On The Insider Security of MLS

Joël Alwen[1], Daniel Jost[2]*, and Marta Mularczyk[3]**

[1] AWS Wickr, alwenjo@amazon.com
[2] New York University, daniel.jost@cs.nyu.edu
[3] ETH Zurich, Switzerland, mumarta@inf.ethz.ch

Abstract. The *Messaging Layer Security* (MLS) protocol is an open standard for end-to-end (E2E) secure group messaging being developed by the IETF poised for deployment to consumers, industry, and government. It is designed to provide E2E privacy and authenticity for messages in long lived sessions whenever possible despite the participation (at times) of malicious insiders that can adaptively interact with the PKI at will, actively deviate from the protocol, leak honest parties' states, and fully control the network.

The core of the MLS protocol (from which it inherits essentially all of its efficiency and security properties) is a *Continuous Group Key Agreement* (CGKA) protocol. It provides asynchronous E2E *group management* by allowing group members to agree on a fresh independent symmetric key after every change to the group's state (e.g. when someone joins/leaves the group).

In this work, we make progress towards a precise understanding of the insider security of MLS (Draft 12). On the theory side, we overcome several subtleties to formulate the first notion of insider security for CGKA (or group messaging). Next, we isolate the core components of MLS to obtain a CGKA protocol we dub *Insider Secure TreeKEM* (ITK). Finally, we give a rigorous security proof for ITK. In particular, this work also initiates the study of insider secure CGKA and group messaging protocols.

Along the way we give three new (very practical) attacks on MLS and corresponding fixes. (Those fixes have now been included into the standard.) We also describe a second attack against MLS-like CGKA protocols proven secure under all previously considered security notions (including those designed specifically to analyze MLS). These attacks highlight the pitfalls in simplifying security notions even in the name of tractability.

* Research supported by the Swiss National Science Foundation (SNF) via Fellowship no. P2EZP2_195410. Work partially done while at ETH Zurich, Switzerland.

** Research supported by the Zurich Information Security and Privacy Center (ZISC).

Table of Contents

1 Introduction

1.1 Background and Motivation

A *Continuous Group Key Agreement* (CGKA) protocol allows an evolving group of parties to agree on a continuous sequence of shared symmetric keys. Most CGKA protocols are designed to be truly practical even when used over an adversarial network by large groups of uncoordinated parties with little, if any, common points of trust.

CGKA protocols should be end-to-end (E2E) secure and use *asynchronous* communication (in contrast to older, highly interactive, Dynamic Group Key Agreement protocols). That is, no assumptions are made about when or for how long parties are online. Instead, an (untrusted) network is expected only to buffer packets for each party until they come online again. As a consequence, all actions a party might wish to take must be performed non-interactively. Moreover, protocols cannot rely on specially designated parties (like the group managers in broadcast encryption). To achieve E2E security, protocols shouldn't rely on trusted third parties including the PKI that distributes long and short term public keys.[4]

Intuitively, CGKA protocols encapsulate the cryptographic core necessary to build higher-level distributed E2E secure group applications like secure messaging (not unlike how Key Encapsulation captures the core of Public Key Encryption). Any change to a group's state (e.g. parties joining/leaving) initiates a new *epoch* in a CGKA session. Each epoch E is equipped with its own uniform and independent epoch key k_E, called the *application secret* of E, which can be derived by all group members in E. The term "application secret" reflects the expectation that k_E will be used by a higher-level cryptographic application during E.[5] For example, k_E might seed a key schedule to derive (epoch specific) symmetric keys and nonces, allowing group members in E to use authenticated encryption for exchanging private and authenticated messages during E.

The Messaging Layer Security Protocol. Probably the most important family of CGKA protocols today is TreeKEM. An initial version was introduced in [32]. This was soon followed by a more precise description in [14]. Next came the improved TreeKEMv2 [10]. Another major revision came with the introduction of the "propose-and-commit" paradigm [13]. The product of this evolution (implicitly) makes up most of the cryptographic core of the latest draft (Draft 12) of the *Messaging Layer Security* (MLS) protocol [9]. It is this most recent version (henceforth TreeKEMv3) which is the main focus of this work.

MLS is being developed under the auspices of the IETF. It aims to set an open standard for E2E secure group messaging; in particular, for very large groups (e.g. 50K users). MLS is being developed by an international collaboration of academic cryptographers and industry actors including Cisco, Cloudflare, Facebook, Google, Twitter, Wickr, and Wire. Together, these already provide messaging services to over 2 billion users across all sectors of society. The IETF is currently soliciting more feedback from the cryptographic community in hopes of finalizing the current draft.

Insider Security. Intuitively, MLS is designed to provide security whenever possible in the face of a weak PKI and despite potential participation by malicious insiders with very powerful adaptive capabilities. These include full control of the network and repeatedly leaking the local states of honest users and even choosing their random coins[6]. However, thus far it has remained open how to formally capture (let alone analyze) such a security notion for CGKA/group messaging. Instead, simplified security models have been used to analyze TreeKEMv2 [3, 5] and TreeKEMv3 [17]. In particular, none of these works let the adversary register public keys in the PKI (let alone without proving knowledge of the corresponding secret keys) nor choose the random coins of corrupt parties (though [17] captures *some* consequences such capabilities can have on honest parties key schedules). Most critically, none of their models let the adversary deliver *arbitrary* packets; a very natural capability for a real-world attacker controlling the network.

In more detail, in [3] the adversary is forced to deliver packets in the same order to all parties and learns nothing about the coins of parties she has compromised. Meanwhile, [5] permits arbitrary

[4] Concretely, the servers distributing keys are normally *not* trusted per se. Instead trust is established by, say, further equipping participants with tools to perform out-of-band audits of the responses they receive from the server.

[5] In the newest draft of MLS the term "application secret" has been changed to "encryption secret".

[6] We stress that adversarially chosen coins can lead to real-world attacks, see e.g. [18].

packet delivery scheduling and leaks the random coins of corrupt parties but still does not allow the adversary to choose corrupt parties' coins. Neither model allows fully active attacks. In [5], the adversary cannot modify/inject packets at all while in [3] she may only deliver modified/injected packets to an honest party if the party will reject the packet.

Finally, [17] focuses exclusively on the pseudorandomness of secrets produced by the key derivation process in MLS. So, unlike other works, they do not consider the general effects a malformed protocol packets can have (e.g. as part of an arbitrary active attack). Instead they focus only on a specific set of effects such packets could have on the key derivation mechanism in MLS. (So for example, they make no statements about authenticity.) In contrast to the other two works, they also only allow for a limited type of adaptivity where adversaries must leak secrets at the moment they are first derived and no later. On the other hand, [17] considers a more fine-grained leakage model where secrets can be individually leaked rather than the whole local state of the victim at once.

1.2 Our Contribution

To help further our understanding of how MLS behaves against such insider attacks, we first precisely define insider security (capturing all of the above mentioned intuitive adversarial capabilities). In particular, our new notion captures correctness as well as the following security goals: security of epoch secret keys, authenticity and agreement on group state. We make these claims formal by building on the security notions in [6] modeling a more accurate and untrusted PKI. E.g., in our model (unlike in [6]) the adversary can register arbitrary (even long term) public keys on behalf of parties and without proving knowledge of corresponding secrets. Of course, security is degraded for epochs in which such keys are used (but, crucially, only those). We note that, in the subsequent works of [26, 7] on CGKA our new insider security notion forms the basis of their security definitions.

Second, we isolate the core features of the full MLS protocol sufficient for realizing an insider secure CGKA protocol and call the result *Insider Secure TreeKEM* (ITK). Specifically, ITK augments TreeKEMv3, with message authentication, tree-signing, confirmation keys and a small part of the MLS's key schedule.

We prove security for ITK. In particular, this is the first work to analyze the tree-signing mechanism in MLS elucidating what it achieves in terms of security guarantees. A lack of (even just intuitive) clarity around what it *should* do lead to differing constructions being proposed and significant debate on the topic within the MLS working group (e.g. [30, 1, 33]). Ultimately, this resulted in the adoption of a flawed design (c.f. Draft 10) as demonstrated in this work via a practical attack on the E2E privacy of MLS.

Beyond this attack, our analysis also unveiled two other new (quite practical) ones on MLS. We proposed fixes for each of the three attacks which have all since been incorporated into to the IETF standard (in Draft 11). Our fixes are already reflected in ITK. In summary, the result of the attacks are as follows:

1. A malicious insider can invite a victim to an artificial group (that includes any number of other honest parties) such that the adversary can continue to derive epoch secrets in the group even after they were supposedly removed from the group by the victim.
2. A malicious insider can break agreement. That is, they can craft two packets delivering each to a different honest user with the result that they will both accept them, agree on their next epoch secret keys, but will in fact be out-of-sync and no longer accept each other's messages.
3. The mode of MLS where ITK packets are not encrypted provides weaker authenticity than intended.

We stress that exploiting these vulnerabilities is only possible for malicious insiders, and hence they are outside the models used so far to analyze MLS, which explains why they went unnoticed until now.[7]

[7] Even the model of [6] which allows active attackers does not consider guarantees for members invited by malicious insiders. Hence the first vulnerability is also outside that model. Moreover, that work analyzed different CGKA protocols than those implicit in MLS.

Complexity. We note that our security definition is quite complex. Unfortunately, we believe this to be, to some extent, inherent given the goals of this project. Namely, to analyze the rather complex ITK (with its six different operations and three types of messages) in executions involving a dynamic group (each member with their own history) subject to attacks by a very powerful adversary with a full array of potential real-world capabilities available to them.

We believe taming this complexity to be an important open problem. But we feel the attacks discovered in this work only serve to underscore the importance of formally capturing the *complete* adversarial capabilities against which MLS intends to defend; in particular also those omitted in previous works. Indeed, we describe a fourth attack that can apply to MLS if the public key encryption scheme it uses were to be instantiated with certain CPA secure constructions. Yet previous analyses of MLS in simpler models implied that CPA security was sufficient [3, 5]. This is again a consequence of oversimplifying the adversary's capabilities. (Fortunately, as implied by [2], the PKE used in MLS is indeed CCA secure which we show to be sufficient for the purposes of this work.)

Outline of the Rest of the Paper. We define insider secure CGKA in Sec. 3. Next we specify the ITK protocol in Sec. 4. The four attacks are described in Sec. 5. Finally, we sketch the security proof in Sec. 6. A more detailed exposition of the proof (and finer details of the security definition) can be found in the appendix.

1.3 Related Work

Concurrently to this work, the paper [15] analyzed insider security of TreeKEM Draft 7 in the symbolic setting (in the sense of Dolev-Yao). Based on the intuition provided in [15], their model seems to cover most if not all security properties considered in our paper (although verifying this would require knowledge of the model's code which is not made public). Thus, their symbolic analysis nicely complements our cryptographic analysis.

However, there are some important differences too. First, [15] has a more fine-grained corruption model. While in [15] the adversary can corrupt individual keys held by parties, in our model it can only corrupt whole states of parties. Second, [15] analyse a version of TreeKEM that does not yet have any tree-signing mechanism. Consequently, they find an attack on TreeKEM Draft 7 (that would also appear in our insider security model) and proposes a strong version of tree signing (aka. "tree-hash based parent hash") that prevents it. Unfortunately, that scheme soon become unworkable (i.e. not correct) as it conflicts with new mechanisms in subsequent Drafts of TreeKEM (namely truncation and unmerged leaves). Thus, Draft 11 TreeKEM/MLS adopted a different, more efficient version tree signing. In this work, we show that this version is too weak and propose an new tree-signing mechanism providing the desired security.

In the cryptographic setting, beyond the results in [3, 5, 17] discussed in the introduction, the recent work of [24] analyzing PCS guarantees provided by MLS in the multi-session setting. Surprisingly, they identify significant inefficiencies in terms of the amount of bandwidth (and computation) required by a multi-session MLS client to return to a fully secure state after a state leakage. They then explore the design space of alternative solutions to remedy this issue.

Besides those implicit in (various versions of) MLS, several other CGKA constructions have been proposed. The first construction, known as the Asynchronous Ratcheting Tree (ART) protocol, was introduced by Cohn-Gordon et al. in [23]. Not long after ART, the TTKEM protocol was introduced in [5] where it was shown to enjoy the same security as TreeKEMv2 (at least with regards to adaptive but passive adversaries). TTKEM is motivated by exhibiting a different computation and cost trade profile to TreeKEMv2. Meanwhile, the RTreeKEM construction of [3] greatly improves on the forward secrecy properties of the TreeKEM family of protocols, albeit by making use of non-standard (but practically efficient) cryptographic components.

The Causal TreeKEM protocol of Weidner [34] supports concurrent changes to the group state (but lacks a formal security analysis). Similarly, the protocol of [16] supports certain types of concurrency within a session, albeit only for groups with fixed membership and in a synchronous communication model.

While the above constructions generally aim for practical efficiency, the three CGKA protocols in [6], eschew this constraint to instead focus on exploring new mechanisms for achieving the increasingly stringent security notions introduced in that work. Up until the results in our work, the later two were the only constructions known to enjoy security of any kind against active adversaries. Moreover, along with weak robustness they also introduce the notion of *strong robustness.* A strongly robust is a weakly robust CGKA with the following additional property. As long as one honest party in an epoch E accepts an arbitrary packet p then all other honest parties currently in E will also accept p (assuming they receive p before some other acceptable packet). Neither ITK (nor MLS) are strongly robust.[8]

All CGKA protocols mentioned here lack an analogue of the tree-signing mechanism used by MLS/ITK.

2 Preliminaries

2.1 Universal Composability

We formalize security in the generalized universal composability (GUC) framework [21], an extension to the UC framework [20]. We moreover use the modification of responsive environments introduced by Camenisch et al. [19] to avoid artifacts arising from seemingly local operations (such as sampling randomness or producing a ciphertext) to involve the adversary.

The (G)UC framework requires a real-world execution of the protocol to be indistinguishable from an ideal world, to an an interactive environment. The real-world experiment consists of the group members executing the protocol (and interacting with the PKI setup). In the ideal world, on the other hand, the protocol gets replaced by dummy instances that just forward all inputs and outputs to an *ideal functionality* characterizing the appropriate guarantees.

The functionality interacts with a so-called simulator, that translates the real-world adversary's actions into corresponding ones in the ideal world. Since the ideal functionality is secure by definition, this implies that the real-world execution cannot exhibit any attacks either.

The Corruption Model. We use the — standard for CGKA/SGM but non-standard for UC — corruption model of continuous state leakage (transient passive corruptions) and adversarially chosen randomness of [6].[9] In a nutshell, this corruption model allows the adversary to repeatedly corrupt parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its current state to the adversary (once), (2) a message $(\texttt{CorrRand}, b)$ sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly.

Restricted Environments. In order to avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment not to corrupt parties at certain times. (This roughly corresponds to ruling out "trivial attacks" in game-based definitions. In simulation-based frameworks, such attacks are no longer trivial, but security against them requires strong cryptographic tools and is not achieved by most protocols.) To this end, we use the technique used in [6] (based on prior work by Backes et al. [8] and Jost et al. [27]) and consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined as part of the ideal functionality $\mathcal{F}$: The functionality can specify certain boolean conditions, and an environment is then called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such condition when interacting with $\mathcal{F}$.

2.2 Notation

We use $v \leftarrow x$ to denote assigning the value x to the variable v and $v \leftarrow\!\!\$\ S$ to denote sampling an element u.a.r. from a set S. If V denotes a variable storing a set, then we write $V \mathrel{+}\!\!\leftarrow x$ and $V \mathrel{-}\!\!\leftarrow x$ as shorthands for $V \leftarrow V \cup \{x\}$ and $V \leftarrow V \setminus \{x\}$, respectively. We further make use of

[8] E.g. a malformed (commit) packet can constructed by an insider such that part of the group accepts it but the rest do not.

[9] Passive corruptions and full network control allow to emulate active corruptions.

associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to denote assignment and retrieval of element i, respectively. Additionally, we denote by $A[*] \leftarrow v$ the initialization of the array to the default value v. Further, we use the following keywords: **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and returns $\perp$. **assert** *cond* is only used to describe functionalities. It means that if *cond* is false, then the given functionality permanently halts, making the real and ideal worlds trivially distinguishable (it is used to validate inputs of the simulator).

3 Continuous Group Key Agreement

This section defines security of CGKA protocols. For better readability, we skip some less crucial details. See App. B for the precise definition.

3.1 Overview

Security via Idealized Services. Analogous to [6], we consider an ideal CGKA functionality that represents an idealized "CGKA service" agnostic to the usage of the protocol. That is, whenever a party performs a certain group operation (e.g. adding a new member) the functionality simply hands back an idealized protocol message to that party — it is then up to the environment to deliver those protocol messages to the other group members, thus not making any assumptions on the underlying network or the architecture of the delivery service.

The Real-World Experiment. In the real-world experiment, the parties execute the protocol that furthermore interacts with the Authentication Service and Key Service PKI functionalities. The former manages long-term identity keys, while the latter allows parties to upload single-use key packages, used by group members to non-interactively add them to the group. The primary interaction with the PKI, i.e., managing the keys, is not group specific and, thus, it is assumed to be handled by the higher-level protocol embedding CGKA. Intuitively, this means that the protocol requires that the environment registers all keys necessary for a given group operation before performing it.

The Ideal World. The ideal world formalizes security guarantees via the ideal functionality $\mathcal{F}_{\text{CGKA}}$. The functionality internally maintains a symbolic representation of the group's evolution in the form of a history graph (introduced in [4]). To formalize confidentiality guarantees, $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **safe**, which, on input a history graph and an epoch within it, decides whether confidentiality of the epoch's secrets is guaranteed. Then, the functionality chooses random and independent secrets for each safe epoch; otherwise the key is arbitrary, i.e., the simulator gets to choose it. Similarly, to formalize authenticity, $\mathcal{F}_{\text{CGKA}}$ is parameterized by **inj-allowed**, which, on input a history graph, an epoch and a party, decides whether messages can be injected on behalf of the party, during the epoch.

As the PKI management is exposed to the environment in the real world, those operations also need to be available in the ideal world. We achieve this by having "ideal-world variants" of the AS and KS, which should be thought of as part of $\mathcal{F}_{\text{CGKA}}$. The ideal AS records which keys have been exposed, which is then used to define the predicates. The actual keys in the ideal world do not convey any particular meaning beyond serving as identifiers — thus in the ideal world we do not run key generation, but instead allow the simulator to choose all honest keys (subject to public keys being unique).

Relation to game-based notions. For readers not familiar with UC, we can translate our notion to a more familiar language as follows. The environment becomes the adversary and different inputs to $\mathcal{F}_{\text{CGKA}}$, as well as corruptions, become different oracles. In the experiment, the challenger flips a bit b and evaluates the adversary's oracles either using the protocol ($b = 0$) or using $\mathcal{F}_{\text{CGKA}}$ and a simulator ($b = 1$), where the simulator is a parameter of the experiment. Security requires that there exists a simulator for which no efficient adversary can guess b.

The advantage of using a simulator is simplicity, since without it in order to interpret injected messages, the game would need additional functionality from the protocol, which is irrelevant for the users.[10] Simulators are very non-standard for game-based notions, so we decided to go for the simulation-based approach.

Roadmap. The rest of this section is structured as follows. First, in Sec. 3.2, we define the two PKI functionalities and their ideal-world variants. Then, in Sec. 3.3, we explain the meaning of different inputs to $\mathcal{F}_{\text{CGKA}}$ (which are also inputs to the protocol) and how they relate to MLS. In Sec. 3.4, we describe history graphs from [4], and in Sec. 3.5 how $\mathcal{F}_{\text{CGKA}}$ builds the graph and formalizes security.

3.2 PKI Setup

In general, we model fully untrusted PKI, where the adversary can register arbitrary keys for any party (looking ahead, security guarantees degrade if such keys are used in the protocol). This especially models insider attacks.[11] All functionalities are formally defined in App. B. Below we give an intuitive description, which should be sufficient to understand our model.

Authentication Service (AS). The AS provides an abstract credential mechanism that maps from user identities, e.g. phone numbers, to long-term identity keys of the given user. Different credential mechanisms of MLS are abstracted by the functionality $\mathcal{F}_{\text{AS}}$, which maintains a set of registered pairs $(\mathsf{id}, \mathsf{spk})$, denoting that user id registered the key spk under their identity. It works as follows:

- A party id can check if a pair $(\mathsf{id}', \mathsf{spk}')$ is registered.
- id can register a new key. In this case, $\mathcal{F}_{\text{AS}}$ generates a key pair $(\mathsf{spk}, \mathsf{ssk})$ (the key-generation algorithm is a parameter), sends spk to id and registers $(\mathsf{id}, \mathsf{spk})$. spk can be later retrieved at any time and then deleted.[12] If id's randomness is corrupted, the adversary provides the key-generation randomness.
- The adversary can register an arbitrary pair $(\mathsf{id}, \mathsf{spk})$.
- When a party's state is exposed, all secret keys it has generated but not deleted yet are leaked to the adversary.

Key Service (KS). The KS allows parties to upload one-time key packages, used to add them to groups while they are offline.[13] This is abstracted by the functionality $\mathcal{F}_{\text{KS}}$. $\mathcal{F}_{\text{KS}}$ maintains pairs $(\mathsf{id}, \mathsf{kp})$, denoting a user's identity and a registered key package. For each $(\mathsf{id}, \mathsf{kp})$, $\mathcal{F}_{\text{KS}}$ stores id's long-term key spk which authenticates the package and for some $(\mathsf{id}, \mathsf{kp})$, it stores the secret key. $\mathcal{F}_{\text{KS}}$ works as follows:

- A party id can request a key package for another party id'. $\mathcal{F}_{\text{KS}}$ sends to id a kp chosen by the adversary in an arbitrary way, i.e. the KS is fully untrusted.
- id can register a new key package. To this end, id specifies a long-term key pair $(\mathsf{spk}, \mathsf{ssk})$ (reflecting that a key package may be signed), $\mathcal{F}_{\text{KS}}$ generates a fresh package $(\mathsf{kp}, \mathsf{sk})$ for id (using a package-generation algorithm that takes as input $(\mathsf{spk}, \mathsf{ssk})$), sends kp to id and registers $(\mathsf{id}, \mathsf{kp})$ with spk.
- id can retrieve all its secret keys (this accounts for the protocol not a priori knowing which key package has been used to add it to the group).
- id can delete one of its secret keys. When its state is exposed, all secret keys it generated but not deleted are leaked to the adversary.

Note that the adversary does not need to register its own packages, since it already determines all retrieved packages.

[10] Looking ahead, the simulator, e.g., identifies the sender of an injected proposal.

[11] In particular, we do not assume so-called key-registration with knowledge. This is a significantly stronger assumption, typically not achieved by the heuristic checks deployed in reality, and it is not needed for security of ITK.

[12] The secret key must be fetched separately, because the key is registered by the environment before the secret key is fetched by the protocol.

[13] In MLS, the KS is implemented as part of the Delivery Service, and in Signal it is called the Key Distribution Center.

Ideal-world variants. The ideal-world variant of AS, $\mathcal{F}_{\text{AS}}^{\text{IW}}$, marks leaked and adversarially registered long-term keys as exposed. The ideal-world variant of KS, $\mathcal{F}_{\text{KS}}^{\text{IW}}$, stores the same mapping between key package and long-term key as $\mathcal{F}_{\text{KS}}$. Intuitively, each key package for which the long-term key spk is exposed (according to AS) is considered exposed. (For simplicity, our ideal world abstracts away key packages. We believe this to be a good trade-off between abstraction and fine-grained guarantees.) Both $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and $\mathcal{F}_{\text{KS}}^{\text{IW}}$ are not parameterized by key-generation algorithms. Instead, on key registration, the adversary is asked to provide a key pair.

3.3 Interfaces of the CGKA Functionality

This section explains different inputs to $\mathcal{F}_{\text{CGKA}}$, which correspond to inputs to the ITK protocol.

Proposals and commits. TreeKEMv3 is a so-called propose-and-commit variant of CGKA, where current group members can propose to *add* new members, *remove* existing ones, or *update* their own key material (for PCS) by sending out a corresponding *proposal message*. The proposals do not affect the group state immediately. Rather, they (potentially) take effect upon transitioning to the next epoch: The party initiating the transition collects a list of proposals in a *commit message* broadcast to the group. Upon receiving such message, each party applies the indicated proposals and transitions to the new epoch. For simplicity, we delegate the buffering of proposals to the higher-level protocol.

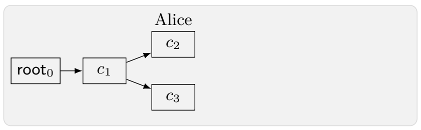
Identity keys. In a real-world deployment, long-term identity keys maintained by the Authentication Service (AS) are likely to be shared across groups. Hence, we also delegate their handling to the higher-level messaging application invoking CGKA. In general, in each group a party uses one signing key at a time. Upon issuing an operation updating the CGKA secrets — i.e., proposing an update or committing — the higher-level may decide to update the signing key as well. Those operations, thus, explicitly take a signing public key spk as input.

Formal syntax. The functionality accepts the following inputs (for simplicity, we treat the party's identity id as implicitly known to the protocol):

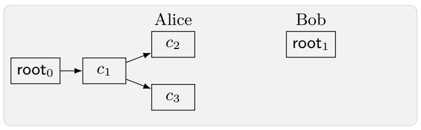
- **Group Creation:** $(\texttt{Create}, \mathsf{spk})$ creates a new group with id being the single member, using the signing public key spk. (This input is only allowed once.)
- **Add, Remove Proposals:** $p \leftarrow (\texttt{Propose}, \mathsf{add\text{-}id}_t)$ (resp., $p \leftarrow (\texttt{Propose}, \mathsf{rem\text{-}id}_t)$) proposes to add (resp., remove) the party id_t. It outputs a proposal message p or $\bot$ if either id is not in the group or id_t already is in the group (resp., is not in the group).
- **Update Proposal:** $p \leftarrow (\texttt{Propose}, \mathsf{up\text{-}spk})$ proposes to update the member's key material, and optionally the long-term signature verification key spk. It outputs an update proposal message p (or $\bot$ if id is not in the group).
- **Commit:** $(c, w) \leftarrow (\texttt{Commit}, \vec{p}, \mathsf{spk})$ commits the vector of proposals $\vec{p}$ and outputs the commit message c. The operation optionally updates the signing public key of the committer.[14]
- **Process:** $(\mathsf{id}_c, \mathsf{propSem}) \leftarrow (\texttt{Process}, c, \vec{p})$ processes the message c, committing the proposals $\vec{p}$ and advances id to the next epoch.[15] It outputs the committer's identity id_c and a vector conveying the semantics of the applied proposals $\vec{p}$.
- **Join:** $(\mathsf{roster}, \mathsf{id}_c) \leftarrow (\texttt{Join}, w)$ allows id (who is not yet a group member) to join the group using the welcome message w. It outputs the roster, i.e. the set of identities and long-term keys of all group members, and the identity id_c of the member who committed the add proposal.
- **Key:** $K \leftarrow \texttt{Key}$ queries the current application secret. This can only be queried once per epoch by each group member (otherwise returning $\bot$).

[14] MLS considers a special type of "add-only" commits. For better clarity, we only consider them in the full definition App. B.

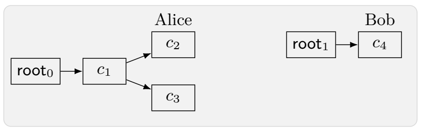
[15] For simplicity, we require that the higher-level protocol that buffers proposals also finds the list p matching c. This is without loss of generality, since ITK uses MLSPlaintext for sending proposals, and c includes hashes of proposals in $\vec{p}$.



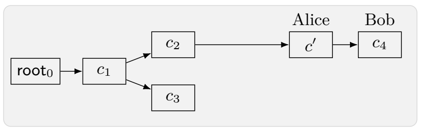
(a) The passive case. Alice processes c_1 and c_2.



(b) Bob joins using injected w'. We don't know where to connect the detached root.



(c) Bob (honestly) commits, creating c_4 in detached tree.



(d) Alice commits with bad randomness and re-computes c' corresponding to w'. We attach the root.

Fig. 1: An example execution with injections and bad randomness, and the corresponding history graph. For simplicity, proposal nodes are excluded.

3.4 History Graph

The history graph is a labeled directed graph that acts as a symbolic representation of a group's evolution. It has two types of nodes: *commit* and *proposal nodes*, representing all executed commit and propose operations, respectively. Note that each commit node represents an epoch. The nodes' labels, furthermore, keep track of all the additional information relevant for defining security. In particular, *all nodes* in the history graph store the following values:

- orig: the party whose action created the node, i.e., the message sender;
- par: the parent commit node, representing the sender's current epoch;
- $\mathsf{stat} \in \{\mathsf{good}, \mathsf{bad}, \mathsf{adv}\}$: a status flag indicating whether secret information corresponding to the node is known to the adversary. Concretely, adv means that the adversary created this node by injecting the message, bad means that it was created using adversarial randomness (hence it is well-formed but the adversary knows the secrets), and good means that it is secure.

Proposal nodes further store the following value:

- $\mathsf{act} \in \{\mathsf{up\text{-}spk}, \mathsf{add\text{-}id}_t\text{-}\mathsf{spk}_t, \mathsf{rem\text{-}id}_t\}$: the proposed action. The history graph here also keeps track of the signature public key spk: $\mathsf{add\text{-}id}_t\text{-}\mathsf{spk}_t$ means that id_t is added with the public key spk_t, and $\mathsf{up\text{-}spk}$ reflects the respective input to the add proposal.

