On The Insider Security of MLS

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

[1] Wickr, `jalwen@wickr.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 a new complex open standard for end-to-end (E2E) secure group messaging being developed by the IETF. Its primary security goal is to provide E2E privacy and authenticity for messages in long lived sessions whenever possible. This, despite the participation (at times) of malicious insiders that can interact with the PKI at will, actively deviate from the protocol, leak honest parties' states, and fully control the network.

The cryptographic 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. CGKA protocols provide asynchronous E2E secure *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 in the form of 3 contributions. On the theory side, we overcome several subtelties to formulate the first notion of insider security for a CGKA (or group messaging) protocol. Next, we isolate the core components of MLS to obtain a CGKA protocol we dubbed *Insider Secure TreeKEM* (ITK). Finally, we give a rigorous proof that ITK provides (adaptive) insider security. In particular, this work also initiates the study of insider secure CGKA protocols, a primitive of interest in its own right.

⋆ 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 . . . 3
 1.1 Background and Motivation . . . 3
 1.2 Our Contribution . . . 4
 1.3 Related Work . . . 7
2 Preliminaries . . . 8
 2.1 Notation . . . 8
 2.2 Universal Composability . . . 9
 2.3 Primitives . . . 9
3 Continuous Group Key Agreement . . . 10
 3.1 CGKA Syntax . . . 10
 3.2 The Security Model . . . 11
 3.3 PKI Setup . . . 12
 3.4 History Graph . . . 14
 3.5 The CGKA Functionality . . . 14
4 The Insider-secure TreeKEM Protocol . . . 18
 4.1 Protocol Overview . . . 18
 4.2 Protocol State . . . 20
 4.3 Setup Algorithms . . . 22
 4.4 Protocol Algorithms . . . 24
 4.5 Simplifications and Deviations . . . 29
5 Security of ITK . . . 30
 5.1 The Safety Predicate for ITK . . . 30
 5.2 Security Statement . . . 32
6 Proof of Theorem 1: Security of ITK . . . 33
 6.1 Modified GSD Security . . . 34
 6.2 Indistinguishability of Hybrids 3 And 4 . . . 37
 6.3 The Rest of the Hybrids . . . 43
7 Different Tree-Signing Methods . . . 45
 7.1 Tree Signing of ITK is Suboptimal . . . 45
 7.2 Alternative Tree-Signing: ITK* . . . 45
 7.3 Security of ITK* . . . 46
A Preliminaries: Cryptographic Primitives . . . 50

1 Introduction

1.1 Background and Motivation

A *Continuous Group Key Agreement* (CGKA) protocol allows an evolving group of users to agree on a continuous sequence of shared symmetric keys. 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.

Therefore, in contrast to older Dynamic Group Key Agreement (DGKA) protocols, CGKA protocols rely only on *asynchronous* communication. No assumptions are made about when or for how long parties are online. Instead, the (untrusted) network is expected only to buffer packets for each party until they come online again. Thus, all actions the party might wish to take must be performed non-interactively. What's more, in contrast to Broadcast Encryption, a CGKA protocol cannot rely on specially designated parties (like group managers) or trusted third parties (like a trusted broadcast channel). The only exception is a trusted PKI that binds identities to their public keys.[4]

Intuitively, CGKA protocols encapsulate the cryptographic core necessary to build higher-level distributed and end-to-end (E2E) secure group applications. To this end, any change to the state of a group (e.g. group members) initiates a new *epoch* for the session. Each epoch E is equipped with a fresh, uniform and independent epoch key k_E, called the *application secret* of E, which can be computed by all group members in E.[5] The term "application secret" reflects the expectation that k_E will be used by a higher-level cryptographic application during E. 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. In light of this paradigm, CGKA encapsulates the basic cryptographic task of managing group membership by providing a continuous stream of private and authenticated symmetric key material to *current* group members to build upon.

The Messaging Layer Security Protocol. Probably the most important family of CGKA protocols today is TreeKEM. An initial version was introduced in [29] followed by a later, more precise, description in [13]. This was followed by the improved TreeKEMv2 [7]. Finally, the most recent "propose-and-commit" variant (henceforth: TreeKEMv3) [12] encapsulates much of the cryptographic core of (the most recent version of) the *Messaging Layer Security* (MLS) protocol [6].

The MLS protocol is being developed by the IETF and aims to define an open standard for end-to-end (E2E) secure group messaging, in particular for very large groups (e.g. 50K users).[6] MLS is being developed as a collaboration between cryptographers at numerous academic institutions around the world as well industry actors including Cloudflare, Cisco, Facebook, Google, Twitter, Wickr and Wire. Together, they already provide secure messaging services to over 2 billion users. Moreover, the IETF is currently explicitly soliciting more feedback from the cryptographic community.

The Security Goal Of MLS. Intuitively, MLS aims to allow for private and authenticated group communication in the face of a fully adversarial network coordinating with malicious group members, aka. *insiders*. We call this (quite ambitious) security goal *insider security* and, in some sense, it sets the bar for what we would like to prove for MLS. The goal of this work is to make substantial progress in this direction.

Clearly, there are limits to what can be achieved against insiders. We can hardly hope for privacy from a malicious group member as long as they explicitly remain part of the group. Thus, a more nuanced security goal is to prevent all but the "unavoidable" attacks. Once the insider has been removed from the group it is reasonable to once again ask for privacy for future epochs. Similarly, we can still hope for privacy for epochs before they ever explicitly joined the group.

[4] In practice, the server distributing idenity/key bindings is often not considered trusted per se. Instead a trusted PKI is implemented by 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] More than that though, it serves as a proof-of-concept of the power and practicality of the CGKA-based protocol design paradigm described above.

PASSIVE SECURITY. Due to its complexity, formal analyses of (the different versions of) MLS have focused on the underlying CGKA protocols. Specifically, the above intuition has given rise to the (collection of) formal notions known as *Forward Secrecy* (FS) and *Post Compromise Security* (PCS) [21]. FS denotes security of a current epoch despite *future* compromises while PCS denotes security of a current epoch despite *past* compromises. Recently, [1] formalized Post Compromise Forward Security (PCFS) as the (strictly stronger) combination of the two holding simultaneously; a current epoch remains secure despite *both* past and future compromises.

The authors of [1] went on to show that TreeKEMv2 has significantly less than optimal FS. That is, privacy for a current epoch E can be violated via the compromise of a group member's local state during any one of a number of subsequent epochs. Unfortunately, this translates directly into poorer than hoped for PCFS for MLS. (Their FS/PCFS attacks easily extend to TreeKEMv3/MLS.)

On a more positive note, [1] showed that TreeKEMv2 does satisfy (an obviously sub-optimal) flavour of PCFS; albeit only against a class of *weakly passive* adversaries. These may (repeatedly) leak group members' local protocol states (including all their key material). However, the network behaves more or less honestly with the adversary unable to modify or inject packets, and required to delivery packets in the same order to all parties. That is, he at most may globally delay, drop, or re-order packets.

A somewhat stronger model was considered by [2] who showed that TreeKEMv2 enjoys PCFS even against *adaptive* and *strongly passive* adversaries. In fact, their adversaries can even learn (but not modify) the random coins of parties, albeit only in conjunction with a state leakage. An "adaptive" adversary can decide on its next action depending on its view of the execution thus far. [2] introduces a general technique for proving adaptive security of TreeKEM-like protocols (including TreeKEMv2 and TreeKEMv3) with polynomial loss (in the random oracle model). Moreover, the "strongly passive" adversaries of [2] can leak group members' states just as in [1] but can freely re-order, delay, and drop packets per recipient while en-route.[7] Still, even in [2] the network remains authenticated in that the adversary may not modify or inject packets.

ACTIVE SECURITY. An active adversary is one which has full control over the network; i.e. it can inject, modify and delete packets. Even just rigorously defining active security for CGKA protocols (let alone for group messaging protocols like MLS), has proven to be surprisingly challenging.

The first step in this direction was only recently made in [4]. In particular, they provide 2 real/ideal style definitions capturing adaptive active adversaries that can additionally set the random coins of honest (and corrupt) parties. (They also consider a third, passively secure notion.) The active security notions guarantee (slightly different versions of) robustness. More precisely, *weak robustness* which ensures that all honest parties accepting some (potentially adversarially produced) packet p will transition to compatible states; in other words they end up in the same epoch.

