texttt{MAC}}$ occurring for c in the experiment emulated by the reduction $\mathcal{B}_{\texttt{MAC}}$'s and in Hybrid *H3* (or *H4*) is upper bounded by the advantage of the reduction $\mathcal{B}_{\texttt{PKE}}$ in the GSD game.
2. If $\texttt{Bad}_{\texttt{MAC}}$ occurs for c, then the reduction $\mathcal{B}_{\texttt{MAC}}$ wins with $(\texttt{confTag}', \texttt{tbm}')$.

For the first item, $\texttt{Bad}_{\texttt{MAC}}$ for c implies that $\textbf{safe}^*(c)$ is true until the event and no party in c is exposed. We can construct $\mathcal{B}_{\texttt{PKE}}$ implicit authentication, i.e, confidentiality proof. The reduction $\mathcal{B}_{\texttt{PKE}}$ embeds the challenge in membKey in c and proceeds the experiment as until $\texttt{Bad}_{\texttt{MAC}}$ for c occurs. Thus it has no effect on the probability of the event.

For the second item, we show that for valid tag ($\texttt{confTag}'$) that is verified by a party accepting the injected c', $(\texttt{confTag}', \texttt{tbm}')$ was not queried by the reduction $\mathcal{B}_{\texttt{MAC}}$ to the MAC oracle. Assume $(\texttt{confTag}', \texttt{tbm}') = (\texttt{confTag}^*, \texttt{tbm}^*)$, where $(\texttt{confTag}^*, \texttt{tbm}^*)$ was queried by the reduction $\mathcal{B}_{\texttt{MAC}}$ to the oracle when it honestly generated packet c^*. We need to prove that the reduction $\mathcal{B}_{\texttt{MAC}}$ has never queried $\texttt{tbm}'$ to its MAC oracle. Note that $\mathcal{B}_{\texttt{MAC}}$ has only queried other $\texttt{tbm}^*$ for other node c^* to its MAC oracle. Thus, for any $c' \neq c^*$, it always holds that $\texttt{tbm}' \neq \texttt{tbm}^*$. The proof for commits requiring a `membTag` can be demonstrated in a similar manner. External commits, on the other hand, do not include a `membTag`; thus, the proof remains consistent with the proof provided for `confTag` above.

Now, consider commits that include only `extAdd` proposals. Note that such commits do not contain a framed MAC tags, e.g., `confTag` or `membTag`. Consequently, the authenticity of these commits relies on the first part of **inj-allowed**, specifically the security of the Sig.

K Proof of Theorem 2

Theorem 2. Assume that PKE is IND-CCA secure and that Sig is SUF-CMA secure. The $\mathsf{ETK}^{\mathrm{PSK}}$ protocol securely realizes $(\mathcal{F}_{\mathcal{AS}}^{\mathcal{IW}}, \mathcal{F}_{\mathcal{KS}}^{\mathcal{IW}}, \mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}})$ in the $(\mathcal{F}_{\mathcal{AS}}, \mathcal{F}_{\mathcal{KS}}, \mathcal{G}_{\mathrm{RO}})$-hybrid model, where $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ uses the predicates **safe** and **inj-allowed** from Figure 7 and calls to HKDF.Expand, HKDF.Extract, and MAC functions are replaced by calls to the global random oracle $\mathcal{G}_{\mathrm{RO}}$.

Proof. For the proof of $\mathsf{ETK}^{\mathrm{PSK}}$, we retain the description of *H1* − *H4*. Since $\mathsf{ETK}^{\mathrm{PSK}}$ differs from ETK only regarding PSK injection, most of the proof is identical to the one of ETK in Appendix J. Hence, we will focus on stating additional requirements and checks in this section. We again prove the indistinguishability between every two adjacent hybrids by Lemma 6 (in Section K.1), Lemma 7 (in Section K.2), and Lemma 8 (in Section K.3).

K.1 Consistency and Correctness of $\mathsf{ETK}^{\mathrm{PSK}}$

Intuition. We prove the indistinguishability between *H1* ($\mathsf{ETK}^{\mathrm{PSK}}$) and *H2* by proving for every input operation that the outputs in $\mathsf{ETK}^{\mathrm{PSK}}$ and $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ of *H2* are the same. The **assert** statements in $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ of *H2* capture that if one party has processed a proposal, (external) commit, or welcome, all parties that process it in the future end up in the same shared group state. Thus, the indistinguishability between *H1* and *H2* indicates that $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ guarantees consistency.

Lemma 6. *Hybrids H1 and H2 are indistinguishable, that is,* $\mathsf{ETK}^{\mathrm{PSK}}$ *guarantees consistency.*

Regular (internal) Proposals. We consider two cases when $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ sends $(\texttt{Propose}, \mathsf{act})$, firstly $\mathsf{act} \in \{\texttt{up-spk}, \texttt{add-id}_t, \texttt{rem-id}_t, \texttt{psk-epoch-use}\}$, or secondly $\mathsf{act} \in \{\texttt{up-spk}, \texttt{rem-id}_t, \texttt{psk-epoch-use}\}$. In both cases, assert statements and consecutively the indistinguishability of the statements remain the same.

External Proposals. All the assert statements and the proof remain the same.

Commits. Additionally, for each PSK proposal contained in $\boldsymbol{p}$, $\mathcal{S}_2$ determines the commit of the history graph the injected PSK pertains to. This cannot be derived from the protocol directly, hence $\mathcal{S}_2$ needs to maintain a separate mapping from commit to PSK. This is possible since $\mathcal{S}_2$ has access to the whole history graph. $\mathcal{S}_2$ returns these c in the order of the PSK proposals as $\boldsymbol{c_{psk}}$. If any pskId appears twice in a commit, $\mathcal{S}_2$ returns $\perp$ instead. Then, $\mathcal{F}_{\mathrm{ECGKA}^{\mathrm{PSK}}}$ runs ***fill-props**. For every proposal p in the input $\boldsymbol{p}$ without a node, $\mathcal{S}_2$ sets orig and act according to p. This is always possible, as the basic check in ***apply-props** by $\mathsf{ETK}^{\mathrm{PSK}}$ guarantees that p is well-formed.

Indistinguishability of Output: An additional **req** statement may have the Commit interface return $\perp$:

– **req** $\texttt{id} \in \texttt{Node}[\texttt{Prop}[p].c_{\text{psk}]}.\texttt{has\_psk}$: ***fixate-psk-refs** guarantees that every PSK proposal refers to a valid c_{psk}. This **req** statement will fail if `id` did not perform a Process of c_{psk} correctly (yet) or the same `pskId` appears twice. Due to the indistinguishability of Process between ETK and $\mathcal{F}_{\text{ECGKA}}$, the same holds for ETK^{PSK}, and the PSKs of c_{psk} would have not been added to `id`'s `psk_store`. Hence ETK^{PSK} would have failed in `*derive-psk-secret`.

Assert Statements: The commit interface includes the following additional **assert** statements:

1. **assert** res $\neq \perp$: holds if ***fixate-psk-refs**$((\texttt{Prop}[p] : p \in \boldsymbol{p}), \boldsymbol{c}_{\text{psk}})$ runs through. It would fail either because $\mathcal{S}_2$ could not find a suitable c_{psk} or because the epoch does not match. The latter must hold because of the direct correspondence of $\texttt{Node}[c].\texttt{epoch}$ in $\mathcal{F}_{\text{ECGKA}}$ and the epoch of c and the way that $\mathcal{S}_2$ choses c_{psk}. The former fails only if there is no `preSharedKeyId` for the epoch in the `psk_store` in which case, due to the direct correspondence of $\texttt{Node}[c].\texttt{has\_psk}$ in $\mathcal{F}_{\text{ECGKA}}$ and the commit's existence in the `psk_store` in ETK^{PSK}, both ***req-correctness** will fail and ack will be `false`.
2. **assert** ***valid-spk** remains the same.
3. **assert** mem $\neq \perp$ will succeed in the following cases.
   – $p = p_{up} + p_{rem} + p_{add} + p_{psk}$ includes the additional p_{psk}. Because of the direct correspondence of $\texttt{Prop}[p].\texttt{act}$ and the action of p, ETK^{PSK} performs the same check and $\mathcal{S}_2$ will return $ack = \texttt{false}$. $\texttt{Prop}[p]$ accurately reflects the action because either p is created by $\mathcal{S}_2$ according to act, or act is set by $\mathcal{S}_2$ extracting act from p.
   – $\texttt{Node}[c].\texttt{epoch} = \texttt{epoch} \wedge \texttt{id} \in \texttt{Node}[c].\texttt{has\_psk} \vee \nexists(\texttt{id}, *) \in G$ states that at least one PSK for each PSK proposal must be in `id`'s possession. This is checked by ETK.
4. **assert** ***valid-successor** has the additional case:
   – $\texttt{Node}[c].\texttt{epoch} = \texttt{Node}[\texttt{Ptr}[\texttt{id}]].\texttt{epoch}$: if $\texttt{Node}[c]$ is a detached root, the epoch was read out of c' and must match, as it is included in the confTransHash, based on which $\mathcal{S}_2$ chooses the root. If it was a pre-existing node, the confTransHash also matches (else c would not be equal to a previous node), and hence the same holds.
5. **assert** $\texttt{Wel}[\texttt{vec\_w}] \in \{\perp, (c, c_{\text{psk}})\}$ guarantees that the vector of welcome message is different for each c. This is achieved by including the `confTag` over the transcript hash in the Welcome's `groupInfo`. In Commit, $\texttt{Wel}[\texttt{vec\_w}]$ can previously only have been set to c and following the argumentation for `Process`, the probability for another $\boldsymbol{p}$ matching c is low, and hence `psk` is also the same according to the standard hybrid argument. The only other option for setting $\texttt{Wel}[\texttt{vec\_w}]$ is in `Join`. There, c and `psk` are set to the value provided by $\mathcal{S}_2$, which retrieves c as the only fitting commit by comparing the confTransHash, or it is set to a detached node. In the latter case, c is providing the link to the detached node, hence ***attach** would have set $\texttt{Wel}[\texttt{vec\_w}]$ to $(c, \boldsymbol{c}_{\text{psk}})$.