Commit nodes further store the following values:

- pro: the ordered list of committed proposals;
- mem: the list of group members and their signature public keys;
- key: the group key;
- chall: a flag indicating whether the application secret has been challenged, i.e., chall is `true` if a random group key has been generated for this node, and `false` if the key was set by the adversary (or not generated);
- exp: a set keeping track of parties corrupted in this node, including whether only their secret state used to process the next commit message or also the current application secret leaked.

3.5 Details of the CGKA Functionality

This section presents a simplified version of $\mathcal{F}_{\text{CGKA}}$. Compared to the precise definition in App. B, we skip some less relevant border cases and details. A pseudo-code-like description is in Figs. 2 to 4 and an example history graph built by $\mathcal{F}_{\text{CGKA}}$ is in Fig. 1. We next build some intuition about how $\mathcal{F}_{\text{CGKA}}$ works.

The passive case. For the start, consider environments that do not inject or corrupt randomness (this relates to parts of the functionality not marked by [Inj] or [RndCor]). Here, $\mathcal{F}_{\text{CGKA}}$ simply builds a history graph, where nodes are identified by messages, and the root is identified by the label root_0 (see Fig. 1a). Moreover, $\mathcal{F}_{\text{CGKA}}$ stores for each party id a pointer $\mathsf{Ptr}[\mathsf{id}]$ to its current history-graph node. If, for example, id proposes to add id_t, $\mathcal{F}_{\text{CGKA}}$ creates a new proposal node identified by a message p chosen by the adversary, and hands p to id. Some other party can now commit p (having received it from the environment), which, analogously, creates a commit node identified by c. Then, if a party processes c, $\mathcal{F}_{\text{CGKA}}$ simply moves its pointer. The graph is initialized by a designated party $\mathsf{id}_{\mathsf{creator}}$, who creates the group with itself as a single member and can then invite additional members.

If a party id is exposed, $\mathcal{F}_{\text{CGKA}}$ records in the history graph which information inherently leaks from its state. This will be used by the predicate **safe** (recall that it determines if the epoch's key is random or arbitrary). In particular, two points are worth mentioning. First, we require that after outputting the group key, id removes it from its state (this is important for forward secrecy of the higher-level messaging protocol). $\mathcal{F}_{\text{CGKA}}$ uses the flag $\mathsf{HasKey}[\mathsf{id}]$ to keep track of whether id outputted the key. Second, id has to store in its state key material for updates and commits it created in the current epoch. Accordingly, upon id's exposure $\mathcal{F}_{\text{CGKA}}$ sets the status stat of all such nodes to bad (note that leaking secrets has the same effect as choosing them with bad randomness).

Injections. The parts of $\mathcal{F}_{\text{CGKA}}$ related to injections are marked by comments containing [Inj]. As an example, say the environment makes id process a commit message c' not obtained from $\mathcal{F}_{\text{CGKA}}$, and hence not identifying any node. $\mathcal{F}_{\text{CGKA}}$ first asks the adversary if c' is simply malformed and, if this is the case, output $\perp$ to id. If the message is not malformed, the functionality creates the new commit node, allowing the adversary to interpret the sender orig'. We guarantee agreement — if any other party transitions to this node, it will output the same committer orig', member set mem', group key etc. (recall that it is contained in the output of process). Note that we also guarantee correctness — if the input of process is an honest message c generated by $\mathcal{F}_{\text{CGKA}}$, then the adversary cannot make the commit fail.

A more challenging scenario is when the environment injects a welcome message w'. Now there are two possibilities. First, w' could lead to an existing node. In this case, $\mathcal{F}_{\text{CGKA}}$ asks the adversary to provide the node c and records that w' leads to it. We require agreement — any party subsequently joining using w' transitions to c.

However, in general, we cannot expect that the adversary (i.e., simulator), given an arbitrary w' computed by the environment, can come up with the whole commit message c' and its position in the history graph.[16] Therefore, in this case $\mathcal{F}_{\text{CGKA}}$ creates a detached root, identified by a unique label $\mathsf{root}_{\mathsf{rootCtr}}$, where $\mathsf{rootCtr}$ is a counter. If afterwards the environment injects c' corresponding to w', then the root is attached and re-labeled as c'. This scenario is depicted in Figs. 1b and 1c. We require consistency — when creating a detached root, the adversary chooses the member set, but when it is attached, we check that it matches the new parent.

Corrupted randomness. The relevant parts of $\mathcal{F}_{\text{CGKA}}$ are marked by [RndCor]. Corrupted randomness leads to two adverse effects. First, the adversary can make parties re-compute existing messages, leading to the following scenarios:

– A party re-computes a message it already computed. In this case, $\mathcal{F}_{\text{CGKA}}$ only checks that the previous message was computed with the same inputs.
– A party re-computes a message previously injected by the environment. Here, $\mathcal{F}_{\text{CGKA}}$ verifies that the semantics of the existing node chosen by the adversary upon injection are consistent with the correct semantics computed using the party's inputs. (Technically, instead of creating a new node, $\mathcal{F}_{\text{CGKA}}$ checks that the node it would have created is consistent with the existing one.)
– A party recomputes a commit c' corresponding to an injected welcome (see Fig. 1d). In this case, $\mathcal{F}_{\text{CGKA}}$ attaches the detached root, just like in case c' was injected into process.

Second, we note that each protocol message in MLS is signed, potentially using ECDSA, which reveals the secret key in case bad randomness is used. Therefore, every time a party id generates a message with bad randomness, $\mathcal{F}_{\text{CGKA}}$ notifies $\mathcal{F}_{\text{AS}}$, which marks all long-term keys of id as exposed.

[16] For instance, say the environment computes a long chain of commits in its head and injects the last one. It is not clear how to construct a protocol for which it is possible to identify all ancestors, without including all their hashes in w.

Functionality $\mathcal{F}_{\text{CGKA}}$: Initialization

Parameters: predicate **safe**(c) (are group secrets in c secure), predicate **inj-allowed**(c, id) (is injecting allegedly from id in c allowed), group creator's identity $\mathsf{id}_{\mathsf{creator}}$.

```
Initialization
    // Pointers, commit nodes, proposal nodes
    $\mathsf{Ptr}[*], \mathsf{Node}[*], \mathsf{Prop}[*] \leftarrow \bot$
    // Welcome message to commit message mapping
    $\mathsf{Wel}[*] \leftarrow \bot$
    $\mathrm{RndCor}[*] \leftarrow \mathsf{good}$; $\mathsf{HasKey}[*] \leftarrow \mathtt{false}$
    $\mathsf{rootCtr} \leftarrow 0$

Input $(\mathtt{Create}, \mathsf{spk})$ from $\mathsf{id}_{\mathsf{creator}}$
    // The group can be created only once.
    req $\mathsf{Node}[\mathsf{root}_0] = \bot \wedge$ *usable-spk$(\mathsf{id}_{\mathsf{creator}}, \mathsf{spk})$
    // Create the root node and transition $\mathsf{id}_{\mathsf{creator}}$ there.
    $\mathsf{Node}[\mathsf{root}_0] \leftarrow$ commit node with $\mathsf{orig} = \mathsf{id}_{\mathsf{creator}}$,
            $\mathsf{mem} = \{(\mathsf{id}_{\mathsf{creator}}, \mathsf{spk})\}$ and $\mathsf{stat} = \mathrm{RndCor}[\mathsf{id}_{\mathsf{creator}}]$.
    $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathsf{root}_0$
    $\mathsf{HasKey}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathtt{true}$

Input $(\mathtt{Propose}, \mathsf{act}), \mathsf{act} \in \{\mathsf{up\text{-}spk}, \mathsf{add\text{-}id}_t, \mathsf{rem\text{-}id}_t\}$ from id
    Send id and all inputs to the adv. and receive $ack$.
    // Adv. can reject invalid inputs.
    if $\neg$*require-correctness('prop', id, act) then
        req $ack$
    // Compute the proposal node this action creates.
    $P \leftarrow$ proposal node with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{orig} = \mathsf{id}$,
            $\mathsf{act} = \mathsf{act}$, $\mathsf{stat} = \mathrm{RndCor}[\mathsf{id}]$.
    if $\mathsf{act} = \mathsf{add}$ then // Adv. can choose the key package for adds.
        Receive $\mathsf{spk}_t$ from the adversary
        $P.\mathsf{act} \leftarrow \mathsf{add\text{-}spk}_t$
    // Insert $P$ into HG.
    Receive $p$ from the adversary.
    if $\mathsf{Prop}[p] = \bot$ then // Passive case: created a new node.
        $\mathsf{Prop}[p] \leftarrow P$
    else// [Inj] [RndCor] Re-computing existing $p$.
        assert *consistent-nodes$(\mathsf{Prop}[p], P)$
    if $\mathrm{RndCor}[\mathsf{id}]$ then // [RndCor] Signed with bad randomness.
        Notify $\mathcal{F}^{\text{IW}}_{\text{AS}}$ that id's spk is compromised.
    return $p$

Input $(\mathtt{Commit}, \vec{p}, \mathsf{spk})$ from id
    Send id and all inputs to the adv. and receive $ack$.
    // Adv. can reject invalid inputs.
    if $\neg$*require-correctness('comm', id, $\vec{p}$, spk) then
        req $ack$
    // [Inj] Adv. interprets injected proposals.
    for $p \in \vec{p}$ s.t. $\mathsf{Prop}[p] = \bot$ do
        $\mathsf{Prop}[p] \leftarrow$ proposal node with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{stat} = \mathsf{adv}$, and
                orig and act chosen by the adversary.
    // Compute the commit node this action creates.
    $C \leftarrow$ commit node with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{orig} = \mathsf{id}$, $\mathsf{stat} = \mathrm{RndCor}[\mathsf{id}]$,
            $\mathsf{pro} = \vec{p}$, and $\mathsf{mem} =$ *members$(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{spk})$
    // Insert $C$ into HG.
    Receive $(c, rt)$ from the adversary.
    if $\mathsf{Node}[c] = \bot \wedge rt = \bot$ then // Passive case: create new node.
        $\mathsf{Node}[c] \leftarrow C$
    else if $\mathsf{Node}[c] \neq \bot$ then [Inj] [RndCor] Re-computing injected $c$.
        assert *consistent-nodes$(\mathsf{Node}[c], C)$
    else // [Inj] [RndCor] $c$ explains a detached root.
        Set $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} \leftarrow \mathsf{Ptr}[\mathsf{id}]$ and then replace
                each occurrence of $\mathsf{root}_{rt}$ in the HG by $c$.
        assert *consistent-nodes$(\mathsf{Node}[c], C)$
    // [Inj] Check that inserting $C$ does not violate authenticity and
    HG-consistency.
    assert *cons-invariant $\wedge$ *auth-invariant
    if $\mathrm{RndCor}[\mathsf{id}]$ then // [RndCor] Commit signed with bad rand.
        Notify $\mathcal{F}^{\text{IW}}_{\text{AS}}$ that id's current spk is compromised.
    Receive $w$ from the adversary.
    if $\mathsf{Wel}[w] \neq \bot$ then
        req *consistent-nodes$(\mathsf{Wel}[w], C)$
    $\mathsf{Wel}[w] \leftarrow c$
    return $(c, w)$

Input $(\mathtt{Process}, c, \vec{p})$ from id
    Send id and all inputs to the adv. and receive $ack$.
    // Adv. can reject invalid inputs.
    if $\neg$*require-correctness('proc', id, $c$, $\vec{p}$) then
        req $ack$
    // [Inj] Adv. interprets injected proposals.
    for $p \in \vec{p}$ s.t. $\mathsf{Prop}[p] = \bot$ do
        $\mathsf{Prop}[p] \leftarrow$ proposal node with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$,
                $\mathsf{stat} = \mathsf{adv}$, and orig and act chosen
                by the adversary.
    // Compute commit node id expects to transition to.
    Receive from the adversary $(\mathsf{orig}', \mathsf{spk}')$.
    $C \leftarrow$ commit node with $\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$, $\mathsf{orig} = \mathsf{orig}'$,
            $\mathsf{pro} = \vec{p}$ , $\mathsf{mem} =$ *members$(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{spk}')$
    // [Inj] If $c$ is injected, then assign a node to it.
    if $\mathsf{Node}[c] = \bot$ then
        // If $c$ explains a detached root, let adv. specify it.
        Receive $rt$ from the adversary.
        if $rt \neq \bot$ then
            Set $\mathsf{Node}[\mathsf{root}_{rt}].\mathsf{par} \leftarrow \mathsf{Ptr}[\mathsf{id}]$ and then
                replace each occurrence of $\mathsf{root}_{rt}$ in
                the HG by $c$.
        else
            $\mathsf{Node}[c] \leftarrow C$
            $\mathsf{Node}[c].\mathsf{stat} \leftarrow \mathsf{adv}$
    // Check that id transitions to expected node.
    assert consistent-nodes$(\mathsf{Node}[c], C)$
    // Transition id.
    if $\exists p \in \vec{p} : \mathsf{Prop}[p].\mathsf{act} = \mathsf{rem\text{-}id}$ then
        $\mathsf{Ptr}[\mathsf{id}] \leftarrow \bot$
    else
        $\mathsf{Ptr}[\mathsf{id}] \leftarrow c$
        $\mathsf{HasKey}[\mathsf{id}] \leftarrow \mathtt{true}$
    return *output-process$(C)$

Input $(\mathtt{Join}, w)$ from id
    Send id and all inputs to the adv. and receive $ack$.
    req $ack$
    // [Inj] If $w$ is injected, then assign a commit node to it.
    if $\mathsf{Wel}[w] = \bot$ then
        // If $w$ leads to existing node, adv. can specify it.
        Receive $c$ from the adversary.
        if $c \neq \bot$ then
            $\mathsf{Wel}[w] \leftarrow c$
        else
            // Create detached root.
            rootCtr++
            $\mathsf{Wel}[w] \leftarrow \mathsf{root}_{\mathsf{rootCtr}}$
            $\mathsf{Node}[\mathsf{root}_{\mathsf{rootCtr}}] \leftarrow$ commit node with
                $\mathsf{par} = \bot$, $\mathsf{pro} = \bot$, $\mathsf{stat} = \mathsf{adv}$, and orig
                and mem chosen by the adv.
    // Transition id.
    $\mathsf{Ptr}[\mathsf{id}] \leftarrow \mathsf{Wel}[w]$
    $\mathsf{HasKey}[\mathsf{id}] \leftarrow \mathtt{true}$
    // Check that joining id does not violate authenticity and HG-
    consistency.
    assert *cons-invariant $\wedge$ *auth-invariant
    return *output-join$(\mathsf{Node}[\mathsf{Wel}[w]])$
```

Fig. 2: $\mathcal{F}_{\text{CGKA}}$ on inputs process and join. Parts related to injections are marked by comments containing [Inj].

Functionality $\mathcal{F}_{\text{CGKA}}$: Corruptions and Group Key

```
Input (Expose, id) from the adversary
    // If id is in the group, the leaked information consists of:
    if Ptr[id] = ⊥ then
        // 1) secrets needed to process other parties' messages and
        potentially the group key
        Node[Ptr[id]].exp +← (id, HasKey[id])
        // 2) secrets needed to process id's own messages
        For each commit or update-proposal node with orig = id and
        par = Ptr[id], set stat ← bad.
        // 3) the signing key
        Notify $\mathcal{F}^{\text{IW}}_{\text{AS}}$ 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 +← (id, true)
    // Disallow adaptive corruptions in some cases.
    This input is not allowed if $\exists c$ s.t Node[$c$].chall = true and ¬safe($c$)

Input (CorrRand, id, $b$), $b \in$ {good, bad} from the adversary
    RndCor[id] ← $b$

Input Key from id
    // Only possible if id has the key.
    req Ptr[id] ≠ ⊥ ∧ HasKey[id]
    // Set the key if id is the first party fetching it in its node.
    (Guarantees consistency across parties.)
    if Node[Ptr[id]].key = ⊥ 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] ← false
    return Node[Ptr[id]].key
```

Fig. 3: $\mathcal{F}_{\text{CGKA}}$: inputs process, join, key and corruptions. Parts related to injections are marked by comments containing [Inj].

Functionality $\mathcal{F}_{\text{CGKA}}$: Helpers

helper *require-correctness('comm', id, c, $\vec{p}$)

Returns true if a) c and each $p \in \vec{p}$ identifies a node with stat ≠ adv, and b) Ptr[id] = Node[c].par, and c) $\vec{p}$ = Node[c].pro.

helper *require-correctness('proc', id, $\vec{p}$, spk)

Returns true if ***usable-spk**(id, spk) and $\forall p \in \vec{p}$: Prop[p] ≠ ⊥ and the vector can be committed by id (in its current node) according to MLS spec.

helper *require-correctness('prop', id, act)

Returns true if act = up-spk and ***usable-spk**(id, spk) or if act = rem-id$_t$ and removing id$_t$ is allowed according to MLS spec.

helper *usable-spk(id, spk)

Returns true if either spk is id's current spk, or id has the secret key according to $\mathcal{F}^{\text{IW}}_{\text{AS}}$.

helper *members(C, id, $\vec{p}$, spk)

Computes the member set after id, currently in C, calls commit with inputs $\vec{p}$ and spk, according to MLS spec. The set consists of tuples (id′, spk′), indicating the member's identity and his identity key.

helper *can-join(C, id)

Returns true if C.pro adds id with spk and, according to $\mathcal{F}^{\text{IW}}_{\text{KS}}$, id has a secret key for some key-package registered with spk.

helper *output-process(C)

Computes committer id$_c$ and proposal semantics propSem, returned by Process when transitioning into C.

helper *output-join(C)

Computes roster and committer id$_c$, returned when joining into C.

helper *consistent-nodes(N, N')

Returns true if all values in proposal or commit nodes N and N' except status match.

helper *auth-invariant

Returns true if there is no proposal or commit node with stat = adv and par s.t. **inj-allowed**(par, id) is false.

helper *cons-invariant

Returns true if HG has no cycles, each id is in the member set of Ptr[id] and for each non-root c, the parent of each p in c's pro vector is c's parent.

Fig. 4: Additional helpers for $\mathcal{F}_{\text{CGKA}}$.

Adaptive corruptions. Adaptive corruptions become a problem if an exposure reveals secret keys that can be used to compute a key that has already been outputted by $\mathcal{F}_{\text{CGKA}}$ at random, i.e. a "challenge" key. Since fully adaptive security is not achieved by TreeKEM (without resorting to programmable random oracles), we restrict the environment not to corrupt if for some nodes with the flag chall set to true this would cause $\mathbf{safe}$ to switch to false.[17]

Remark 1 (Correctness). Having the environment deliver messages is rather non-standard for interactive protocols. In a more "classical" UC treatment this would be done by the adversary. Our formulation, modeling arbitrary instead of worst-case network behavior, allows us to additionally consider *correctness.* In contrast, "classical" treatment typically permits trivial protocols that just reject all messages with the simulator just not delivering them in the ideal world.

4 The Insider-secure TreeKEM Protocol

This section provides a (high-level) description of the Insider-Secure TreeKEM (ITK) protocol. A formal description of the protocol can be found in App. C.

Distributed state. The primary object constituting the distributed state of the ITK protocol is the *ratchet tree* τ. The ratchet tree is a labeled binary tree (i.e., a binary tree where nodes have a number of named properties), where each group member is assigned to a leaf and each internal node represents the sub-group of parties whose leaves are part of the node's sub-tree.

To give a brief overview, each node has two (potentially empty) labels pk and sk, storing a key pair of a PKE scheme. Leaves have an additional label spk, storing a long-term signature public key of the leaf's owner. The root has a number of additional shared symmetric secret keys as labels (see below). See Fig. 5 for an example of a ratchet tree with the labels. The *public part* of τ consists of the tree structure, the leaf assignment, as well as all public labels, i.e., those storing public keys. The *secret part* consists of the labels storing secret keys and the symmetric keys. The ITK protocol maintains two invariants:

Invariant (1): The public part of τ is known to all parties.
Invariant (2): The secret labels in a node v are known only to the owners of leaves in the sub-tree rooted at v.

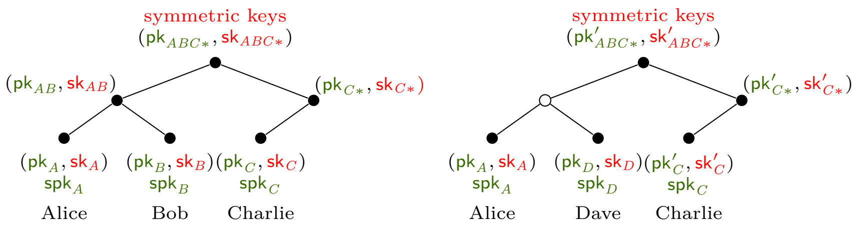

symmetric keys
$(\mathsf{pk}_{ABC*}, \mathsf{sk}_{ABC*})$
$(\mathsf{pk}_{AB}, \mathsf{sk}_{AB})$
$(\mathsf{pk}_{C*}, \mathsf{sk}_{C*})$
$(\mathsf{pk}_A, \mathsf{sk}_A)$ spk_A
Alice
$(\mathsf{pk}_B, \mathsf{sk}_B)$ spk_B
Bob
$(\mathsf{pk}_C, \mathsf{sk}_C)$ spk_C
Charlie
symmetric keys
$(\mathsf{pk}'_{ABC*}, \mathsf{sk}'_{ABC*})$
$(\mathsf{pk}'_{C*}, \mathsf{sk}'_{C*})$
$(\mathsf{pk}_A, \mathsf{sk}_A)$ spk_A
Alice
$(\mathsf{pk}_D, \mathsf{sk}_D)$ spk_D
Dave
$(\mathsf{pk}'_C, \mathsf{sk}'_C)$ spk_C
Charlie

Fig. 5: (Left) An example ratchet tree τ for a group with three members. For Invariant (1), the public labels (green) are known to all parties. For Invariant (2), the secret labels (red) in a node v are only known to parties in v's subtree, e.g. Bob knows sk_B, sk_{AB} and sk_{ABC*}. (Right) the tree after Charlie commits removing Bob and adding Dave. The empty node $\circ$ is blank. Messages to Alice and Dave are encrypted under its resolution $(\mathsf{pk}_A, \mathsf{pk}_D)$.

Evolving the tree. Each epoch has one fixed ratchet tree τ. Proposals represent changes to τ, and a commit chooses which changes should be applied when advancing to the next epoch.

A *remove* proposal represents removing from τ all keys known to the removed party (see Fig. 5). That is, its leaf is cleared, and all keys in its *direct path* — i.e., the path from the party's leaf to the

[17] In game based definitions, such corruptions are usually disallowed, as they allow to trivially distinguish. Our notion achieves the same level of adaptivity.

root — are *blanked*, meaning that all their labels are cleared. This is followed by shrinking the tree by removing unneeded leaves from the right side of the tree.[18] Note that until a blanked node gets a new key pair assigned (as explained shortly), in order to encrypt to the respective subgroup one has to encrypt to the node's children instead (and recursing if either child is blanked as well). The minimal set of non-blanked nodes covering a given subgroup is called the subgroup's *resolution*.

An *update* proposes removing all keys currently known to the party (and hence possibly affected by state leakage), and replacing the public key in their leaf (and possibly the long-term verification key) by a fresh one, specified in the proposal. Hence, τ is modified as in a remove proposal, but instead of clearing the leaf, its key is replaced.

Finally, an *add* proposal indicates the new member's identity (defined on a higher application level), its long-term public key from the AS, and an ephemeral public key from KS. It represents the following modification: First, a leaf has to be assigned, with the public label set according to the public key from the proposal. If there exists a currently unused leaf, then this can be reused, otherwise a new leaf is added to the tree. In order to satisfy invariant (2), the party committing the add proposal would then have to communicate to the new member all secret keys on its direct path. Unfortunately, it can only communicate the keys for nodes above the least common ancestor of its and the new member's leaves. For all other nodes, the new member is added to a so-called *unmerged leaf set*, which can be accounted for when determining the node's resolution.

Re-keying. Whenever a party *commits* a sequence of proposals, they additionally replace their leaf key (providing an implicit update) and re-key their direct path. In order to maintain invariant (1) on the group state, the committer includes all new public keys in the commit message.

To minimize the number of secret keys needed to be communicated as part of the commit message, the committer samples the fresh key pairs along the path by "hashing up the tree". That is, the committer derives a sequence of *path secrets* s_i, one for each node on the path, where s_0 for the leaf is random and s_{i+1} is derived from s_i using the $\mathsf{HKDF.Expand}$ function (cf. App. A.2). Then, each s_i is expanded again (with a different label) to derive random coins for the key generation. The secret s_n for the root, called the *commit secret*, is not used to generate a key pair, but instead used to derive the epoch's symmetric keys (see below). This implies that each other party only needs to be able to retrieve the path secret of the least common ancestor of their and the committer's leaves. Hence, invariant (2) can be maintained by including in the commit each path secret encrypted to (the resolution of) the node's child not on the direct path.

Note that for PCS, the new secret keys must not be computable using the committer's state from before sending the commit (we want that a commit heals the committer from a state). Hence, the committer simply stores all new secrets explicitly until the commit is confirmed.

Key schedule. Each epoch has several associated symmetric keys, four of which are relevant for this paper: The *application secret* is the key exported to the higher-level protocol, the *membership key* is used for protecting message authenticity, the *init secret* is mixed into the next epoch's key schedule, and the *confirmation key* ensures agreement on the cryptographic material.

The epoch's keys are derived from the commit secret computed in the re-keying process, mixed with (some additional context and) the previous epoch's init secret. This ensures that only parties who knew the prior epoch's secrets can derive the new keys. One purpose of this is improving FS: corrupting a party in an epoch, say, 5 must not allow to derive the application secret for a prior epoch, say, 3. As, however, some internal nodes of the ratchet tree remain unchanged between epochs 3 and 5, it might be possible for the adversary to decrypt the commit secret of epoch 3, given the leakage from epoch 5. Mixing in the init secret of epoch 2 thus ensures that this is information is of no value per se (unless some party in epoch 2 was already corrupted.)

Welcoming members. Whenever a commit adds new members to the group, the committer must send a *welcome message* to the new members, providing them with the necessary state. First, the welcome message contains the public group information, such as the public part of the ratchet tree. Second, it includes (encrypted) *joiner secret*, which combines current commit secret and previous init secret and allows the new members to execute the key schedule. Finally, it contains the seed to derive the secrets on the joint path, which the committer just re-keyed. (Recall that for the other

[18] The exact conditions under which truncation is performed are presented in App. C. We note that truncation is best-effort and does not necessarily lead to a tree of optimal depth.

nodes on the new party's direct path they are simply added to the unmerged leaves set, indicating that they do not know the corresponding secrets.) The above seeds, as well as the joiner secret, are encrypted under the public key (obtained from KS), specified in the add proposal (which thus serves dual purposes).

Security mechanisms. All messages intended for existing group members — commit messages and proposals — are subject to *message framing*, which binds them to the group and epoch, indicates the sender, and protects the message's authenticity. The sender first signs the group identifier, the epoch, his leaf index, and the message using his private signing key. This in particular prevents impersonation by another (malicious) group member.

Since the signing key, however, is shared across groups and its replacement is also not tied to the PCS guarantees of the group, each package is additionally authenticated using shared key material. Proposals are MACed using the membership key, while commit messages are protected using the confirmation tag (see below). Further, commit messages that include remove proposals are additionally MACed using the membership key, since the confirmation tag cannot be verified by the removed members. In summary, to tamper or inject messages an adversary must both know at least the sender's signing key as well as the epoch's symmetric keys.

The protocol makes use of two (running) hashes on the communication transcript to authenticate the group's history. For authentication purposes, it uses the *confirmed transcript hash*, which 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 then computed by hashing the confirmed transcript hash with the confirmation tag. Each commit message moreover contains a so-called confirmation tag that allows the receiving members to immediately verify whether they agree on the new epoch's key-schedule. To this end, the committer computes a MAC on the confirmed transcript hash under the new epoch's confirmation key.

Finally, ITK uses a mechanism called *tree signing* to achieve a certain level of insider security. We discuss this aspect in detail in Sec. 5.3.

Remark 2 (Simplifications and Deviations.). While ITK closely follows the IETF MLS protocol draft, there are some small deviations as well as some omissions. In particular, our model assumes a fixed protocol version and ciphersuite, and omits features such as advanced meta-data protection, external proposals and commits, exporters, preshared keys, as well as extensions. We discuss those deviations and their implications on our results in more detail in App. C.4.

5 Insider Attacks On MLS

We first discuss three insider attacks on the design of MLS (as it stood prior to applying the fixes proposed as part of this work). Each is practical yet violates the design goals of MLS. Next, we present an insider attack on MLS made possible when its ciphersuite is replaced by a weaker one that still meets assumptions deemed sufficient in previous analyses. Together these attacks highlight the limitations of past security notions. In Sec. 7 we also discuss several areas where the security of MLS could (at least in principle) be further improved. While some of areas will likely soon be improved upon in MLS, other areas only have either incomplete solutions or incur significant external costs (like precluding a FIPS compliant mode for MLS).

5.1 An Attack on Authenticity in Certain Modes

MLS supports two wire formats for packets: The first (called MLSCiphertext) is meant to provide extra metadata protection by applying an extra layer of authenticated symmetric encryption to the message. The second mode (MLSPlaintext) allows for additional server-assisted efficiency improvements. As part of our analysis, we realized that an MLSCiphertext (unintentionally) provides stronger authentication guarantees than an MLSPlaintext: Forging the latter requires only signature keys of a group member while the former also requires knowing the current epoch's key. (Signature keys will be rotated much less frequently than epoch keys.) To bring the authenticity guarantees in line, we proposed adding a MAC to MLSPlaintexts [11].