However, as opposed to [2], the notion of [4] also requires *optimal* security in the intuitive sense that an adversary should only be able to break the security of epoch keys that the parties it corrupted could trivially compute themselves. Unfortunately, it is clear that neither TreeKEMv2 nor TreeKEMv3 (nor MLS itself) are optimally secure in this sense [8]. (Indeed, to meet their notions, [4] goes on to introduce 3 new, but impractical, CGKA protocols.)

INSIDER SECURITY. Active adversaries (as captured in [2]) can choose the coins of corrupt parties and send arbitrary network packets on their behalf. Yet, there remains one more capability available to a malicious insider: interacting with the PKI on behalf of the corrupt parties. Thus, security against active attackers (regardless of details) does not tell us a complete story about insider security.

1.2 Our Contribution

In a nutshell, this work contains three high-level contributions.

[7] Such attacks are sometimes called *splitting* or *forking* attacks as they (unavoidably) can be used to partition the group into subsets that evolve independently of each other.

[8] For instance, they allow so-called *cross-group* attacks where the adversary uses state leakage from one branch after a forking attack to target another branch.

1. On the theory side, we close the gap between active and insider adversaries by formulating the first security notion considering the latter. Along the way we also provide the strongest security guarantees to date for new members joining a (potentially adversarially generated) session.
2. Second, we isolate the core features of the full MLS protocol sufficient for encapsulating an insider secure CGKA protocol which we call *Insider Secure TreeKEM* (ITK). In particular, ITK augments TreeKEMv3, with message authentication, "tree-signing", confirmation keys and a small part of the MLS's key schedule.
3. Finally, we formulate and prove the accurate security guarantees provided by ITK in the presence of malicious insiders and adaptive corruptions, giving strong evidence for the type of insider security guaranteed by MLS. Along the way we identified three places where security of MLS (at the time of writing) could be improved by small protocol changes. Fixes for the first two issues are already incorporated into the standard, and hence included in ITK. The fix for the last issue related to tree-signing affects deniability, and hence we analyze the improved security it provides separately.

Defining Insider Security. Though incomplete for our purposes, the notion of weakly robust, adaptive and active security in [4] serves as a helpful starting point for defining insider security. Indeed, we extend it to formalize adversaries that can either act as malicious insiders, active attackers, or attackers leaking state. Technically, capturing the PCS guarantees against the latter two types is more challenging than one might think at first glance.

To model malicious insiders, one might begin with a model like the one in [28], where temporary corruption of a party lets the adversary dictate how that party behaves during the corruption. The issue is that during the corruption the adversary can arbitrary alter the party's local state. Hence, at the end of a corruption it would not be clear if and when we can expect the party to return to a secure state via continued honest protocol execution; a feature mandated by the PCS property targeted by MLS.[9] In other words, modeling insiders this way makes it difficult simultaneously provide non-trivial PCS guarantees.

Thus we follow [4] by modeling the moment an honest party P becomes a malicious insider P^* using an adversary that leaks P's state, fully controls honest P's network access, can choose P's randomness coins, and (new to this work) can interact with the PKI on behalf of P. Formally, P continues to be a part of the execution as an honest party, as does P^*, albeit within the adversary. This resolves the above conflict, because on the one hand we can now meaningfully ask for a return to a secure state if/when the adversary lets P participate in the execution again. On the other hand, we also effectively capture security against P^* because we have endowed the adversary with every possible capability P^* could have made use of if we were to make P^* more explicit.

With this technique in mind, it remains to flesh out the adversaries interaction with the PKI. In contrast to prior works, we have opted for a much more detailed and faithful modeling of MLS's PKI as described in its architecture document [27]. In more detail, MLS uses two distinct PKI services: the *Authentication Service* (AS) for binding identities to their long-term (signature) keys and the *Key Service* (KS) which maintains ephemeral public *Key Packages* that are used to non-interactively add new members to a group while they are offline. We model the KS as little more than an untrusted database. For the AS, although we also let the adversary register arbitrary public keys on behalf of parties, we, intuitively, only expect security for epochs that involve keys exclusively registered by honest parties (and that have not been leaked to the adversary in the meantime).

ITK: The Insider Secure CGKA Underpinning MLS. Armed with the above security notion, we turned to extending TreeKEMv3 with the extra cryptographic machinery applied to it in MLS. Thus we obtain the insider secure CGKA protocol ITK, implicit in MLS. For this, we used MLS in its "MLSPlaintext" common packet framing mode; that is where packets are authenticated and sent over the wire in the clear. Intuitively, the alternative mode (MLSCiphertext) only improves in terms of meta-data hiding which is outside the scope of this work. Moreover,

[9] Indeed, it is not even clear what continued execution would mean when starting from an arbitrary local state.

one can assume that if a protocol is secure when packets are authenticated but sent in the clear then the same security is obtained by additionally encrypting those packets; the converse being of course much less clear.

ITK contains the following machinery on top of TreeKEMv3 not analyzed in prior work.

Key Schedule: TreeKEMv3, like any CGKA, provides a stream of symmetric keys. MLS combines those keys using in a continuous key schedule mixing in the outputs of TreeKEMv3. ITK includes the core of this key schedule. Intuitively, this serves to translate FS guarantees of TreeKEMv3 into PCFS guarantees for ITK.

Confirmation Tag: To both guarantee weak robustness and authenticate TreeKEMv3 packets that initiate a new epoch, ITK extends those packets to include a confirmation tag: a MAC of the entire CGKA transcript up to and including that packet. The MAC key is derived from the new epoch's key schedule, proving knowledge thereof to the receiver (which in turn also implies knowing the old epoch's key schedule).

Packet MACing: To extend the same type of authenticity to the remaining protocol packets, ITK requires them to be MACed using a key derived from the current epoch's key schedule.

Packet Signing: To authenticate a sender's identity to a receiver ITK includes singing keys for each group member as part of the group's cryptographic state. Protocol packets must be signed by their sender.

Tree Signing: Tree-singing is a critical (and previously unexplored) part of how MLS tries to provide meaningful security guarantees to new members when they join a group. In a nutshell, tree-signing demands that all new public keys added to an ongoing session's distributed cryptographic state are signed under a long-term key of the party introducing them. The signatures are included in the state and are verified by the new member when they join. The idea is that if a long-term signature key does not belong (or leak) to an insider, then the signed public keys are honestly generated and we can give additional guarantees to a joining party, even if they are invited by a malicious insider to an artificial group.

Accurate Security Guarantees of ITK. Our goal was to prove a strong security statement about the ITK protocol as-is. Below we briefly summarize our main findings.

Issues with MLS with Fixes Incorporated Into the Standard. We discovered two issues, the fixes for which are now incorporated into MLS [9, 10]:

1. An MLSCiphertext implicitly, via authenticated encryption, provided stronger authentication than an MLSPlaintext: Forging the former requires the epoch secrets, while forging the latter requires only the signing key of some member (which may leak in a different context). To bring the authenticity guarantees in line, we proposed adding a MAC to MLSPlaintexts.
2. The way the transcript hash was computed and included in the confirmation tag lead to counter-intuitive behavior, where parties think they are in-sync and agree on all current-epoch secrets, but in fact they are out-of-sync, meaning that they will never progress to the same next epoch. Our fix has been incorporated into MLS.

Security Guarantees For New Group Members. One of the explicit goals of ITK, in part achieved by tree-signing, is to provide non-trivial security guarantees to new members joining a group *regardless* of who sent the invitation packet. Expressing such guarantees is considered out of scope in [4], where security is only extended to an (adversarial) epoch E if (roughly speaking) at least one honest party already in a group accepted the (adversary's) protocol packet transitioning to E. Hence, we improve on their definition and formalize the security gained by tree signing.

Tree Signing. In more detail, we prove that the tree-signing mechanism provides the following security property: a new member P that joins a group (even if invited by a malicious insider) ends up in a secure epoch if all members with the following types of long-term signature keys are removed from the group. Either A) the keys are corrupted (i.e., registered by an insider or leaked), or B) are being used in a different group that itself includes a corrupted signing key.