Process. $\mathsf{ETK}^{\mathrm{PSK}}$ distinguishes the processing of external commits and regular commits by `id` handing in either an empty vector or the full proposal vector. Both for regular and (for both cases of) external commits, the simulator $\mathcal{S}_2$ retrieves c's epoch, orig', spk' from c. $\mathcal{S}_2$ also extracts the proposal vector $\boldsymbol{p}$ from c and the vector $\boldsymbol{c_{psk}}$ consisting of the commits of all injected PSKs or $\perp$ if any of the `pskId` appear twice.

Indistinguishability of Output:

For the regular commits, in addition to cases mentioned in ETK, the `Process` returns $\perp$ in the following cases:

- **req** $\mathsf{res} \neq \perp$: holds if **req** ***fixate-psk-refs**(`Prop`$[p] : p \in \boldsymbol{p}$) runs through. This is the case for a valid commit, where c_{psk} has been set before. Otherwise, **req** ***req-correctness** would have failed, and ETK would either also have failed or provided the necessary correct commits in $\boldsymbol{c_{psk}}$.
- **req** (`id` $\in$ `Node`$[c_{\mathrm{psk}}]$.`has_psk` $\vee \nexists(\mathtt{id}, *) \in G$) holds as for `Commit`.

For the external commit we consider the following additional cases where `Process` returns $\perp$:

- **req** `ExtCommitProps` $\neq \perp$ holds as for `Commit`: if ***fixate-psk-refs** fails because of an unseen commit, ***req-correctness** would have failed, and ETK would have failed if it did not have the appropriate $\boldsymbol{c_{psk}}$.
- **req** `id` $\in$ `Node`$[c_{\mathrm{psk}}]$.`has_psk`: due to the direct correspondence between `has_psk` and the PSK being in `id`'s `psk_store` in ETK, this fails exactly if ETK fails.

Assert statements : The `Process` interface includes the following **assert** statements for regular commits:

- **assert** $\mathsf{mem} \neq \perp$ means `*apply-props` failed. If c already existed, then `*apply-props` must have been run on the exact same parameters before, especially on $\mathsf{Ptr}[\mathtt{id}]$ (or ***req-correctness** would have failed while checking `Node`$[c]$.$\mathsf{par} = \mathsf{Ptr}[\mathtt{id}]$) and passed. Else, ***req-correctness** would have failed. If `Node`$[c] = \perp$ or c were malformed according to ETK (i.e. a proposal removing the committer), $\mathcal{S}_2$ would return $ack =$ `false` and `Process` would unwind. This means that c either corresponds to a correct detached root or a correct new node. It is left to show that every single **req** in `*apply-props` is either fulfilled or $\mathcal{S}_2$ would have returned $ack =$ `false`, unwinding the function. This holds for the same reason as for `Commit`, the only exception being PSK proposals, since `Node`$[c]$.$\mathsf{epoch} = \mathsf{epoch} \wedge \mathtt{id} \in$ `Node`$[c]$.`has_psk` $\vee \nexists(\mathtt{id}, *) \in G$ might hold for the committer but not the processing party. In such a case of a valid commit with unprocessable PSK proposals, ***req-correctness** would have failed and $\mathcal{S}_2$ would have returned $ack =$ `false`. The same holds for an injected commit.

In the case of an external commit, `*process-ec`(c) contains the following **assert** statements:

– **assert** mem $\neq \bot$ for both $\texttt{Node}[c] = \bot$ and $\texttt{Node}[c] \neq \bot$: holds if `*apply-props` runs fully. The only additional case that results in the unwinding of `*process-ec` is as follows:
  - $\exists \texttt{Node}[c].\texttt{epoch} = \texttt{epoch} \wedge \texttt{id} \in \texttt{Node}[c].\texttt{has\_psk} \vee \nexists(\texttt{id}, *) \in G$ states that at least one PSK for each PSK proposal must be in `id`'s possession. This is guaranteed by the **req** in the line before.

Join. The `Join` returns $\bot$ only in the following case:

– **req** $\texttt{id} \in \texttt{Node}[c_{\text{psk}}].\texttt{has\_psk}$: if $\bot$ were invalid, $\mathcal{S}_2$ would have returned $ack = \texttt{false}$. Due to the direct correspondence of `has_psk` and the PSK being in `id`'s PSK store, this **req** fails exactly if ETK does.

Assert Statements: The proof for the **assert** statements of the `Join` interface remains the same.

Consistency: The proof showing that every joiner `id` that executes `Join` with a vector of welcome message `vec_w` and every group member $\texttt{id}'$ that executes `Process` with a commit c' must end up in the same state remains the same.

External Commit. After $\mathcal{S}_2$ constructs the external commit and generates all required parameters and variables and makes the necessary checks as described in ETK proof, it additionally sends the vector $\boldsymbol{c_{psk}}$ consisting of the commits of all injected PSKs or $\bot$ if any of the `pskId`'s appear twice.

Indistinguishability of Output:
$\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ returns c and g' generated according to the protocol. The `ExtCommit` returns $\bot$ additionally in the following cases:

– If **req** $\texttt{ExtCommitProps} \neq \bot$ fails then because $\mathcal{S}_2$ could not supply suitable c_{psk} or there was the same `pskId` twice. In this case, $\mathsf{ETK}^{\text{PSK}}$ would also fail.
– **req** $\texttt{id} \in \texttt{Node}[c_{\text{psk}}].\texttt{has\_psk}$: Due to the direct correspondence of `has_psk` and the PSK being in `id`'s `psk_store`, this **req** fails exactly if $\mathsf{ETK}^{\text{PSK}}$ does.

Assert Statements: The `ExtCommit` interface includes changes only for the following **assert** statements:

– **assert** mem $\neq \bot \wedge (\texttt{id}, \mathsf{spk}) \in$ mem: is given as `*apply-props` executed on any $\boldsymbol{p}$ contains at least the committer (in this case `id`) when run successfully. It runs successfully because:
  - the requirements on `props` of `*apply-props` are fulfilled through the construction in `ExtCommit`.
  - **req** $(\texttt{id}_c, *) \in G$ (resync) or $(\texttt{id}_c, *) \notin G$ (external commit): is equally covered by ***req-correctness** in combination with the $\mathsf{ETK}^{\text{PSK}}$ execution.
  - **req** $\texttt{id}_s = \texttt{id}_t = \texttt{id}_c$ and $\texttt{id}_s \in G$ is fulfilled through the construction in `ExtCommit` and the previous **req**.

- $\exists \texttt{Node}[c].\mathsf{epoch} = \mathsf{epoch} \wedge \texttt{id} \in \texttt{Node}[c].\texttt{has\_psk} \vee \nexists(\texttt{id}, *) \in G$ states that at least one PSK for each PSK proposal must be in id's possession. This is guaranteed by the **req** in the line before.