5.2 Breaking Agreement

The way the transcript hash was computed and included in the confirmation tag in the original proposal of MLS lead to counter-intuitive behavior, where parties think they are in-sync and agree on all relevant state when they are not.

More concretely, the package's signature was not included into the confirmed transcript hash, but it was included it into the interim transcript hash. Suppose that a malicious insider creates two valid commit messages c and c', which only differ in the signatures, and sends them to Alice and Bob respectively. If both signatures check out (which for most signatures an insider can achieve) then Alice and Bob both end up with the same confirmed transcript hash and, thus, with the same confirmation tag. Therefore, they both transition to the new epoch, agree on all epoch secrets and can exchange application messages. However, MLS messages Alice sends now include confirmation tags computed using the mismatching interim transcript hash, and hence are not accepted by Bob. Our fix that moves the signature into the confirmed transcript hash has been incorporated into MLS [12].

5.3 Inadequate Joiner Security (Tree-Signing)

The role of the tree-signing mechanism of MLS is to provide additional guarantees for joiners by leveraging the long-term signature keys distributed by the PKI. Intuitively, we may hope for the following guarantee: A joiner (potentially invited by a malicious insider to a non-existing group) ends up in a secure epoch once all malicious insiders, i.e., members whose long-term signature keys are corrupt, have been removed. A bit more precisely, a key is corrupt if the secret key is registered by or leaked to malicious insiders.

Surprisingly, we can show that the initial tree signing mechanism proposed in TreeKEMv3 does not achieve this guarantee. Rather, it achieves something much weaker: A joiner ends up in a secure epoch once all members with the following types of long-term signature keys have been removed: A) corrupt keys, B) keys used in a different epoch that includes a key of type A). We believe this to be an unexpectedly weak guarantee. In particular, it means that malicious insiders can read messages after being removed.

The attack on tree-singing. We call ITK using the tree-signing mechanism from TreeKEMv3 ITK*. We next present a simple and highly practical attack against the tree signing of ITK*. It results in groups with epochs containing no keys of type A) yet for which the epoch key is easy to compute by the malicious insiders.

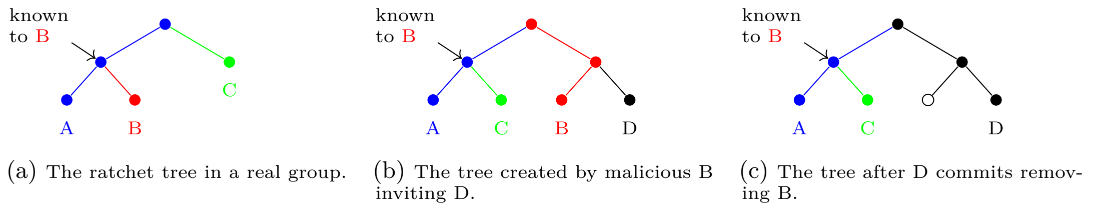
We first recall the tree signing of ITK*. It works by storing in each ratchet-tree node v a value $v.\mathsf{parentHash}$ computed as follows.

```
**if** $v.\mathsf{isroot}$ **then** $v.\mathsf{parentHash} \leftarrow \epsilon$
**else** $v.\mathsf{parentHash} \leftarrow \mathsf{Hash}(v.\mathsf{parent}.\mathsf{pk}, v.\mathsf{parent}.\mathsf{parentHash})$
```

Further, each leaf contains a signature over all its contents, including its parentHash, under the long-term key of its owner. This means that during each commit the committer signs the new parentHash of their leaf, which binds all new PKE public keys they generated. We say that the committer's signature attests to the new PKE keys. Now joiners can verify that each PKE public key in the ratchet tree they receive in the welcome message is attested to by some group member who generated it. (The joiners check the validity of the long-term keys in the PKI.)

We next show that the above check is insufficient to achieve the desired security goals. The attack is illustrated in Fig. 6. Intuitively, the problem is that committers attest to the key pairs they (honestly) generated, but *not* to which parties they informed of the secret keys (for Invariant (2) of ITK*). This allows a malicious insider to create his own ratchet tree, where he knows secrets of nodes that are not on his direct path. Therefore, removing him from the fake group doesn't cause removal of every key he knows.

Fixing tree signing. In essence, we can prevent the attack by modifying the parent hash such that committers attest to the key pairs they generated *and* to which parties were informed about the secret keys. In general, we can achieve this by computing the parent hash as $v.\mathsf{parentHash} \leftarrow \mathsf{Hash}(w.\mathsf{pk}, w.\mathsf{parentHash}, w.\mathsf{memberCert})$ where w is v's parent and memberCert attests to the set



(a) The ratchet tree in a real group. (b) The tree created by malicious B inviting D. (c) The tree after D commits removing B.

Fig. 6: The attack on the tree signing of ITK*. In the real group (a), A and C are honest, B is malicious. The keys in blue, red and green nodes are attested to by (the signatures of) A, B and C, respectively. In the attack, B copies two lowest nodes on A's path and C's leaf, and builds the ratchet tree in (b) for a fake group where he invites D. After D removes B, the group in (c) contains no insiders. However, the new application secret is known to B, because, due to the invariant (2) of ITK* in the real group, he knows the secret generated by A for the marked node.

of parties informed about the w.sk. It is left to find a good candidate for memberCert; one that is secure and easy to compute. We next discuss 3 candidates for memberCert.

The first candidate is called *the leaf parent hash*. This is the most direct solution which simply sets w.memberCert to the list of all leaves in the subtree of v.sibling that are not unmerged at w. Observe that, by Invariant (2) of ITK, the owners of these leaves, and only they, were informed about w.sk (recall that the unmerged leaves are defined as those that do not know w.sk). One disadvantage of the leaf hash is that it is not very implementation-friendly.

The second candidate, called *the tree parent hash*, has been initially considered for MLS [33]. It basically sets w.memberCert to the tree hash of v.sibling with the unmerged leaves omitted (recall that ITK computes the tree hash as the Merkle hash of the ratchet tree). Observe that the tree hash binds strictly more than the leaf hash. The tree hash would be more straightforward to compute. Unfortunately, it is not workable due to other mechanisms of MLS.[19]

Therefore, we propose a new candidate called *the resolution parent hash*. It improves upon the leaf hash in 2 ways: it is more implementation-friendly and it has slightly better deniability properties.[20] The resolution hash sets memberCert to the PKE public keys of nodes in u.origChildResolution where u.origChildResolution is the resolution of u with the unmerged leaves of u.parent omitted. Observe that u.origChildResolution is the resolution of u at the time the last committer in the subtree of v generated the key pair of w.

The reason this works is less direct than in the case of leaf and tree hashes. Intuitively, assume all long-term keys in the subtree of w are uncorrupted. The honest committer who generated w's key pair attests to w.pk and all PKE keys in u.origChildResolution, i.e. those they encrypted w.sk to. These PKE keys are in turn attested to by the honest members in their subtrees who generated them. Applying this argument recursively and relying on the security of the encryption scheme, we can conclude that all key pairs in the ratchet tree remain secure.

5.4 IND-CPA Security Is Insufficient

Most statements about security of MLS's CGKA protocol only assume IND-CPA security of the PKE scheme (in fact, to our knowledge, [17] is the only one assuming IND-CCA). However, there are PKE schemes that are IND-CPA secure but that make MLS clearly insecure against active attackers — despite MLS employing signatures and MACs to protect authenticity — highlighting the inadequacies of those works' simplified security models to account for all relevant aspects (and the danger of analyzing too piecemeal protocols without considering their composition in general). In particular, we show an attack where a malicious insider can decrypt messages after he has been removed from the group.

We will use an IND-CPA secure scheme PKE* with the following property: a ciphertext ctx containing a message m can be modified into ctx_i, such that decrypting ctx_i outputs $\perp$ if and only

[19] With adds and removes, the subtree of v can grow or shrink since the last commit, changing the tree hash. It is not clear how to revert these changes.

[20] With the leaf hash, members sign each other's credentials, thus attesting to being in a group together. The resolution hash gets rid of this side effect.

if the i-th bit of m is 0, and otherwise decrypting ctx_i outputs m (we explain how to construct such scheme shortly). Now say a malicious insider Bob and an honest Alice are on the opposite ends of the ratchet tree (Alice on the left), and Alice sends a commit c. Let ctx be the ciphertext included in c that encrypts the path secret s for the left child of the root. Bob can learn s bit-by-bit by injecting Alice's message with ctx replaced by ctx_i, for different i, to different users in the left subtree. If a user accepts (which Bob can see because the user sends messages for the new epoch) then the i-th bit of the seed is 1, else 0. (Note that Bob knows the key schedule for both the old and the new epoch, so he can compute any confirmation keys.) With s, Bob knows the secret key sk in the left child of the root. If a user in the right subtree removes Bob, then sk allows him to compute all group secrets in the new epoch.

Finally, we note that PKE^* can be easily obtained as a straightforward adaptation of the symmetric encryption scheme by Krawczyk [28] to the public key setting. The artificial scheme from [28] was used to show that the authenticate-then-encrypt paradigm is not secure in general.

6 Security of ITK

Security of ITK is expressed by the predicates **safe**(c, id) and **inj-allowed**(c, id), where c is a commit message identifying a history graph node and id is a party. The predicates are formally stated in Fig. 7. They are defined using recursive deduction rules **know**(c, id) and **know**$(c, \text{'epoch'})$, indicating that the adversary knows id's secrets (such as the leaf secret), and that it knows the epoch secrets (such as the init secret), respectively. In more detail:

- **know**(c, id) consists of three conditions, the last two being recursive. Condition a) is true if id's secrets in c are known to the adversary because they leaked as part of an exposure or were injected by the adversary in id's name (due to many attack vectors, this can happen in many ways, see Fig. 7). The conditions b) and c) reflect that in ITK only commits sent by or affect id (id updates, is added, or removed) are guaranteed to modify all id's secrets. If c is not of this type, then **know**(c, id) is implied by **know**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})$ (condition b)). If a child c' of c is not of this type, then it is implied by **know**(c', id) (condition c)).
- **know**$(c, \text{'epoch'})$ takes into account the fact that ITK derives epoch secrets using the initSecret from the previous epoch, and hence achieves slightly better FS compared to parties' individual secrets.
  In particular, the adversary knows the epoch secrets in c only if it corrupted a party in c, or knows the epoch secrets in c's parent and knows individual secret of some party id in c. The latter condition allows the adversary to process c using id's protocol and is formalized by the ***can-traverse** predicate.
- The only difference between $\neg$**safe**(c) and **know**$(c, \text{'epoch'})$ is that the application secret is not leaked if id is exposed in c after outputting it.

Remark 3. Previous works [5, 3] defined a simpler **safe** predicate by defining the set of history graph nodes where application secrets are affected by an exposure. Then, a node's secret is secure if there is no exposure that affects it. However, in our setting a set of simultaneous exposures may leak information that is not leaked by any of the exposures alone, as illustrated in Fig. 8.

With the predicates **safe** and **inj-allowed**, we can now state the following security statement for ITK.

Theorem 1. *Assuming that* PKE *is* IND-CCA *secure, and that* Sig *is* EUF-CMA *secure, the* ITK *protocol securely realizes* $(\mathcal{F}^{\text{IW}}_{\text{AS}}, \mathcal{F}^{\text{IW}}_{\text{KS}}, \mathcal{F}_{\text{CGKA}})$ *in the* $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where* $\mathcal{F}_{\text{CGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Fig. 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 (Sktech). We here provide the high level proof idea; the complete proof is presented in App. D. The proof proceeds in three steps. The first step is to show that various consistency mechanisms, such as MACing the group context, guarantee consistency of the distributed group state. More precisely, the real world (Hybrid 1) is indistinguishable from the following Hybrid

Predicate safe

Knowledge of parties' secrets.

$\textbf{know}(c, \mathsf{id}) \iff$

a) // id's state leaks directly e.g. via corruption (see below):
$\quad$ $\textbf{*state-directly-leaks}(c, \mathsf{id}) \;\vee$

b) // know state in the parent:
$\quad$ $(\mathsf{Node}[c].\mathsf{par} \neq \bot \wedge \neg\textbf{*secrets-replaced}(c, \mathsf{id}) \wedge \textbf{know}(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})) \;\vee$

c) // know state in a child:
$\quad$ $\exists c' : (\mathsf{Node}[c'].\mathsf{par} = c \wedge \neg\textbf{*secrets-replaced}(c', \mathsf{id}) \wedge \textbf{know}(c', \mathsf{id}))$

$\textbf{*state-directly-leaks}(c, \mathsf{id}) \iff$

a) // id has been exposed in c:
$\quad$ $(\mathsf{id}, *) \in \mathsf{Node}[c].\mathsf{exp} \;\vee$

b) // c is in a detached tree and id's spk is exposed
$\quad$ $\exists rt : \mathsf{*ancestor}(\mathsf{root}_{rt}, c) \wedge \exists \mathsf{spk} : (\mathsf{id}, \mathsf{spk}) \in \mathsf{Node}[c].\mathsf{mem} \wedge \mathsf{spk} \in \mathsf{Exposed} \;\vee$

c) // id's secrets in c are injected by the adversary:
$\quad$ $((\mathsf{id}, \mathsf{spk}) \in \mathsf{Node}[c].\mathsf{mem} \wedge \textbf{*secrets-injected}(c, \mathsf{id}))$

$\textbf{*secrets-injected}(c, \mathsf{id}) \iff$

a) // id is the sender of c and c was injected or generated with bad randomness
$\quad$ $(\mathsf{Node}[c].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Node}[c].\mathsf{stat} \neq \mathsf{good}) \;\vee$

b) // c commits an update of id that is injected or generated with bad randomness
$\quad$ $\exists p \in \mathsf{Node}[c].\mathsf{pro} : (\mathsf{Prop}[p].\mathsf{act} = \mathsf{up}\text{-}* \wedge \mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Prop}[p].\mathsf{stat} \neq \mathsf{good}) \;\vee$

c) // c adds id with corrupted spk
$\quad$ $\exists p \in \mathsf{Node}[c].\mathsf{pro} : (\mathsf{Prop}[p].\mathsf{act} = \mathsf{add\text{-}id\text{-}spk} \wedge \mathsf{spk} \in \mathsf{Exposed})$

$\textbf{*secrets-replaced}(c, \mathsf{id}) \iff \mathsf{Node}[c].\mathsf{orig} = \mathsf{id} \vee \exists p \in \mathsf{Node}[c].\mathsf{pro} :$
$\quad$ $\mathsf{Prop}[p].\mathsf{act} \in \{\mathsf{add\text{-}id\text{-}}*, \mathsf{rem\text{-}id}\} \vee (\mathsf{Prop}[p].\mathsf{act} = \mathsf{up}\text{-}* \wedge \mathsf{Prop}[p].\mathsf{orig} = \mathsf{id})$

Knowledge of epoch secrets.

$\textbf{know}(c, \text{'epoch'}) \iff \mathsf{Node}[c].\mathsf{exp} \neq \varnothing \vee \textbf{*can-traverse}(c)$

// Can the adversary process c using exposed individual secrets and parent's init secret?
$\textbf{*can-traverse}(c) \iff$

a) // orphan root with a corrupted signature public key:
$\quad$ $(\mathsf{Node}[c].\mathsf{par} = \bot \wedge (*, \mathsf{spk}) \in \mathsf{Node}[c].\mathsf{mem} \wedge \mathsf{spk} \in \mathsf{Exposed}) \;\vee$

b) // commit to an add proposal that uses an exposed key package:
$\quad$ $(\exists p \in \mathsf{Node}[c].\mathsf{pro} : \mathsf{Prop}[p].\mathsf{act} = \mathsf{add\text{-}id\text{-}spk} \wedge \mathsf{spk} \in \mathsf{Exposed}) \;\vee$

c) // secrets encrypted in the welcome message under an exposed leaf key
$\quad$ $\textbf{*leaf-welcome-key-reuse}(c) \;\vee$

d) // know necessary info to traverse the edge:
$\quad$ $(\textbf{know}(c, *) \wedge (c = \mathsf{root}_* \vee \textbf{know}(\mathsf{Node}[c].\mathsf{par}, \text{'epoch'})))$

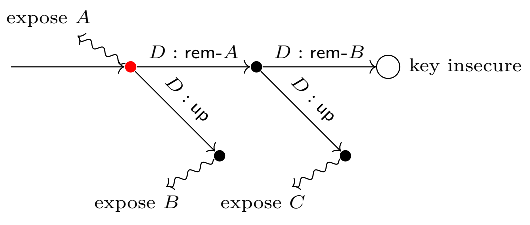
$\textbf{*leaf-welcome-key-reuse}(c) \iff \exists \mathsf{id}, p \in \mathsf{Node}[c].\mathsf{pro} : \mathsf{Prop}[p].\mathsf{act} = \mathsf{add\text{-}id\text{-}}* \wedge \exists c_d : \mathsf{*ancestor}(c, c_d)$
$\quad$ $\wedge\ (\mathsf{id}, *) \in \mathsf{Node}[c_d].\mathsf{exp} \wedge$ no node c_h with $\textbf{*secrets-replaced}(c_h, \mathsf{id})$ on c-c_d path

Safe and can-inject.

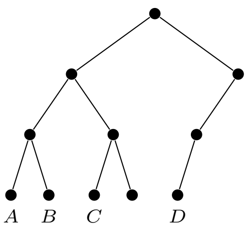
$\textbf{safe}(c) \iff \neg\Big((*, \mathtt{true}) \in \mathsf{Node}[c].\mathsf{exp} \vee \textbf{*can-traverse}(c)\Big)$

$\textbf{inj-allowed}(c, \mathsf{id}) \iff \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}] \in \mathsf{Exposed} \wedge \textbf{know}(c, \text{'epoch'})$

Fig. 7: The safety and injectability predicates for the CGKA functionality reflecting the sub-optimal security of the ITK protocol.



(a) The history graph.



(b) The ratchet tree in the red node.

Fig. 8: An execution illustrating that many simultaneous corruptions leak information that cannot be deduced from any single corruption. Exposing A reveals the $\mathsf{initSecret}$. Exposing B reveals secrets on his direct path (untouched by D's update). Together this allows to process D's commit removing A and compute the next $\mathsf{initSecret}$. Together with C's exposed secrets, this allows to process the commit removing B and compute the group key. Notice that states of A and B cannot be used to process the last commit (they are not group members).

2: The experiment includes a modified CGKA functionality, $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$, which differs from $\mathcal{F}_{\text{CGKA}}$ in that it uses **safe** = `false` and **inj-allowed** = `true`. The functionality interacts with the trivial simulator who sets all keys and messages according to the protocol. The second step is to show that IND-CCA of the PKE scheme guarantees confidentiality: Hybrid 2 is indistinguishable from Hybrid 3 where application and membership secrets in safe epochs are random, i.e. the original **safe** is restored. The final step is to show that unforgeability of the MAC and signature schemes implies that Hybrid 3 is indistinguishable from the ideal world, where the original **inj-allowed** is restored as well. (Considering confidentiality before integrity, while somewhat unusual, is necessary, because we must first argue secrecy of MAC keys. We note that IND-CPA would be anyway insufficient, because some injections are inherently possible.)

In this overview, we sketch the core of our proof, which is the second step concerning confidentiality. For simplicity, we do not consider randomness corruptions. We now proceed in two parts: first, we consider only passive environments, which do not inject messages. In the second part, we show how to modify the passive strategy to deal with active environments.

Part 1: Passive security. For simplicity, consider $\mathcal{F}_{\text{CGKA}}^{\texttt{rand}}$, which uses the original **safe** only for the first (safe) key it sets (think of the first step in the hybrid argument). The goal is to show that *IND-CPA* security of the PKE scheme implies that $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$ and $\mathcal{F}_{\text{CGKA}}^{\texttt{rand}}$, both with the trivial simulator, are indistinguishable for *passive* environments.

Unfortunately, already the passive setting turns out challenging for the following reason: The path secrets in a (safe) commit c are encrypted under public keys created in another commit c', which contains encryptions of the corresponding secret keys under public keys created in another commit c'', and so on. Moreover, the keys are related by hash chains (of path secrets). Even worse, the environment can adaptively choose who to corrupt, revealing some subset of the secret keys, which mean that we cannot simply apply the hybrid argument to replace encryptions of secret keys by encryptions of zeros.[21]

To tackle adaptivity and related keys, we adapt the techniques of [31, 5]. Namely, we define a new security notion for PKE, called (modified) Generalized Selective Decryption (GSD),[22] which generalizes the way ITK uses PKE together with the hash function to derive its secrets. Roughly speaking, the GSD game creates a graph, where each node stores a secret seed. The adversary can instruct the game to 1) create a node with a random seed, 2) create a node v where the seed is a hash of the seed of another node u, 3) use a (different) hash of the seed in a node u to derive a key pair, use the public key to encrypt the seed in a node v and send the public key and ciphertext to the adversary. Each of the actions 2) and 3) creates an edge (u, v) to indicate their relation. Moreover, the adversary can adaptively corrupt nodes and receive their seeds. For the challenge of

[21] Observe that at the time a ciphertext is created we do not know if the key it contains will be used to create a safe epoch, or if some receiver will be corrupted.

[22] GSD was first defined for symmetric encryption [31] and then extended to prove security of TreeKEM [5]. Our notion is an extension of [5].

the game, she receives either a seed from a sink node or a random value. (See the full proof for a precise definition.)[23] It remains to be shown that 1) GSD security implies secrecy of ITK keys, and 2) IND-CPA security implies GSD security. The latter proof is adapted from [5], so we now focus on 1).

To be a bit more concrete, assume an environment $\mathcal{Z}$ distinguighes between $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$ and $\mathcal{F}_{\text{CGKA}}^{\texttt{rand}}$ (each with the trivial simulator). We construct an adversary $\mathcal{A}$ against GSD security of the PKE scheme in the standard way: $\mathcal{A}$ executes the code of $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$ and the trivial simulator, except for all honest commits and updates, public keys and epoch keys are created using the GSD game. If a party is corrupted, $\mathcal{A}$ corrupts all GSD nodes needed to compute its state. Finally, $\mathcal{A}$ replaces the first key outputted by $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$ by its challenge.

Part 2: Injections. We sketch the main points of how the strategy from the passive setting can be adapted to show that IND-CCA security of PKE implies secrecy of keys in the presence of active environments. There are three types of messages $\mathcal{Z}$ can inject: proposals, commits and welcome messages. Proposals are the least problematic. Say $\mathcal{Z}$ injects an update proposal p' with public key pk' on behalf of Alice. Since Alice will never process a commit containing p', allegedly from her, that she did not send (see Fig. 15 on Page 35), all epochs created by such commits and their descendants are not safe until Alice is removed. This also removes pk' and any secrets encrypted to it. So, $\mathcal{A}$ can generate all secrets sent to pk' itself, as they don't matter for any safe epoch.

Now say $\mathcal{Z}$ makes Bob process an injected commit c' and assume Bob uses an honest key, i.e., one created in the GSD game for an uncorrupted node. Say Bob's ciphertext in c' is *ctxt*. There are a few possible scenarios:

– $\mathcal{A}$ has never seen *ctxt* (e.g. because $\mathcal{Z}$ computed a commit in his head). Clearly, IND-CPA is not sufficient here. Hence, we extend the GSD game by a decrypt oracle (which does not work on ciphertexts that allow to trivially compute the challenge) and prove that the new notion is implied by IND-CCA.
– $\mathcal{A}$ generated *ctxt* using the GSD game, as part of a commit message c creating a safe epoch (note that c and c' may differ in places other than *ctxt*). Now the decrypt oracle cannot be used, but fortunately the confirmation tag comes to the rescue. Indeed, any tag accepted by Bob allows $\mathcal{A}$ to extract the joiner in c from $\mathcal{Z}$'s RO queries (we soon explain how) and compute the application secret in c. Hence, $\mathcal{A}$ can request GSD challenge for this secret and win.
  For simplicity, assume c and c' are siblings, i.e., Bob is currently in c's parent (see the full proof for other cases). Recall that the tag is a MAC under the new epoch's confirmation key over the transcript hash, and that the transcript hash contains the whole commit message c or c' (except the tag). The MAC is modeled as an RO call on input (confirmation key, transcript hash), so the only way for $\mathcal{Z}$ to compute a valid tag for c' is to query the RO on input (confirmation key in c', transcript hash updated with c'). Moreover, the confirmation key is a hash of the joiner secret, so $\mathcal{A}$ can extract the joiner secret in c' as well (note that the joiner secret is never encrypted). Now observe that the joiner secret is a hash of the init and commit secrets. Moreover, the init secret is the same in c and c', since they are siblings, as is the commit secret due to *ctxt* being the same. Hence, the joiner secret of c is the same as the one extracted from c'. □

7 Sub-optimal Security of ITK

Our analysis uncovered a couple of aspects where security of ITK could be improved, but the changes were not adopted by MLS for non-cryptographic reasons.

Forward-secrecy. Forward-secrecy of ITK is sub-optimal for two reasons. First, when adding a party id_t, the same key pair is used for the leaf of id_t and to encrypt id_t's secrets in the welcome message. As a result, if id_t is exposed before it updates its leaf, the key can be used to process the welcome message and recover secrets for all past epochs. (Using separate key pairs for the two tasks is a simple and cheap solution to this issue.)

[23] The GSD game in the full proof is inherently more complex. For example, recall that joiner secret is a hash of init and commit secrets. Accordingly, the adversary is allowed to create nodes whose seeds are hashes of two other seeds.

Second, since a committer refreshes only keys on their direct path, exposing a party allows to recover past commit secrets. Now leaking a past init secret (e.g. by exposing some other party that is later removed) allows to recover past epoch secrets. Using a key-updatable encryption scheme would improve this [3] but doing this would require first designing a compatible tree-signing mechanism.

Protection against bad randomness. ITK supports ECDSA signatures, which leak the secret key in case randomness is reused. Independently, the effect of bad randomness on all operations can be (partly) mitigated by a so-called randomness pool, mixing the fresh randomness with the secret state. Then, bad randomness would not compromise a previously secure state but just hamper PCS. Unfortunately, not supporting ECDSA would leave MLS with no FIPS compliant mode which can be problematic in practice. [24]

Multi-group security. We prove security of ITK for a single group. In particular, this means that re-using PKI keys between groups is outside of our model. Interestingly, our statement would imply multi-group security by composition if we used a global PKI functionality in the sense of [21]. Unfortunately, this does not work for ITK for two reasons. First, GUC was envisioned to formalize strong deniability, where protocol executions can be simulated without the secret PKI keys and hence do not constitute a proof of participation. However, the fact that ITK signs messages makes it non-deniable.[25] Moreover, the typical techniques used to cope with non-deniable protocols in GUC (e.g. [22]) rely on strict domain separation on the cryptographic primitives, e.g., a values signed during the execution of one protocol instance cannot be used in another instance. However, this is not true for ITK, where for example, signed key packages can be used across groups. We leave proving or disproving multi-group security of ITK as an important open problem.

References

[1] Messagying layer security (mls) wg - meeting minutes for interim 2020-1, January 2020. `https://datatracker.ietf.org/doc/minutes-interim-2020-mls-01-202001110900/`.

[2] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. Analysing the HPKE standard. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 87–116. Springer, Heidelberg, October 2021.

[3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of mls. In *ACM SIGSAC Conference on Computer and Communications Security — CCS 2021*, page 1463–1483, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284, 2021. Full version: `https://eprint.iacr.org/2019/1489`.

[6] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreementwith active security. In *Theory of Cryptography — TCC 2020*, 2020. Full version: `https://eprint.iacr.org/2020/752.pdf`.

[7] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456, 2021. `https://ia.cr/2021/1456`.

[8] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS 2006*, volume 4189 of *LNCS*, pages 424–443. Springer, Heidelberg, September 2006.

[9] R. Barnes, B. Beurdouche, , J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol (draft-ietf-mls-protocol-12). Technical report, IETF, Mar 2020. `https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/12/`.

[24] The non-FIPS compliant ciphersuites do not suffer from this issue and so, somewhat ironically, using them may result in a *more* secure insanitation than the FIPS ones.

[25] Encrypting the signatures would not help, as corrupted parties leak decryption keys.

[10] Richard Barnes. Subject: [MLS] Remove without double-join (in TreeKEM). MLS Mailing List, 06 August2018 13:01UTC. https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik.

[11] Richard Barnes. MLS Protocol Pull Requests #396: Authenticate group membership in MLSPlaintext, 18 August 2020. https://github.com/mlswg/mls-protocol/pull/396.

[12] Richard Barnes. MLS Protocol Pull Requests #416: Inlclude the signature in the confirmation tag, 18 August 2020. https://github.com/mlswg/mls-protocol/pull/416.

[13] Richard Barnes. Subject: [MLS] Proposal: Proposals (was: Laziness). MLS Mailing List, 22 August 2019 22:17UTC. https://mailarchive.ietf.org/arch/msg/mls/5dmrkULQeyvNu5k3MV_sXreybj0/.

[14] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups, May 2018. Published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8.

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

[16] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.

[17] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. https://eprint.iacr.org/2021/137.

[18] Bushing, Marcan, Segher, and Sven. Console hacking 2010 — PS3 epic fail. In *27th Chaos Communication Congress — 27C3*, 2010. https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.

[19] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.

[20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[21] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[22] Ran Canetti, Daniel Shahaf, and Margarita Vald. Universally composable authentication and key-exchange with global PKI. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 265–296. Springer, Heidelberg, March 2016.

[23] 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 David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.