We believe this to be an unexpectedly weak guarantee. In particular, one might hope that in any group all epochs without type A) keys are secure. (After all, ITK does mandate that any secrets introduced into a group's state be freshly and independently sampled.) Surprisingly, that

is not the case. Indeed, requiring the removal of keys of type B) to ensure security is not just an artifact of our proof. It is relatively easy to construct a scenario where an insider in a real group G_1 with other honest parties and cryptographic state s_1 can construct a state s_2 for a fake group G_2 such that A) G_2 only contains honest members of G_1 and B) s_2 consists only of a subset of values introduced to s_1 by those honest parties. The upshot is that s_2 contains no keys of type A). Nonetheless, as a member of G_1 it is not difficult for the insider to learn some (if not even all) the secrets in s_2.

Intuitively, the reason for this is that (in an effort to achieve better deniability properties) the signatures in MLS's/ITK's tree-signing mechanism authenticate only the introduced public keys, but nothing about the context they were created for. Crucially, the signatures do not authenticate which other group members were explicitely sent each new secret key. This leaves the adversary wiggle room to "cut-and-paste" parts of the group state from different existing groups to produce plausible (but otherwise artificial) new group states. To mount an attack, the adversary can copy over keys from a "real" group that were sampled and signed by honest parties who then sent the corresponding secret keys to the adversary. But in the artificial group state, the adversary instead arranges things in a way that (erroneously) indicate they were *not* sent the secret keys. Thus removing the adversary from that group wont result those keys being refreshed. In other words, the adversary remains aware of secret keys used by the group even after being kicked out of the group.

Modified Tree Signing. Interestingly, an alternative tree-signing method has been considered for MLS [30], but rejected due to having worse deniability properties but providing an "unclear" (at the time) benefit for other security properties. We revisit that method to prove that ITK modified accordingly does achieve the expected security, namely, all epochs without type A) keys are secure.

Precise PC/FS Guarantees. Following previous works on the security of CGKA [2, 1, 4], we formalize the precise PCS/FS guarantees provided using a so-called *safe predicate* — a predicate that, given a symbolic representation of a CGKA execution and an epoch E, decides whether the key k_E is secure. However, as previous works considered weaker types of adversaries and/or analyzed protocols with stronger security properties, they got away with formalizing relatively simple safe predicates of the form "k_E is secure if it is not leaked by any *single* exposure of a party in epoch E'". Unfortunately, this is no longer possible for insider security of ITK. In particular, a combination of exposures might allow computing other keys that were not explicitly leaked. Hence, our safe predicate takes the form of recursive "deduction rules", reflecting the adversary collecting information from different exposures.

Multi-Group Security. Based on [4], our security definition is phrased in the Universal Composability (UC) framework [17], and it considers a single group, i.e., a single *session* of ITK. Intuitively, one would expect that multi-session security follows by the UC composition theorem if the PKI is modeled as a global (i.e., shared) functionality [18]. Unfortunately, we believe that (both of the) tree-signing methods considered in this work preclude any hope of using a general composition theorem this way. In particular, both methods call for parties to sign cryptographic data without sufficient (or any) binding to the session the material belongs to. This leads to the potential for reusing said material in concurrent sessions in a way where events in one session directly affect the security of the other. Therefor we believe multi-session security may instead have to be (formally) dealt more directly e.g. by considering a multi-session security game. We leave this for future work.

1.3 Related Work

To date, little formal analysis of (components of) MLS have been made public. Beyond the results in [2, 1] discussed in the introduction, the only other example we are aware of is the recent work of [22] 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. [10]

[10] We are also aware of the work [14] described on `https://hal.inria.fr/hal-02425229` analyzing MLS using automated verification tools but are unfortunately unable to find a public copy of the paper.

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-Gorden et al. in [20]. The authors showed that (at least for a group with fixed membership) ART is secure against adaptive and strongly passive adversaries. The security loss in their proof is exponential in the group size. [11]

Not long after ART, the TTKEM protocol was introduced in [2] where it was shown to enjoy the same security as TreeKEMv2 (at least with regards to adaptive and strongly passive adversaries). TTKEM is motivated by exhibiting an different computation and cost trade profile to TreeKEMv2. The authors run experiments comparing TreeKEM variants with TTKEM that show TTKEM enjoying significant efficiency advantages in some plausible use cases especially for larger groups (e.g. when groups are managed by a small set of "administrators" in charge of adding/removing members). Meanwhile, the rTreeKEM construction of [1] 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 [31] supports concurrent changes to the group state (although it lacks a formal security analysis). Similarly, the protocol of [15] supports a certain types of concurrency with in 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 [4], eschew this constraint to instead focus on exploring new mechanisms for achieving the increasingly stringent security notions introduced in that work. As discussed above, up until 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.[12]

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

2 Preliminaries

2.1 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 $x := (x_1, \ldots, x_n)$ and $y := (y_1, \ldots, y_m)$ we denote the concatenation by $x \mathbin{+\!\!+} y = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ and use $x \mathbin{+\!\!+}\!\!\leftarrow v$ as a shorthand for $x \leftarrow x \mathbin{+\!\!+} (v)$. Moreover, let $x.\mathsf{reverse}() := (x_n, x_{n-1}, \ldots, x_1)$ 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 $\mathsf{zip}(x, y) := ((x_1, y_1), \ldots, (x_n, y_n))$ 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 $S[x] := \{y \mid (x, y) \in S\}$, 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 $A[*, j] := \{A[i, j] \mid i \in \mathcal{I}\}$ and for a set $S \subseteq \mathcal{I} \times \mathcal{J}$ we write $(i, *) \in S$ as a shorthand for the condition $\exists j \in \mathcal{J} : (i, j) \in S$.

[11] Typically, a quantitative similar security statement with polynomial loss against non-adaptive adversaries can be inferred using the technique of complexity leveraging.

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

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).

2.2 Universal Composability

We formalize security in the generalized universal composability (GUC) framework [18], an extension to the UC framework [17]. We moreover use the modification of responsive environments introduced by Camenisch et al. [16] 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 [4].[13] 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 introduced in [4] (based on prior work by Backes et al. [5] and Jost et al. [24]). More concretely, we 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 by the ideal functionality $\mathcal{F}$ with statements of the form **restrict** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such *cond* when interacting with $\mathcal{F}$.

2.3 Primitives

The protocol ITK makes use of standard cryptographic primitives. For completeness, we outline them in Appendix A.

[13] As explained in the introduction, passive corruptions together with full network control allow to emulate active corruptions.

3 Continuous Group Key Agreement

This section defines syntax and security of Continuous Group Key Agreement (CGKA) protocols. The security definition is quite generic — different PS/FS guarantees of protocols can be expressed by specifying different safe predicates. We define the predicates for MLS's ITK and its version with more secure tree signing ITK^* after explaining the protocols in Sections 5 and 7, respectively.

3.1 CGKA Syntax

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 selects a list of proposals and indicates it in a *commit message*. Upon receiving such message, a party applies the indicated proposals and transitions to the new epoch.

It is generally up to the higher-level protocol to decide which subsets of proposals are deemed to be legal in a commit as part of a *group policy.* Akin to MLSv9, we however require that the proposal vector must first contain all updates, then all removes, and finally all adds. For simplicity, we delegate the buffering of proposals to the higher-level protocol as well.

Handling of 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.

The formal syntax. We consider a stateful protocol for a single group that accepts the following inputs. For simplicity, we treat the party's identity id as implicitly known to the protocol.

- **Group Creation:** $(\texttt{Create}, \mathsf{spk})$ initializes a new group with id being the single member, using the signing public key spk. (In our model, this input is only allowed once.)
- **Add, Remove Proposals:** $p \leftarrow (\texttt{Propose}, \mathsf{add}\text{-}\mathsf{id}_t)$ (resp., $p \leftarrow (\texttt{Propose}, \mathsf{rem}\text{-}\mathsf{id}_t)$) proposes to add (resp., remove) the party id_t. It outputs a proposal message p or $\perp$ if either id is not in the group or id_t already is in the group (resp., is not in the group).
- **Update Proposal:** $p \leftarrow (\texttt{Propose}, \mathsf{up}\text{-}\mathsf{spk})$ 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 $\perp$ if id is not in the group).
- **Commit:** $(c, w) \leftarrow (\texttt{Commit}, \vec{p}, \mathsf{spk}, \mathsf{force}\text{-}\mathsf{rekey})$ commits the vector of proposals $\vec{p}$ and outputs the commit message c. If the proposals contain at least one add, then it also outputs a single welcome message w that is sent to all freshly added members. The operation optionally updates the signing public key of the committer. The flag $\mathsf{force}\text{-}\mathsf{rekey}$ forces an implicit update of the committer (see below).
- **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.[14] It outputs the committer's identity id_c as well as 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 $\perp$).