– ***consistent-ext-comm**() holds for a similar reason as for proposals and commits. An external commit does not include a membership tag, but it does include the confirmation tag, a MAC over the GroupContext's confirmed transcript hash, which provides the same uniqueness guarantee of a commit c. $\texttt{Node}[c].\mathsf{pro}$ and $\texttt{Node}[c].\mathsf{par}$ will hence only be set to the proposals included in c and the parent of c included in the confTransHash, or else to $\perp$ in case of a detached root. As a consequence of these two properties, $\texttt{Node}[c].\mathsf{mem}$ will always be set to the same membership set handed to ***consistent-ext-comm**. As ExtCommitProps is directly constructed from $\boldsymbol{p}$, as a consequence, all its parameters are also equivalent, the only exception being c_{psk}, whose injection, however, is encoded in the confTransHash.

K.2 Confidentiality of $\mathsf{ETK}^{\text{PSK}}$

Lemma 7. *If* PKE *is* GSD *secure, then Hybrids H2 and H3 are indistinguishable, that is,* $\mathsf{ETK}^{\text{PSK}}$ *guarantees confidentiality.*

Proof. We show the indistinguishability by hybrid games. More concretely, for any $i > 1$, we define two adjacent sub-hybrids Hs_{i-1} and Hs_i that behave differently only in epoch i: Hs_{i-1} sets the epoch secret in epoch i according to $\mathcal{F}_{\text{ECGKA}}$ (depending on whether **safe**(c) holds), while Hs_i sets is according to the simulator $\mathcal{S}$. Further, we reduce the indistinguishability between every two adjacent hybrid games to the GSD security of the underlying PKE. More concretely, we show that if the environment $\mathcal{Z}$ can distinguish between a random key and a simulator chosen *application* key for c with **safe**(c) = true, the reduction $\mathcal{B}$ (i.e., the adversary that aims to break GSD game) can use the environment $\mathcal{Z}$ to win the GSD game with $\mathsf{gsd\text{-}exp}((c, u)) = \texttt{false}$ for some associated secret u.

The individual confidentiality proof steps of the $\mathsf{ETK}^{\text{PSK}}$ are organized as follows:

Proof part 2.b): Confidentiality

Show indistinguishability:

I) when not allowing any injections or corruption of randomness by reasoning through all possible cases of $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ (Section K.2):
   (a) the corruption of (c, u_{app})
   (b) the corruption of both (c, u_{joi}) and (c, u_{psk}) .
   (c) if all three $\mathsf{gsd\text{-}exp}$ $((c, u_{\text{com}}))$ and $\mathsf{gsd\text{-}exp}$ $((\text{Node}[c].\text{par}, u_{\text{ini}}))$ and $\mathsf{gsd\text{-}exp}$ $((c, u_{\text{psk}}))$ are true.

II) when allowing injections for cases (a) - (c) (Section K.2).

III) when allowing the corruption of a client's randomness for cases (a) - (c) (Section K.2).

Here we mostly note down the psk injection effect on the proof as many proof steps identical to Section J.2

Part 2.b.I) No Injection or Corrupted Randomness Allowed

Here in case of neither malicious injection nor randomness corruption is allowed, we prove by reduction that $\mathsf{gsd\text{-}exp}((c,u)) \implies \neg\mathbf{safe}(c)$. Reduction $\mathcal{B}$ has the following modifications on the code of the functionalities and the simulator in the context where no injection and no corrupted randomness are permitted.

Following this, we systematically examine each possible case of $\mathsf{gsd\text{-}exp}((c,u))$, corresponding to the three cases specified above, and illustrate how our proof captures the requirements of the safe predicate.

As described in the *gsd* experiment explanation; psk injection affects our model, for $\{\texttt{add}, \texttt{extAdd}, \texttt{psk}\}$-only commits, no `updatePath` is executed, and all of `id` 's secrets remain the same. This is addressed in $\textbf{*secrets-replaced}(c, \texttt{id})$ with color coding. Different than ETK in Section J.2, for no `updatePath` is executed case, we will also consider `psk` proposals. To this end, we also consider the commits including `add`, `extAdd`, `psk` proposals only, as well as those that involve any combination of these proposal types.

From draft 12 to RFC: All the changes related to leaf key separation, welcome messages and init key usage are also valid for $\mathsf{ETK}^{\mathrm{PSK}}$. Additionally:

- Proposal `psk`: Whenever $\mathcal{Z}$ chooses an `pskIds`, the reduction $\mathcal{B}$ stores `pskIds` in the new proposal node.
- Applying Proposals: Additionally, for each `psk` proposal with list $\texttt{pskIds} = \{\texttt{pskId}_1, \texttt{pskId}_2, ...\}$ (recall that the reduction has already stored `pskIds` in the corresponding node), the reduction $\mathcal{B}$ recursively queries the **Join-Hash** oracle with input $(\mathsf{temp}_i, \texttt{pskId}_i, \mathsf{temp}_{i+1}, \mathsf{lbl} = \mathsf{derive\text{-}psk})$ for $i \geq 1$, where $\mathsf{temp}_i = 0$. Note that the secret included in the node, which is derived from the last **Join-Hash** query, is indeed the `psk-secret` of the `psk` proposal. We denote this node by u_{psk}.
- Commits by reference: Compared with the ETK case in Section J.2, the `*rekey-path` operation is *not* executed if the proposal list in a regular commit satisfies the condition that it contains only `add`, `extAdd`, or `psk` proposals.

  On the other hand, if the commit includes any `psk` proposal, the reduction $\mathcal{B}$ emulates the `*derive-keys` as follows:
  1. *Add vertices to the GSD graph:* Additionally the reduction $\mathcal{B}$ creates a $u_{\text{joi-psk}}$ vertex by querying the **Join-Hash** oracle with input $(u_{\mathrm{joi}}, u_{\mathrm{psk}}, u_{\text{joi-psk}}, \mathsf{lbl} = \mathsf{joi\text{-}psk})$. Afterwards, the reduction $\mathcal{B}$ generates vertices u_{app}, u_{mem}, u_{conf}, u_{ini}, u_{ext}, u_{psk} by querying the **Hash** oracle as in [4], but with inputs $(u_{\text{joi-psk}}, u_{\mathrm{lbl}}, \mathsf{lbl})$ for $\mathsf{lbl} \in \{\mathsf{app}, \mathsf{mem}, \mathsf{conf}, \mathsf{ini}, \mathsf{ext}, \mathsf{psk}\}$.
  2. *Create the packet:* To complete the set of values needed to compute the commit packet, the reduction $\mathcal{B}$ additionally has to corrupt u_{psk}.
  3. *Create the welcome message:* The welcome message is a component of the welcome vector, consisting of three parts. The final part, `wel_type`,

indicates whether the message contains a single welcome message or multiple welcome messages. In the first part, for each new member $\texttt{id}_t$, the encryptions of `joinerSec` and $\texttt{id}_t$'s `pathSec` together with `pskIds` under the init-key in $\texttt{id}_t$'s key package (obtained from KS by the party adding $\texttt{id}_t$). Let u_i be the GSD vertex corresponding to the `pathSec` sent to $\texttt{id}_t$. If $\texttt{id}_t$'s init-key is of the form (gsd, u)[12], $\mathcal{A}$ obtains the encryptions by creating encryption edges from u to u_{joi} and from u to u_i. Otherwise, it corrupts u_i and u_{joi} encrypts the values itself. The second part of the welcome message contains the encryption of `groupInfo`, under the `welcome_secret` which is simply the combination of `joinerSec`, `psk-secret` (derived from `pskIds`) and the vector of welcome messages. To compute the second part of the vector of welcome messages, the reduction $\mathcal{B}$ creates the vertex u_{wel} by querying **Hash** oracle with input $(u_{\text{joi-psk}}, u_{\text{wel}}, \mathsf{wel})$. The reduction $\mathcal{B}$ gets the respective secrets by corrupting corresponding vertices.

– Expose: Additionally, the state of `id` includes `psk-secret`, as specified by ***update-stat-after-exp**.

As the basis for the proof, considering the key derivation that also considers the `psk` key material illustrated in Figure 2, $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ can only hold true in one of the following cases:

2.b.I.a) The reduction $\mathcal{B}$ corrupts the vertex (c, u_{app}). This case identical to ETK.

2.b.I.b) The reduction $\mathcal{B}$ corrupts the vertices (c, u_{joi}) and (c, u_{psk}) to ultimately gain access to the application secret. This happens if and only if the reduction $\mathcal{B}$ computes a vector of welcome messages for $\texttt{id}_t$ using an exposed key bundle. Upon exposure of the key bundle, the initial key also becomes corrupted along with u_{psk} (since we are considering only the resumption `psk`'s). So the mechanism remains as previously described.