[24] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477, 2019. https://eprint.iacr.org/2019/477.

[25] Yevgeniy Dodis, Thomas Ristenpart, John P. Steinberger, and Stefano Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 348–366. Springer, Heidelberg, August 2012.

[26] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient pkes. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1441–1462, 2021.

[27] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 180–210. Springer, Heidelberg, December 2019.

[28] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331. Springer, Heidelberg, August 2001.

[29] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.

[30] Matthew A. Miller. Messaging layer security (mls) wg - meeting minutes for ietf105, August 2019. https://datatracker.ietf.org/doc/minutes-105-mls/.

[31] Saurabh Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 21–40. Springer, Heidelberg, February 2007.

[32] Eric Rescorla. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List, 03 May 2018 14:27UTC. `https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no`.

[33] Nick Sullivan. Subject: [MLS] Virtual interim minutes. MLS Mailing List, 29 January 2020 21:39UTC. `https://mailarchive.ietf.org/arch/msg/mls/ZZAz6tXj-jQ8nccf7SyIwSnhivQ/`.

[34] Matthew Weidner. Group messaging for secure asynchronous collaboration. MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann, 2019. `https://mattweidner.com/acs-dissertation.pdf`.

A Preliminaries

A.1 Notation

We denote the security parameter by κ and all our algorithms implicitly take 1^κ as input. For an algorithm A, we write $A(\cdot; r)$ to denote that A is run with explicit randomness r. We use $v \leftarrow x$ to denote assigning the value x to the variable v and $v \leftarrow\$ S$ to denote sampling an element u.a.r. from a set S.

Data structures. If V denotes a variable storing a set, then we write $V \mathrel{+\!\!\leftarrow} x$ and $V \mathrel{-\!\!\leftarrow} x$ as shorthands for $V \leftarrow V \cup \{x\}$ and $V \leftarrow V \setminus \{x\}$, respectively. For vectors $x \coloneqq (x_1, \ldots, x_n)$ and $y \coloneqq (y_1, \ldots, y_m)$ we denote the concatenation by $x \mathbin{+\!\!+} y = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ and use $x \mathrel{+\!\!+\!\!\leftarrow} v$ as a shorthand for $x \leftarrow x \mathbin{+\!\!+} (v)$. Moreover, let $x.\mathsf{reverse}() \coloneqq (x_n, x_{n-1}, \ldots, x_1)$ and let $x.\mathsf{indexof}(z)$ denote the smallest $i \in \mathbb{N}$ such that $x_i = z$ (or $\perp$ if not such i exists). Finally, let $\mathsf{zip}(x, y) \coloneqq ((x_1, y_1), \ldots, (x_n, y_n))$ if $n = m$, or $\perp$ otherwise. We further make use of associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to denote assignment and retrieval of element i, respectively. Additionally, we denote by $A[*] \leftarrow v$ the initialization of the array to the default value v. In a slight abuse of notation, for sets of tuples $S \subseteq \mathcal{X} \times \mathcal{Y}$, we define $S[x] \coloneqq \{y \mid (x, y) \in S\}$, akin to associative arrays.

For simplicity we moreover use wildcard notation when dealing with sets of tuples and multi-argument associative arrays. For instance, for an array with domain $\mathcal{I} \times \mathcal{J}$, we write $A[*, j] \coloneqq \{A[i, j] \mid i \in \mathcal{I}\}$ and for a set $S \subseteq \mathcal{I} \times \mathcal{J}$ we write $(i, *) \in S$ as a shorthand for the condition $\exists j \in \mathcal{J} : (i, j) \in S$.

Keywords. In the pseudocode, we use the following keywords:

- **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and immediately returns $\perp$.
- **parse** $(m_1, \ldots, m_n) \leftarrow m$ denotes an attempt to parse a message m as a tuple. If m is not of the correct format, the current function unwinds all state changes and immediately returns $\perp$.
- **try** $y \leftarrow *\mathsf{func}(x)$ is a shorthand notation for calling a helper $*\mathsf{func}$ and executing **req** $y \neq \perp$.
- **assert** *cond* is only used to describe functionalities. It denotes that if *cond* is false, then the given functionality permanently halts, making the real and ideal worlds trivially distinguishable (this is used to validate inputs of the simulator).

A.2 Cryptographic Primitives

We recall the basic cryptographic primitives used throughout this work.

Signature Scheme. A signature scheme is a tuple of PPT algorithms $\mathsf{Sig} \coloneqq (\mathsf{Sig.kg}, \mathsf{Sig.sign}, \mathsf{Sig.vrf})$. For a public/secret key pair $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{Sig.kg}()$ from the key-generation algorithm, we denote signing by $\mathsf{sig} \leftarrow \mathsf{Sig.sign}(\mathsf{ssk}, m)$, and the verification by $\mathsf{Sig.vrf}(\mathsf{spk}, \mathsf{sig}, m)$. We require the standard existential unforgeability under chosen message attacks (EUF-CMA) notion.

Public Key Encryption. A public key encryption scheme is a tuple of algorithms $\mathsf{PKE} \coloneqq (\mathsf{PKE.kg}, \mathsf{PKE.enc}, \mathsf{PKE.dec})$. For a public/secret key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{pk}()$ from the key-generation algorithm, we denote encryption by$c \leftarrow \mathsf{PKE.enc}(\mathsf{pk}, m)$, and decryption by $m \leftarrow \mathsf{PKE.dec}(\mathsf{sk}, c)$. We require the standard indistinguishability under chosen ciphertext (IND-CCA2) notion.

Message Authentication Code. A message authentication code (MAC) scheme is a tuple of algorithms $\mathsf{MAC} \coloneqq (\mathsf{MAC.tag}, \mathsf{MAC.vrf})$. For a uniformly random key k, we denote by $t \leftarrow \mathsf{MAC.tag}(k, m)$ the tagging algorithm and by $\mathsf{MAC.vrf}(k, t, m)$ the respective verification algorithm.

Proving ITK secure requires two non-standard assumptions on the MAC: *extractability* and *collision resistance*. The first assumption means that from a valid tag, it is possible to extract the corresponding message and key (in the sense of a proof of knowledge). The second assumption means that an adversary should not be able to come up with any collision $\mathsf{MAC.tag}(k_1, m_1) = \mathsf{MAC.tag}(k_2, m_2)$ for $(k_1, m_1) \neq (k_2, m_2)$. Neither assumption is implied by EUF-CMA security.

To this end, we model the MAC in the random oracle model (ROM). That is, in the security proof we simply replace all calls to $\mathsf{MAC.tag}(k, m)$ by invocations of $RO(k, m)$ and $\mathsf{MAC.vrf}$ simply comparing the tags. Note that for HMAC, as used by MLS, this assumption is valid if the underlying compression function is assumed to be a random oracle [25].

HKDF. The *HMAC-based Extract-and-Expand Key Derivation Function* is a tuple of algorithms $\mathsf{HKDF} = (\mathsf{HKDF.Extract}, \mathsf{HKDF.Expand})$. The extraction algorithm $k \leftarrow \mathsf{HKDF.Extract}(s_0, s_1)$ outputs a u.a.r key if either s_0 or s_1 has high min-entropy. The expansion algorithm $k_{\mathsf{lbl}} \leftarrow \mathsf{HKDF.Expand}(k, \mathsf{lbl})$, given a key k, outputs an independent u.a.r. key for each (public) label lbl.

We model its security in the ROM. Note that MLS' requirement of the extraction being secure if either input has high (conditional) min-entropy anyway deviates from the HKDF RFC and the respective standard security notion [29].

Hash Function. Finally, we use a generic hash function Hash, mapping from an arbitrary input space to a fixed length output. For security, we use the ROM as well.

B Details of the Security Model

*The PKI functionalities.*The formal description of the Authentication Service and Keypackage Service functionalities can be found in Fig. 9.

Note that the ideal-world version of the Authentication Service, $\mathcal{F}_{\text{AS}}^{\text{IW}}$, the adversary gets to provide both the secret and public keys, whenever a party requests to register a spk. This reflects that those keys in the ideal world merely function as identifiers and do not convey any significance with respect to security. Indeed, whether a key is considered secure or not, is tracked via the $\mathsf{Exposed}$ set, which reflects whether a given key is known to the adversary in the *real world* (e.g., having leaked or having been sampled with bad randomness). Moreover, observe that the functionality treats keys registered by the adversary conservatively: it only treats keys from other honest parties as secure (e.g., when registering the spk of id, that has never leaked, also for id') and assumes for all other keys that the adversary might know the corresponding secret key.

For the Keypackage Service, observe that in the using an ssk to register a key package with bad randomness is assumed to leak ssk. As a result, the Authentication Service is notified of this leakage, and (in the real world) ssk is handed to the adversary. (In the ideal world, the adversary chose ssk in the first place.) This behavior reflects that the key package is (potentially) signed using ssk and that MLSallows to use of ECDSA that exhibits this leakage.

*The CGKA functionality.*The formal description of the $\mathcal{F}_{\text{CGKA}}$ functionality is depicted in Figs. 10 to 12. We refer to Sec. 3.5 for a high-level description thereof.

One difference from Sec. 3.5 we would like to highlight is the functionality related to so-called "add-only" mode of commits in MLS. That is, if the committed proposal vector contains only adds (and is not empty), then MLS *permits* skipping the implicit update of the committer. We model this with the $\mathsf{force\text{-}rekey}$ flag inputted to commit: if $\mathsf{force\text{-}rekey} = \mathtt{false}$, then an add-only commit does not do the implicit update, whereas if either $\mathsf{force\text{-}rekey} = \mathtt{true}$ or there are non-add proposals, then the implicit update is performed. (Skipping the update also implies ignoring the new spk.)

C Details on the ITK Protocol

C.1 Protocol State

The ratchet tree. Formally, the ratchet tree τ is a left-balanced binary tree with n nodes, LBBT_n.

Definition 1 (Left-Balanced Binary Tree). *For $n \in \mathbb{N}$ the n^{th}* left-balanced binary tree *is denoted by LBBT_n. Specifically, LBBT_1 is the tree consisting of one node. Furthermore, if $m = \mathsf{mp2}(n) := \max\{2^p : p \in \mathbb{N} \wedge 2^p < n\}$, then LBBT_n is the (undirected) tree whose root has left and right subtrees LBBT_m and LBBT_{n-m}.*

We use the following indexing of nodes (see Fig. 13 for an example): all nodes are numbered left to right — i.e., according to an in-order depth-first traversal of the tree — starting with 0.

For a node v of a LBBT τ, we use standard object oriented notation as outlined in Table 1. (Observe that every *internal* node always has both children.)

Functionality $\mathcal{F}_{\text{AS}}$ and $\boxed{\mathcal{F}^{\text{IW}}_{\text{AS}}}$

The functionality is parameterized by a key generation algorithm $\mathsf{gen\text{-}sk}()$.

```
Initialization
  Registered ← ∅ // registered identity key pairs
  Exposed ← ∅ // exposed public keys
  SSK[*, *] ← ⊥ // honestly generated secret keys
  RndCor[*] ← good

Inputs from a party id

Input (register-spk)
  if RndCor[id] = good then
      (spk, ssk) ←$ gen-sk()
  else
      Send (rnd, id) to the adversary and receive $r$.
      (spk, ssk) ←$ gen-sk($r$)
  Send (sample-ssk, id) to the adversary and receive (spk, ssk).
  if RndCor[id] ≠ good then
      Exposed +← spk
  SSK[id, spk] ← ssk
  Registered +← (id, spk)
  Send (register-spk, id, spk) to the adversary.
  Send spk to the party id.

Input (verify-cert, id′, spk)
  Send (id′, spk) ∈ Registered to id.

Input (get-ssk, spk)
  Send SSK[id, spk] to the party id.

Input (del-ssk, spk)
  SSK[id, spk] ← ⊥

Inputs from the adversary

Input (register-spk, id, spk)
  if (*, spk) ∉ Registered then
     Exposed +← spk
  Registered +← (id, spk)

Input (expose, id)
  Exposed +← {spk | SSK[id, spk] ≠ ⊥}
  Send SSK[id, *] to the adversary.

Input (corRand, id, $b$), $b$ ∈ {good, bad}
  RndCor[id] ← $b$

Inputs from $\mathcal{F}_{KS}$.

Input (exposed, id, spk)
  Exposed +← spk
  Send SSK[id, spk] to the adversary.

Inputs from $\mathcal{F}_{CGKA}$.

Input (has-ssk, spk, id)
  Send SSK[id, spk] ≠ ⊥ to $\mathcal{F}_{CGKA}$.
```

Functionality $\mathcal{F}_{\text{KS}}$ and $\boxed{\mathcal{F}^{\text{IW}}_{\text{KS}}}$

The functionality is parameterized by a key-package generation algorithm $\mathsf{gen\text{-}kp}(\mathsf{id}, \mathsf{spk}, \mathsf{ssk})$.

```
Initialization
  SK[*, *], SPK[*, *] ← ⊥ // secret keys and spk's corresponding to
  honestly generated keys
  RndCor[*] ← good

Inputs from a party id

Input (register-kp, spk, ssk)
  if RndCor[id] = good then
      (kp, sk) ←$ gen-kp(id, spk, ssk)
      if kp = ⊥ then return
  else
      Send (rnd, id) to the adversary and receive $r$.
      (kp, sk) ← gen-kp(id, spk, ssk; $r$)
      if kp = ⊥ then return
      Send (exposed, id, spk) to $\mathcal{F}_{AS}$.
      Send ssk to the adversary.
  Send (sample-sk, id, spk, ssk) to the adversary and receive
  (kp, sk, $ack$).
  if ¬$ack$ then return
  if RndCor[id] ≠ good then
      Send (exposed, id, spk) to $\mathcal{F}_{AS}$.
  SK[id, kp] ← sk
  SPK[id, kp] ← spk
  Send (register-pk, id, spk, kp) to the adversary.
  Send kp to the party id.

Input get-sks
  Send {(kp, SK[id, kp]) | SK[id, kp] ≠ ⊥} to id.

Input (get-kp, id′)
  Send (get-kp, id, id′) to the adversary and receive kp.
  Send kp to id.

Input (del-sk, spk)
  SK[id, kp] ← ⊥

Inputs from the adversary

Input (expose, id)
  Send SK[id, *] to the adversary.

Input (corRand, id, $b$), $b$ ∈ {good, bad}
  RndCor[id] ← $b$
```

Fig. 9: The Authentication and Key Service functionalities $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$, and their ideal-world counterparts $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and $\mathcal{F}^{\text{IW}}_{\text{KS}}$. Code marked by $\boxed{\text{solid}}$ or dashed boxes are executed only by the respective version, whereas code outside those boxes is shared by both variants.

Functionality $\mathcal{F}_{\text{CGKA}}$

The functionality expects as part of the instance's session identifier sid the group creator's identity $\mathsf{id}_{\mathsf{creator}}$. It is parameterized in the predicates **safe**(c), specifying which keys are confidential, and **inj-allowed**(c, id), specifying when authenticity is not guarantees.

```
Initialization
  $Ptr[*], Node[*], Prop[*], Wel[*] \leftarrow \bot$
  $RndCor[*] \leftarrow good$; $HasKey[*] \leftarrow \texttt{false}$
  $rootCtr \leftarrow 0$

Inputs from $id_{creator}$
Input (Create, spk)
  req $Node[root_0] = \bot \wedge$ *valid-spk$(id_{creator}, spk)$
  $mem \leftarrow \{id_{creator}, spk\}$
  $Node[root_0] \leftarrow$ *create-root$(id_{creator}, mem, RndCor[id_{creator}])$
  $HasKey[id_{creator}] \leftarrow \texttt{true}$; $Ptr[id_{creator}] \leftarrow root_0$

Inputs from a party id

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

Input (Commit, $\vec{p}$, spk, force-rekey)
    req $Ptr[id] \neq \bot$
    Send (Commit, id, $\vec{p}$, spk, force-rekey) to the adversary
        and receive $(ack, c, w, rt)$.
    if $\neg$*req-correctness('comm', id, $\vec{p}$, spk, force-rekey) then
        req $ack$
    *fill-props$(id, \vec{p})$
    if $\neg$force-rekey $\wedge$ *only-adds$(\vec{p})$ then
        $spk \leftarrow Node[Ptr[id]].mem[id]$
    assert *valid-spk(id, spk)
    $mem \leftarrow$ *members$(Ptr[id], id, \vec{p}, spk)$
    assert $mem \neq \bot \wedge (id, spk) \in mem$
    if $Node[c] = \bot \wedge rt = \bot$ then
        if $\neg$force-rekey $\wedge$ *only-adds$(\vec{p})$ then $stat \leftarrow bad$
        else $stat \leftarrow RndCor[id]$
        $Node[c] \leftarrow$ *create-child$(Ptr[id], id, \vec{p}, mem, stat)$
    else
        if $Node[c] = \bot$ then $c' \leftarrow root_{rt}$
        else $c' \leftarrow c$
        *consistent-comm$(c', id, \vec{p}, mem)$
        if $c \neq c'$ then *attach$(c, c', id, \vec{p})$
    assert $w \neq \bot$ iff $\exists p \in \vec{p} : Node[p].act = add\text{-}*$
    if $w \neq \bot$ then
        assert $Wel[w] \in \{\bot, c\}$
        $Wel[w] \leftarrow c$
    assert cons-invariant $\wedge$ auth-invariant
    if $RndCor[id] = bad$ then
        Send (exposed, id, $Node[Ptr[id]].mem[id]$) to $\mathcal{F}_{AS}$.
    return $(c, w)$

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

Input (Process, $c, \vec{p}$)
    Send (Process, id, $c, \vec{p}$) to the adv, receive $(ack, rt, orig', spk')$.
    if $\neg$*req-correctness('proc', id, $c, \vec{p}$) then
        req $ack$
    *fill-props$(id, \vec{p})$
    if $Node[c] = \bot \wedge rt = \bot$ then
        $mem \leftarrow$ *members$(Ptr[id], orig', \vec{p}, spk')$
        assert $mem \neq \bot \wedge$ inj-allowed$(Ptr[id], id)$
        $Node[c] \leftarrow$ *create-child$(Ptr[id], orig', \vec{p}, mem, adv)$
    else
        if $Node[c] = \bot$ then $c' \leftarrow root_{rt}$
        else $c' \leftarrow c$
        $id_c \leftarrow Node[c'].orig$
        $spk_c \leftarrow Node[c'].mem[id_c]$
        $mem \leftarrow$ *members$(Ptr[id], id_c, \vec{p}, spk_c)$
        assert $mem \neq \bot$
        *valid-successor$(c', id, \vec{p}, mem)$
        if $c \neq c'$ then *attach$(c, c', id, \vec{p})$
    if $\exists p \in \vec{p} : Prop[p].act = rem\text{-}id$ then
        $Ptr[id] \leftarrow \bot$
    else
        assert $id \in Node[c].mem$
        $Ptr[id] \leftarrow c$
        $HasKey[id] \leftarrow \texttt{true}$
    assert cons-invariant $\wedge$ auth-invariant
    return *output-proc$(c)$

Input (Join, $w$)
    Send (Join, id, $w$) to the adv, receive $(ack, c', orig', mem')$.
    req $ack$
    $c \leftarrow Wel[w]$
    if $c = \bot$ then
        if $Node[c'] \neq \bot$ then $c \leftarrow c'$
        else
            rootCtr++
            $c \leftarrow root_{rootCtr}$
            $Node[c] \leftarrow$ *create-root$(orig', mem', adv)$
        $Wel[w] \leftarrow c$
    $Ptr[id] \leftarrow c$
    $HasKey[id] \leftarrow \texttt{true}$
    assert $id \in Node[c].mem \wedge$ cons-invariant $\wedge$ auth-invariant
    return *output-join$(c)$

Corruptions

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

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

Fig. 10: The CGKA functionality. The helper functions are defined in Figs. 11 and 12. The safety predicates for ITK are defined in Fig. 7.

Functionality $\mathcal{F}_{\text{CGKA}}$: Bookkeeping Helpers

```
// Creating nodes
helper *create-child($c, \mathsf{id}, \vec{p}, \mathsf{mem}, \mathsf{stat}$)
    return new node with $\mathsf{par} \leftarrow c$, $\mathsf{orig} \leftarrow \mathsf{id}$, $\mathsf{pro} \leftarrow \vec{p}$,
        $\mathsf{mem} \leftarrow \mathsf{mem}$, $\mathsf{stat} \leftarrow \mathsf{stat}$.

helper *create-root($\mathsf{id}, \mathsf{mem}, \mathsf{stat}$)
    return new node with $\mathsf{par} \leftarrow \bot$, $\mathsf{orig} \leftarrow \mathsf{id}$,
        $\mathsf{pro} \leftarrow \bot$, $\mathsf{mem} \leftarrow \mathsf{mem}$, $\mathsf{stat} \leftarrow \mathsf{stat}$.

helper *create-prop($c, \mathsf{id}, \mathsf{act}, \mathsf{stat}$)
    return new proposal with $\mathsf{par} \leftarrow c$, $\mathsf{orig} \leftarrow \mathsf{id}$,
        $\mathsf{act} \leftarrow \mathsf{act}$, $\mathsf{stat} \leftarrow \mathsf{stat}$.

helper *fill-props($\mathsf{id}, \vec{p}$)
    for $p \in \vec{p}$ s.t. $\mathsf{Prop}[p] = \bot$ do
        Send ($\texttt{Proposal}, p$) to the adversary and receive ($\mathsf{orig}, \mathsf{act}$).
        $\mathsf{Prop}[p] \leftarrow$ *create-prop($\mathsf{Ptr}[\mathsf{id}], \mathsf{orig}, \mathsf{act}, \mathsf{adv}$)

// Does the vector of proposals create an add-only commit?
helper *only-adds($\vec{p}$)
    return $\vec{p} \neq () \wedge \forall p \in \vec{p} : \mathsf{Prop}[p] \neq \bot \wedge \mathsf{Prop}[p].\mathsf{act} = \mathsf{add}\text{-}*$

// Output of process and join
helper *output-proc($c$)
    $(*, \mathsf{propSem}) \leftarrow$ *apply-props($c, \mathsf{Node}[c].\mathsf{pro}$)
    return ($\mathsf{Node}[c].\mathsf{orig}, \mathsf{propSem}$)

helper *output-join($c$)
    return ($\mathsf{Node}[c].\mathsf{mem}, \mathsf{Node}[c].\mathsf{orig}$)

// Is the (new) $\mathsf{spk}'$ valid for update or commit?
helper *valid-spk($\mathsf{id}, \mathsf{spk}'$)
    $\mathsf{spk} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}[\mathsf{id}]$
    if $\mathsf{spk} \neq \bot \wedge \mathsf{spk}' = \mathsf{spk}$ then return true
    Send ($\texttt{has-ssk}, \mathsf{spk}', \mathsf{id}$) to $\mathcal{F}_{\text{AS}}$ and receive $ack$
    return $ack$

// Generating the group key (secure or insecure)
helper *set-key($c$)
    if ¬safe($c$) then
        Send ($\texttt{Key}, \mathsf{id}$) to the adversary and receive $I$.
        $\mathsf{Node}[c].\mathsf{key} \leftarrow I$
        $\mathsf{Node}[c].\mathsf{chall} \leftarrow \texttt{false}$
    else
        $\mathsf{Node}[c].\mathsf{key} \leftarrow\!\!\$\ \mathcal{I}$
        $\mathsf{Node}[c].\mathsf{chall} \leftarrow \texttt{true}$

// Corruptions
helper *update-stat-after-exp($\mathsf{id}$)
    for each $p$ s.t. $\mathsf{Prop}[p] \neq \bot$ and
        (a) $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$ and
        (b) $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id}$ and
        (c) $\mathsf{Prop}[p].\mathsf{act} = \mathsf{up}$
    do $\mathsf{Prop}[p].\mathsf{stat} \leftarrow \mathsf{bad}$
    for each $c$ s.t. $\mathsf{Node}[c] \neq \bot$ and
        (a) $\mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$ and
        (b) $\mathsf{Node}[c].\mathsf{orig} = \mathsf{id}$
    do $\mathsf{Node}[c].\mathsf{stat} \leftarrow \mathsf{bad}$
```

Functionality $\mathcal{F}_{\text{CGKA}}$: Consistency Helpers

```
helper *consistent-prop($p, \mathsf{id}, \mathsf{act}, \mathsf{stat}$)
    // Preexisting $p$ valid for $\mathsf{id}$ proposing $\mathsf{act}$?
    assert $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Prop}[p].\mathsf{act} = \mathsf{act}$
    assert $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$

helper *valid-successor($c, \mathsf{id}, \vec{p}, \mathsf{mem}$)
    // Preexisting node valid for $\mathsf{id}$ processing $(c, \vec{p})$?
    assert $\mathsf{Node}[c] \neq \bot \wedge \mathsf{Node}[c].\mathsf{mem} = \mathsf{mem}$
    assert $\mathsf{Node}[c].\mathsf{pro} \in \{\bot, \vec{p}\} \wedge \mathsf{Node}[c].\mathsf{par} \in \{\bot, \mathsf{Ptr}[\mathsf{id}]\}$

helper *consistent-comm($c, \mathsf{id}, \vec{p}, \mathsf{mem}$)
    // Preexisting $c$ valid for $\mathsf{id}$ committing $\vec{p}$?
    assert *valid-successor($c, \mathsf{id}, \vec{p}, \mathsf{mem}$)
    assert $\mathrm{RndCor}[\mathsf{id}] \neq \mathsf{good} \wedge \mathsf{Node}[c].\mathsf{orig} = \mathsf{id}$

helper *attach($c, c', \mathsf{id}, \vec{p}$)
    // Attach detached root $c'$ under new name $c$ as successor of $\mathsf{id}$'s node.
    assert $c' \neq \mathsf{root}_0$
    $\mathsf{Node}[c'].\mathsf{par} \leftarrow \mathsf{Ptr}[\mathsf{id}]$; $\mathsf{Node}[c'].\mathsf{pro} \leftarrow \vec{p}$; $\mathsf{Node}[c] \leftarrow \mathsf{Node}[c']$; $\mathsf{Node}[c'] \leftarrow \bot$
    for $w : \mathsf{Wel}[w] = c'$ do $\mathsf{Wel}[w] \leftarrow c$
```

Functionality $\mathcal{F}_{\text{CGKA}}$: Correctness Helpers

```
helper *req-correctness('comm', $\mathsf{id}, \vec{p}, \mathsf{spk}, \mathsf{force}\text{-}\mathsf{rekey}$)
    return *apply-props($\mathsf{id}, \vec{p}, \mathsf{spk}$) $\neq \bot$ // $p$ is valid
        $\wedge$ (*valid-spk($\mathsf{id}, \mathsf{spk}$) $\vee$ (*only-adds($\vec{p}$) $\wedge \neg\mathsf{force}\text{-}\mathsf{rekey}$))

helper *req-correctness('proc', $\mathsf{id}, c, \vec{p}$)
    return $\mathsf{Node}[c] \neq \bot \wedge \mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}] \wedge \mathsf{Node}[c].\mathsf{pro} = \vec{p}$
        $\wedge\ \mathsf{Node}[c].\mathsf{stat} \neq \mathsf{adv} \wedge \forall p \in \vec{p} : \mathsf{Prop}[p].\mathsf{stat} \neq \mathsf{adv}$

helper *req-correctness('prop', $\mathsf{id}, \mathsf{act}$)
    if $\mathsf{act} = \mathsf{rem}\text{-}\mathsf{id}_t$ then
        return $\mathsf{id}_t \in \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}$
    else if $\mathsf{act} = \mathsf{up}\text{-}\mathsf{spk}$ then
        return *valid-spk($\mathsf{id}, \mathsf{spk}$)
    else if $\mathsf{act} = \mathsf{add}\text{-}\mathsf{id}_t$ then
        return false // Adv can always deliver bad key package
```

Functionality $\mathcal{F}_{\text{CGKA}}$: Invariants

```
// No injections when authenticity guaranteed.
helper auth-invariant
return true iff
 a) $\forall c$ with $c_p = \mathsf{Node}[c].\mathsf{par}$ and $\mathsf{id} = \mathsf{Node}[c].\mathsf{orig}$,
      if $\mathsf{Node}[c].\mathsf{stat} = \mathsf{adv}$ then inj-allowed($c_p, \mathsf{id}$) and
 b) $\forall p$ with $c_p = \mathsf{Prop}[p].\mathsf{par}$ and $\mathsf{id} = \mathsf{Prop}[p].\mathsf{orig}$,
      if $\mathsf{Prop}[p].\mathsf{stat} = \mathsf{adv}$ then inj-allowed($c_p, \mathsf{id}$).