[14] 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 $\vec{p}$.

Add-only commits. Our syntax reflects the special “add-only” mode of commits in MLS. That is, if $\vec{p}$ only contains add proposals (and is not empty), then MLS *permits* skipping the implicit update of the committer. We model this with the force-rekey flag: if force-rekey = `false`, then an add-only commit does not do the implicit update, whereas if either force-rekey = `true` or there are non-add proposals, then the implicit update is performed. (Skipping the update also implies ignoring the new spk.)

3.2 The Security Model

Security via Idealized Services. Analogous to [4], 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. creating a proposal or commit) 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. Additionally, this also allows us to consider *correctness* and *robustness* guarantees, in contrast to more “classical” UC treatments that let the adversary deliver the messages. (Such models typically permit trivial protocols that just reject all messages with the simulator just not delivering them in the ideal world.)

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 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. We reflect this by the protocol transparently forwarding those queries from the environment to the PKI. For instance, the environment can instruct the Authentication Service (via the party’s protocol) to register a new key for a party. As a result, the AS generates a new key pair for the party and hands the public key to the environment, making the secret key available to the party’s protocol upon request. The PKI is defined in detail in the next section.

The Ideal World. The ideal world formalizes the security guarantees via the ideal functionality $\mathcal{F}_{\text{CGKA}}$, which internally maintains a so-called *history graph.* History graphs were introduced as a core technique to define security of CGKA in [3] and first used in [4]. 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. For instance, proposal nodes have a label that stores the proposed action, and commit nodes have labels that store the epoch’s application secret and the set of parties corrupted in the given epoch.

Security of the application secrets is then formalized by the functionality choosing a random and independent key for each commit node whenever security is guaranteed; otherwise the simulator gets to choose the key. Whether security is guaranteed in given node, is determined via an explicit *safe predicate* on the node and the history graph. In addition to the secrecy of the keys, the functionality also formalizes authenticity by appropriately disallowing injections.

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 interacting with $\mathcal{F}_{\text{CGKA}}$. Those variants essentially record which keys have been exposed, which in turn is then used to define the safe predicate. The actual keys in the ideal world do not convey any particular meaning beyond serving as identifiers — thus in the ideal world we can leak all secret keys to the simulator (they are necessary to simulate signatures on protocol messages). We note that this roughly corresponds to treating the PKI setup as local rather than global (in the sense UC versus GUC).

A remark on multi-group security. Our security definition considers a single group. Ideally, we would model a proper global PKI (that does not reveal keys to the simulator) which would imply multi-group security by composition. Unfortunately, a number of obstacles make being able to use composition unlikely.

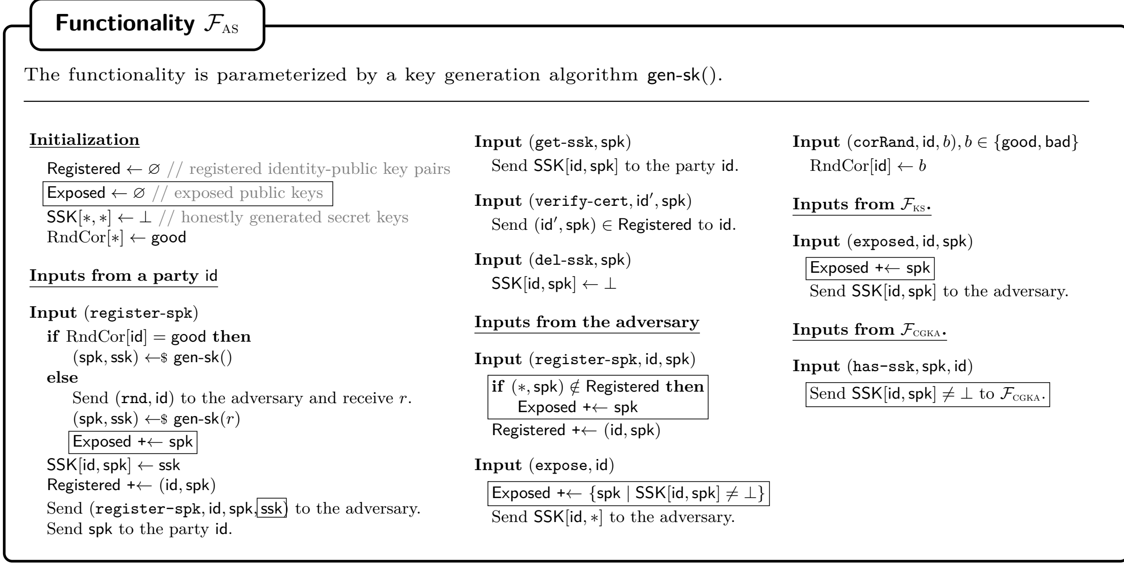

Fig. 1: The Authentication Service $\mathcal{F}_{\text{AS}}$ and its ideal-world variant $\boxed{\mathcal{F}^{\text{IW}}_{\text{AS}}}$.

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.[15] Moreover, the typical techniques used to cope with non-deniable protocols in GUC (e.g. [19]) 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 signed key packages can be used across groups.

Second, the composition theorem of UC assumes that parties have pre-agreement on the session id (i.e., the group identifier). Models [26] more akin to ITK, where parties simply get the group identifier upon joining the group (without pre-agreement), suffer from significantly diminished composition guarantees.

3.3 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.[16]

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. The functionality $\mathcal{F}_{\text{AS}}$, defined in Fig. 1, can be seen as an abstraction of the different credential mechanisms of MLS.

The functionality $\mathcal{F}_{\text{AS}}$ allows a party, identified by id (e.g., a phone number) to register a fresh key pair and verify if a public key spk has been registered by another party (input verify-cert). On registration, the new key pair is generated for the party id by $\mathcal{F}_{\text{AS}}$. If id's randomness is currently corrupted, $\mathcal{F}_{\text{AS}}$ asks the adversary to provide key-generation randomness. The output of id is the new public key spk. The secret key can be retrieved by it at any time with an additional command get-ssk. The secret key can also be deleted, in which case it does not leak upon corruption of id.

The adversary can register arbitrary public keys in the name of any party. Moreover, when a party's state is exposed, all secret keys it generated but not deleted are leaked to the adversary.

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

[16] 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.

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

The functionality is parameterized by a key-package generation algorithm gen-kp(id, spk, 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 $\mathcal{F}_{\text{AS}}$.
    Send ssk to the adversary.
  SK[id, kp] ← sk
  SPK[id, kp] ← spk
  Send (register-pk, id, spk, kp, [sk]) 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), $b \in$ {good, bad}
  RndCor[id] ← b

Fig. 2: The Key Service $\mathcal{F}_{\text{KS}}$ and its ideal-world variant $\boxed{\mathcal{F}_{\text{KS}}^{\text{IW}}}$ that leaks all secrets.

Security is then modeled by having in the ideal-world a variant of $\mathcal{F}_{\text{AS}}$, called $\mathcal{F}_{\text{AS}}^{\text{IW}}$, that marks leaked and adversarially registered keys as exposed (see parts in [boxes]). Looking ahead, the guarantees of $\mathcal{F}_{\text{CGKA}}$ will depend on whether a certain key has been exposed or not.

Furthermore, $\mathcal{F}_{\text{AS}}$ allows the KS functionality $\mathcal{F}_{\text{KS}}$ to signal that a certain ssk leaked. (Formally, $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$ can be understood as a joint functionality.) The role of this command will become clear when we describe $\mathcal{F}_{\text{KS}}$. Finally, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ always leaks all secret keys to the simulator. (Recall that this does not affect security, but simply means $\mathcal{F}_{\text{CGKA}}$ is a local functionality only.)