2.b.I.c) This case considers the following three elements to be true: $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$, $\mathsf{gsd\text{-}exp}((\texttt{Node}[c].\mathsf{par}, u_{\text{ini}}))$, and $\mathsf{gsd\text{-}exp}((c, u_{\text{psk}}))$. We need to show that $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ implies $\mathsf{know}(c, *)$. This implication follows by case d) of ***can-traverse**. Here, $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ is only possible in one of the following cases that are identical to ETK.

1) the reduction $\mathcal{B}$ corrupts path secrets u_i, and uses them to compute $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$
2) the reduction $\mathcal{B}$ calls the **Enc** oracle to encrypt path secrets under relevant keys
3) after the commit is processed, the reduction $\mathcal{B}$ exposes a party `id` that holds u_i's secret

Additionally the reduction $\mathcal{B}$ needs to assure $\text{Node}[c].\texttt{psk} \neq \texttt{good}$. This only possible if the reduction $\mathcal{B}$ corrupts some `id` who owns the `psk` (not necessarily $\texttt{id}_t$ that commits c) or exposes `pskId` that generates the `psk-secret` during the commit.

[12] Following [4], a secret key of the form (gsd, u) means that it is unknown to the environment $\mathcal{Z}$.

Considering case 3) any actions of `id` requiring `updatePath` removes u_i's secret from its state, thus, the proof remains the same. For $\{\texttt{add}, \texttt{extAdd}, \texttt{psk}\}$-only commits, since no `updatePath` is executed, all of `id`'s secrets remain the same. Thus, if `id` is corrupted, $\mathsf{know}(c, \texttt{id})$ is true until an `updatePath` issued. We complete the proof by showing that for cases 1) and 2) the combination of $\mathsf{gsd\text{-}exp}((\texttt{Node}[c].\mathsf{par}, u_{\text{ini}}))$ and $\mathsf{gsd\text{-}exp}((c, u_{\text{psk}}))$ implies $\mathsf{know}(c, \text{'epoch'})$. For these cases the proof follows the same pattern as the encapsulating proof for u_{app}, as both u_{ini} and u_{psk} have the exact same lifetime as its epoch's u_{app}. This is because the resumption `psk`'s with *application* usage are generated from each epoch secret. The proof's case c) hence recursively proceeds until at some point, $\texttt{Node}[c].\mathsf{pars} = \mathsf{root}_x$. At that point, case d) of $\textbf{*can-traverse}(c)$ in $\textbf{safe}(c)$ becomes true.

External Commit. The proof is identical to that of ETK presented in Section J.2, with the exception of a minor modification in Case 2.b.I.c). Figure 30 visualizes the GSD game with the effect of a `psk` injection. There is an edge between (c, u_{exini}) and (c, u_{joi}), meaning that c is an external commit. As the `*rekey-path-upon-join` proceeds in the same way as `*rekey-path` in a regular commit, the reasoning for $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ implying $\mathsf{know}(c, *)$ also remains the same. The only change occurs when the combination of $\mathsf{gsd\text{-}exp}((\texttt{Node}[c].\mathsf{par}, u_{\text{exini}}))$ and $\mathsf{gsd\text{-}exp}((c, u_{\text{psk}}))$ implies $\mathsf{know}(c, '\text{epoch}')$.

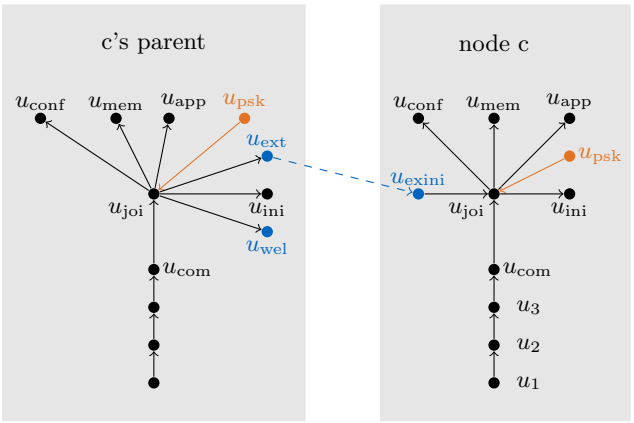


Fig. 30: GSD game graph of an external commit, following an image from [4]. Each circle represents a node in the GSD graph. Dashed arrows mark an encryption of the sink using the key of the source, while bold arrows mark a Hash call on the source. The additionally added nodes of the graph are marked in ●. The external secret encrypts the committer-generated seed s_{exini} that is used as the init secret of epoch c.

Similar to the analysis for regular commits, we consider two cases: i) u_{exini} is compromised together with u_{psk} when a client `id` is exposed while processing the

commit, in which case **safe**(c) does not hold, or ii) the reduction $\mathcal{B}$ calculates u_{exini} and u_{psk} itself if the seed s_{exini} and s_{psk} are exposed. Case i) happens if an `id`'s key bundle is exposed during the commit processing, which again implies **safe**(c) is false, while case ii) happens if it is encrypted under a node whose private key is compromised. The seed s_{exini} is encrypted under the key from the u_{ext} node of epoch $\texttt{Node}[c].\texttt{par}$. A u_{ext} node is exposed in the exact same cases as a u_{app}, and as in a commit, and the the u_{psk} has the same life time as u_{app}, proof continues recursively until c is a root and hence **safe**(c) is false.

Part 2.b.II) Allowing Injections In the case of allowing the injection of malicious injections, additional to the capabilities we mentioned in Section J.2, including `psk` injection, the functionalities and the simulator incorporate the following adversarial modifications:

From draft 12 to RFC

- Injected proposals: Additionally, in the case of `psk` proposal, the reduction $\mathcal{B}$ stores `pskIds` included in the `psk` proposal in a new proposal node.
- Commits injected to process: In the case that $\mathcal{Z}$ makes `id` process an injected commit c' from a party $\texttt{id}_c \neq \texttt{id}$, reduction $\mathcal{B}$ attempts to build new commit node's state. For {`add`, `extAdd`, `psk` }-only commits, the reduction $\mathcal{B}$ only applies the proposals, as these commits do not require rekey operation. For other commits along with the ciphertexts `ctxt` that `id` would decrypt with keys in the ratchet tree, the reduction $\mathcal{B}$ proceeds as follows:

  i-ii) remain the same.

  iii) if the `ctxt` is copied from an honest commit that was generated earlier by $\mathcal{B}$, for which the GSD node associated to both `appSecret` is still a valid challenge, we must show that `id` accepting such a commit c would enable $\mathcal{B}$ to succeed in the GSD game. The trick in this step is to reduce the forgery of the confirmation tag `confTag` in the commit c to the GSD security of the underlying PKE. Let $\mathcal{B}$ challenge the GSD node u and extract the correct seed from $\mathcal{Z}$'s random oracle calls as follows:

  1. Note that $\mathsf{appSecret} = \mathsf{Hash}(\texttt{joinerSec}, \texttt{psk-secret}, .)$ and $\texttt{joinerSec} = \mathsf{Hash}(\mathsf{initSecret}, \texttt{commitSec}, .)$, where HKDF is modeled as RO. Moreover, note also that `commitSec` must be the same in c and c' for the shared `ctxt` to be accepted.
  2. The reduction $\mathcal{B}$ needs to extract the `joinerSec` and `psk-secret` of c'. To this end, the reduction $\mathcal{B}$ proceeds as follows: First, recall that the `confTag` is a MAC of `confKey` and `confTransHash`. The only way for $\mathcal{Z}$ to compute a valid tag is to query RO on (`confKey`, `confTransHash`). The reduction $\mathcal{B}$ can search the RO query history and extract `confKey`. Second, recall that `confKey` is derived by applying HKDF to `joinerSec` and `psk-secret`. The environment $\mathcal{Z}$ needs to query RO on $(\texttt{joinerSec}, \texttt{psk-secret})$ for deriving `confKey`. The reduction $\mathcal{B}$ can then extract `joinerSec` and `psk-secret` of c' in a similar way.
  3. This case also remains the same.