// The history graph is consistent.
helper cons-invariant
return true iff
 a) $\forall c$ s.t. $\mathsf{Node}[c].\mathsf{par} \neq \bot$: $\mathsf{Node}[c].\mathsf{pro} \neq \bot$ and
      $\forall p \in \mathsf{Node}[c].\mathsf{pro} : \mathsf{Prop}[p].\mathsf{par} = \mathsf{Node}[c].\mathsf{par}$ and
 b) $\forall \mathsf{id}$ s.t. $\mathsf{Ptr}[\mathsf{id}] \neq \bot : \mathsf{id} \in \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}$ and
 c) the graph contains no cycles
```

Fig. 11: The helper functions for $\mathcal{F}_{\text{CGKA}}$.

Functionality $\mathcal{F}_{\text{CGKA}}$: Group State Helpers

```
helper *members(c, id_c, p⃗, spk_c)
    $(G, *) \leftarrow$ *apply-props$(\mathsf{id}_c, \vec{p}, \mathsf{spk}_c)$
    if $(G, *) = \bot$ then return $\bot$
    else return $G$

helper *apply-props(c, id_c, p⃗, spk_c)
    // Returns group members $G$ and proposal semantics $P$ resulting
    from applying $\vec{p}$ to state $\mathsf{Node}[c]$, or $\bot$ if $\vec{p}$ is invalid.
    req $\mathsf{Node}[c] \neq \bot \wedge (\mathsf{id}_c, *) \in \mathsf{Node}[c].\mathsf{mem}$
    req $\forall p \in \vec{p} : \mathsf{Prop}[p] \neq \bot \wedge \mathsf{Prop}[p].\mathsf{par} = c$
    req $\vec{p} = \vec{p}_{\mathsf{up}} +\!\!+ \vec{p}_{\mathsf{rem}} +\!\!+ \vec{p}_{\mathsf{add}}$ for some $\vec{p}_{\mathsf{up}}, \vec{p}_{\mathsf{rem}}, \vec{p}_{\mathsf{add}}$
                with $\forall \mathsf{act}\, \forall p \in \vec{p}_{\mathsf{act}} : \mathsf{Node}[p].\mathsf{act} = \mathsf{act}\text{-}*$
    $G \leftarrow \mathsf{Node}[c].\mathsf{mem}$; $G \mathrel{-\!\!\leftarrow} (\mathsf{id}_c, *)$; $G \mathrel{+\!\!\leftarrow} (\mathsf{id}_c, \mathsf{spk}_c)$
    $L \leftarrow \{\mathsf{id}_c\}$ // set of updated parties
    for $p \in \vec{p}_{\mathsf{up}}$ do
        $(\mathsf{id}_s, \mathsf{up\text{-}spk}) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
        req $\mathsf{id}_s \in G \setminus L$
        $G \mathrel{-\!\!\leftarrow} (\mathsf{id}_s, *)$; $G \mathrel{+\!\!\leftarrow} (\mathsf{id}_s, \mathsf{spk})$
        $L \mathrel{+\!\!\leftarrow} \mathsf{id}_s$
    for $p \in \vec{p}_{\mathsf{rem}}$ do
        $(\mathsf{id}_s, \mathsf{rem\text{-}id}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
        req $\mathsf{id}_s \in G \wedge \mathsf{id}_t \in G \setminus L$
        $G \mathrel{-\!\!\leftarrow} (\mathsf{id}_t, *)$
    for $p \in \vec{p}_{\mathsf{add}}$ do
        $(\mathsf{id}_s, \mathsf{add\text{-}id}_t\text{-}\mathsf{spk}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
        req $\mathsf{id}_s \in G \wedge \mathsf{id}_t \notin G$
        $G \mathrel{+\!\!\leftarrow} (\mathsf{id}_t, \mathsf{spk}_t)$
    $P \leftarrow ((\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act}) : p \in \vec{p})$
    return $(G, P)$
```

Fig. 12: The helper functions for $\mathcal{F}_{\text{CGKA}}$.

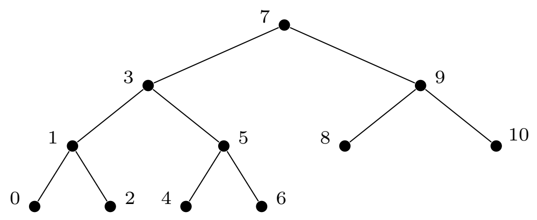


Fig. 13: The tree LBBT_6 with node indices.

A basic operation of ITK requires adding leaves to (data structures that represent) LBBTs. We describe the algorithm addLeaf which takes as input an LBBT and a new leaf inserting it to obtain an output tree LBBT_{n+1}.

Definition 2 (addLeaf). *The algorithm* $\mathsf{addLeaf}(\tau, v)$ *takes input a tree* τ *with root* r *and a fresh leaf* v *and returns a new tree* τ'. *Let* τ_{L} *and* τ_{R} *be the left and right subtrees of* r.

– *If* $\tau = \mathsf{FT}_n$ *(for some* $n \in \mathbb{N}$*) then create a new root* r' *for* τ'. *Attach* r *as the left child and* v *as the right child.*
– *Otherwise let* $\tau' = \tau$ *except that* τ_{R} *is replaced by* $\mathsf{addLeaf}(\tau_{\mathsf{R}}, v)$.

Lemma 1 (from [3]). $\tau = \mathsf{LBBT}_n \implies \mathsf{addLeaf}(\tau, v) = \mathsf{LBBT}_{n+1}$.

Moreover, observe that addLeaf preserves node indices and, thus, in particular also leaf indices. This will turn out to be a crucial property for the ITK protocol, which addresses group members by leaf indices.

Second, ITK occasionally truncates the tree by pruning the right-most leaf, using the following operation.

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

Table 1: Object oriented notation for LBBTs.

Definition 3 ($\mathsf{pruneRightmost}$)**.** *The algorithm* $\mathsf{pruneRightmost}(\tau)$ *takes input a tree* τ *with rightmost node* v_{max} *and returns a new tree* τ' *in which* v_{max} *is removed and its parent* $v_{\mathsf{max}}.\mathsf{parent}$ *replaced by the parent's former left child* $v_{\mathsf{max}}.\mathsf{parent}.\mathsf{lchild}$.

Lemma 2. $\tau = \mathsf{LBBT}_n \implies \mathsf{pruneRightmost}(\tau) = \mathsf{LBBT}_{n-1}$.

Proof. This follows by realizing that $\mathsf{pruneRightmost}$ essentially undoes $\mathsf{addLeaf}$.

Node Labels. Recall that each node v of the LBBT has several labels associated. They are outlined in Table 2. To simplify the protocol's description, we will furthermore make use of the helper methods from Table 3. Observe that the *direct path* of a leaf consists of the (ordered list) of all nodes on the path from the leaf to the root, without the leaf itself. The *co path*, on the other hand, consists of the children of the direct path's nodes that are not on the direct path themselves. That is, for every node on the direct path its sibling node is on the co path. Note that the co path contains the sibling leaves but not the root and, thus, is of equal length to the direct path. The *resolution* of a node v is the minimal set of descendant non-blank nodes that covers the whole sub-tree rooted at v, i.e., such that for every descendant u of v there exists node w in the resolution such that w is non-blank and w an ancestor of u.

Additional State. The protocol's state γ consists of the ratchet tree $\gamma.\tau$ and a number of additional variables, listed in Table 4 (recall that the protocol implicitly knows the party's identity id).

There are two aspects worth mentioning. First, the state contains three hashes: the tree hash of the LBBT's public part and two transcript hashes called *confirmed transcript hash* and *interim transcript hash*. The latter additionally contains the authentication data of the last commit message, which the confirmed transcript hash cannot contain yet to avoid cyclic dependencies. Second, if the member issued an update proposal or commit message that did not get confirmed by the delivery service yet, then the corresponding secret keys are stored in the $\gamma.\mathsf{pendUp}$ and $\gamma.\mathsf{pendCom}$ maps, respectively.

The so-called *group context* is comprised of the group id, the epoch number, the tree hash, and the confirmed transcript hash together. The corresponding helper method is defined in Table 5.

C.2 Setup Algorithms

Figure 14 depicts the algorithms $\mathsf{gen\text{-}sk}$ and $\mathsf{gen\text{-}kp}$, which are used by the Authentication Service and Key Service functionalities $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$, respectively.

The algorithm $\mathsf{gen\text{-}sk}$ generates a new key pair of a signature scheme. The algorithm $\mathsf{gen\text{-}kp}$ samples a fresh key pair of a PKE scheme and outputs the secret-key and a so-called *key package*. The key package is a signed tuple consisting the party's identity id, the PKE public key pk, and the verification key spk. As the same key package format is also used as the data structure stored in leaves, it can optionally also contain a parent hash. We model this here as an optional input which is set to ϵ if not provided — in the MLS protocol draft, the parent hash is an extension of the key package.

$v.\mathsf{pk}$	The public key of a public-key encryption scheme.
$v.\mathsf{sk}$	The corresponding secret key.
$v.\mathsf{parentHash}$	A hash value binding the node to all of its ancestors.
$v.\mathsf{unmergedLvs}$	The set of leaf indices below v, for which the corresponding party does not know $v.\mathsf{sk}$.
$v.\mathsf{id}$	If $v.\mathsf{isleaf}$: the identity associated with that leaf.
$v.\mathsf{leafId}$	If $v.\mathsf{isleaf}$: a unique identifier of the leaf (set by the protocol).
$v.\mathsf{spk}$	If $v.\mathsf{isleaf}$: an associated verification key of a signature scheme.
$v.\mathsf{sig}$	If $v.\mathsf{isleaf}$: A signature of the leaf's labels under the singing key corresponding to $v.\mathsf{spk}$.

Table 2: The node labels of the LBBTs.

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 $\bot$.
τ.roster()	Returns the identities of all parties in the tree.
τ.leaves[leafId]	Returns the leaf with identifier leafId.
τ.leafof(id)	Returns the leaf identifier of the v for which v.id = id.
τ.allotLeaf(newId)	Checks that no leaf with id newId exists (otherwise fails). Then, finds the leaf v with the lowest nodeIdx for which $\neg v$.inuse(), or adds a new leaf v using addLeaf. Finally, assigns v.leafId $\leftarrow$ newId.
τ.directPath(leafId)	Returns the direct path, excluding the leaf, as an ordered list from the leaf to root.
τ.coPath(leafId)	Returns the co-path to τ.directPath(leafId) as an ordered list.
τ.lca(leafId$_1$, leafId$_2$)	Returns the least common ancestor of the two leafs.
τ.blankPath(leafId)	Calls v.blank() on all $v \in \tau$.directPath(leafId).
τ.mergeLeaves(leafId)	Sets v.unmergedLvs $\leftarrow \varnothing$ for all $v \in \tau$.directPath(leafId)
τ.unmergeLeaf(leafId)	Sets v.unmergedLvs $+\!\!\leftarrow$ leafId for all v returned by τ.directPath(leafId)
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 $\bot$.
v.blank()	Sets all labels except parentHash to $\bot$.
v.resolution()	Return $\begin{cases} (v) \mathbin{+\!\!+} v.\mathsf{unmergedLvs} & \text{if } v.\mathsf{inuse}() \\ v.\mathsf{lchild.resolution}() \mathbin{+\!\!+} v.\mathsf{lchild.resolution}() & \text{else if } \neg v.\mathsf{isleaf} \\ () & \text{else.} \end{cases}$
v.resolvent(u)	For a descendant u of v, returns the (unique) node in v.resolution() which is an ancestor of u.

Table 3: Helper methods defined on the LBBT nodes.

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 public keys to each party id$'$.
γ.pendUp[$*$]	A mapping associating the secret keys for each pending update proposal issued by id.
γ.pendCom[$*$]	A mapping associating the new group state for each pending commit issued by id.
γ.appSecret	The current epoch's CGKA key.
γ.membKey	The key used to MAC packages.
γ.initSecret	The next epoch's init secret.

Table 4: The protocol state.

γ.leafId()	Returns $\gamma.\tau$.leafof(id).leafId.
γ.groupCtxt()	Returns (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash).

Table 5: Helper method on the protocol state.

Protocol ITK : Setup Algorithms

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

Algorithm gen-kp(id, spk, ssk, parentHash = ε)
    (pk, sk) ← PKE.kg()
    sig ← Sig.sign(ssk, (id, pk, spk, parentHash))
    kp ← (id, pk, spk, parentHash, sig)
    return (kp, sk)
```

Fig. 14: The algorithms gen-sk and gen-kp, used by $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$, respectively.

C.3 Protocol Algorithms

The main (UC) protocol is depicted in Fig. 15. The helper functions are depicted in Figs. 16 to 18. In contrast to the main protocol they handle state explicitly, clearly indicating what state they rely on (as input) and what state they modify (as return value).

Group creation. The group can be created (by the designated group creator $\mathsf{id}_{\mathsf{creator}}$ in our UC model) using the input $(\mathtt{Create}, \mathsf{spk})$.This input sets up the state of a group with a single member, whose initial signature public key spk to be used is specified as part of the input. The creator then fetches the respective signing key ssk from the setup $\mathcal{F}_{\text{AS}}$ using the helper method `*fetch-ssk-if-nec` from Fig. 17.

Proposals. To create an update proposal, the protocol generates a fresh key package kp together with the respective secret key sk. The key package kp is used as the proposal, whereas sk is stored in $\gamma.\mathsf{pendUp}$ to be used once the proposal is applied. In case a new signing key ssk is passed, the protocol furthermore fetches the respective secret key from $\mathcal{F}_{\text{AS}}$. To create an add proposal, the protocol fetches a key package for the added party from $\mathcal{F}_{\text{KS}}$. The proposal then simply consists of the key package, which includes the party's identity. A remove proposal simply consists of the removed party's leaf index.

All proposals are then *framed* using `*frameProp` (see Fig. 18). Framing first signs the proposal P together with the string '$\mathsf{proposal}$', the group context, the group id, the epoch index, and the sender's leaf index to bind it to the current cryptographic context. This in particular prevents impersonation by another (malicious) group member. Since the signing key, however, is shared across groups and its replacement is also not tied to the PCS guarantees of the group, everything (including the signature) is additionally MACed using the membership key. In summary, to tamper or inject messages an adversary must both know at least the sender's signing key as well as the epoch's symmetric keys. The actual proposal package p then consists of everything except the group context.

Commit. Upon an input $(\mathtt{Commit}, \vec{p}, \mathsf{spk}, \mathsf{force\text{-}rekey})$, the protocol initializes the next epoch's state by copying the current one. It then proceeds to apply the proposals using `*apply-props` (see Fig. 16). Alongside, it verifies the validity of each proposal, in particular their MACs and signatures.

If it is not an add-only commit (i.e, not all proposals are adds or $\mathsf{force\text{-}rekey}$ is $\mathtt{true}$) the protocol then re-keys its direct path using the helper method `*rekey-path`. The keys are derived bottom to top using the $\mathsf{HKDF.Expand}$ function (cf. App. A.2) with the labels 'node' and 'path' for key's randomness and the next seed, respectively. The seeds are then encrypted to the resolution of the respective child in the co-path.

To complete the implicit update, the protocol furthermore generates a new leaf key package. This leaf key package gets bound to its ancestor nodes (i.e., the committers freshly sampled direct path) by including a *parent hash* which is computed top to bottom by each node storing a hash of its parent node (see `*set-parent-hash` from Fig. 17). This process is called *tree signing.* It is supposed to guarantee newly joining parties that each internal node has been sampled by one of the parties contained in its subtree. As a consequence, once all malicious parties have been removed from (an arbitrary) group, all keys have been generated by the remaining honest parties.

Next, ITK prepares a preliminary commit message C including hashes of the applied proposals and the updated direct path (including the leaf). This commit message is then signed alongside the cryptographic context (using `*signCommit`) analogous to the framing of proposals. Afterwards, the protocol computes the so-called *confirmation tag* (see `*conf-tag`) — a MAC on the confirmed transcript hash updated by C and the signature (see `*set-conf-trans-hash`) under the new epoch's $\mathsf{confKey}$. The confirmation tag also serves the purpose of a MAC included in framing of proposals.

Observe that removed members cannot verify $\mathsf{confTag}$, because they do not know the new epoch's $\mathsf{confKey}$. Therefore, if some members are removed, ITK additionally MAC's the commit message under the current epoch's membership key. The MAC is only verified by the removed members and serves the purpose of the MAC insluded in framing of proposals.

If new members were added, ITK generates a *welcome message* for them using `*welcome-msg`. The welcome message contains the public group state — the group identifier, the current epoch, the public part of the ratchet tree, and the confirmed and interim transcript hashes — as well as

Protocol ITK

```
Input (Create, spk)
  req $\gamma = \bot \wedge \mathsf{id} = \mathsf{id}_{\mathsf{creator}}$
  $\gamma.\mathsf{groupId}, \gamma.\mathsf{initSecret} \leftarrow\$ \{0,1\}^\kappa$
  $\gamma.\mathsf{epoch} \leftarrow 0$
  $\gamma.\mathsf{interimTransHash} \leftarrow \epsilon$
  $\gamma.\mathsf{certSpks}[*], \gamma.\mathsf{pendUp}[*], \gamma.\mathsf{pendCom}[*] \leftarrow \bot$
  $\gamma.\tau \leftarrow \mathrm{new}\ \mathsf{LBBT}_1$
  try $\gamma.\mathsf{ssk} \leftarrow$ *fetch-ssk-if-nec($\gamma, \mathsf{spk}$)
  $(\mathsf{kp}, \mathsf{sk}) \leftarrow\$ \mathsf{gen\text{-}kp}(\mathsf{id}, \mathsf{spk}, \mathsf{ssk}, \epsilon)$
  $\gamma.\tau.\mathsf{leaves}[0].\mathsf{leafId} \leftarrow \mathsf{Hash}(\mathsf{kp})$
  $\gamma.\tau.\mathsf{leaves}[0].\mathsf{assignKp}(\mathsf{kp})$
  $\gamma.\tau.\mathsf{leaves}[0].\mathsf{sk} \leftarrow \mathsf{sk}$

Input (Propose, up-spk)
  req $\gamma \neq \bot$
  try $\mathsf{ssk} \leftarrow$ *fetch-ssk-if-nec($\gamma, \mathsf{spk}$)
  $(\mathsf{kp}, \mathsf{sk}) \leftarrow\$ \mathsf{gen\text{-}kp}(\mathsf{id}, \mathsf{spk}, \mathsf{ssk}, \epsilon)$
  $P \leftarrow (\text{`upd'}, \mathsf{kp})$
  $p \leftarrow$ *frameProp($\gamma, P$)
  $\gamma.\mathsf{pendUp}[p] \leftarrow (\mathsf{ssk}, \mathsf{sk})$
  return $p$

Input (Propose, add-id$_t$)
  req $\gamma \neq \bot \wedge \mathsf{id}_t \notin \gamma.\tau.\mathsf{roster}()$
  $\mathsf{kp}_t \leftarrow$ query (get-pk, $\mathsf{id}_t$) to $\mathcal{F}_{\mathrm{KS}}$
  try $\gamma \leftarrow$ *validate-kp($\gamma, \mathsf{kp}_t, id_t, \epsilon$)
  $P \leftarrow (\text{`add'}, \mathsf{kp}_t)$
  $p \leftarrow$ *frameProp($\gamma, P$)
  return $p$

Input (Propose, rem-id$_t$)
  req $\gamma \neq \bot \wedge \mathsf{id}_t \in \gamma.\tau.\mathsf{roster}()$
  $\mathsf{leafId}_t \leftarrow \gamma.\tau.\mathsf{leafof}(it_t)$
  $P \leftarrow (\text{`rem'}, \mathsf{leafId}_t)$
  $p \leftarrow$ *frameProp($\gamma, P$)
  return $p$

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

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

Input (Join, $w$)
  req $\gamma = \bot$
  parse $(\mathsf{encGroupSecs}, \mathsf{groupInfo}) \leftarrow w$
  $\gamma.\mathsf{certSpks}[*], \gamma.\mathsf{pendUp}[*], \gamma.\mathsf{pendCom}[*] \leftarrow \bot$
  parse $(\mathsf{groupInfoTBS}, \mathsf{sig}) \leftarrow \mathsf{groupInfo}$
  parse $(\gamma.\mathsf{groupId}, \gamma.\mathsf{epoch}, \gamma.\mathsf{treeHash}, \gamma.\mathsf{confTransHash},$
                         $\gamma.\mathsf{interimTransHash}, \gamma.\tau, \mathsf{confTag}, \mathsf{senderId})$
                                                      $\leftarrow \mathsf{groupInfoTBS}$
  req $\mathsf{Sig.vrf}(\gamma.\tau.\mathsf{leaves}[\mathsf{senderId}].\mathsf{spk}, \mathsf{sig}, \mathsf{groupInfoTBS})$
  try $\gamma \leftarrow$ *vrf-tree-state($\gamma$)
  $v \leftarrow \gamma.\tau.\mathsf{leaves}[\gamma.\mathsf{leafId}()]$
  try $\gamma.\mathsf{ssk} \leftarrow$ *fetch-ssk-if-nec($\gamma, v.\mathsf{spk}$)
  $\mathsf{kbs} \leftarrow$ query get-sks to $\mathcal{F}_{\mathrm{KS}}$
  $\mathsf{joinerSec}, \mathsf{pathSec} \leftarrow \bot$
  for $e \in \mathsf{encGroupSecs}$ do
      parse $(\mathsf{hash}, \mathsf{cipher}) \leftarrow e$
      for $(\mathsf{kp}, \mathsf{sk}) \in \mathsf{kbs}$ do
          if $\mathsf{hash} = \mathsf{Hash}(\mathsf{kp})$ then
              $v.\mathsf{sk} \leftarrow \mathsf{sk}$
              req $v.\mathsf{kp}() = \mathsf{kp}$
              parse $(\mathsf{joinerSec}, \mathsf{pathSec}) \leftarrow \mathsf{PKE.dec}(\mathsf{sk}, \mathsf{cipher})$
  req $\mathsf{joinerSec} \neq \bot$
  if $\mathsf{pathSec} \neq \epsilon$ then
      $v \leftarrow \gamma.\tau.\mathsf{lca}(\gamma.\mathsf{leafId}(), \mathsf{senderId})$
      while $v \neq \bot$ do
          $\mathsf{nodeSec} \leftarrow \mathsf{HKDF.Expand}(\mathsf{pathSec}, \text{`node'})$
          $(\mathsf{sk}, v.\mathsf{sk}) \leftarrow \mathsf{PKE.kg}(\mathsf{nodeSec})$
          req $v.\mathsf{sk} = \mathsf{sk}$
          $\mathsf{pathSec} \leftarrow \mathsf{HKDF.Expand}(\mathsf{pathSec}, \text{`path'})$
          $v \leftarrow v.\mathsf{parent}$
  $(\gamma, \mathsf{confKey}) \leftarrow$ *derive-epoch-keys($\gamma, \mathsf{joinerSec}$)
  req *vrf-conf-tag($\gamma, \mathsf{confKey}, \mathsf{confTag}$)
  return $(\gamma.\tau.\mathsf{roster}(), \gamma.\tau.\mathsf{leaves}[\mathsf{senderId}].\mathsf{id})$

Input Key
  req $\gamma \neq \bot$
  $(k, \gamma.\mathsf{appSecret}) \leftarrow (\gamma.\mathsf{appSecret}, \bot)$
  return $k$
```

Fig. 15: The (UC) protocol ITK as run by party id. The group creator's identity $\mathsf{id}_{\mathsf{creator}}$ is encoded a part of the instance's session identifier.

Protocol ITK : Commit. Process, and Join Helpers

```
helper *init-epoch($\gamma$)
  $\gamma' \gets \gamma$.clone()
  $\gamma'$.epoch $\gets \gamma'$.epoch $+ 1$
  $\gamma'$.pendUp[$*$], $\gamma'$.pendCom[$*$] $\gets \bot$
  return $\gamma'$

helper *rekey-path($\gamma'$, id, spk)
  directPath $\gets \gamma'.\tau$.directPath($\gamma'$.leafId())
  coPath $\gets \gamma'.\tau$.coPath($\gamma'$.leafId())
  updatePathNodes $\gets$ ()
  pathSecs[$*$] $\gets \bot$
  leafSec $\gets_\$ \{0,1\}^\kappa$
  leafNodeSec $\gets$ HKDF.Expand(leafSec, 'node')
  pathSec $\gets$ HKDF.Expand(leafSec, 'path')
  for $(v, c) \in$ zip(directPath, coPath) do
      pathSecs[$v$] $\gets$ pathSec
      nodeSec $\gets$ HKDF.Expand(pathSec, 'node')
      ($v$.pk, $v$.sk) $\gets$ PKE.kg(nodeSec)
      encPathSecs $\gets$ ()
      for $t \gets c$.resolution() do
          encPathSecs $+\!\!\gets$ PKE.enc($t$.pk, pathSec)
      updatePathNodes $+\!\!\gets$ ($v$.pk, encPathSecs)
      pathSec $\gets$ HKDF.Expand(pathSec, 'path')
  commitSec $\gets$ pathSec
  $\gamma'.\tau$.mergeLeaves($\gamma'$.leafId())
  $\gamma' \gets$ *set-parent-hash($\gamma'$, $\gamma'$.leafId())
  try ssk $\gets$ *fetch-ssk-if-nec($\gamma'$, spk)
  $v \gets \gamma'.\tau$.leaves[$\gamma'$.leafId()]
  $r \gets$ leafNodeSec
  (kp, sk) $\gets$ gen-kp(id, spk, ssk, $v$.parentHash; $r$)
  req $\gamma'.\tau$.leaves[Hash(kp)] $= \bot$
  $v$.leafId $\gets$ Hash(kp)
  $v$.assignKp(kp)
  $v$.sk $\gets$ sk
  $\gamma' \gets$ *set-tree-hash($\gamma'$)
  updatePath $\gets$ (kp, updatePathNodes)
  return ($\gamma'$, commitSec, updatePath, pathSecs)

helper *apply-rekey($\gamma'$, senderId, updatePath)
  parse (kp, updatePathNodes) $\gets$ updatePath
  directPath $\gets \gamma'.\tau$.directPath(senderId)
  coPath $\gets \gamma'.\tau$.coPath(senderId)
  lca $\gets \gamma'.\tau$.lca($\gamma'$.leafId(), senderId)
  for ($v$, $c$, updatePathNode) $\in$ zip(directPath,
                                   coPath, updatePathNodes) do
      parse ($v$.pk, encPathSecs) $\gets$ updatePathNode
      if $v =$ lca then
          $r \gets c$.resolvent($\gamma'.\tau$.leaves[$\gamma'$.leafId()])
          $i \gets c$.resolution().indexof$r$
          pathSec $\gets$ PKE.dec($r$.sk, encPathSecs[$i$])
      if pathSec $\neq \bot$ then
          nodeSec $\gets$ HKDF.Expand(pathSec, 'node')
          (pk, $v$.sk) $\gets$ PKE.kg(nodeSec)
          req $v$.pk $=$ pk
          pathSec $\gets$ HKDF.Expand(pathSec, )
  commitSec $\gets$ pathSec
  $\gamma'.\tau$.mergeLeaves(senderId)
  $\gamma' \gets$ *set-parent-hash($\gamma'$, senderId)
  $v \gets \gamma'.\tau$.leaves[senderId]
  try $\gamma' \gets$ *validate-kp($\gamma'$, kp, $v$.id, $v$.parentHash)
  req $\gamma'.\tau$.leaves[Hash(kp)] $= \bot$
  $v$.leafId $\gets$ Hash(kp)
  $v$.assignKp(kp)
  $\gamma' \gets$ *set-tree-hash($\gamma'$)
  return ($\gamma'$, commitSec)

helper *truncate-tree($\gamma'$)
  $v \gets \gamma'.\tau$.leaves[$|\gamma'.\tau$.leaves$| - 1$]
  if $\neg v$.inuse($\wedge$($\neg v$.parent.inuse() $\vee$ $v$.parent $= \gamma'.\tau$.root) then
      $\gamma'.\tau \gets \tau$.pruneRightmost()
      $\gamma' \gets$ *truncate-tree($\gamma'$)
  return $\gamma'$

helper *apply-props($\gamma, \gamma', \vec{p}$)
  $upd, rem, add \gets$ ()
  for $p \in \vec{p}$ do
      try (senderId, $P$) $\gets$ *unframeProp($\gamma, p$)
      id$_s \gets \gamma.\tau$.leaves[senderId].id
      parse (type, val) $\gets P$
      if type $=$ 'upd' then
          req (id$_s$, $*$) $\notin upd \wedge rem =$ () $\wedge$ $add =$ ()
          try $\gamma' \gets$ *validate-kp($\gamma'$, val, id$_s$, $\epsilon$)
          $\gamma'.\tau$.leaves[senderId].assignKp(val)
          $\gamma'.\tau$.blankPath(senderId)
          if senderId $= \gamma$.leafId() then
              parse (ssk, sk) $\gets \gamma$.pendUp[$p$]
              $\gamma'.\tau$.leaves[senderId].sk $\gets$ sk
              $\gamma'$.ssk $\gets$ ssk
          spk $\gets \gamma'.\tau$.leaves[senderId].spk
          req $\gamma'.\tau$.leaves[Hash(kp)] $= \bot$
          $\gamma'.\tau$.leaves[senderId].leafId $\gets$ Hash(kp)
          $upd$ $+\!\!\gets$ (id$_s$, 'upd'-spk)
      else if type $=$ 'rem' then
          id$_t \gets \gamma.\tau$.leaves[val].id
          req val $\neq$ senderId $\wedge$ $\gamma'.\tau$.leaves[val] $\neq \bot$
          req $\gamma'.\tau$.leaves[val].inuse() $\wedge$ (id$_t$, $*$) $\notin upd \wedge add =$ ()
          $\gamma'.\tau$.leaves[val].blank()
          $\gamma'.\tau$.leaves[val].blankPath(val)
          $\gamma' \gets$ *truncate-tree($\gamma'$)
          $rem$ $+\!\!\gets$ (id$_s$, 'rem'-id$_t$)
      else if type $=$ 'add' then
          parse (id$_t$, $*$, spk, $*$, $*$) $\gets$ val
          req id$_t \notin \gamma'.\tau$.roster()
          try $\gamma' \gets$ *validate-kp($\gamma'$, val, id$_t$, $\epsilon$)
          newId $\gets$ Hash(val)
          try $\gamma'.\tau$.allotLeaf(newId)
          $\gamma'.\tau$.leaves[newId].assignKp(val)
          $\gamma'.\tau$.unmergeLeaf(newId)
          $add$ $+\!\!\gets$ (id$_s$, 'add'-id$_t$-spk)
      else return $\bot$
  return ($\gamma'$, $upd, rem, add$)

helper *welcome-msg($\gamma, \gamma'$, $add$, joinerSec, pathSecs, confTag)
  groupInfoTBS $\gets$ ($\gamma'$.groupId, $\gamma'$.epoch, $\gamma'$.treeHash,
        $\gamma'$.confTransHash, $\gamma'$.interimTransHash,
        $\gamma'.\tau$.public(), confTag, $\gamma'$.leafId())
  sig $\gets$ Sig.sign($\gamma'$.ssk, groupInfoTBS)
  groupInfo $\gets$ (groupInfoTBS, sig)
  encGroupSecs $\gets$ ()
  for ($*$, 'add'-id$_t$-spk$_t$) $\in add$ do
      leafId$_t \gets \gamma'.\tau$.leafof(id$_t$)
      $v_t \gets \gamma'.\tau$.leaves[leafId$_t$]
      lca $\gets \gamma'.\tau$.lca($\gamma'$.leafId(), leafId$_t$)
      encGroupSec $\gets$ PKE.enc($v_t$.pk, (joinerSec, pathSecs[lca]))
      encGroupSecs $+\!\!\gets$ (Hash($v_t$.kp), encGroupSec)
  $w \gets$ (encGroupSecs, groupInfo)
  return ($\gamma'$, $w$)

helper *vrf-tree-state($\gamma'$)
  req $\gamma'$.treeHash $=$ *tree-hash($\gamma'.\tau$.root)
  for $v \in \gamma'.\tau$.nodes : $v$.inuse() $\wedge$ $\neg v$.isleaf do
      lchild $\gets v$.lchild
      rchild $\gets$ *origRChild($v$.rchild)
      phr $\gets$ *parent-hash-cochild($v$, $v$.rchild)
      phl $\gets$ *parent-hash-cochild($v$, $v$.lchild)
      req (lchild.inuse $\wedge$ lchild.parentHash $=$ phr)
            $\vee$ (rchild.inuse $\wedge$ rchild.parentHash $=$ phl)
  $mem \gets \varnothing$
  for $v \in \gamma'.\tau$.nodes : $v$.inuse() $\wedge$ $v$.isleaf do
      req $v$.id $\notin mem$
      $mem$ $+\!\!\gets$ $v$.id
      try $\gamma' \gets$ *validate-kp($\gamma'$, $v$.kp(), $v$.id, $v$.parentHash)
  return $\gamma'$

helper *origRChild($v$)
  if $v$.inuse $\vee$ $v$.isleaf then return $v$
  else return *origRChild($v$.lchild)
```

Fig. 16: The helper methods related to creating and processing the commit and welcome messages.