Key Service (KS). The KS allows parties to upload one-time key packages, used to add them to groups while they are offline.[17]

The KS is formalized by the functionality $\mathcal{F}_{\text{KS}}$, defined in Fig. 2. Similar to $\mathcal{F}_{\text{AS}}$, a party id can register a fresh key package, which is generated by $\mathcal{F}_{\text{KS}}$ using the algorithm gen-kp(id, spk, ssk), which takes as input the party's identity and a long-term signature key pair (reflecting that a key package may be signed) and outputs a key package public key and secret key. If id's randomness is corrupted, the randomness is taken from the adversary. Moreover, using the signature secret key ssk with bad randomness may leak ssk.[18] Hence, the functionality signals to $\mathcal{F}_{\text{AS}}$ that spk is exposed and sends ssk to the adversary.

A party id can request another party's key package (input get-kp). The returned key fully is specified by the adversary. The party can also retrieve all its (not yet deleted) secrets alongside the respective key packages using the input get-sks. (This accounts for the protocol not a priori knowing which key package has been used to add it to the group.) The other inputs are analogous to $\mathcal{F}_{\text{AS}}$.

Remark 1. Unlike $\mathcal{F}_{\text{AS}}^{\text{IW}}$, $\mathcal{F}_{\text{KS}}^{\text{IW}}$ does not keep track of exposed key packages. The reason is that our security definition abstracts away key packages and is only aware of long-term keys from AS. This makes the security guarantees simpler and more comprehensible. On the other hand, we do not

[17] In MLS, the KS is implemented as part of the Delivery Service, and in Signal it is called the Key Distribution Center.
[18] This is true e.g. for ECDSA, which is one of the schemes allowed by MLS.

guarantee security in some border cases where it would be provided.[19] We believe this to be a good trade-off between abstraction and fine-grained guarantees.

3.4 History Graph

We now proceed by formally defining the history graph used by $\mathcal{F}_{\text{CGKA}}$. Recall that the history graph is a labeled directed acyclic graph, in our case, a forest, with nodes, representing sent or received messages. We first list the nodes' labels. *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;
- 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:

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

Having introduced the history graphs and the PKI, we are ready to define the ideal functionality $\mathcal{F}_{\text{CGKA}}$, formally defined in Fig. 3, with helper functions outsourced to Figs. 4 to 6 $\mathcal{F}_{\text{CGKA}}$ is parameterized by predicates **safe** and **inj-allowed**, specifying which application secrets are secure, and when authenticity is guaranteed, respectively. The predicates are defined (for ITK) in Fig. 15.

State. The functionality maintains the session's history graph. It addresses proposal nodes by the (idealized) proposal message p and non-root commit nodes by the (idealized) commit message c. The root node corresponding to session initialization is addressed by the label root_0. Moreover, other roots may be created without a commit message, e.g. when a party uses an injected welcome message to join an adversarially created epoch, not directly related to the main group. Such roots are addressed by labels root_i for $i > 0$ and their trees are called *detached*.

The functionality also stores for each party id a pointer Ptr[id] to its current history graph node, or $\mathsf{Ptr}[\mathsf{id}] = \bot$ for parties who currently are not in the group.

Interfaces. $\mathcal{F}_{\text{CGKA}}$ offers interfaces for creating the group, creating a proposal, committing a list of proposals, processing a commit, joining, and retrieving the current group key. The designated party $\mathsf{id}_{\mathsf{creator}}$ (specified as part of the session id) initially creates the group with itself as a single member, to which he can then invite additional members.[20] All interfaces except create and join are for group members only (i.e., parties for which $\mathsf{Ptr}[\mathsf{id}] \neq \bot$).

[19] For example, if id's long-term key ssk leaks, but then id manages to generate an honest key package kp using ssk, kp is considered exposed. In our abstraction, kp cannot be distinguished from key packages generated by the adversary using ssk.

[20] Note that parties might join adversarially generated group states before $\mathsf{id}_{\mathsf{creator}}$ created the real group.

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 leaked via corruptions and **inj-allowed**(c, id), specifying when authenticity is not guarantees.

```
Initialization
  Ptr[*], Node[*], Prop[*], Wel[*] ← ⊥
  RndCor[*] ← good; HasKey[*] ← false
  rootCtr ← 0

Inputs from id_creator
Input (Create, spk)
  req Node[root_0] = ⊥ ∧ *valid-spk(id_creator, spk)
  mem ← {id_creator, spk}
  Node[root_0] ← *create-root(id_creator, mem, RndCor[id_creator])
  HasKey[id_creator] ← true
  Ptr[id_creator] ← root_0

Inputs from a party id

Input (Propose, act), act ∈ {up-spk, add-id_t, rem-id_t}
    Send (Propose, id, act) to the adversary and
        receive (p, spk_t, ack).
    req ack
    if act = up-spk then req*valid-spk(id, spk)
    if act = add-id_t then act ← add-id_t-spk_t
    if Prop[p] = ⊥ then
        Prop[p] ← *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 F_AS.
    return p

Input (Commit, p⃗, spk, force-rekey)
    req Ptr[id] ≠ ⊥
    Send (Commit, id, p⃗, spk, force-rekey) to the adversary
        and receive (ack, c, w, rt).
    req*should-succeed-comm(id, p⃗, spk, force-rekey) ∨ ack
    *fill-props(id, p⃗)
    if ¬force-rekey ∧ *only-adds(p⃗) then
        spk ← Node[Ptr[id]].mem[id]
    req *valid-spk(id, spk)
    mem ← *members(Ptr[id], id, p⃗, spk)
    assert mem ≠ ⊥ ∧ (id, spk) ∈ mem
    if Node[c] = ⊥ ∧ rt = ⊥ then
        if ¬force-rekey ∧ *only-adds(p⃗) then stat ← bad
        else stat ← RndCor[id]
        Node[c] ← *create-child(Ptr[id], id, p⃗, mem, stat)
    else
        if Node[c] = ⊥ then c' ← root_rt
        else c' ← c
        *consistent-comm(c', id, p⃗, mem)
        if c ≠ c' then *attach(c, c', id, p⃗)
    assert w ≠ ⊥ iff ∃p ∈ p⃗ : Node[p].act = add-*
    if w ≠ ⊥ then
        assert Wel[w] ∈ {⊥, c}
        Wel[w] ← c
    assert cons-invariant ∧ auth-invariant
    if RndCor[id] = bad then
        Send (exposed, id, Node[Ptr[id]].mem[id]) to F_AS.
    return (c, w)

Input Key
    req Ptr[id] ≠ ⊥ ∧ HasKey[id]
    if Node[Ptr[id]].key = ⊥ then *set-key(Ptr[id])
    HasKey[id] ← false
    return Node[Ptr[id]].key

Input (Process, c, p⃗)
    Send (Process, id, c, p⃗) to the adversary and
        receive (ack, rt, orig', spk').
    req *should-succeed-proc(id, c, p⃗) ∨ ack
    *fill-props(id, p⃗)
    if Node[c] = ⊥ ∧ rt = ⊥ then
        mem ← *members(Ptr[id], orig', p⃗, spk')
        assert mem ≠ ⊥ ∧ inj-allowed(Ptr[id], id)
        Node[c] ← *create-child(Ptr[id], orig', p⃗, mem, adv)
    else
        if Node[c] = ⊥ then c' ← root_rt
        else c' ← c
        id_c ← Node[c'].orig
        spk_c ← Node[c'].mem[id_c]
        mem ← *members(Ptr[id], id_c, p⃗, spk_c)
        assert mem ≠ ⊥
        *valid-successor(c', id, p⃗, mem)
        if c ≠ c' then *attach(c, c', id, p⃗)
    if ∃p ∈ p⃗ : Prop[p].act = rem-id then
        Ptr[id] ← ⊥
    else
        assert id ∈ Node[c].mem
        Ptr[id] ← c
        HasKey[id] ← true
    assert cons-invariant ∧ auth-invariant
    return *output-proc(c)

Input (Join, w)
    Send (Join, id, w) to the adversary and
        receive (ack, c', orig', mem').
    req ack
    c ← Wel[w]
    if c = ⊥ then
        if Node[c'] ≠ ⊥ then c ← c'
        else
            rootCtr++
            c ← root_rootCtr
            Node[c] ← *create-root(orig', mem', adv)
        Wel[w] ← c
    Ptr[id] ← c
    HasKey[id] ← true
    assert id ∈ Node[c].mem ∧ cons-invariant
        ∧ auth-invariant
    return *output-join(c)

Corruptions

Input (Expose, id)
    if Ptr[id] ≠ ⊥ then
        Node[Ptr[id]].exp +← (id, HasKey[id])
        *update-stat-after-exp(id)
        Send (exposed, id, Node[Ptr[id]].mem[id]) to F_AS.
    Send (get-sk) to F_KS and receive SK and SPK.
    for each c, kp s.t. SK[id, kp] ≠ ⊥ ∧ SPK[id, kp] = spk
      ∧ ∃p ∈ Node[c].pro : Prop[p].act = add-id-spk do
        Node[c].exp +← (id, true)
    restrict ∀c, if Node[c].chall = true then safe(c)

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

Fig. 3: The CGKA functionality. The helper functions and the safe predicate are defined in Figs. 4 to 6 and Fig. 15, respectively.

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