– Injected vector of welcome messages vec_w: When $\mathcal{Z}$ injects a welcome message $w = (\texttt{encGroupSec}, \texttt{encGroupInfo})$ in the welcome vector vec_w, $\mathcal{B}$ does as follows:

1. Similar to proof of ETK, the reduction $\mathcal{B}$ searches for a key package kp such that $\mathsf{SK}[\texttt{id}, \texttt{kp}] \neq \perp$ and $\mathsf{ISK}[\texttt{id}, \texttt{kp}] \neq \perp$ and $\mathsf{H}(\texttt{kp})$ matches an entry $e \in \texttt{encGroupSecs}$ and aborts if no such kp exists.
2. If e is copied from a vector of welcome messages vec_w generated by $\mathcal{B}$ while creating a commit node c and ISK[id, kp] is a GSD node, to win the game, reduction $\mathcal{B}$ proceeds as follows: $\mathcal{B}$ needs to distinguish whether the injected welcome message is associated with the commit c that the e stems from. To this end, $\mathcal{B}$ compares the encGroupSec. Since e is copied, the joinerSec and pskIds underlying different welcome messages that include the same e must be the same. Moreover, psk-secret must be the same as is it derived from pskIds. Furthermore, an equal encGroupInfo implies an equal confTransHash. In this case, $\mathcal{B}$ moves id to c.
   If the encGroupInfo in the vector of welcome message vec_w is different from the one derived by the reduction $\mathcal{B}$ for c, that means that e was copied but used to create the welcome for a non-existent commit c'. This is not possible as it would allow $\mathcal{B}$ to win the GSD game: In order to construct a encGroupInfo that is decryptable using the secrets stored in the encGroupSecs, the environment $\mathcal{Z}$ must be in possession of the welcome_secret (as $\mathcal{Z}$ cannot copy encGroupInfo from c). In order to derive the welcome_secret, $\mathcal{Z}$ must derive it from the joinerSec and psk-secret using the RO calls. The reduction $\mathcal{B}$ can extract joinerSec and psk-secret by checking queries that have been sent to RO one by one as follows: First, $\mathcal{B}$ attempts to parse the input of RO into joinerSec and psk-secret candidates. If the parse fails, $\mathcal{B}$ directly omits the following steps and attempts for the next RO input. Second, the reduction $\mathcal{B}$ derives welcome_secret and confKey from the joinerSec and psk-secret candidates, respectively. Third, the reduction $\mathcal{B}$ derives welcome_key and welcome_nonce from welcome_secret. Fourth, the reduction $\mathcal{B}$ decrypts encGroupInfo to recover groupInfo. Fifth, the reduction $\mathcal{B}$ parses groupInfoTBS from groupInfo and further confTag from groupInfoTBS. Finally, the reduction $\mathcal{B}$ verifies the confTag by using the confKey. If no error occurs, then reduction $\mathcal{B}$ found the correct joinerSec and psk-secret. If any error occurs, then the reduction $\mathcal{B}$ continues to check the next RO query. Note that the environment $\mathcal{Z}$ can send only polynomial queries to RO. The checks that $\mathcal{B}$ needs to execute must also be in polynomial time.
   From joinerSec and psk-secret, the reduction $\mathcal{B}$ can calculate the appSecret for the valid GSD challenge node c and hence win the game. Copying e is hence not possible.
3. Else, if e has not been copied from any vector of welcome messages generated by $\mathcal{B}$, the reduction $\mathcal{B}$ can simply obtain the encrypted joinerSec, pskIds and pathSec either by using the secret in $\mathsf{ISK}[\texttt{id}, \texttt{kp}]$, if it has been compromised and therefore known by $\mathcal{B}$, or otherwise by querying $\mathcal{B}$'s GSD

decryption oracle **Dec**. Related to `pskIds`, we consider two cases. First, all `psk` in the list `pskIds` have been exposed. In this case, the reduction $\mathcal{B}$ also knows them and therefore can derive `psk-secret` by itself. By using the `joinerSec` and `psk-secret`, the reduction $\mathcal{B}$ can decrypt and verify the `encGroupInfo` in the vector of welcome messages. If the verification passes, then $\mathcal{B}$ further checks whether the decrypted `groupInfo` indicates an existing node c (by comparing the $\mathsf{confTransHash} \in$ `groupInfo`) or a new node c'. If `groupInfo` indicates an existing node c, then $\mathcal{B}$ moves `id` to c. Otherwise, `groupInfo` indicates a new node c'. In this case, $\mathcal{B}$ creates c' with labels taken from `groupInfo` and the ratchet tree set to the public part of τ from `groupInfo`. Then, for any node of τ with a public key for which it has a secret key stored (in another ratchet tree or in ISK and SK), $\mathcal{B}$ copies the secrets into τ. Moreover, $\mathcal{B}$ updates the ratchet tree secrets to those derived from `pathSec` (if any secret key was set to a GSD node, $\mathcal{B}$ uses `pathSec` to win the game), and computes the secrets of the epoch from `joinerSec`. For all other unknown secrets in τ, $\mathcal{B}$ keeps them as $\perp$.
Second, there exists some PSK with identifier $\texttt{pskId} \in$ `pskIds` that is unexposed. In this case, the environment $\mathcal{Z}$ must create `welcome_secret` by querying RO. Similar to above case for "$\mathsf{H}($`kp`$) \in$ `encGroupSecs` but `encGroupInfo` is forged", we can prove that this case is impossible otherwise $\mathcal{B}$ can easily win the GSD game.

Injected `groupInfo`. The proof is identical to ETK for injected `groupInfo`.

External Commit Injected to Process.

The reduction $\mathcal{B}$ constructs the new node including the $\mathsf{initSecret}$ as follows: On one hand, if the $\mathsf{external\text{-}key}$, which encrypts the shared secret s_{exini} in the `extInit` proposal of the external commit, is a GSD node, i.e., neither $\mathcal{B}$ nor $\mathcal{Z}$ has knowledge of it, $\mathcal{B}$ uses the **Dec** oracle of GSD to retrieve the shared secret s_{exini}. On the other hand, if the $\mathsf{external\text{-}key}$ is a value, meaning that both $\mathcal{B}$ and $\mathcal{Z}$ know it, $\mathcal{B}$ directly decrypts s_{exini}. The $\mathsf{external\text{-}key}$ cannot be $\perp$, as it is not a key that the adversary can inject. From s_{exini}, $\mathcal{B}$ can then derive the new $\mathsf{initSecret}$. The only case where $\mathcal{B}$ will not be able to decrypt s_{exini} is if the `kem_output` stems from an honest external commit c' previously generated by $\mathcal{B}$ and copied by $\mathcal{Z}$ and whose $\mathsf{appSecret}$ is still a valid challenge. If an `id` accepts such a copied `kem_output`, $\mathcal{B}$ can use it to win the GSD-game similarly to a copied path secrets in commits:

1. Since the `kem_output` is copied, the $\mathsf{ext\text{-}initSec}$ is identical in both the injected external commit c and the honest external commit c'. At the same time, $\mathsf{confTransHash}$ must differ; otherwise, c and c' would be identical.
2. For $\mathcal{Z}$ to compute a valid `confTag` for the injected commit c, it must query the RO on $\mathsf{H}(\mathsf{confTransHash}, \texttt{confKey})$. From this query, $\mathcal{B}$ can extract the `confKey`. Since `confKey` is derived from (`joinerSec`, `psk-secret`), $\mathcal{Z}$ must also query RO on (`joinerSec`, `psk-secret`) to compute a valid tag. This allows the reduction $\mathcal{B}$ to extract `psk-secret`.

3. For $\mathcal{Z}$ to compute the `joinerSec` for the injected commit c, it needs to query RO on Hash(ext-initSec, `commitSec`). From this query, $\mathcal{B}$ can extract the ext-initSec, which is shared with c', by searching the RO query history.

4. $\mathcal{B}$ can then corrupt the `commitSec` of c', which does not impact the challenge, and compute the `joinerSec` of c'. Using this `joinerSec` and `psk-secret`, the reduction $\mathcal{B}$ calculates the appSecret.

Thus, this case cannot occur.

The reduction $\mathcal{B}$ handles the `updatePath` of the commit the same way as a regular one, as the argumentation of an injected commit relies on calculating the initSecret and `psk-secret` of c from the RO calls and corrupting the initSecret and `psk-secret` of c' directly, regardless of its origin. Note that in the case of an injected external commit, both the initSecret and `psk-secret` are always retrieved by $\mathcal{B}$ directly via decryption. This does not affect the argument showing it is impossible to copy `updatePath` secrets from an honest commit.

The final case to consider is when both s_{exini} and ctxt are copied. If they are copied from different commits, the `confTag` can be used to break the challenge for both commits in the same way as before. Figure 31 illustrates how $\mathcal{B}$ utilizes $\mathcal{Z}$ to win the GSD game in case of a copied ctxt.