Protocol ITK : Confirmation-Tag

```
helper *conf-tag($\gamma'$, confKey)
  return MAC.tag(confKey, $\gamma'$.confTransHash)

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

Protocol ITK : Tree-Hash

```
helper *set-parent-hash($\gamma'$, leafId)
  path $\leftarrow$ $\gamma'.\tau$.directPath(leafId)
  path $\leftarrow$ path.reverse()
  path +$\leftarrow$ $\gamma'.\tau$.leaves[leafId]
  for $v \in$ path do
    if $v$.isroot then
      $v$.parentHash $\leftarrow \epsilon$
    else
      $v$.parentHash $\leftarrow$ *parent-hash-cochild($v$.parent, $v$.sibling)
  return $\gamma'$

helper *parent-hash-cochild($v, u$)
  origChildResolution $\leftarrow$ $u$.resolution() $\setminus$ $u$.parent.unmergedLvs
  return Hash($v$.pk, $v$.parentHash, origChildResolution)

helper *set-tree-hash($\gamma'$)
  $\gamma'$.treeHash $\leftarrow$ *tree-hash($\gamma'.\tau$.root)
  return $\gamma'$

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

Protocol ITK : Transcript-Hash

```
helper *set-conf-trans-hash($\gamma, \gamma'$, senderId, $C$, sig)
  commitContent $\leftarrow$ ($\gamma$.groupId, $\gamma$.epoch, senderId, ‘commit’, $C$, sig)
  $\gamma'$.confTransHash $\leftarrow$ Hash($\gamma$.interimTransHash, commitContent)
  return $\gamma'$

helper *set-interim-trans-hash($\gamma'$, confTag)
  $\gamma'$.interimTransHash $\leftarrow$ Hash($\gamma'$.confTransHash, confTag)
  return $\gamma'$
```

Protocol ITK : Key-Schedule

```
helper *derive-keys($\gamma, \gamma'$, commitSec)
  $s \leftarrow$ HKDF.Extract($\gamma$.initSecret, commitSec)
  joinerSec $\leftarrow$ HKDF.Expand($s, \gamma'$.groupCtxt())
  ($\gamma'$, confKey) $\leftarrow$ *derive-epoch-keys($\gamma'$, joinerSec)
  return ($\gamma'$, confKey, joinerSec)

helper *derive-epoch-keys($\gamma'$, joinerSec)
  $s \leftarrow$ HKDF.Expand($\gamma$.joinerSec, ‘member’)
  memberSec $\leftarrow$ HKDF.Extract($s$, 0)
  $e \leftarrow$ HKDF.Expand(memberSec, ‘epoch’)
  epSec $\leftarrow$ HKDF.Extract($e, \gamma'$.groupCtxt())
  confKey $\leftarrow$ HKDF.Expand(epSec, ‘confirm’)
  $\gamma'$.appSecret $\leftarrow$ HKDF.Expand(epSec, ‘app’)
  $\gamma'$.membKey $\leftarrow$ HKDF.Expand(epSec, ‘membership’)
  $\gamma'$.initSecret $\leftarrow$ HKDF.Expand(epSec, ‘init’)
  return ($\gamma'$, confKey)
```

Protocol ITK : Setup Interaction

```
helper *fetch-ssk-if-nec($\gamma$, spk)
  if $\gamma.\tau$.leaves[$\gamma$.leafId()].spk $\neq$ spk then
    ssk $\leftarrow$ query (get-ssk, spk) to $\mathcal{F}_{\text{AS}}$
  else
    ssk $\leftarrow$ $\gamma$.ssk
  return ssk

helper *validate-kp($\gamma$, kp, id, parentHash)
  parse (id', pk, spk, parentHash', sig) $\leftarrow$ kp
  req id = id' $\wedge$ parentHash = parentHash'
  if spk $\notin$ $\gamma$.certSpks[id] then
    $succ \leftarrow$ query (verify-cert, id', spk) to $\mathcal{F}_{\text{AS}}$
    req $succ$
    $\gamma$.certSpks[id] +$\leftarrow$ spk
  req Sig.vrf(spk, sig, (id, pk, spk, parentHash))
  return $\gamma$
```

Fig. 17: Various helper methods for the protocol ITK.

Protocol ITK : Message-Framing

```
helper *signComit$(\gamma, C)$
  tbs $\leftarrow$ $(\gamma$.groupCtxt(), $\gamma$.groupId, $\gamma$.epoch, $\gamma$.leafId(), 'commit', $C)$
  sig $\leftarrow$ Sig.sign($\gamma$.ssk, tbs)
  return sig

helper *frameCommit$(\gamma, C$, confTag, sig, membTag)
  return ($\gamma$.groupId, $\gamma$.epoch, $\gamma$.leafId(), 'commit', $C$, confTag, sig,
                                                              membTag)

helper *unframeCommit$(\gamma, c)$
  parse (groupId, epoch, senderId, contentType, $C$, confTag, sig,
                                                      membTag) $\leftarrow c$
  req contentType = 'commit' $\wedge$ groupId = $\gamma$.groupId
  reqepoch = $\gamma$.epoch
  tbs $\leftarrow$ ($\gamma$.groupCtxt(), groupId, epoch, senderId, 'commit', $C$)
  req $\gamma.\tau$.leaves[senderId] $\neq \perp$
        $\wedge$ $\gamma.\tau$.leaves[senderId].inuse()
        $\wedge$ Sig.vrf($\gamma.\tau$.leaves[senderId].spk, sig, tbs)
  return (senderId, $C$, confTag, sig, membTag)

helper *frameProp$(\gamma, P)$
  tbs $\leftarrow$ ($\gamma$.groupCtxt(), $\gamma$.groupId, $\gamma$.epoch, $\gamma$.leafId(), 'proposal', $P$)
  sig $\leftarrow$ Sig.sign($\gamma$.ssk, tbs)
  tbm $\leftarrow$ (tbs, sig)
  membTag $\leftarrow$ MAC.tag($\gamma$.membKey, tbm)
  return ($\gamma$.groupId, , $\gamma$.epoch, $\gamma$.leafId(), 'proposal', $P$, sig, membTag)

helper *unframeProp$(\gamma, p)$
  parse (groupId, epoch, senderId, contentType, $P$, sig, membTag) $\leftarrow p$
  req contentType = 'proposal' $\wedge$ groupId = $\gamma$.groupId
  reqepoch = $\gamma$.epoch
  tbs $\leftarrow$ ($\gamma$.groupCtxt(), groupId, epoch, senderId, 'proposal', $P$)
  tbm $\leftarrow$ (tbs, sig)
  req $\gamma.\tau$.leaves[senderId] $\neq \perp$
        $\wedge$ $\gamma.\tau$.leaves[senderId].inuse()
        $\wedge$ Sig.vrf($\gamma.\tau$.leaves[senderId].spk, sig, tbs)
        $\wedge$ MAC.vrf($\gamma$.membKey, membTag, tbm)
  return (senderId, $P$)
```

Fig. 18: The helper methods related to message framing.

for each party an encryption of the joiner secret (to derive the epoch secrets) and seed of the least common ancestor of the party and the committer.

Finally, ITK computes the next epoch's interim transcript hash by hashing the confirmed transcript hash and the confirmation tag. Moreover, the next epoch's state is stored in γ.pendCom.

Process. Consider an input (`Process`, $c, \vec{p}$). If the party created this commit message c (and the proposals match), then the protocol can simply retrieve the new epoch's state from γ.pendCom. Otherwise, it proceeds as follows.

First, the protocol "unframes" the message, i.e., it verifies the signature and checks that it belongs to the correct group and epoch. Next, it verifies that $\vec{p}$ match the proposals mentioned in c and, if so, applies them using `*apply-props`.

Afterwards, ITK applies the re-key using `*apply-rekey` (see Fig. 16). That is, it updates all the public keys and decrpyts the least common ancestor's seed to derive the secret keys shared between the direct paths of the committer and the processing party. Moreover, this updates the parent hash on the re-keyed path and in particular verifies that the one signed as part as the new leaf's key package matches.

The protocol then derives the new epoch's key schedule by first computing the confirmed transcript hash and then deriving the keys. Based on the new schedule, the confirmation tag is then verified. Finally, it completes the new epoch's state with the interim transcript hash.

Join. Upon input (`Join`, w), ITK sets up a new state and copies the public group information from the welcome message. This state is then verified by first verifying the sender's signature on the group information as well as verifying the public part of the ratchet tree.

Crucially, this entails verifying the tree-signing mechanism. Intuitively, we would like to maintain the following invariant:

> **Invariant:** For each non-blank internal node v, either of its children is non-blank and has a parent hash (with co-path) stored that matches the `*parent-hash-cochild`(v, u), where u denotes the other child.

See Fig. 19 for an explanation of the tree-signing mechanism and the invariant in particular.

Unfortunately, the above invariant is too idealistic for the following reason. If adding a member results in adding a leaf to v's subtree, v's right child u may be replaced by a fresh blank node u' — in which case u, storing v's parent hash, becomes the left child of u' — without re-keying v. (Recall the definition of left-balanced binary trees.) To account for this possibility, `*vrf-tree-state` first (see Fig. 16) first searches for the "real" right child of v using the helper `*origRChild`.

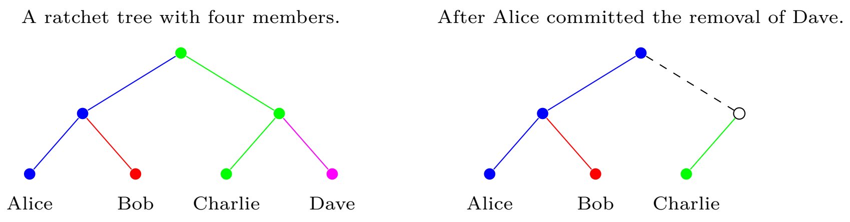

A ratchet tree with four members.
Alice
Bob
Charlie
Dave
After Alice committed the removal of Dave.
Alice
Bob
Charlie

Fig. 19: An example of the tree-signing mechanism. In the left configuration, the keys in blue, red, green, and purple nodes have been sampled and certified by Alice, Bob, Charlie, and Dave, respectively. When Alice now commits the removal of Dave, first Dave's direct path gets blanked (including the root). Then Alice re-keys her direct path. Her leaf now contains a signature of the hash of the following things about the internal node shared with Bob: (1) the internal node's public-key, the list of public-keys to whom the corresponding secret key has been encrypted (in this case Bob's public key), and (3) the internal node's respective parent hash (of the root) binding the leaf to the entire direct path.
At this point, the parent hash of Charlie's leaf no longer matches the now blanked parent node, for which now neither child node has a valid parent hash. Note however that ITK always blanks a party's entire direct path. Hence, for each non-blank node, the binding child node is still in the tree. Thus, each non-blank internal node and the set of public key knowing its corresponding secret key is always bound by its child sampled by the same party, ultimately certified by the corresponding leaf node.

For each leaf, `*vrf-tree-state` furthermore verifies the signature on the key package, which includes the parent hash. Overall, this mechanism ensures that each internal node has been sampled by one of the parties in the respective sub-tree (or the party's signing key has been compromised).

ITK then proceeds by decrypting the private information — the joiner secret and the seed of the least common ancestor — from the welcome message. To this end, it fetches all its key-package/secret-key pairs $(\mathsf{kp}, \mathsf{sk})$ from $\mathcal{F}_{\text{KS}}$ and determines the one that has been used for the welcome message based on the hash of kp.

Analogous to `Process` it then derives the secret keys on the common path segment to the root and finally the next epoch's key schedule. It moreover also verifies the confirmation tag.

Key. The input `Key` outputs the current epoch's application secret and then deletes it from the local state.

C.4 Simplifications and Deviations

While ITK closely follows the IETF MLS protocol draft, there are a number of small deviations and omissions.

Omitted modes and optional features. The ITK protocol omits the following modes and optional features of the MLS protocol draft.

Protocol versions and ciphersuites. In the MLS draft, each group has a protocol version and a ciphersuite associated. Our analysis, on the other hand, simply assumes a single protocol version with a fixed set of underlying primitives. As they are specified upon group initialization by the group creator (rather than negotiated) and remain unchanged over the group's lifetime, we do not, a priori, see any major potential for downgrade and other attacks. Additionally, those parameters are incorporated into the key schedule, ensuring agreement. However, we leave a more complete analysis for future work.

Meta-data protection. The MLS draft supports two message framing formats: encrypted MLSCiphertexts and unencrypted MLSPlaintexts. While using the former is mandatory for protecting application messages' confidentiality, it is only recommended for handshake message to thwart basic meta-data analysis. Since we only consider handshake messages (not application messages) and do not take meta-data protection into account, we fix all frameing to be MLSPlaintexts. In particular, it is immediate that additionally encrypting the packets (as done by MLSCiphertext framing) does not undermine any of the security properties analyzed for MLSPlaintexts in this work.

External proposals. The MLS protocol draft allows for non-members to propose adding themselves to an existing group, and also allows for pre-provisioned parties to send (arbitrary) proposals, e.g., for a server to remove stale members. In both cases, it is up to the group policy to decide on the validity of such external proposals. We did not take either mechanism into consideration.

Extensions. Similar to the TLS protocol, the MLS protocol draft is extensible in a number of places. We did not analyze any extensions.

Preshared keys. Groups which have an out-of-band mechanism to agree upon pre-shared keys can incorporate these into the MLS key schedule for additional security. We did not analyze this mechanism.

Exporters. The MLS key schedule provides a mechanism to export additional secrets to higher-level applications. As they are derived from the key schedule similarly to the application secret (and are otherwise unused by the protocol), their security should follow analogously.

Minor simplifications. Furthermore, our model of the protocol deviates from the draft in a number of minor aspects.

No membership MAC on commits that do not include removes. ITK uses an explicit MAC to ensure the authenticity of proposals. The MLS protocol also includes the MAC for commits (when using MLSPlaintext) [11]. For the security properties considered in this work this inclusion is redundant in case the commit does not include remove proposals, as the confTag already provides the same (and more) authenticity guarantees. But in practical terms, the MAC can provide somewhat better denial-of-service mitigation than relying only on confTag. In particular, verifying the MAC may allow quickly rejecting malformed commit packets without needing to first derive the next epoch's key schedule (a comparatively costly computation) needed to verify the confTag.

Simplified primitives. While the MLS protocol draft imposes a particular use of HKDF for key derivation (ExpandWithLabel), our model simply uses HKDF directly, not mixing in the same amount of context as in the spec. We note that this modification can only weaken security. So our results carry over to the more inclusive version in the RFC. For our analysis we treated HKDF's expand and extract functions as a random oracle. Moreover, in lieu of explicitly imposing the KEM-DEM paradigm (with the HPKECiphertext structure in commit messages) we simply model this as public-key encryption. Thus, formally speaking, it remains to show that HPKE, in the mode used in by MLS, implements a PKE scheme as modeled in our work (from reasonable assumptions). (Given the simplicity of that mode of HPKE we believe this to be quite straight forward.)

Expiration of key packages and certificates. Key packages are mandated to have explicit lifetimes, which we do not account for. Neither does our model of the Authentication Service account for the expiration of certificates.

Simplified welcome message format. The protocol in the MLS draft always encrypts the (public) group context in welcome messages, analogously to MLSCiphertexts and not offering a mode analogous to MLSPlaintexts. As we do not take meta-data protection into account, our model forgoes this additional complexity. In particular, all of our results carry trivially cary over to the MLS variant that performs the extra encryption. Additionally, we always put the public part of the ratchet tree as part of the welcome message, not taking into account alternative means of delivery (e.g. via the DS). But here too, our model implies security for such delivery methods. Indeed, we use an insecure network (modeled by the environment) which means our model provides no guarantees on what is ultimately delivered to new joining members. Instead, it is up to the protocol to extract an guarantees from whatever packet is delivered.

More restrictive proposal lists. Our analysis assumes that the proposal vectors inside a commit message follow a strict ordering of first update proposals, then remove proposals, and finally add proposals. The current MLS draft (no longer) imposes such a restriction on the vector, but requires them to be applied respecting this order, i.e., not necessarily in the order specified. (We believe our techniques carry over essentially unchanged to this more permissive version of MLS.)

D Proof of Theorem 1: Security of ITK

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

The proof is structured as follows. First, in Apps. D.2 to D.4 we prove that ITK* realizes $\mathcal{F}_{\text{CGKA}}$ with a worse safety predicate **safe***. Then, in App. D.5 we extend the proof to show full security of ITK. Formally, **safe*** differs from the original predicate in that the condition b) of ***state-directly-leaks** (cf. Fig. 7) is replaced by

b) // c is in a detached tree and id's spk appears in some exposed node:
$(\exists c_a : {*}\mathsf{ancestor}(c_a, c) \wedge \mathsf{Node}[c_a].\mathsf{par} = \perp \wedge (\mathsf{id}, \mathsf{spk}) \in \mathsf{Node}[c].\mathsf{mem}$
$\wedge\ (\mathsf{spk} \in \mathsf{Exposed} \vee \exists c_e : (*, \mathsf{spk}_e) \in \mathsf{Node}[c_e].\mathsf{mem} \wedge \mathsf{spk}_e \in \mathsf{Exposed})$

The proof that ITK* realizes the weakened $\mathcal{F}_{\text{CGKA}}$ proceeds in a sequence of hybrids, transitioning from the real world to the ideal world. The simulator is introduced gradually together with the hybrids. Each hybrid introduces a different security property provided by ITK*: the first hybrid is the real world, the second hybrid introduces consistency, the third – confidentiality and the fourth – authenticity. The fourth hybrid is the ideal world. Intuitively, if two consecutive hybrids are indistinguishable then ITK* provides the introduced security property. Formally, we have

Hybrid 1. This is the real world. We make a syntactic change: the simulator $\mathcal{S}_1$ interacts with a dummy functionality $\mathcal{F}_{\text{DUMMY}}$, which routs all inputs and outputs through $\mathcal{S}_1$, who executes ITK*.

Hybrid 2. This hybrid introduces consistency. $\mathcal{F}_{\text{DUMMY}}$ is replaced by $\mathcal{F}_{\text{AS}}^{\text{IW}}$, $\mathcal{F}_{\text{KS}}^{\text{IW}}$ and $\mathcal{F}_{\text{CGKA}}$, except **safe***$(\cdot) = \mathtt{false}$ and **inj-allowed**$(\cdot,\cdot) = \mathtt{true}$. That is, all application secrets are set by the simulator and injections are always allowed. The simulator $\mathcal{S}_2$ still sets all messages and keys according to the protocol.

Hybrid 3. This hybrid introduces confidentiality. $\mathcal{F}_{\text{CGKA}}$ uses the original **safe*** predicate. The simulator $\mathcal{S}_3$ sets only those application secrets for which **safe*** is false.

Hybrid 4. This hybrid introduces authenticity. $\mathcal{F}_{\text{CGKA}}$ uses the original **inj-allowed**. The simulator remains the same. This is the ideal world.

In Apps. D.2 to D.4 we prove that each pair of consecutive hybrids is indistinguishable.

D.1 ITK* Guarantees Consistency

Claim. Hybrids 1 and 2 are indistinguishable, that is ITK* guarantees consistency.

To prove the claim, we describe in detail the simulator $\mathcal{S}_2$, and argue that it does not violate any statements executed by $\mathcal{F}_{\text{CGKA}}$ within **assert**, and that the outputs of ITK and $\mathcal{F}_{\text{CGKA}}$ are the same. Observe that $\mathcal{S}_2$ knows the whole history graph, including the application secrets (since **safe*** is false in Hybrid 2). Moreover, each history graph node has a unique confTransHash, because the transcript hash includes all messages c leading to it, i.e., all parents (except the last confTag, but this is uniquely determined by the last c).

PROPOSALS. When $\mathcal{F}_{\text{CGKA}}$ sends $(\mathtt{Propose}, \mathsf{id}, \mathsf{add})$ to $\mathcal{S}_2$, the simulator executes the ITK protocol to obtain the packet p. Recall that for proposals adding id_t, ITK fetches the key package kp_t for id_t from $\mathcal{F}_{\text{KS}}$, and that $\mathcal{F}_{\text{KS}}$ asks $\mathcal{Z}$ to provide kp_t. $\mathcal{S}_2$ executes the code of both ITK and $\mathcal{F}_{\text{KS}}$, which means it uses kp_t provided by $\mathcal{Z}$.
If $p = \perp$, $\mathcal{S}_2$ sends to $\mathcal{F}_{\text{CGKA}}$ *ack* $= \mathtt{false}$. Else, it sends $(p, \mathsf{spk}_t, \mathtt{true})$, where spk_t is taken from kp_t (by inspection, the protocol guarantees that kp_t is well formed and contains spk_t).
Assert statements: The only **assert** statement executed on proposals is a part of ***consistent-prop**, which enforces that proposals computed by id in node c are different than those computed in node c' (even if id can never get to these nodes). This is guaranteed by including in

proposals $\mathsf{membTag}$ — a MAC, modeled as a random oracle,[26] over $\mathsf{groupCtxt}$, which includes $\mathsf{confTransHash}$ (c.f. framing in Fig. 18).[27]

<u>Commits.</u> $\mathcal{S}_2$ computes the packets c and w according to ITK and sets $ack = \mathtt{false}$ if $c = \bot$. If $ack = \mathtt{true}$, it first checks if c corresponds to a detached root — if $\mathsf{Node}[c] = \bot$ and there exists a w such that $\mathsf{Wel}[w] = \mathsf{root}_{rt}$ and $\mathsf{confTransHash}$ in w (included as a part of $\mathsf{groupInfo}$) matches that in c (the latter can be computed), sends rt to $\mathcal{F}_{\text{CGKA}}$ (alongside c and w).
Then, $\mathcal{F}_{\text{CGKA}}$ runs ***fill-props**. For each proposal p without a node, $\mathcal{S}_2$ sets orig and act according p (the basic checks executed by ITK guarantee that p is well formed).
Assert statements: ***consistent-comm** succeeds for the same reason as ***consistent-prop**. All other asserted statements trivially hold by inspection and the fact that all messages include a MAC over the transcript hash (note that in the invariant, **inj-allowed** is false in these hybrids).

<u>Process.</u> $\mathcal{S}_2$ executes the protocol to check if the receiver would accept the inputs and sends $ack = \mathtt{false}$ if this is not the case. Else, it checks if c corresponds to a detached root exactly as in Commits above. If c creates a new node (i.e., there was no detached root and $\mathsf{Node}[c] = \bot$, $\mathcal{S}_2$ retrieves orig' and spk' from c (the latter can be found in the committer's key package in the $\mathsf{updatePath}$.
The fact that all statements in **assert** are true follows easily by inspection. To see why the outputs of ITK and $\mathcal{F}_{\text{CGKA}}$ are the same, observe first that since a commit contains hashes of all proposals, with overwhelming probability, for each c there is only one $\vec{p}$ such that $(\mathtt{Process}, c, \vec{p})$. Second, the output of process is determined by $\vec{p}$ and the member set in c's parent (moreover, this output is computed the same way by ITK and ***output-proc** in $\mathcal{F}_{\text{CGKA}}$). By the standard hybrid argument, this implies that outputs are the same.

<u>Join.</u> When a party id joins using a welcome message w, $\mathcal{S}_2$ first executes id's protocol to determine if joining succeeds and sets the ack flag accordingly. If $ack = \mathtt{false}$, or if $\mathsf{Wel}[w] \neq \bot$, then $\mathcal{S}_2$ simply sends $(ack, \bot, \bot, \bot)$ to $\mathcal{F}_{\text{CGKA}}$. Note that in the latter case, $\mathcal{F}_{\text{CGKA}}$ already knows the semantics of w and ignores any values received from the simulator other than ack. Else, $\mathcal{S}_2$ sends to $\mathcal{F}_{\text{CGKA}}$ values c', orig' and mem' that interpret the (injected) w. It computes them as follows.
First, $\mathcal{S}_2$ parses $w = (\mathsf{encGroupSecs}, \mathsf{groupInfo})$ (if w was not of this format, joining would have failed) and determines orig' and mem' based on $\mathsf{groupInfo}$. Then, $\mathcal{S}_2$ chooses c' as the history graph node for which $\mathsf{confTransHash}$ matches that in $\mathsf{groupInfo}$, or sets $c' = \bot$ if no such node exists (if $c' = \bot$, $\mathcal{F}_{\text{CGKA}}$ creates a new detached root).
It is left to argue that the joiner id ends up in a state that is consistent with the state of any other party id' transitioning into c' via process or join. For this, 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 collision resistance) 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's 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'. □

D.2 A New Security Notion for PKE

Our proofs that ITK^* provides confidentiality and authenticity rely on security of the PKE scheme (for authenticity, PKE protects the MAC keys). In the setting with adaptive corruptions, reducing ITK^* security to the standard IND-CCA security is difficult for the following reason. Recall that ITK^* generates key pairs $(\mathsf{pk}_i, \mathsf{sk}_i)$ and encrypts sk_i under different pk_j (without creating cycles). Say a secret message m is encrypted under pk_1. To argue that IND-CCA security implies confidentiality

[26] The claim is not implied by any standard security notion for MAC's. What we would need is that even given the secret key, it is hard to find two messages with the same tags. While possible to formalize, for simplicity we instead model the MAC as the RO (this is anyway necessary for the MAC used to compute the confirmation tag).

[27] Note that the epoch counter is not unique — it is in fact the same for all commit nodes with the same depth.

of m, we would have to first introduce a sequence of hybrids that replace encryptions of secret keys that allow to compute sk_i via a chain of decryptions by encryptions of some unrelated messages. It is important to replace only those, since replacing others allows to distinguish the experiments by corrupting secret keys. However, with adaptive adversaries we do not know if an encryption should be replaced at the moment it is created. Guessing this for each encryption incurs exponential loss.

To deal with this, we follow the strategy of [5] and define a stronger security notion for PKE, called *(Modified) Generalized Selective Decryption (GSD)*. In contrast to [5], our version considers active attackers and takes into account the full key schedule. We then prove that IND-CCA security implies GSD security in the (non-programmable but observable) ROM. The proof is an adaptation of the proof in [5] showing that IND-CPA implies their weaker version of GSD.

The modified GSD game. GSD security is formalized by the game defined in Fig. 20. It is parameterized by a hash function Hash and a number N. In essence, the game maintains a (hyper)graph with N vertices, where each vertex u stores a seed s_u (initially $\perp$), from which a key pair can be derived by running key generation with randomness set to the hash of s_u. Edges correspond to dependencies between seeds: one seed being a hash of another or being encrypted under a key derived from another. In general, if a vertex is a source of an edge, then the public key is known to the adversary (note that an outputted ciphertext may already reveal it). Otherwise, the public key is secret and the seed should be indistinguishable from random. (Note that a secure seed can be used as a symmetric key.) To put the definition in the context of a ITK execution, the GSD hypergraph created by a ITK commit is given in Fig. 21.