```
// Creating nodes
helper *create-child(c, id, p⃗, mem, stat)
  return new node with par ← c, orig ← id, pro ← p⃗, mem ← mem, stat ← stat.

helper *create-root(id, mem, stat)
  return new node with par ← ⊥, orig ← id, pro ← ⊥, mem ← mem, stat ← stat.

helper *create-prop(c, id, act, stat)
  return new proposal with par ← c, orig ← id, act ← act, stat ← stat.

helper *fill-props(id, p⃗)
  for p ∈ p⃗ s.t. Prop[p] = ⊥ do
    Send (Proposal, p) to the adversary and receive (orig, act).
    Prop[p] ← *create-prop(Ptr[id], orig, act, adv)

// Output of process and join
helper *output-proc(c)
  (*, propSem) ← *apply-props(c, Node[c].pro)
  return (Node[c].orig, propSem)

helper *output-join(c)
  return (Node[c].mem, Node[c].orig)

// Does the vector of (existing) proposals create an add-only commit?
helper *only-adds(p⃗)
  return p⃗ ≠ () ∧ ∀p ∈ p⃗ : Prop[p] ≠ ⊥ ∧ Prop[p].act = add-*

// Is the (new) spk′ valid for update or commit?
helper *valid-spk(id, spk′)
  spk ← Node[Ptr[id]].mem[id]
  if spk ≠ ⊥ ∧ spk′ = spk then return true
  Send (has-ssk, spk′, id) to F_AS and receive ack
  return ack