If they are copied from the same commit but the confTransHash differs from the original commit (e.g., when the external commit is injected for a different party), the `joinerSec` and `psk-secret`, retrievable via the `confKey` RO call, are the same between the two commits. The reason behind is that the same ciphertext indicates same path secret and proposals, which further indicate the same `commitSec` and `psk-secret`. Combing the fact that s_{exini} is copied from the same commit, which yields the same initSecret, the `joinerSec` must also be same. Thus, this allows the reduction $\mathcal{B}$ to compute the appSecret to be computed from it.

Proof of Indistinguishability. We show that $\mathbf{safe}(c) \implies \mathsf{gsd\text{-}exp}((c, u))$ for some u. As a reminder, this is part **2.b.II** of the main proof of $\mathsf{ETK}^{\mathrm{PSK}}$. The subproof is structured as follows:

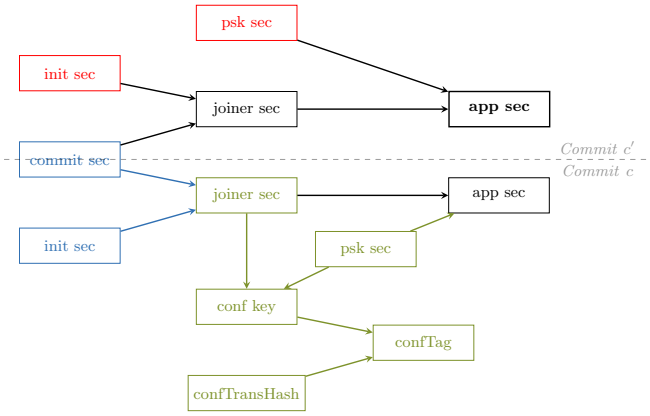


Fig. 31: Visualization of $\mathcal{B}$ using the `confTag` to win the GSD game for a commit c' if the environment $\mathcal{Z}$ successfully copies a ciphertext from a commit c' into an injected commit c. The commit secret is shared between both commits. In the first step (in ●), $\mathcal{Z}$ needs to call the random oracle RO to derive the `confTag` from the `confKey` and the confTransHash, which $\mathcal{B}$ now learns from the query. $\mathcal{Z}$ also needs to query the RO to derive `confKey` from `joinerSec` and `psk-secret`. $\mathcal{B}$ now knows `joinerSec` and `psk-secret` of c. In order to derive `joinerSec` for commit$c$, $\mathcal{Z}$ also needs to query Hash(`commitSec`, initSecret), which $\mathcal{B}$ now learns (in ●). $\mathcal{B}$ now knows the `commitSec` of commit c', and corrupts the initSecret and `psk-secret` of commit c' (in ●). $\mathcal{B}$ can now calculate appSecret of c'.

Part 2.b.II): Confidentiality when allowing injections

The reduction $\mathcal{B}$ succeeds in winning the GSD game for commit c' if the environment $\mathcal{Z}$ successfully copies a ciphertext from commit c' into an injected commit c. This can be formulated as follows:

a) Show that if the reduction $\mathcal{B}$ corrupts (c, u_{app}) then $\neg$**safe**(c)
b) Show that if the reduction $\mathcal{B}$ corrupts (c, u_{joi}) and (c, u_{psk}) then $\neg$**safe**(c)
c) Show that if gsd-exp$((c, u_{\text{com}}))$ and gsd-exp$((\texttt{Node[c].par}, u_{\text{exini}}))$ and gsd-exp$((c, u_{\text{psk}}))$ are true then $\neg$**safe**(c). For this, consider all cases where gsd-exp$((c, u_i))$ is true:
   c.1) During an exposure of an `id` that stores u_i
   c.2) When a node $\tau.v$ used in `*rekey-path` is exposed
   c.3) If the secret in $\tau.v$ is $\perp$, which is the case if:
      i) $\mathcal{Z}$ injects a new commit c' on behalf of `id`
      ii) the injected commit c' commits $\tau.v$ as an updated injected leaf
      iii) the injected c' commits $\tau.v$ as an injected key package

In the case that c is a regular commit, the proof is extended the same way as in step 2 of the original proof in [4], except for two differences at b) and c).

In case b), we still need to prove that the corruption of (c, u_{joi}) and (c, u_{psk}) implies $\neg\textbf{safe}(c)$, where now the node (c, u_{psk}) might be created by the psk-proposal with `pskIds` injected by the environment $\mathcal{Z}$.

Below, we prove that allowing the environment $\mathcal{Z}$ to inject PSK proposals with `pskIds` does not affect $\mathcal{Z}$'s advantages by considering the following two additional sub-cases: whether the environment $\mathcal{Z}$ has exposed all of the PSK values of $\texttt{pskId} \in \texttt{pskIds}$ or not, i.e., $\texttt{Node}[c].\texttt{psk} = \texttt{good}$ or `bad`.

Recall that PSK injection allows the environment $\mathcal{Z}$ to inject the `pskId`, but not the PSK values themselves. The environment cannot manipulate the exact PSK values and further the yielding `psk-secret`. In the first sub-case, if the environment $\mathcal{Z}$ has exposed all PSK values with identifiers $\texttt{pskId} \in \texttt{pskIds}$, i.e., $\texttt{Node}[c].\texttt{psk} = \texttt{bad}$, then the environment $\mathcal{Z}$ can compute `psk-secret`. At the same time, we have $\mathsf{gsd\text{-}exp}((u, u_{\text{psk}})) = \texttt{true}$. However, in order to derive the correct epoch secret $\texttt{epochSec} = \mathsf{HKDF}(\texttt{joinerSec}, \texttt{psk-secret})$, the environment $\mathcal{Z}$ still needs to query RO with correct `joinerSec`. Except for correctly guessing `joinerSec` with negligible probability, the only way that $\mathcal{Z}$ can obtain `joinerSec` is to corrupt (c, u_{joi}). Otherwise, in the second sub-case, the environment $\mathcal{Z}$ does not expose all PSK values of the corresponding $\texttt{pskId} \in \texttt{pskIds}$, i.e., $\texttt{Node}[c].\texttt{psk} = \texttt{good}$. In this case, we have $\mathsf{gsd\text{-}exp}((c, u_{\text{psk}})) = \texttt{false}$. In order to derive the correct epoch secret $\texttt{epochSec} = \mathsf{HKDF}(\texttt{joinerSec}, \texttt{psk-secret})$, the environment $\mathcal{Z}$ has to query RO with the correct `psk-secret`, which is generated by the recursive invocation of $\mathsf{HKDF}(., \texttt{psk}_{\texttt{pskId}})$ for all $\texttt{pskIds} \in \texttt{pskIds}$, where $\texttt{psk}_{\texttt{pskId}}$ is the PSK value of corresponding `pskIds`. Thus, the environment $\mathcal{Z}$ has to query RO with correct $\texttt{psk}_{\texttt{pskId}}$ for all $\texttt{pskId} \in \texttt{pskIds}$, in particular, for the `pskId` that $\mathcal{Z}$ has not exposed. Thus, this case also happens with negligible probability.

Changes in case c), on the other hand, are identical to the proof for ETK, except for minor modifications in c.1) and c.2). Commits *that include an* `updatePath`, which encompass any form of regular commit but exclude `add`, `extAdd`-only commits, are considered as `add`, `extAdd`, `psk`.

In the case that c is an external commit - if there is an edge between $(c, u_{\texttt{exini}})$ and (c, u_{joi}) - the proof proceeds as follows: $\mathsf{gsd\text{-}exp}((c, u_{\text{app}}))$ is only true in three cases:

a) $\mathcal{B}$ corrupts (c, u_{app}) - same as in step 1
b) $\mathcal{B}$ corrupts (c, u_{joi}) and (c, u_{psk}) - same as in step 1
c) if all three $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$, $\mathsf{gsd\text{-}exp}((\texttt{Node[c].par}, u_{\text{exini}}))$ and $\mathsf{gsd\text{-}exp}((c, u_{\text{psk}}))$ are true:
   - For $\mathsf{gsd\text{-}exp}((c, u_{\text{com}}))$ the argumentation remains the same as for regular commit. Note that an injected external commit c' (case i)) hence implies $\mathsf{know}(c)$.
   - In addition, $\mathsf{gsd\text{-}exp}((\texttt{Node}[c].\mathsf{par}, u_{\text{exini}}))$ can also be true if injected: When the environment $\mathcal{Z}$ injecting the seed s_{exini} in the `extInit` proposal, the reduction $\mathcal{B}$ decrypts it and calculates u_{exini} itself. In the case of an

injected external commit, case e) of ***can-traverse** immediately holds, and hence **safe**(c) is false.