We now describe the GSD oracles in more detail.

- Enc(u, v) creates an edge from u to v with label e and outputs an encryption of the seed s_v under the public key derived from s_u. This query also, if necessary, initializes s_u and s_v to random values. (ITK context: encrypting path secrets during rekey.)
- Hash(u, v, lbl) creates an edge from u to v with label lbl and computes v as $\mathsf{Hash}(s_u, \mathsf{lbl})$. Since the hash is deterministic, we require that s_v is not initialized yet and no other seed has been computed from s_u using lbl. (ITK context: hash chain of path secrets.)
- Join-Hash(u, u', v, lbl) is similar to Hash, but instead of s_u, it uses the pair $(s_u, s_{u'})$. (ITK context: joiner secret is the hash of init and commit secrets.)
- Dec and Chal oracles are analogous to those in the IND-CCA game, except the restrictions which nodes can be queried. The Corr oracle outputs the seed and records it in the Corr set.

The crucial aspect of the game is the **gsd-exp**(u) function, which determines if the seed in a vertex u is exposed due to corruptions, or its secrecy is guaranteed. That is, **gsd-exp** for vertices is analogous to $\neg$**safe*** for application secrets. Specifically, u is exposed if it is corrupted, or if there is an edge to u that can be traversed. The latter is true iff all sources of the edge are exposed. (Notice the similarity to how our **safe*** is defined.)

Definition 4. *Let* $\mathrm{Adv}^{\mathrm{GSD}}_{\mathsf{PKE},\mathcal{A}} := 2\Pr[\mathrm{GSD}_{\mathsf{PKE},\mathcal{A}} = \mathtt{true}] - 1$ *denote the advantage of $\mathcal{A}$ against the game defined in Fig. 20. A scheme* PKE *is* GSD *secure, if for all PPT adversaries $\mathcal{A}$,* $\mathrm{Adv}^{\mathrm{GSD}}_{\mathsf{PKE},\mathcal{A}}$ *is negligible in κ.*

IND-CCA security implies GSD security. We next show the following theorem.

Theorem 2 (adapted from [5]). *If* PKE *is* IND-CCA *secure and* Hash *is modeled as a (observable, non-programmable) random oracle, then* PKE *is* GSD *secure.*

Proof. The proof is adapted from [5]. There, the authors first show that IND-CPA implies in the ROM the standard GSD security, GSD$^-$, i.e., the notion formalized by the game from Fig. 20 without the Hash, Join-Hash and Dec oracles. This proof solves the main technical challenges, and we refer the reader to [5] for the details. Then, [5] includes a proof sketch showing that the reduction for GSD$^-$ can be easily modified to account for certain additional hash queries, namely, the ones that in our game correspond to Hash queries with a fixed label $\mathsf{lbl} = 1$. (While the proof sketch of [5] involves programming of the RO, we believe this is not necessary.) We show that additional Hash, Join-Hash and Dec queries do not affect (the modification of) the reduction.

Game $\mathrm{GSD}_{\mathcal{A}}$

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
$b \leftarrow\$ \{0,1\}$
$s' \leftarrow\$ \{0,1\}^\kappa$
$b' \leftarrow \mathcal{A}_{\mathsf{PKE}}^{\mathbf{Enc},\mathbf{Dec},\mathbf{Corr},\mathbf{Chal},\mathbf{Hash},\mathbf{Join\text{-}Hash}}$
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 $s_v = \bot \wedge (u, *, \mathtt{h}\text{-}\mathsf{lbl}) \notin E$ // hash is deterministic
gen-key-if-nec$(u)$
$s_v \leftarrow \mathsf{Hash}(s_u, \mathsf{lbl})$
gen-key-if-nec$(v)$
$E \mathrel{+}\!\!\leftarrow (u, v, \mathtt{h}\text{-}\mathsf{lbl})$
return $\mathsf{pk}_u$
```

Oracle Corr(u)

```
req $s_u \neq \bot$
$\mathsf{Corr} \mathrel{+}\!\!\leftarrow u$
return $s_u$
```

Oracle Join-Hash(u, u', v, lbl)

```
req $s_v = \bot \wedge ((u, u', \mathsf{lbl}), *, \mathtt{h}\text{-}\mathsf{lbl}) \notin E$
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$(v)$
$E \mathrel{+}\!\!\leftarrow ((u, u'), v, \mathtt{h}\text{-}\mathsf{lbl})$
return $(\mathsf{pk}_u, \mathsf{pk}_{u'})$
```

Oracle Enc(u, v)