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

// Corruptions
helper *update-stat-after-exp(id)
  for each p s.t. Prop[p] ≠ ⊥ and
      (a) Prop[p].par = Ptr[id] and
      (b) Prop[p].orig = id and
      (c) Prop[p].act = up
  do Prop[p].stat ← bad
  for each c s.t. Node[c] ≠ ⊥ and
      (a) Node[c].par = Ptr[id] and
      (b) Node[c].orig = id
  do Node[c].stat ← bad
```

Fig. 4: The helper functions for creating and maintaining the history graph.

Proposals. When a party id wants to create a proposal, $\mathcal{F}_{\text{CGKA}}$ notifies the adversary, who then provides a flag *ack*, a node identifier (i.e., a message) p and a public key spk_t. Sending $ack = \mathtt{false}$ allows the adversary to specify that the proposal fails, i.e., the output is $\perp$ (e.g. because an already existing member is being added). If the proposal succeeds, and if no node with identifier p exists, then $\mathcal{F}_{\text{CGKA}}$ creates it with status set according to whether id's randomness is currently corrupted. For add proposals, it additionally extends the action by the long-term public key spk_t (specified by the adversary) of the added party id_t. (Note that in the real world, spk_t is a part of a key package for id_t, retrieved by id from $\mathcal{F}_{\text{KS}}$ that allows the adversary to choose the package.)

In certain situations, a proposal may not create a new node. For example, this can happen if id proposes to remove the same party twice within the same epoch. Another such situation is when the adversary uses id's leaked state to generate a proposal itself, makes another party commit it (this creates the proposal node) and then id proposes the same action using corrupted randomness. In such cases, the adversary can specify the preexisting p. $\mathcal{F}_{\text{CGKA}}$ then enforces, using ***consistent-prop** of Fig. 6, that the values stored in the existing node match the expected ones.

Finally, the proposal identifier p is returned to the calling party.

Commits. To create a commit, a party id specifies a list of proposals $\vec{p}$, a (possibly fresh) signature public key spk and a flag force-rekey. If, as determined by force-rekey and $\vec{p}$, the commit is add-only, then the new spk is ignored. Then, $\mathcal{F}_{\text{CGKA}}$ forwards all inputs to the adversary and receives identifiers c and w of commit and welcome nodes, respectively, as well as an *ack* flag. For correctness, we require that committing a valid proposal list — as specified by ***should-succeed-comm** from Fig. 5 — succeeds. Otherwise the adversary can make the commit fail by specifying $ack = \mathtt{false}$. If the commit succeeds, $\mathcal{F}_{\text{CGKA}}$ first asks the adversary to interpret those proposals in $\vec{p}$ for which no node has been created, i.e., the injected proposals. It then computes the member set resulting from applying $\vec{p}$ by calling ***members** from Fig. 6 (which returns $\perp$ if $\vec{p}$ is invalid).

Then, analogously to creating proposals, $\mathcal{F}_{\text{CGKA}}$ either creates a new node, or verifies that the existing node is consistent. The randomness status is set analogously to the proposals. Note that add-only commits have status bad, reflecting that they never heal the committer. It may happen that the existing node is the root of a detached tree. In such case, $\mathcal{F}_{\text{CGKA}}$ attaches it to id's current

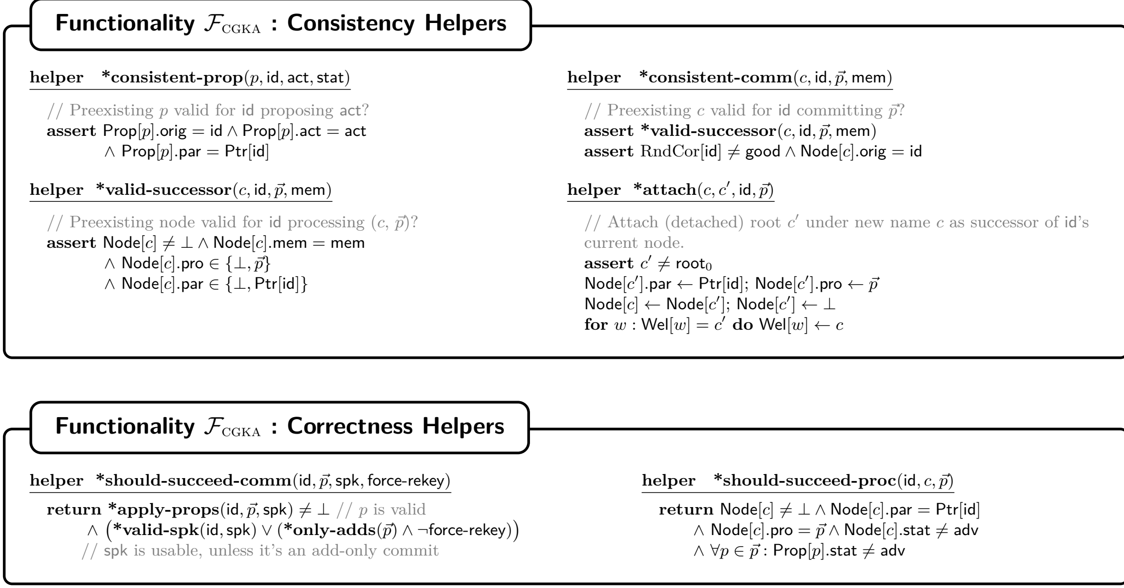

Fig. 5: The additional helper functions. Consistency Helpers ensure that preexisting history graph nodes correspond to the expected actions.

node using ***attach**. This helper also now assigns c as the proper identifier of the node, replacing the temporary root_i identifier. This ensures *weak robustness* — a detached root can be attached to at most one parent, and all parties transitioning into it (both from the main group and using injected welcome messages) end up in consistent states. Moreover, once attached, the root's tree enjoys the same (extended) security guarantees as the main tree. Since attaching an orphan node changes the history graph, $\mathcal{F}_{\text{CGKA}}$ also enforces two invariants: **cons-invariant** enforcing that the graph remains consistent, and **auth-invariant** enforces the authenticity guarantees.

Finally, in case add proposals are committed, $\mathcal{F}_{\text{CGKA}}$ records that the new welcome message leads to the created commit node.

Processing commits. To process a commit, a party id specifies the commit message c and the list of committed proposals $\vec{p}$. (Recall that we require the higher-level protocol to buffer and match them.) We require that processing a valid commit and the matching proposal list succeeds, and otherwisethe adversary can make processing fail using the *ack* flag.

If processing succeeds, the adversary first interprets messages for which nodes do not exist: proposals p are interpreted as in creating a commit, and c is interpreted by specifying the committer orig' and its (possibly fresh) public key in the new group spk'. If $\mathsf{Node}[c]$ already exists, $\mathcal{F}_{\text{CGKA}}$ enforces that it is a valid successor of id's current node and, if it is an orphan node, attaches it.

Finally, depending on whether c removes id, $\mathcal{F}_{\text{CGKA}}$ either moves id's pointer to the new node, or sets the pointer to $\perp$. The committer's identity and the semantics of the applied proposals are returned to the calling party.

Joining. A party can join the group using a welcome message w and a key package kp. As usual, $\mathcal{F}_{\text{CGKA}}$ enforces that joining with valid inputs succeeds. Then, it identifies the commit $c = \mathsf{Wel}[w]$ corresponding to w. If this is the first time $\mathcal{F}_{\text{CGKA}}$ sees w, i.e. $\mathsf{Wel}[w] = \perp$, the adversary chooses c (we require that afterwards all parties joining with w end up in c). If the commit node for c does not exist, it is created as an orphan node with all stored values chosen by the adversary (if the node is attached to a parent, $\mathcal{F}_{\text{CGKA}}$ enforces that these values are consistent). The functionality returns to id the state of the joined group.

Group Keys and Corruptions. The current group key can be fetched via the input `Key`. Keys for which the protocol guarantees secrecy, as identified by the **safe** predicate, are chosen at random, and

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

helper *members$(c, \mathsf{id}_c, \vec{p}, \mathsf{spk}_c)$

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

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

// 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**
  $(\mathsf{id}_s, \mathsf{up}\text{-}\mathsf{spk}) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
  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**
  $(\mathsf{id}_s, \mathsf{rem}\text{-}\mathsf{id}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
  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**
  $(\mathsf{id}_s, \mathsf{add}\text{-}\mathsf{id}_t\text{-}\mathsf{spk}_t) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
  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)

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, 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, 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. 6: The helper functions to determine the group state after applying a commit and the history graph invariants.

insecure keys are set by the simulator. The predicate **safe** uses information related to corruptions, which is recorded by $\mathcal{F}_{\text{CGKA}}$ as follows. When the state of a current group member id is exposed, the functionality records leakage of the following information:

– Any key material necessary to process future control messages, as well as the group key, if not retrieved yet. This is recorded by adding the pair $(\mathsf{id}, \mathsf{HasKey}[\mathsf{id}])$ to the exposed set of id's node, where the flag $\mathsf{HasKey}[\mathsf{id}]$ indicates if id currently stores the group key.
– The key material for updates and commits created by id in the current epoch. This is recorded by setting the status of all child nodes created by id to bad, indicating they no longer heal id.

In addition, exposure of a party id that is not a group member reveals key material that can be used to process welcome messages. Accordingly, $\mathcal{F}_{\text{CGKA}}$ adds id to the exposed sets of all nodes where id can join.

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 **safe** to switch to false.[21]

4 The Insider-secure TreeKEM Protocol

4.1 Protocol Overview

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

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

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. 7 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.

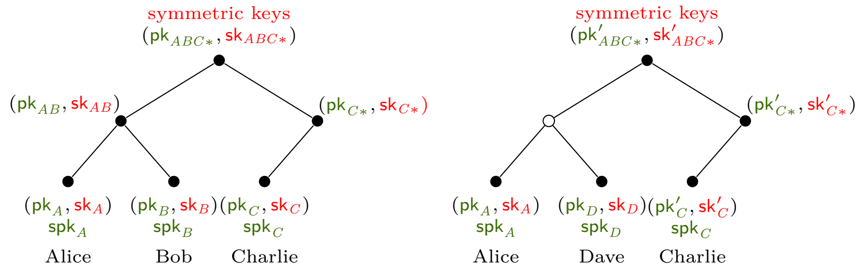

Fig. 7: (Left) An example ratchet tree τ for a group with three members. For Invariant (1), the public labels (green) are known to all parties. For Invariant (2), the secret labels (red) in a node v are only known to parties in v's subtree, e.g. Bob knows sk_B, sk_{AB} and sk_{ABC*}. (Right) the tree after Charlie commits removing Bob and adding Dave. The empty node $\circ$ is blank. Messages can now be encrypted to Alice and Dave by encrypting them to the resolution of the blank node, in this case pk_A and 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. 7). That is, its leaf is cleared, and all keys in its *direct path* — i.e., the path from the party's leaf to the root — are *blanked*, meaning that all their labels are cleared.[22] 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

[22] The leaf is simply marked as unused, implying that the tree does not actually shrink. Hence, the protocol's efficiency is potentially logarithmic in the maximum group size rather than the current group size. Shrinking the tree has been proposed but is not considered in this work.

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.[23]

Re-keying. Whenever a party *commits* a sequence of proposals, they additionally replace their leaf key (providing an implicit update) and re-keys 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. Appendix A). 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 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. The protocol uses a number of additional security features, such as *transcript hashes* to agree on a common history, *confirmation tags* to ensure key confirmation, and *tree signing* to provide some guarantees to newly joint parties. We will discuss those as part of the more formal protocol description.

4.2 Protocol State

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

[23] Another solution would be to blank the new member's direct path, however, this would negatively impact efficiency.

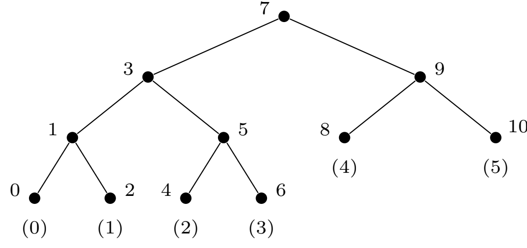

Fig. 8: The tree LBBT_6 with node indices (next to each node) and leaf indices (below the leaves).

Definition 1 (Left-Balanced Binary Tree). *For $n \in \mathbb{N}$ the nth* 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. 8 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. We additionally index leafs from left to right, starting with (0). Observe that for LBBTs this means that leaf indices are equal to their node indices divided by two.

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.)

τ.root	Returns the root.
τ.nodes	Returns the set of all nodes in the tree.
v.isroot	Returns true iff $v = \tau$.root.
v.isleaf	Returns true iff v has no children.
v.parent	Returns the parent node of v (or $\perp$ if v.isroot).
v.lchild	Returns the left child of v (or $\perp$ if v.isleaf).
v.rchild	Returns the right child of v (or $\perp$ if v.isleaf).
v.nodeIdx	Returns the node index of v.
v.leafIdx	Returns the leaf index of v (or $\perp$ if $\neg v$.isleaf).

Table 1: Object oriented notation for LBBTs.

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

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.

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.

4.3 Setup Algorithms

Figure 9 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.

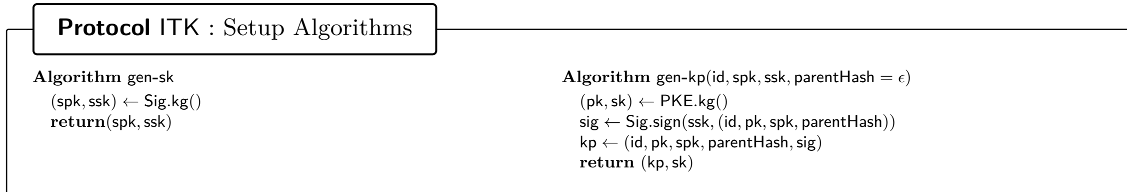

Fig. 9: The algorithms $\mathsf{gen\text{-}sk}$ and $\mathsf{gen\text{-}kp}$, used by $\mathcal{F}_{\text{AS}}$ and $\mathcal{F}_{\text{KS}}$, respectively.

$v.\mathsf{pk}$	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 rooted 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.
$v.\mathsf{spk}$	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[leafIdx]	Returns the leaf with leaf index leafIdx.
τ.leafof(id)	Returns the leaf index of the v for which v.id = id.
τ.getLeaf()	Returns the leaf v with the lowest leafIdx for which $\neg v$.inuse(). If no such leaf exists, adds a new leaf using addLeaf and returns that.
τ.directPath(leafIdx)	Returns the direct path, excluding the leaf, as an ordered list from the leaf to root.
τ.coPath(leafIdx)	Returns the co-path to τ.directPath(leafIdx) as an ordered list.
τ.lca(leafIdx$_1$, leafIdx$_2$)	Returns the least common ancestor of the two leafs.
τ.blankPath(leafIdx)	Calls v.blank() on all $v \in \tau$.directPath(leafIdx).
τ.mergeLeaves(leafIdx)	Sets v.unmergedLvs $\leftarrow \varnothing$ for all $v \in \tau$.directPath(leafIdx)
τ.unmergeLeaf(leafIdx)	Sets v.unmergedLvs $+\!\!\leftarrow$ leafIdx for all v returned by τ.directPath(leafIdx)
v.kp()	Returns (v.id, v.pk, v.spk, v.parentHash, v.sig) (undefined if $\neg v$.isleaf).
v.assignKp(kp)	Sets (v.id, v.pk, v.spk, v.parentHash, v.sig) from kp (only allowed if v.isleaf).
v.inuse()	Returns `false` iff all labels except parentHash are $\bot$.
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.leafIdx	The party's leaf index in τ.
γ.treeHash	A hash of (the public part) of τ.
γ.confTransHash	The confirmed transcript hash.
γ.interimTransHash	The interim transcript hash for the next epoch.
γ.ssk	The current signing key.
γ.certSpks[$*$]	A mapping associating the set of validated signature verification keys to each party 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.groupCtxt()	Returns (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash).

Table 5: Helper method on the protocol state.

4.4 Protocol Algorithms

The main (UC) protocol is depicted in Fig. 10. The helper functions are depicted in Figs. 11 to 13. 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 ($\texttt{Create}, \mathsf{spk}$).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. 12.

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. 13). 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 ($\texttt{Commit}, \vec{p}, \mathsf{spk}, \mathsf{force\text{-}rekey}$), 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. 11). 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 $\mathsf{force\text{-}rekey} = \texttt{true}$) 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. Appendix A) 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. 12). 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.

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 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 $\gamma.\mathsf{pendCom}$.

Protocol ITK