In detached trees, the proof remains the same as ETK.

Part 2.b.III) Allowing Bad Randomness

Proposals by Value. In case of a `psk` proposal, the reduction $\mathcal{B}$ computes the proposal message p by using randomness provided by $\mathcal{Z}$ (as 'nonce' value), the current `memberSec`, and `id`'s spk (all of which are known by $\mathcal{B}$), and stores the `pskId` in `pskIds`.

Commit by Reference and by Value. Given randomness provided by $\mathcal{Z}$, $\mathcal{B}$ computes the commit and required secrets as:

1. The reduction $\mathcal{B}$ executes `*rekey-path` to obtain the path secrets, `commitSec`, and commitContent. If a `psk` is required, it computes `pskId` using the provided randomness. It then generates a valid confTransHash.
2. $\mathcal{B}$ creates a new `joinerSec`, which is a hash of the current initSecret and the newly computed `commitSec`, i.e., `joinerSec` = H (initSecret, `commitSec`, ...). If initSecret stores a GSD node u, $\mathcal{A}$ queries the $\mathsf{H}(u, u_{\text{ctr}}, \texttt{commitSec})$, increments u_{ctr}, and uses the fresh, uncorrupted node as the `joinerSec`. Otherwise, if initSecret stores a value, $\mathcal{A}$ computes the `joinerSec` itself.
3. Additionally, $\mathcal{B}$ is required to compute `psk-secret`. Since $\mathcal{B}$ knows the full `pskIds` list, $\mathcal{B}$ can compute `psk-secret` same as in Step 1. Using the secrets obtained, $\mathcal{B}$ executes the key schedule to compute the `confTag`. It then computes the commit message c and the vector of welcome messages `*welcome-msg` accordingly.

Validity of the Challenge and Randomness in `psk`. The reduction $\mathcal{B}$'s actions and the proof remain the same for commits that involve fresh randomness. Recall that the helper predicate ***secrets-replaced** captures the update of the path secret, except for the proposal-list in regular commit c is *non-empty* or only includes regular add, external self-add, or PSK proposals. Any commit c'' (except only regular add, external self-add, or PSK proposals) with ***secrets-replaced**(c'', `id`) would hence replace secrets injected by randomness.

Now we will show with above changes, randomness corruption does not affect existing challenges, where **safe**(c) holds. With corrupted randomness the environment $\mathcal{Z}$ cannot manipulate the exact values of any psk, and therefore `psk-secret`. Instead, $\mathcal{Z}$ can inject `pskId` by using corrupted "nonce" value, i.e, `Node`[c].`psk` = `bad`.

This case is same with part 2 of the proof: although $\mathcal{Z}$ can compute `psk-secret`, $\mathcal{Z}$ cannot derive the correct `epochSec` unless gsd-exp((u, u_{psk})) = `true`, because the environment $\mathcal{Z}$ needs to query RO on `joinerSec` and `psk-secret` to compute `epochSec` = HKDF(`joinerSec`, `psk-secret`). Hence, reduction $\mathcal{B}$ cannot get additional advantage by `psk-secret`.

We adapt the proof that randomness corruption does not affect the validity of a challenge, and gsd-exp(c) $\implies$ $\neg$**safe**(c) still holds. Due to above mentioned

changes, no additional corruption is performed by $\mathcal{B}$ on the `joinerSec`, case b) of gsd-exp holds as before.

Only case that is affected is case c). Here, the proof remains in agreement with [4], showing that $\mathsf{gsd\text{-}exp}((c, u_{\text{com}})) \implies \neg\mathsf{know}(c, *)$ also when allowing corruptions. This is the same bug we mentioned in ETK proof.

Randomness in Detached Trees. The proof remains identical to the ETK proof, allowing randomness in detached trees.

External Commit from `id`. Similar to regular commit, using the randomness provided by $\mathcal{Z}$, reduction $\mathcal{B}$ computes the external commit and the secrets in new commit node, as follows:

1. $\mathcal{B}$ uses the randomness provided by $\mathcal{Z}$ to execute `*rekey-path-upon-join`, obtaining all path secrets, the `commitSec`, and the intermediate commit packet. It computes `pskId` using the same randomness. Then, it signs the commit packet using the `id` 's spk, along with $\mathcal{Z}$'s randomness, and sets the confTransHash accordingly.
2. A key distinction from a regular commit is that the initSecret for an external commit is determined by the client. As a result, corrupted randomness can expose this initSecret. Reduction $\mathcal{B}$ computes the ext-initSec as follows:
   - If the external secret from the previous epoch stores a GSD node u, $\mathcal{B}$ queries $\mathbf{Enc}(u, u_{\text{ctr}})$, corrupts u_{ctr}, and sets the external seed s_{exini}, increments u_{ctr}. $\mathcal{B}$ then uses this seed as the initSecret for the external commit.
   - If the external secret from the previous epoch stores a value (and is therefore known to $\mathcal{B}$), $\mathcal{B}$ computes the `joinerSec` directly.
3. Using the `joinerSec`, `psk-secret`, and the confTransHash from previous steps, $\mathcal{B}$ runs the key schedule, computes the `confTag`, and finalizes the commit message c.

Proof for External Commit. Case a) of the proof remains the same. Case b): The reduction $\mathcal{B}$ now also corrupts (c, u_{joi}) and (c, u_{psk}) when executing an external commit with compromised randomness. Any node that is injected with bad randomness has $\texttt{Node}[c].\texttt{stat} = \texttt{bad}$ and $\texttt{Node}[c].\texttt{psk} = \texttt{bad}$. Additionally, external commit nodes are marked via the sender type `new_member`. An external commit with bad randomness hence fulfills case e) of $\textbf{*can-traverse}(c)$, and hence $\textbf{safe}(c) = \texttt{false}$.

Case c): To show $\mathsf{gsd\text{-}exp}((c, u_{\text{com}})) \implies \neg\mathsf{know}(c, *)$ for external commits, we proceed similar as for regular commits. $\mathsf{gsd\text{-}exp}((c, u_i))$ can also be true if the secret in a ratchet tree node $\tau.v$ used in `*rekey-path-upon-join` stores a seed s generated during an action executed with bad randomness. Consider an external commit c' that inserts s into $\tau.v$. Unlike a regular commit, an external commit injects only the secrets it generates during the commit process. It does not, for instance, include an update proposal from another `id` or a corrupted key package. This injection is covered by case a) of $\textbf{*secrets-injected}(c, \texttt{id})$. Now consider the case where c itself is the external commit. Since the external committer utilizes

the same public ratchet tree as a member performing a regular commit, the proof remains the same to regular commit.

K.3 Authenticity

We follow the same intuition with ETK to prove authenticity; specifically, we aim to demonstrate that *H3* and *H4* are indistinguishable by showing that **auth-invariant** is never triggered.

Lemma 8. *If* Sig *is* SUF-CMA *secure,* MAC *is* EUF-CMA *secure, and* PKE *is* GSD *secure, then Hybrids H3 and H4 are indistinguishable, that is,* $\mathsf{ETK}^{\mathrm{PSK}}$ *guarantees authenticity.*

Proof. Similarly, the hybrids *H3* and *H4* are identical unless the event $\mathtt{Bad}$ occurs. We further decompose the event $\mathtt{Bad}$ into two sub-events: $\mathtt{Bad}_{\mathtt{sig}}$ and $\mathtt{Bad}_{\mathtt{MAC}}$. The proof is concluded by combining Lemma 9 and Lemma 5, where we respectively prove that the events $\mathtt{Bad}_{\mathtt{sig}}$ and $\mathtt{Bad}_{\mathtt{MAC}}$ will never happen except for negligible probability. For $\mathsf{ETK}^{\mathrm{PSK}}$, the case where $\mathtt{Bad}_{\mathtt{MAC}}$ is triggered remains identical to that in ETK. Therefore, we focus here on examining the scenario where the event $\mathtt{Bad}_{\mathtt{sig}}$ occurs.

Lemma 9. *For any environment* $\mathcal{Z}$*, there exists a reduction* $\mathcal{B}_{\mathtt{sig}}$ *that succeeds in the* SUF-CMA *game with a probability that is only polynomially smaller than the probability of* $\mathcal{Z}$ *triggering* $\mathtt{Bad}_{\mathtt{sig}}$*.*

Proof. For any environment $\mathcal{Z}$, $\mathcal{B}_{\mathtt{sig}}$ emulates the functionalities and simulator by embedding its challenge spk as one of the public keys honestly created during the experiment. To emulate commits signed under ssk, $\mathcal{B}_{\mathtt{sig}}$ calls **Sign** oracle. When $\mathtt{Bad}_{\mathtt{sig}}$ occurs, reduction $\mathcal{B}_{\mathtt{sig}}$ stops the experiment and sends to its challenger to forgery consisting of the sig', $\mathtt{tbs}'$ from the injected node c'.