```
gen-key-if-nec$(u)$; gen-key-if-nec$(v)$
$E \mathrel{+}\!\!\leftarrow (u, v, \mathsf{e})$
$c \leftarrow \mathsf{PKE.enc}(\mathsf{pk}_u, s_v)$
$\mathsf{Ctxt} \mathrel{+}\!\!\leftarrow (u, c)$
return $(\mathsf{pk}_u, c)$
```

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 ITK, 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. 20: The GSD game, modified to explain ITK executions.

Decryption. While [5] considers IND-CPA security, we note that their reduction generates the seeds in all GSD vertices except one "challenge" vertex itself. Hence, answers to decrypt queries for non-challenge vertices can simply be computed, and for the challenge vertex — sent to the IND-CCA oracle (requiring $(u, c) \notin \mathsf{Ctxt}$ makes sure that the IND-CCA challenge is valid).

Hash and Join-Hash. Here we need a bit more details of the reduction from [5]. Assume $\mathcal{A}_{\mathsf{gsd}}$ is a GSD adversary. The authors define an event E on the GSD execution with $\mathcal{A}_{\mathsf{gsd}}$ as follows (here adapted to our setting)

> Event E. At some point, $\mathcal{A}_{\mathsf{gsd}}$ queries RO on a value that contains a seed s_u for a non-challenge vertex u for which **gsd-exp** is false (at the time of the RO query).

Then, [5] presents two reductions: the reduction (1) constructs an IND-CCA adversary $\mathcal{A}_{\mathsf{cca}}$, given a GSD adversary $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ that triggers E with small probability, and the reduction (2) constructs a GSD adversary $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ that triggers E with small probability, given a GSD adversary $\mathcal{A}_{\mathsf{gsd}}^{E}$ which triggers E with large probability.

Reduction (1). We first argue that in the reduction (1), $\mathcal{A}_{\mathsf{cca}}$ can easily deal with the additional hash edges in the GSD experiment it simulates for $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$. In essence, $\mathcal{A}_{\mathsf{cca}}$ defined in [5] guesses an edge (v, u, e), where u is the GSD challenge issued by $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ (for now, just assume the edge is given; see [5] for details). Then, $\mathcal{A}_{\mathsf{cca}}$ samples seeds for all vertices except u itself, replaces the public key in v by its challenge key pk (unrelated to v's seed), and embeds the IND-CCA challenge in the encryption query creating the (v, u, e) edge. (The IND-CCA challenge is queried on two random seeds, and $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$'s challenge is answered with the first one.)

Clearly, any hash query that does not involve u or v can be simulated by evaluating the RO. Any query involving v is simulated by evaluating the RO on v's seed. Since u is a challenge and there is a v-u edge, **gsd-exp** is false for v. Hence, the fact that $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ does not trigger E implies that it does not query RO on v's seed and hence cannot verify that it is inconsistent with the public key. Finally, if u is created via a hash query (note that as a challenge, u is a sink), $\mathcal{A}_{\mathsf{cca}}$ can simply ignore this edge (i.e., choose u at random instead). Again, the inconsistency of this edge cannot be verified without triggering E. (Note that if u is created via join-hash of u, u', then for **gsd-exp** to be false in u, it must be false for at least one of u, u'. Since verifying the hash requires querying the RO on both u and u', it triggers E.)

Reduction(2). Second, consider the reduction (2). Given a GSD adversary $\mathcal{A}^{E}_{\mathsf{gsd}}$ that triggers E, [5] defines $\mathcal{A}^{\neg E}_{\mathsf{gsd}}$ that does not trigger E roughly as follows. $\mathcal{A}^{\neg E}_{\mathsf{gsd}}$ simulates the experiment for $\mathcal{A}^{E}_{\mathsf{gsd}}$ using its oracles, and halts as soon as E turns true (it can realize that E is true by checking each RO query of $\mathcal{A}^{E}_{\mathsf{gsd}}$ against all public keys). Moreover, it guesses the vertex v corresponding to the RO query that makes E true. The idea is to challenge v as soon as its seed is defined (since now v must be a sink in $\mathcal{A}^{\neg E}_{\mathsf{gsd}}$'s game, outgoing edges from v and its public key are simulated using a special vertex $N + 1$), obtain a seed s and search for s in $\mathcal{A}^{E}_{\mathsf{gsd}}$'s RO queries. If s is the real seed (and the guess for v is correct), then it is queried to the RO and $\mathcal{A}^{\neg E}_{\mathsf{gsd}}$ outputs 0. Else, if s is random, then it is independent of $\mathcal{A}^{E}_{\mathsf{gsd}}$'s view, so with high probability it is not queried and, accordingly, $\mathcal{A}^{\neg E}_{\mathsf{gsd}}$ outputs 1 (when E is triggered for a different node, or $\mathcal{A}^{E}_{\mathsf{gsd}}$ halts).

We only need to argue that the additional (join-)hash queries do not affect the simulation before E is triggered, as afterwards the reduction halts. The reason this holds is analogous to the reasoning for the reduction (1) — the only inconsistency is in the vertex v, where edges $(u, v, *)$ are generated using v's actual seed, and edges $(v, u, *)$ are generated using the special vertex $N + 1$. However, this inconsistency cannot be verified without querying the RO on the seed from v or $N + 1$. As for both of these vertices **gsd-exp** is false (for v by assumption that the guess was correct, and for $N + 1$ since it is a source and cannot be corrupted, as it does not appear in $\mathcal{A}^{E}_{\mathsf{gsd}}$'s game), such query would trigger E. $\square$

D.3 ITK* Guarantees Confidentiality

Claim. If PKE is GSD secure, then Hybrids 2 and 3 are indistinguishable, that is, ITK* guarantees confidentiality.

We define two sequences of hybrids: $\mathcal{H}^{\mathsf{appSecret}}_{i}$ and $\mathcal{H}^{\mathsf{membKey}}_{i}$ for $i \in [N]$. $\mathcal{H}^{\mathsf{appSecret}}_{i}$ is the same as Hybrid 2, except the first i application secrets chosen by $\mathcal{F}_{\text{CGKA}}$ are sampled as in Hybrid 3, i.e. they are random if **safe*** is true. $\mathcal{H}^{\mathsf{membKey}}_{i}$ is the same as $\mathcal{H}^{\mathsf{appSecret}}_{N}$, except the first i membership keys used by the simulator are as in Hybrid 3, i.e. they are random if **safe*** is true. In the following, we show that $\mathcal{H}^{\mathsf{appSecret}}_{i-1}$ and $\mathcal{H}^{\mathsf{appSecret}}_{i}$ are indistinguishable. The proof for $\mathcal{H}^{\mathsf{membKey}}_{i}$ is analogous.

Assume that an environment $\mathcal{Z}$ has a non-negligible advantage in distinguishing between hybrids $\mathcal{H}^{\mathsf{appSecret}}_{i-1}$ and $\mathcal{H}^{\mathsf{appSecret}}_{i}$, and let M be an upper bound on the number of secret keys (including PKE secret keys and symmetric keys) created in an execution with $\mathcal{Z}$. We construct an adversary $\mathcal{A}$ against the GSD game with M nodes as follows. On a high level, $\mathcal{A}$ emulates for $\mathcal{Z}$ the interaction with $\mathcal{F}_{\text{CGKA}}$, $\mathcal{F}^{\text{IW}}_{\text{KS}}$, $\mathcal{F}^{\text{IW}}_{\text{AS}}$ and the simulator. In particular, $\mathcal{A}$ executes the code of all these functionalities and the simulator as defined in $\mathcal{H}^{\mathsf{appSecret}}_{i-1}$, except secure PKE key pairs are generated with the help of GSD oracles and the i-th group key embeds the GSD challenge. Note that if for the i-th application secret **safe*** is false, then the challenge is not embedded, but in this case the hybrids proceed exactly the same.

We now explain in detail how $\mathcal{A}$ modifies the code of the functionalities and the simulator. First, instead of storing a separate state for each party (as the simulator executing the protocol would), $\mathcal{A}$ keeps a single group state for each commit node and a separate state for each proposal. Relevant to the reduction, the group state contains a ratchet tree τ with a key pair in each node and a number of symmetric keys, such as memberSec and appSecret. A proposal node's state contains a key package. In general, a secret key (symmetric or asymmetric) can have one of three values: (1) if it is unknown to $\mathcal{Z}$, then it is equal to (gsd, u), where $u \in \mathbb{N}$ is a GSD vertex, (2) if it is known to both $\mathcal{Z}$ and $\mathcal{A}$, then it is set to the actual value, and (3) if it known to $\mathcal{Z}$ but unknown to $\mathcal{A}$ (e.g. an injected public key), then it is set to $\perp$.

For bookkeeping, $\mathcal{A}$ keeps a counter u_{ctr} (initially 1), denoting the largest vertex in the GSD game used so far. We write $\mathsf{pk} \leftarrow *\mathsf{get\text{-}pk}(u)$ to denote that $\mathcal{A}$ obtains the public key pk for a vertex u by calling the oracle $\mathrm{Enc}(u, 0)$ (the special vertex 0 is only used here).

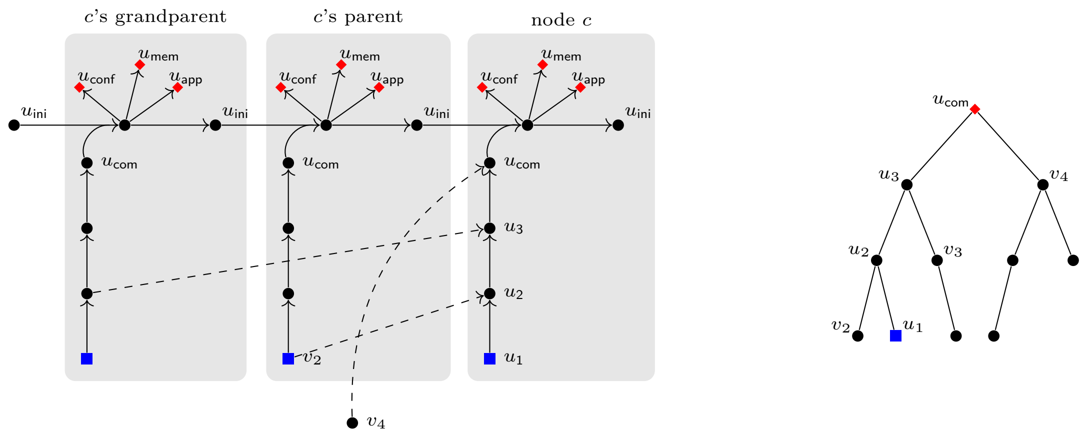


Fig. 21: An example commit creating a node c: (right) the ratchet tree in c and its parent, (left) the corresponding GSD graph created by the reduction $\mathcal{A}$. The sinks and sources are marked by ◆ and ■, respectively. The continuous and dashed edges denote hash and encryption edges, respectively. The rightmost gray area is created upon the commit. The committer id first generates a sequence of path secrets, while $\mathcal{A}$ creates vertices $u_1, ..., u_{\mathsf{com}}$ connected by hash edges. Then id derives next-epoch secret from $\mathsf{joinerSec}$ obtained by combining $\mathsf{commitSec}$ and previous $\mathsf{initSecret}$. Accordingly, $\mathcal{A}$ creates u_{joi} as the destination of a join-hash edge from u_{com} and u_{ini}. Finally, id encrypts u_i under v_i, while $\mathcal{A}$ obtains ciphertexts from encryption edges (v_4 was created in a previous commit).

The remainder of the proof consists of three steps. First, we consider the simplified setting, where $\mathcal{Z}$ never injects messages *or key packages*, and never corrupts randomness. In the next two steps, we remove the former and the latter assumption, respectively.

Step 1: No Injections, No Bad Randomness We describe how $\mathcal{A}$ modifies the code of the functionalities and the simulator. First, unlike $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$ and $\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}$, it does not delete secret keys (but records the deletion event). Then, it processes different inputs as follows.

Key-package registration. When $\mathcal{Z}$ instructs a party id to register a key package in the emulated $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$, $\mathcal{A}$ creates a new GSD vertex by executing $\mathsf{pk} \leftarrow *\mathsf{get\text{-}pk}(u_{\mathsf{ctr}})$. It uses pk to generate the public part of the key package kp, sets the secret key $\mathsf{SK}[\mathsf{id}, \mathsf{kp}]$ to $(\mathsf{gsd}, u_{\mathsf{ctr}})$, and sets u_{ctr}++.

Proposal add-id_t from id. Recall that whenever the protocol requests a key package for id_t from $\mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}$, $\mathcal{Z}$ gets to choose it. Accordingly, $\mathcal{A}$ stores in the new proposal node the pk taken from the key package chosen by $\mathcal{Z}$.

Proposal up from id. Analogous to registering key packages, $\mathcal{A}$ creates the new key pair as $*\mathsf{get\text{-}pk}(u_{\mathsf{ctr}})$ and $(\mathsf{gsd}, u_{\mathsf{ctr}})$, stores it in the new proposal node and uses it to compute the message. It sets u_{ctr}++.

Applying proposals. The only difference from `*apply-props` (executed by the simulator as part of the protocol) is that for each update proposal, the leaf's secret key is set to the value stored in the proposal node, and for each add proposal, its set to the value in the SK array.

Commit from id. After applying the proposals, $\mathcal{A}$ emulates `*rekey-path` as follows (see Fig. 21 for an example). First, consider the case where for all public keys used by id in `*rekey-path`, the secret keys stored in the ratchet tree are of the form $(\mathsf{gsd}, *)$.

1. *Add vertices to the GSD graph:* $\mathcal{A}$ adds the following vertices: $u_1, \ldots, u_n$ (path secrets), u_{joi} (joiner secret), u_{app}, u_{mem}, u_{conf} (their values are set to u_{ctr}, u_{ctr}+1, ... and u_{ctr} is incremented). The vertices are created as follows: for each $i \in [n-1]$, query $\mathrm{Hash}(u_i, u_{i+1}, \mathsf{path})$. Then, query $\mathrm{Join\text{-}Hash}(u_{\mathsf{par\text{-}ini}}, u_{\mathsf{n}}, .)$, where $(\mathsf{gsd}, u_{\mathsf{par\text{-}ini}})$ is stored in the $\mathsf{initSecret}$ of $\mathsf{Ptr}[\mathsf{id}]$. For $\mathsf{lbl} \in \mathsf{app}, \mathsf{mem}, \mathsf{ini}, \mathsf{conf}$, query $\mathrm{Hash}(u_{\mathsf{joi}}, u_{\mathsf{lbl}}, \mathsf{lbl})$.
2. *Create the packet:* $\mathcal{A}$ creates encryptions of path secrets by creating corresponding encryption edges. Then, it corrupts u_{mem} and stores the result as the $\mathsf{memberSec}$ of the new node.

Finally, it corrupts u_{conf}, which completes the set of values needed to compute the commit packet.

3. *Create the welcome message:* The welcome message contains, for each new member id_t, the encryptions of joinerSec and id_t's pathSec under the key in id_t's leaf in the new epoch's ratchet tree (obtained from KS by the party adding id_t).[28] Let u_i be the GSD vertex corresponding to the pathSec sent to id_t. If id_t's leaf key is of the form (gsd, u), $\mathcal{A}$ obtains the encryptions by creating encryption edges from u to u_{joi} and from u to u_i. Else, it corrupts u_i and u_{joi} and encrypts the values itself.

Now assume that for some key used in `*rekey-path`, the secret key is not $(\mathsf{gsd}, *)$. Let u_i be the smallest that should be encrypted under such key. After adding the vertices, $\mathcal{A}$ corrupts u_i and uses it to compute $u_{i+1}, ..., u_n$ and encrypts $u_i, ..., u_n$ itself. It creates the packet as before.

<u>KEY IN NODE c.</u> $\mathcal{A}$ modifies the ***set-key** as follows. Assume this is the j-th call to ***set-key**. If **safe**$^*(c)$ is true and $j < i$, output a random value. Else, if **safe**$^*(c)$ is true and $j = i$, let $(\mathsf{gsd}, u_{\mathsf{app}})$ be the value stored in appSecret of c. Query challenge on u_{app} and output the result. Else, output the real group key, corrupting the GSD node if necessary.

<u>EXPOSE id.</u> The state of id's contains the following secrets: 1) the secret key for each ratchet tree node on id's direct path such that id is not in unmerged leaves of this node, 2) epoch secrets, and 3) the key packages secret keys generated by id. For each of the above secrets, if the secret key that is equal to (gsd, u), $\mathcal{A}$ corrupts u. Then, for each vertex v s.t. **gsd-exp**(v) becomes true, $\mathcal{A}$ replaces all occurrences of (gsd, v) by the seed s_v, computed using previously obtained ciphertexts and corrupted seeds.

If a history graph node c stores a symmetric key $(\mathsf{gsd}, u_{\mathsf{lbl}})$, we refer to the GSD vertex as (c, u_{lbl}). Assume $\mathcal{A}$ queries challenge on a vertex (c, u_{app}) (if $\mathcal{A}$ does not query a challenge, i.e. **safe**$^*(c)$ is flase, then the hybrids are exactly the same, so $\mathcal{Z}$'s advantage is 0). We now show that in the GSD execution with $\mathcal{A}$, **safe**$^*(c)$ implies that **gsd-exp**$((c, u_{\mathsf{app}}))$ is false, and hence $\mathcal{A}$ can win the game by outputting whatever $\mathcal{Z}$ outputs.

We first observe that for any commit c' with corresponding welcome message w', each party who could join using w' is equivalent to a current group member who already processed c' (ignoring encryptions of joinerSec in w', which we will consider separately). This is for the following reason: recall that for any joining id_t, w' contains the encryption of the pathSec that id_t would receive as part of c', was he a member in the parent epoch of c' with the ratchet tree leaf as in c'. Moreover, when id_t is corrupted, he is immediately added to the corrupt set of any node where he could join,[29] including c' (see the "for each" loop in the Expose input). This is the same as if id_t was a current group member, transitioned to c' and was corrupted. Therefore, we now only consider current group members.

Recalling the definition of **gsd-exp** (Fig. 20) and the GSD graph created by $\mathcal{A}$ (Fig. 21), **gsd-exp**$((c, u_{\mathsf{app}}))$ can only be true in one of the three cases:

(a) $\mathcal{A}$ corrupts the vertex (c, u_{app}). This happens iff $\mathcal{A}$ computes the state of a party exposed in c, which immediately implies $\neg$**safe**$^*(c)$.
(b) $\mathcal{A}$ corrupts the vertex (c, u_{joi}). This happens iff $\mathcal{A}$ computes a welcome message for a party id_t added with an exposed key bundle. Recall that if id_t is exposed, $\mathcal{F}_{\text{CGKA}}$ adds it to the exposed set exp of each node where it can join using a currently held key package (the "for each" loop of input expose). Hence, id_t must be in the exp set of c and **safe*** is false.
(c) Both **gsd-exp**$((c, u_{\mathsf{com}}))$ and **gsd-exp**$((\mathsf{Node}[c].\mathsf{par}, u_{\mathsf{ini}}))$ are true. For this case, we show below that **gsd-exp**$((c, u_{\mathsf{com}}))$ implies **know**$(c, *)$. Then, the claim follows by condition d) of ***can-traverse**.
   – It is easy to see (c.f. Fig. 21) that **gsd-exp**$((c, u_{\mathsf{com}}))$ is true if and only if **gsd-exp**$((c, u_i))$ is true for some path secret u_i created by $\mathcal{A}$ when generating c. This, in turn, is true iff either
   (1) during the commit, $\mathcal{A}$ corrupts u_i, or
   (2) during the commit, $\mathcal{A}$ calls the Enc oracle to encrypt u_i under a key in (c, u) and **gsd-exp**$((c, u))$,

[28] ITK actually encrypts both secrets together in one ciphertext, but without loss of generality here we treat them as two separate ciphertexts.
[29] Note that he cannot join to epochs where he was added using an old, deleted spk_t.

(3) during an exposure of an id storing u_i's secret (after processing the commit).

- For Case (3), notice that any action of id removes u_i's secret from its state (it is blanked for id's proposals and rekeyed for its commits). Hence, if id is corrupted in c's descendant c' while still storing u_i's secret, we clearly have **know**(c', id) and $\neg$***secrets-replaced**(c'', id) for each c'' on the c-c' path.
- Next, we consider Cases (1) and (2). Observe that (1) happens only if for some key used in `*rekey-path` to encrypt u_i, the secret key stores a seed. This means that this key was created as a GSD node (c, u) and then set during exposure, because **gsd-exp**$((c, u))$ became true (c.f. $\mathcal{A}$'s behavior on expose). Hence, we only have to show that **gsd-exp**$((c, u))$ for some (c, u) stored in a vertex used in `*rekey-path` implies **know**$(c, *)$.
- Let $\tau.v$ be the ratchet tree node that stores the exposed vertex (c, u) and let $\mathsf{id}_1, \ldots, \mathsf{id}_n$ be the parties with leaves in $\tau.v$'s subtree. Consider the subgraph G of the history graph containing all commit nodes (with incoming edges) where (c, u) is stored in $\tau.v$. Since there are no injections and no bad randomness, G is a tree (c.f. the example in Fig. 22).
- First, consider the case where $\tau.v$ is not a leaf. Then, the root of G is the commit that inserts (c, u) into the ratchet tree, i.e., the first ancestor of c where an id_i is the committer. The leaves of G are commits that remove (c, u), i.e., any commits sent by an id_i or commits that remove an id_i.
  There are two possible reasons for which (c, u) is exposed. First, this happens if some id_i is exposed in a node c_e in G and $\mathcal{A}$ has to compute its secret state. In this case, observe that by the definition of ***secrets-replaced**, for *each* id_i, every non-leaf node in G is reachable from c via recursive evaluation of **know**(c, id). Moreover, we clearly have ***state-directly-leaks**(c_e, id_i).
  Second, this can happen if **gsd-exp**$((c', u'))$ is true for some u' used to encrypt the seed in (c, u). If $\tau.v'$ is not a leaf, we repeat the above reasoning for (c', u') and the ratchet tree node $\tau.v'$ storing (c', u') (the procedure terminates, since the protocol guarantees that $\tau.v'$ is in $\tau.v$'s subtree of $\tau.v$, so the subtree of $\tau.v'$ is smaller).
- Now consider the case where $\tau.v$ is a leaf and let id be its owner. Only id's actions affect $\tau.v$. In particular, the root of G is a commit by id or one that includes a proposal updating or adding it. Similarly, leaves of G are commits by id, or ones that include proposals updating or removing it. In other words, these are exactly commits c' for which ***secrets-replaced**(c', id) is true.[30] This means that G is exactly those nodes that are reachable from c via the recursive condition of **know**$(\cdot, \mathsf{id})$.
  Now a leaf secret is always a source in the GSD graph (which $\mathcal{A}$ generates when id updates, commits, or registers a key package), and $\mathcal{A}$ only corrupts id's leaf when id holds the secret key (note that this secret is not encrypted during a commit), i.e. when id is in a node of G. Hence, there is a node c' in G where id is exposed, making ***state-directly-leaks**(c', id) true.

Step 2: Allowing Injections We extend $\mathcal{A}$ to deal with different types of injected messages as follows.

<u>Injected proposals.</u> $\mathcal{A}$ creates the new node using the public keys from the message and the secret key set to $\perp$ (even if the public key is already stored somewhere else). Note that for add proposals, the secret key stored in the SK array may be $\perp$.

<u>Applying proposals.</u> This works exactly the same, i.e. a secret key equal to $\perp$ is copied to the ratchet tree leaf.
Note that a party id never enters a node where its leaf's key is injected (i.e., the secret is $\perp$), as ITK trivially detects this situation. Hence, storing $\perp$ has no effect, for leaf keys only being used by the party itself.

<u>Commit from id.</u> $\mathcal{A}$ proceeds the same as before. Secret keys equal to $\perp$ are treated the same as those with known seeds, i.e, $\mathcal{A}$ corrupts the smallest u_i that is encrypted under a public key, where the secret key is a seed or $\perp$.

[30] Note that a leaf of G cannot add id, since its already in the group, and similarly the root cannot remove id.

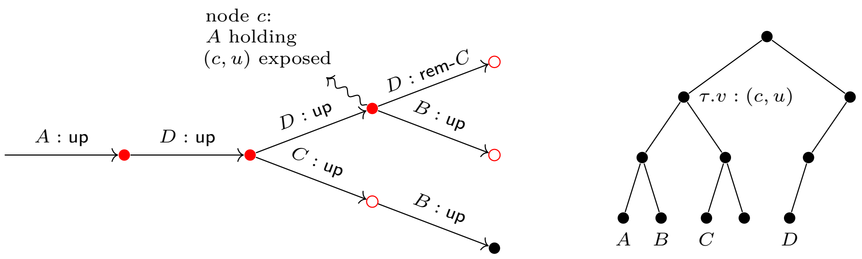


Fig. 22: An illustration for the proof that **gsd-exp**$((c,u))$ implies **know**$(c,*)$: the history graph (left) and the ratchet tree in the exposed node c (right). The history graph subtree G is marked by ● and the leafs are marked by ○. In the root of G, the committer A inserts (c,u) as one of the path secrets created during the commit. The leaves of G remove (c,u) by either B replacing it during the commit, or D blanking it to remove C.

<u>Commits injected to process.</u> Assume $\mathcal{Z}$ makes id process an injected commit c from $\mathsf{id}_c \neq \mathsf{id}$, and that id accepts it. $\mathcal{A}$ attempts to build the new commit node's state as follows. First, it applies proposals (copying the $\perp$ keys if necessary) to the ratchet tree in id' node. Then, it normally applies the rekey operation, except for each ciphertext *ctxt* that id would decrypt with keys in a ratchet-tree node v. To apply the rekey for those ciphertexts *ctxt*, $\mathcal{A}$ then does as follows.

- If the secret in v is not a GSD node, $\mathcal{A}$ simply decrypts *ctxt*. (Observe that the secret seed in v is not $\perp$, as id's (real-world) protocol would reject c in that case.)
- If the secret is (gsd, u) and *ctxt* can be queried to the decrypt oracle, $\mathcal{A}$ decrypts it this way.
- The only reason why decrypting would not be possible, is that *ctxt* must had been copied from an "honest" commit c', generated earlier by $\mathcal{A}$, for which the GSD node u associated to appSecret is still a valid challenge. (Recall that upon exposure, $\mathcal{A}$ immediately computes all secrets it can given the new information.) We now argue that this situation cannot occur due the the (valid) confirmation tag included in c and, in fact, show that id accepting such a c would allow $\mathcal{A}$ to win the GSD game. To this end, let $\mathcal{A}$ challenge the GSD node u; and extract the correct seed from $\mathcal{Z}$'s random oracle calls as follows.
  1. Observe that appSecret can be derived from joinerSec, which in turn is computed as $\mathsf{joinerSec} = \mathsf{Hash}(\mathsf{initSecret}, \mathsf{commitSec}, .)$, modeling HKDF as a RO. Moreover, observe that commitSec must be the same in c and c', due to the shared (honestly generated) *ctxt* accepted in both states.
  2. We now proceed towards extracting joinerSec of c'. Recall to this end that the tag is a MAC, modeled as RO, of confKey and confTransHash, and that confTransHash includes the whole message c except the confirmation tag itself and the membership tag. Since the latter two are unique given the rest of c, confTransHash is unique for c as well. Hence, the only way for $\mathcal{Z}$ to compute a valid confirmation tag is to query RO on $(\mathsf{confKey}, \mathsf{confTransHash})$, and $\mathcal{A}$ can extract confKey. Analogously, as confKey is derived by hashing joinerSec with an appropriate label, it can extract joinerSec (of c) from the queries as well.
  3. Now consider two cases. First, if initSecret is the same in c and c', then $\mathsf{joinerSec} = \mathsf{Hash}(\mathsf{initSecret}, \mathsf{commitSec}, .)$ is the same in c and c' (by commitSec being the same). Second, if initSecret is different, then joinerSec in c is the hash of an honestly generated commitSec with a different initSecret. Since the protocol only uses commitSec once with the correct initSecret, the only way for $\mathcal{Z}$ to compute the joiner is to query the RO, and commitSec can be extracted from the queries. Now $\mathcal{A}$ corrupts initSecret in c' and combines it with commitSec to compute joinerSec. Note that this does not affect u being a valid challenge, since the node corresponding to commitSec (of c') is not exposed.

     In either case, $\mathcal{A}$ can now compute appSecret and compare it to the result from the GSD challenge to determine the bit b.

<u>Injected welcome messages.</u> In case $\mathcal{Z}$ makes id process an injected welcome message $w =$ (encGroupSecs, groupInfo), $\mathcal{A}$ does as follows.

1. *Join to an existing node.* If there exists a node c with confTransHash matching that in groupInfo, $\mathcal{A}$ searches for a key package kp such that $\mathsf{SK}[\mathsf{id}, \mathsf{kp}] \neq \perp$ and Hash(kp) matches an entry $e \in$ encGroupSecs.
   If e is copied from a welcome message generated by $\mathcal{A}$ while creating a commit node c and SK[id, kp] is a GSD node, $\mathcal{A}$ moves id to c. Else, it uses either the secret in SK[id, kp] or the GSD decrypt oracle to check if id would process the message and moves id if this is the case.
2. *Join a new node: create the public part.* If no c with matching transcript hash is found, and id accepts the message, $\mathcal{A}$ 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 SK), it copies the secret into τ (other secrets remain $\perp$).
3. *Join a new node: decrypt the secrets.* $\mathcal{A}$ searches for a key package kp such that $\mathsf{SK}[\mathsf{id}, \mathsf{kp}] \neq \perp$ and Hash(kp) matches an entry $e \in$ encGroupSecs and aborts if no such kp exists.
   Similarly to injected commit messages, id will not accept e if it is copied from a welcome message generated by $\mathcal{A}$ while creating a commit node c and SK[id, kp] is a GSD node. To this end, observe that $\mathcal{A}$ could then use confTag from w to compute appSecret in c and win the GSD game as follows. Recall that confTag = Hash(confKey, confTransHash), where confKey is derived from joinerSec id decrypts and confTransHash is taken from w. Since e is copied, joinerSec used (implicitly) for the tag is the same as in c. On the other hand, confTransHash in w and c differ (else, id would have joined to c). Hence, joinerSec inn c can be extracted from $\mathcal{Z}$'s RO queries and used to compute appSecret in c.
   Otherwise, $\mathcal{A}$ can obtains the encrypted joinerSec and pathSec using the stored secret or the Dec oracle. It updates ratchet tree secrets to those derived from pathSec (if any secret key was set to a GSD node, $\mathcal{A}$ uses pathSec to win the game), and computes the epoch secrets from joinerSec.

We argue that with the above changes, $\mathcal{A}$'s GSD challenge (c, u_{app}) is still valid, as long as $\mathbf{safe}^*(c)$ is true. Assume towards a contradiction that $\mathbf{gsd\text{-}exp}((c, u_{\mathsf{app}}))$ is true.

The main tree. The proof is almost the same as in Step 1 (no injections). The only difference is in case (c), where we show that $\mathbf{gsd\text{-}exp}((c, u_{\mathsf{com}}))$ implies **know**$(c, *)$. We modify the proof of (c) as follows.

– $\mathbf{gsd\text{-}exp}((c, u_i))$ is true for some u_i in one of 3 cases:
  (a) During an exposure of an id who (supposedly) stores u_i's secret,
  (b) (As in Step 1) the secret key in some ratchet tree node $\tau.v$ used in `*rekey-path` stores a seed from a GSD vertex (c, u) with $\mathbf{gsd\text{-}exp}((c, u))$,
  (c) The secret in $\tau.v$ is $\perp$.
  We show that all cases imply **know**$(c, *)$.
– *Case (a).* Assume id is exposed in a commit node c' and a ratchet tree node $\tau'.v'$ on its direct path has u_i's public key pk_i. This can occur in 2 cases. First, if id processed c and has not performed any action — in this case, the reasoning is the same as in Step 1.
  Second, pk_i can be injected into $\tau'.v'$. We claim that this case cannot occur, since id will never process a commit that injects an honestly generated (as part of c) key into its direct path. Indeed, if $\tau'.v'$ is id's leaf, then the only way to inject pk_i is via update or commit sent by id, or by adding id. However, id's protocol does not accept proposals or commits from id that were not actually sent (the corresponding secrets are indexed by the whole messages), and id does not join a group with a key package it did not generate. If, on the other hand, $\tau'.v'$ is an internal node, then pk must be a part of an injected commit. If any party in the subtree of $\tau'.v'$ accepts such commit $\mathcal{A}$ can use the confirmation tag to win the GSD game.
– *Case (b).* Similar to Step 1, we consider the subgraph G of the history graph, containing all commit nodes where $\tau.v$ stores (c, u)'s public key pk.
  Using the exact same argument as in Case (a), we can argue that (c, u) is not exposed in any commit node outside of G. Hence, we use the same analysis as in Step 1.

– *Case (c).* The secret in $\tau.v$ is set by $\mathcal{A}$ to $\perp$ only when the public key pk in $\tau.v$ is injected during a commit c', i.e., if (a) $\mathcal{Z}$ injects c' on behalf of a party id in $\tau.v$'s subtree, or (b) $\tau.v$ is id's leaf and c' commits an update injected on behalf of id, (c) $\tau.v$ is id's leaf and c' commits an add proposal that uses an injected key package (either injected to KS, or directly to an injected commit).
  The first two cases exactly correspond to the respective cases a) and b) of ***secrets-injected**(c', id). In case (c), the add proposal must contain a key package for id with pk not generated by id. Since key packages are signed using id's spk (and the signature is always validated on process by `*validate-kp`), this means that spk is exposed (or $\mathcal{Z}$ can be used to break EUF-CMA), implying condition c) of ***secrets-injected**(c', id).
  Moreover, any commit c'' that heals id replaces all keys in its direct path, including $\tau.v$. Hence, c' is reachable from c via the recursive evaluation of **know**.

Orphan trees. Assume $\mathcal{A}$ challenges (c, u_{app}) for a node c in a detached tree rooted at root_{rt}. We show that if **safe**$^*(c)$ true, then the GSD challenge is valid.

First, observe that in a detached tree, **safe*** is true only if no spk of a group member in c is exposed. This is because **know**$(\mathsf{root}_{rt}, \text{'epoch'})$ is true (c.f. condition a) of ***can-traverse**) and **know**(c', id) is true for any c' in the detached tree as soon as id's spk is exposed (c.f. condition b) of ***state-directly-leaks**).

Second, we show that **safe**$^*(c)$, in particular clause b) of ***state-directly-leaks**, implies that each secret key in the ratchet tree τ of c stores a GSD vertex. With this, it is easy to see that **gsd-exp**$((c, u_{\mathsf{app}}))$ implies **know**$(c, *)$, where the argument is the same as for the main tree.

Take any node $\tau.v$ in c's ratchet tree. If $\tau.v$ is a leaf, then its public key is set to a value only if (1) its owner (with current spk) is corrupted or (2) a message (an update, a commit or a key package sent to KS) is injected on behalf of the owner. In case (1) the spk is explicitly marked as exposed. In case (2), it must have been marked as exposed, or $\mathcal{Z}$ can be used to break EUF-CMA.

If $\tau.v$ is an internal node, then its public key pk is included in the $\mathsf{parentHash}$ stored in the leaf of the party id_c in $\tau.v$'s subtree whose commit introduced pk. (There is such leaf, since the protocol rejects any welcome or commit that introduces a ratchet tree without it. Note that id_c is still in the group, since otherwise $\tau.v$ would have been blanked by the commit removing id_c.) Let spk_c denote id_c current signature key. This $\mathsf{parentHash}$ is signed by id_c and (assuming **safe***) spk_c is not exposed, pk was generated by $\mathcal{A}$ as a GSD vertex for id_c's (honest) commit c_c (or $\mathcal{Z}$ can be used to break EUF-CMA). Such pk is set to a value only in two situations:

1. A party id_e is corrupted in a descendant c_e of c_c before id_c performs any action (in which case its direct path, including pk, would be blanked). In this case, c_e's group contains id_c with (unchanged) spk_c and id_e with exposed spk_e, making ***state-directly-leaks** true in c.
2. A secret key for a ratchet tree node $\tau_c.v_c$ (in c_c) involved in `*rekey-path` executed while generating c_c is not a GSD node. In this case, $\tau_e.v_e$'s tree must have been injected by, or leaked upon corruption of a party id_e in $\tau_e.v_e$'s subtree. Moreover, id_e is still in the group and has not performed any action, else $\tau_e.v_e$ would have been blanked or replaced. This means that his key spk_e in c_c is exposed (it must have been exposed to enable the injection, or marked as exposed on corruption). Hence, c_c contains id_c with spk_c and id_e with exposed spk_e, making ***state-directly-leaks** true in c.

Step 3: Allowing Bad Randomness Finally, we modify $\mathcal{A}$ to deal with actions executed using bad randomness as follows.

PROPOSAL FROM id. $\mathcal{A}$ computes the proposal message p (and, in case of an update, the new key package $(\mathsf{kp}, \mathsf{sk})$) using the randomness provided by $\mathcal{Z}$, the current $\mathsf{membKey}$ and the id's spk (all of which are always known to $\mathcal{A}$). If p does not identify an existing node $\mathcal{A}$ creates it. In case of an update proposal, it sets the secret key in p's node to sk.

COMMIT FROM id. Given the randomness provided by $\mathcal{Z}$, $\mathcal{A}$ computes the commit and welcome messages, and the secrets in the new commit node, as follows.

1. $\mathcal{A}$ uses $\mathcal{Z}$'s randomness to execute `*rekey-path` and obtains: all path secrets, the $\mathsf{commitSec}$, and the intermediate commit packet C. Then, it signs C using id's spk (and, again, $\mathcal{Z}$'s randomness) and sets the confirmed transcript hash accordingly.

2. $\mathcal{A}$ computes the new $\mathsf{joinerSec}$, which is a hash of the current $\mathsf{initSecret}$ and the freshly computed $\mathsf{commitSec}$: If $\mathsf{initSecret}$ stores a GSD node u, $\mathcal{A}$ queries Hash with input $(u, u_{\mathsf{ctr}}, \mathsf{commitSec})$, corrupts u_{ctr}, sets $\mathsf{joinerSec}$ to the result and increments u_{ctr}. Else, if $\mathsf{initSecret}$ stores a value, $\mathcal{A}$ computes $\mathsf{joinerSec}$ itself.
3. Using $\mathsf{joinerSec}$ and the transcript hash from Step 1, $\mathcal{A}$ runs the key schedule, computes the confirmation tag, and finishes computing the commit message c and the welcome message w.

We claim that the above changes do not affect validity of $\mathcal{A}$'s challenge (c, u_{app}). First, observe that a corruption of the GSD node needed to compute $\mathsf{joinerSec}$ during a commit c' with bad randomness does not affect (c, u_{app}). This is because by condition a) of ***secrets-injected**, **safe*** is false in all descendants of c' until a commit is executed with good randomness. For this honest commit, $\mathsf{commitSec}$ corresponds to a GSD node with **gsd-exp** false, and hence **gsd-exp** is false for the $\mathsf{joinerSec}$ as well.

Second, we modify the proof that **gsd-exp**$((c, u_{\mathsf{com}}))$ implies **know**$(c, *)$. For this, observe that now **gsd-exp**$((c, u_i))$ can be true also if the secret in a ratchet tree node $\tau.v$ used in `*rekey-path` stores a seed s generated during an action executed with bad randomness. Consider the commit c' that inserts s into $\tau.v$.

– If c' is generated by id with bad randomness, then by condition a) of ***secrets-injected**, **know**(c', id) if true. Moreover, since any commit c'' with ***secrets-replaced**(c'', id) would replace the key in $\tau.v$, there is no such c'' on the c'-c path and **know**(c, id) is true.
– If $\tau.v$ is id's leaf and c' commits id's update executed with bad randomness, by condition b) of ***secrets-injected**, **know**(c', id) is true and, for the same reason as above, **know**(c, id) is true as well.
– Finally, assume $\tau.v$ is id's leaf and c' adds id using a key package kp generated with bad randomness. When kp was generated with bad randomness, $\mathcal{F}_{\text{KS}}^{\text{IW}}$ marked the used spk as exposed. Hence, c' must be adding id with an exposed spk, which, by condition c) of ***secrets-injected**, implies that **know**(c', id) is true. As before, this implies that **know**(c, id) is true as well.

Finally, we note that any action executed with bad randomness marks the used spk as exposed, and hence we no longer guarantee security in commit nodes in detached trees where the group contains a member with spk. Hence, the simulation becomes trivial in such nodes even if $\mathcal{Z}$ learns ssk and injects arbitrary messages.

D.4 ITK^* Guarantees Authenticity

Claim. If Sig and MAC are EUF-CMA secure and PKE is GSD secure, then Hybrids 3 and 4 are indistinguishable, that is ITK^* guarantees authenticity.

Observe that the hybrids are identical unless in there exists an injected history graph node in $\mathcal{F}_{\text{CGKA}}$ for which **inj-allowed** requires authenticity. (In Hybrid 3, the functionality ignores such nodes, while in Hybrid 4 it halts forever as soon as any such node appears.) More precisely, the hybrids are identical unless the following happens:

Event Bad. There exists a (commit or proposal) node with $\mathsf{stat} = \mathsf{adv}$ and **inj-allowed**$(c, \mathsf{id}) =$ `false` for its parent c and creator id.

Further, recall that (see Fig. 7) **inj-allowed**(c, id) is `false` if either id's signature public key spk in c is not exposed, or the adversary "does not know" the epoch key in c. Accordingly, we define two sub-events of Bad:

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

We next show that if Sig is secure, then the probability of $\mathsf{Bad}_{\mathsf{sig}}$ is negligible. Further, if PKE is secure, then the probability of $\mathsf{Bad}_{\mathsf{MAC}}$ is negligible (recall that the PKE is used to encrypt MAC keys). In both statements, MAC is modeled as an RO.

Lemma 3. *For any environment $\mathcal{Z}$, there exists a reduction $\mathcal{A}_{\mathsf{sig}}$ who wins the EUF-CMA game with probability only polynomially smaller than the probability that $\mathcal{Z}$ triggers* $\mathsf{Bad}_{\mathsf{sig}}$.

Lemma 4. *For any environment $\mathcal{Z}$, there exist reductions $\mathcal{A}_{\mathsf{MAC}}$ and $\mathcal{A}_{\mathsf{PKE}}$ such that the probability that $\mathcal{Z}$ triggers $\mathsf{Bad}_{\mathsf{MAC}}$ is upper bounded by $p \cdot (\epsilon_{\mathsf{MAC}} + \epsilon_{\mathsf{PKE}})$, where p is a polynomial, ϵ_{MAC} is the advantage of $\mathcal{A}_{\mathsf{MAC}}$ against the security of* MAC *and ϵ_{PKE} is the advantage of $\mathcal{A}_{\mathsf{PKE}}$ against the GSD security of* PKE.

Proof (of Lemma 3). Let $\mathcal{Z}$ be any environment. $\mathcal{A}_{\mathsf{sig}}$ emulates the functionalities and the simulator for $\mathcal{Z}$ as in their definitions, except it embeds its challenge spk as one of the public keys honestly created during the experiment. To emulate commits and proposals signed under the challenge ssk, $\mathcal{A}_{\mathsf{sig}}$ uses the sign oracle. When $\mathsf{Bad}_{\mathsf{sig}}$ occurs, $\mathcal{A}_{\mathsf{sig}}$ stops the experiment and sends to its challenger the forgery consisting of the signature sig' and the signed content tbs' from the injected node c'. We assume c' is a commit node; the proof for proposals is analogous. We claim that $\mathsf{Bad}_{\mathsf{sig}}$ occurs and $\mathsf{spk} = \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}]$, then $\mathcal{A}_{\mathsf{sig}}$ wins. Indeed, we have:

– sig' as a valid signature over tbs'. The reason is that the injected node was created when some party accepted c', which means that it verified sig' under $\mathsf{spk} = \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}]$.
– $\mathcal{A}_{\mathsf{sig}}$ simulates the experiment perfectly until $\mathsf{Bad}_{\mathsf{sig}}$ occurs. Indeed, the only situation in which the simulation differs from the experiment is when $\mathsf{spk} = \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}]$ is corrupted. This does not happen, since the event guarantees $\mathsf{spk} = \mathsf{Node}[c].\mathsf{mem}[\mathsf{id}] \notin \mathsf{Exposed}$.
– $\mathcal{A}_{\mathsf{sig}}$ did not query $(\mathsf{sig}', \mathsf{tbs}')$ to the sign oracle. Indeed, assume that $(\mathsf{sig}', \mathsf{tbs}')$ is the same as $(\mathsf{sig}^*, \mathsf{tbs}^*)$ queried by $\mathcal{A}_{\mathsf{sig}}$ to the sign oracle for c^*. We will show that this implies $c' = c^*$. Since c' is injected and c^* is honestly generated by $\mathcal{A}_{\mathsf{sig}}$, this is a contradiction.
Recall that c' contains $(\mathsf{groupId}', \mathsf{epoch}', \mathsf{senderId}', \text{‘}\mathsf{commit}', C', \mathsf{confTag}', \mathsf{sig}', \mathsf{membTag}')$ and tbs' contains $(\mathsf{groupCtxt}', \mathsf{groupId}', \mathsf{epoch}', \mathsf{leafId}', \text{‘}\mathsf{commit}', C')$, and c^* contains analogous values. This means that c' and c^* can only differ on $\mathsf{confTag}' \neq \mathsf{confTag}^*$ or $\mathsf{membTag}' \neq \mathsf{membTag}^*$. Furthermore, recall that $\mathsf{membTag}' = \mathsf{MAC.tag}(\mathsf{membKey}', C')$ and $\mathsf{membTag}^* = \mathsf{MAC.tag}(\mathsf{membKey}^*, C^*)$. We have $C' = C^*$, because they are included in $\mathsf{tbs}' = \mathsf{tbs}^*$. Moreover, $\mathsf{tbs}' = \mathsf{tbs}^*$ includes $\mathsf{groupCtxt}' = \mathsf{groupCtxt}^*$. Since the group context uniquely determines the epoch and key schedule, it follows that $\mathsf{membKey}' = \mathsf{membKey}^*$. Therefore, $\mathsf{membTag}' = \mathsf{membTag}^*$. We can prove analogously that $\mathsf{confTag}' = \mathsf{confTag}^*$. □

Proof (of Lemma 4). We consider the case where the injected packet triggering $\mathsf{Bad}_{\mathsf{MAC}}$ is a commit and that it does not remove any party, i.e., all receivers verify the MAC tag $\mathsf{confTag}$ and not $\mathsf{membTag}$. The case for proposals and the removed parties who verify $\mathsf{membTag}$ is analogous.

Let $\mathcal{Z}$ be any environment. The reduction $\mathcal{A}_{\mathsf{MAC}}$ first guesses the node c that makes $\mathsf{Bad}_{\mathsf{MAC}}$ occur. Further, it runs $\mathcal{Z}$, emulating the UC experiment exactly, except instead of the MAC key $\mathsf{membKey}$ in epoch c, it uses its oracles in the MAC EUF-CMA game. If $\mathsf{Bad}_{\mathsf{MAC}}$ occurs for parent node c and injected child c', $\mathcal{A}_{\mathsf{MAC}}$ stops the experiment and outputs the forgery $(\mathsf{confTag}', \mathsf{tbm}')$, denoting, respectively, the confirmation MAC tag and the MAC'ed content from c'. We claim that

1. The difference between the probability of $\mathsf{Bad}_{\mathsf{MAC}}$ occurring for c in the experiment emulated by $\mathcal{A}_{\mathsf{MAC}}$'s and in Hybrid 3 (or 4) is upper-bounded by the advantage of a reduction $\mathcal{A}_{\mathsf{PKE}}$ in the GSD game.
2. If $\mathsf{Bad}_{\mathsf{MAC}}$ occurs for c, then $\mathcal{A}_{\mathsf{MAC}}$ wins with $(\mathsf{confTag}', \mathsf{tbm}')$.

For the first claim, observe that $\mathsf{Bad}_{\mathsf{MAC}}$ for c implies that $\mathbf{safe}^*(c)$ is true until the event and no party in c is exposed. Therefore, we can construct $\mathcal{A}_{\mathsf{PKE}}$ the same way as in the proof that ITK^* guarantees confidentiality (indistinguishability of Hybrids 2 and 3). In particular, $\mathcal{A}_{\mathsf{PKE}}$ embeds the challenge in the MAC key $\mathsf{membKey}$ in c and emulates the experiment as described in that proof, until $\mathsf{Bad}_{\mathsf{MAC}}$ occurs for c. Then, it stops the experiment, since the rest of it has no effect on the probability of the event.[31]

[31] Note that afterwards the emulation may be easily distinguishable. Indeed, if a party in the parent of c, i.e., one who can compute the real $\mathsf{confKey}$ used to authenticate c', is corrupted, then its state is inconsistent with the random key in the experiment emulated by $\mathcal{A}_{\mathsf{PKE}}$. In the proof of indistinguishability of Hybrids 2 and 3, such corruptions would be disallowed, in order to avoid the commitment problem. Here we can prove a stronger statement without restrictions, since the experiment after the event doesn't matter.

For the second claim, we know that $\mathsf{confTag}'$ is a valid tag on tbm', because it was verified by a party accepting the injected c'. It is left to show that $(\mathsf{confTag}', \mathsf{tbm}')$ was not queried by $\mathcal{A}_{\mathsf{MAC}}$ to the MAC oracle. Assume towards a contradiction that $(\mathsf{confTag}', \mathsf{tbm}') = (\mathsf{confTag}^*, \mathsf{tbm}^*)$, where $(\mathsf{confTag}^*, \mathsf{tbm}^*)$ was queried by $\mathcal{A}_{\mathsf{MAC}}$ to the oracle when it honestly generated a packet c^*. Recall that tbm' and tbm^* are equal to the confirmed transcript hashes $\mathsf{confTransHash}'$ and $\mathsf{confTransHash}^*$. If $\mathsf{confTransHash}'$ and $\mathsf{confTransHash}^*$, then $c' = c^*$, because the transcript is computed by hashing the last message. This is a contradiction, since c^* is honest and c' is injected. □

D.5 Stronger Security of ITK

Notice that the only difference between **safe** and **safe*** concerns how detached history-graph nodes are handled, so it suffices to consider these. Specifically, for a detached node c, **safe** only guarantees that the epoch doesn't include an exposed signing key spk. We argue that this implies that no node in the entire ratchet tree of c contains an exposed key pk (that is, the secret key to pk was not leaked to, or chosen by, the environment). In particular, that would mean that no ciphertext in the most recent commit (and/or welcome messages) leading to the epoch can be decrypted by the environment. (More formally, the plaintexts are indistinguishable from random to the environment.) Thus, the environment is essentially unable to query the RO at the points it would need to compute the new application secret, meaning it too would look random, meaning the epoch is indeed secure.

It remains to argue that **safe** returning true for epoch c implies that no node $\tau.v$ in the epoch's ratchet tree τ contains an exposed key pk. The proof is by (strong) induction on the height of $\tau.v$. We showed already in the proof of Theorem 1 that when $\tau.v$ is a leaf with an exposed pk then spk at v must also have been exposed, which contradicts **safe**. Now take any internal node $\tau.v$ and assume that for any node in τ with height smaller than the height of $\tau.v$, the pk is not exposed.

We first argue that all signatures in τ must verify. Note that by definition for a history-graph node introduced by the simulator to exist, it must be that at least one party has accepted either a commit or welcome message leading to that node. Let c_a be the ancestor epoch of c referenced in the first two clauses of b) in **safe**. Since the node exists it must be that a party accepted a welcome message leading to the node.[32] Thus, it must be that all signatures at the leaves of c_a verify. For each of the subsequent epochs up to and including c, each new signature inserted into the group state not present in the previous epoch must have been verified, at least once; namely, by the party that accepted a commit message causing the epoch's history-graph node to be created. Thus, we can conclude that all signatures in τ verify.

A similar argument lets us conclude that the parent hash of $\tau.v$ is included (via a hash chain) in the $\mathsf{parentHash}$ value signed at some leaf in the subtree rooted at $\tau.v$. Let id_s denote the owner of this leaf. (We can assume that id_s is in $\tau.v$'s subtree, as otherwise no party would have accepted the commit or welcome message leading to this epoch.) Let $\tau.u$ and $\tau.w$ denote $\tau.v$'s children, where id_s is in $\tau.u$'s subtree. The parent hash of $\tau.v$ includes $\tau.v.\mathsf{pk}$ and $\tau.w.\mathsf{origChildResolution}$.

By assumption id_s's signature public key spk_s is not exposed so (due to the unforgeability of the signature scheme) $\tau.v.\mathsf{pk}$ was generated honestly by id_s as part of some commit message c_s (possibly for a different session). Moreover, committing with bad randomness would expose spk_s, so c_s was generated with good randomness. This means that the secret key $\tau.v.\mathsf{sk}$ is not chosen by the environment. Therefore, the only way $\tau.v.\mathsf{pk}$ can be exposed is if id_s used an exposed public key to encrypt a path secret for a node on the path from id_s's leaf to $\tau.v$.[33]

Let τ_s denote the tree in epoch c_s. Due to unforgeability of the signature scheme, the path from id_s's leaf to $\tau.v$ is the same in τ_s and τ. Similarly, the $\mathsf{origChildResolution}$ of $\tau.w$ (including $\tau.w$) in τ is equal to $\tau_s.w$'s resolution in τ_s (note that $\tau_s.v$ has no unmerged leaves in τ_s.)

Encrypting a path secret for a node on the path from id_s's leaf to $\tau.u$ under an exposed key would expose pk at $\tau.u$, which contradicts the induction hypothesis (since $\tau.u$ is below $\tau.v$). The only

[32] Since c_a has no parent no party accepted a commit message leading to c_a which only leaves a welcome message as having triggered the creation of c_a.

[33] Note that after c_s is sent, $\tau.v$'s secret key never leaves the local state of any party who knows it, that is any party in $\tau.v$'s subtree. This easily follows by inspection, for example, any commit by such party replaces $\tau.v$'s key pair.

remaining path secret is the one for $\tau.v$, encrypted by id_s under all keys in $\tau.w$'s $\mathsf{origChildResolution}$. However, exposing any of these nodes again contradicts the induction hypothesis. (Note that the induction hypothesis concerns only nodes in τ, so we used unforgeability of the signature scheme to argue that the relevant nodes in τ are the same as those in τ_s, actually used by id_s. Note also that without the signature, we would not be able to assume anything about the height of nodes used by id_s.)