First assume c' is a commit node. We show that if $\mathtt{Bad}_{\mathtt{sig}}$ occurs and $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}[\mathtt{id}]$, then $\mathcal{B}_{\mathtt{sig}}$ wins.

1. We know sig' is a valid signature over $\mathtt{tbs}'$. The injected node was created when some party accepted c' which means that it verified sig' under $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}$.
2. Simulation differs from the experiment when $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem}$ is corrupted, however this does not happen as the event guarantees $\mathsf{spk} = \mathtt{Node}[c].\mathsf{mem} \notin \mathsf{Exposed}$.
3. The reduction $\mathcal{B}_{\mathtt{sig}}$ has never queried **Sign** on $\mathtt{tbs}'$ for sig' to its signing oracle. Assuming $(\mathsf{sig}', \mathtt{tbs}')$ is the same as $(\mathsf{sig}^*, \mathtt{tbs}^*)$ queried by $\mathcal{B}_{\mathtt{sig}}$ to the sign oracle for c^*, where c' is injected and c^* is honestly generated by $\mathcal{B}_{\mathtt{sig}}$. By showing our assumption implies $c' = c^*$, we will achieve a contradiction. Now we will divide the proof for two, according to the type of the commit c'.
   - We first consider that c' is a regular commit. c' contains $(\mathsf{groupId}', \mathsf{epoch}', \mathsf{leafIdx}', '\mathtt{Commit}', C', \mathtt{confTag}', \mathsf{sig}', \mathtt{membTag}')$ and $\mathtt{tbs}' = (\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \mathtt{senderIdx}', '\mathtt{Commit}', C')$ and c^* contains the analogous

values. Then c' and c^* can only differ on $\texttt{confTag}' \neq \texttt{confTag}^*$ or $\texttt{membTag}' \neq \texttt{membTag}^*$. Note that $\texttt{membTag}' = \mathsf{MAC.tag}(\mathsf{membKey}', C')$ and that $\texttt{membTag}^* = \mathsf{MAC.tag}(\mathsf{membKey}^*, C^*)$. According to our assumption $\texttt{tbs}' = \texttt{tbs}^*$ implies $C' = C^*$ and $\mathsf{groupCtxt}' = \mathsf{groupCtxt}^*$. As group context and psk (if there exists) determines the epoch and key schedule. We have $\mathsf{membKey}' = \mathsf{membKey}^*$ and $\texttt{psk-secret}' = \texttt{psk-secret}^*$, and further $\texttt{confKey}' = \texttt{confKey}*$. Accordingly $\texttt{membTag}' = \texttt{membTag}^*$ and $\texttt{confTag}' = \texttt{confTag}^*$, which gives a contradiction.
   - Then we consider that c' is an external commit. c' contains two components: $\texttt{framedContent} = (\mathsf{groupId}', \mathsf{epoch}', \texttt{'new\_member\_commit'}, \texttt{'commit'}, C', \mathsf{sig}')$ and $\texttt{framedContentAuth}' = (\mathsf{sig}', \texttt{confTag}')$ and c^* contains the analogous values. Moreover, $\texttt{tbs}' = (\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \texttt{senderIdx}', \texttt{'Commit'}, C')$ and $\texttt{tbs}^*$ contains the analogous values. Similar to regular commit, c' and c^* can only differ on $\texttt{confTag}' \neq \texttt{confTag}^*$. By definition it holds that $\texttt{confTag}' = \mathsf{Hash}(\mathsf{confTransHash}', \texttt{confKey}')$ and $\texttt{confTag}^* = \mathsf{Hash}(\mathsf{confTransHash}^*, \texttt{confKey}^*)$. Since confTag is derived on the framed content, we have $\texttt{confTag}' = \texttt{confTag}^*$. However this gives us contradiction as c' is injected and c^* is honestly generated.
4. Now consider all proposals nodes except extAdd-only commits, p' contains $(\mathsf{groupId}', \mathsf{epoch}', \mathsf{leafIdx}', \texttt{'Propose'}, P', \mathsf{sig}', \texttt{membTag}')$ and $\texttt{tbs}' = (\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \mathsf{leafIdx}()', \texttt{'Propose'}, P')$ and c^* contains the analogous values. Note that proposal have membTag but not confTag. The proof for this case is analogous to the proof for commits.
5. For the extAdd proposal on the other hand, p' contains $(\texttt{tbs}', \mathsf{sig}')$, where $\texttt{tbs}' = (\mathsf{groupId}', \mathsf{epoch}', \texttt{new\_member}', \texttt{'Propose'}, P')$ and c^* contains the analogous values. Note that here as there is no membTag or confTag, commits c' and c^* that includes only extAdd proposals can only differ on $\texttt{psk-secret}' \neq \texttt{psk-secret}^*$. According to our assumption $\texttt{tbs}' = \texttt{tbs}^*$ implies $C' = C^*$ and $\mathsf{groupCtxt}' = \mathsf{groupCtxt}^*$. As group context and psk determines the epoch and key schedule. So $\texttt{psk-secret}' = \texttt{psk-secret}^*$ must hold. This results in a contradiction.

Changelog

- Version 1.0, February 14, 2025: Initial draft.
- Version 1.1, February 19, 2025:
  - Fixed minor typo.
- Version 2.0, April 11, 2026:
  - Added Changelog.
  - Extended security results in Section 5.2.
  - Substantial additions including simplified versions of ETK, $\mathsf{ETK}^{\mathrm{PSK}}$, and helper functions.
  - Updated the parent-hash computation from the Draft 12 formulation to the version adopted in MLS since Draft 13.
  - Updated technical details and improved consistency regarding the corruption of isk and ssk in Figures 13 and 15.

Table of Contents

ETK: External-Operations TreeKEM and the Security of MLS in RFC 9420 . . . 1
Cas Cremers[1][0000−0003−0322−2293], Esra Günsay[1,2][0000−0001−6672−4253], Vera Wesselkamp[3][0009−0002−8481−9103], and Mang Zhao[4(✉)][0009−0002−4720−6551]
1 Introduction . . . 2
2 Background . . . 4
2.1 The Evolution of the MLS Protocol . . . 4
2.2 Formal Security Analysis of MLS Draft 12 . . . 6
2.3 Notation . . . 10
3 Extended Continuous Group Key Agreement: Syntax . . . 10
4 Security Model and Functionality . . . 12
4.1 History Graph . . . 12
4.2 Functionality $\mathcal{F}_{\text{ECGKA}}$ . . . 13
5 RFC 9420 and ETK: Extended Operations TreeKEM . . . 21
5.1 The ETK Protocol . . . 21
5.2 Security Results for ETK and $\mathcal{F}_{\text{ECGKA}}$ . . . 23
6 Stronger Security for Group Re-Synchronization via External Commit 25
6.1 Resyncs and Resumption Pre-Shared Keys . . . 25
6.2 The Stronger Functionality $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ . . . 26
6.3 The ETK^{PSK} Protocol . . . 27
6.4 Security Results for ETK^{PSK} and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ . . . 28
A Overview of Supplementary Material . . . 32
B Preliminaries . . . 33
B.1 Public Key Encryption . . . 33
B.2 Digital Signature . . . 35
B.3 Message Authentication Code . . . 36
B.4 Random Oracle . . . 36
C Simplifications . . . 38
D Details on $\mathcal{F}_{\text{ECGKA}}$ and $\mathcal{F}_{\text{ECGKA}^{\text{PSK}}}$ Functionalities . . . 40
E Details on ETK and ETK^{PSK} Protocols . . . 50
F Proof Sketch of Theorem 1 . . . 60
G Proof Sketch of Theorem 2 . . . 61
H Comparison with the History Graph in [4] . . . 62
I Comparison between $\mathcal{F}_{\text{CGKA}}$ [4] and our $\mathcal{F}_{\text{ECGKA}}$ . . . 63
J Proof of Theorem 1 . . . 64
J.1 Consistency and Correctness . . . 64
J.2 Confidentiality . . . 73
J.3 Authenticity . . . 87
K Proof of Theorem 2 . . . 90

K.1 Consistency and Correctness of $\mathsf{ETK}^{\mathrm{PSK}}$ 90
K.2 Confidentiality of $\mathsf{ETK}^{\mathrm{PSK}}$ 94
K.3 Authenticity 106