```
Input (Create, spk)
  req γ = ⊥ ∧ id = id_creator
  γ.groupId, γ.initSecret ←$ {0,1}^κ
  γ.epoch ← 0
  γ.interimTransHash ← ϵ
  γ.certSpks[∗], γ.pendUp[∗], γ.pendCom[∗] ← ⊥
  γ.τ ← new LBBT_1
  γ.leafIdx ← 0
  try ssk ← *fetch-ssk-if-nec(γ, spk)
  γ.ssk ← ssk
  (kp, sk) ←$ gen-kp(id, spk, ssk, ϵ)
  γ.τ.leaves[0].assignKp(kp)
  γ.τ.leaves[0].sk ← sk

Input (Propose, up-spk)
  req γ ≠ ⊥
  try ssk ← *fetch-ssk-if-nec(γ, spk)
  (kp, sk) ←$ gen-kp(id, spk, ssk, ϵ)
  P ← ('upd', kp)
  p ← *frameProp(γ, P)
  γ.pendUp[p] ← (ssk, sk)
  return p

Input (Propose, add-id_t)
  req γ ≠ ⊥ ∧ id_t ∉ γ.τ.roster()
  kp_t ← query (get-pk, id_t) to F_KS
  try γ ← *validate-kp(γ, kp_t, id_t, ϵ)
  P ← ('add', kp_t)
  p ← *frameProp(γ, P)
  return p

Input (Propose, rem-id_t)
  req γ ≠ ⊥ ∧ id_t ∈ γ.τ.roster()
  leafIdx_t ← γ.τ.leafof(it_t)
  P ← ('rem', leafIdx_t)
  p ← *frameProp(γ, P)
  return p

Input (Commit, p⃗, spk, force-rekey)
  req γ ≠ ⊥
  γ' ← *init-epoch(γ)
  try (γ', upd, rem, add) ← *apply-props(γ, γ', p⃗)
  req (∗, 'rem'-id) ∉ rem ∧ (id, ∗) ∉ upd
  if force-rekey ∨ p⃗ = () ∨ upd ≠ () ∨ rem ≠ () then
     try (γ', commitSecret, updatePath, pathSecrets)
                                   ← *rekey-path(γ', id, spk)
  else
     commitSecret ← 0
     updatePath ← ϵ
     pathSecrets[∗] ← ϵ
  propIDs ← ()
  for p ∈ p⃗ do
     propIDs +← Hash(p)
  C ← (propIDs, updatePath)
  sig ← *signCommit(γ, C)
  γ' ← *set-conf-trans-hash(γ, γ', γ.leafIdx, C, sig)
  (γ', confKey, joinerSecret) ← *derive-keys(γ, γ', commitSecret)
  confTag ← *conf-tag(γ', confKey)
  c ← *frameCommit(γ, C, confTag, sig)
  if add ≠ () then
     (γ', w) ← *welcome-msg(γ', add, joinerSecret,
                                   pathSecrets, confTag)
  else
     w ← ⊥
  γ' ← *set-interim-trans-hash(γ', confTag)
  γ.pendCom[c] ← (γ', p⃗, upd, rem, add)
  return (c, w)

Input (Process, c, p⃗)
  req γ ≠ ⊥
  (senderIdx, C, confTag, sig) ← *unframeCommit(γ, c, sig)
  id_c ← γ.τ.leaves[senderIdx].ID
  if (senderIdx = γ.leafIdx) then
     parse (γ', p⃗', upd, rem, add) ← γ.pendCom[c]
     req p⃗ = p⃗'
     return (id_c, upd ++ rem ++ add)
  parse (propIDs, updatePath) ← C
  for i ← 1, ..., |p⃗| do
     req Hash(p⃗[i]) = propIDs[i]
  γ' ← *init-epoch(γ)
  try (γ', upd, rem, add) ← *apply-props(γ, γ', p⃗)
  req (∗, id_c) ∉ rem ∧ (id_c, ∗) ∉ upd
  if (∗, 'rem'-id) ∈ rem then
     γ ← ⊥
  else
     if updatePath ≠ ϵ then
        (γ', commitSecret) ← *apply-rekey(γ', senderIdx, updatePath)
     else
        req p⃗ ≠ () ∧ upd = () ∧ rem = ()
        commitSecret ← 0
     γ' ← *set-conf-trans-hash(γ, γ', senderIdx, C, sig)
     (γ', ∗) ← *derive-keys(γ, confKey, γ', commitSecret)
     req *vrf-conf-tag(γ', confKey, confTag)
     γ' ← *set-interim-trans-hash(γ', confTag)
  return (id_c, upd ++ rem ++ add)

Input (Join, w)
  req γ = ⊥
  parse (encGroupSecrets, groupInfo) ← w
  γ.certSpks[∗], γ.pendUp[∗], γ.pendCom[∗] ← ⊥
  parse (groupInfoTBS, sig) ← groupInfo
  parse (γ.groupId, γ.epoch, γ.treeHash, γ.confTransHash,
            γ.interimTransHash, γ.τ, confTag, senderIdx) ← groupInfoTBS
  req Sig.vrf(γ.τ.leaves[senderIdx].spk, sig, groupInfoTBS)
  try γ ← *vrf-tree-state(γ)
  γ.leafIdx ← γ.τ.leafof(id)
  v ← γ.τ.leaves[γ.leafIdx]
  try γ.ssk ← *fetch-ssk-if-nec(γ, v.spk)
  kbs ← query get-sks to F_KS
  joinerSecret, pathSecret ← ⊥
  for e ∈ encGroupSecrets do
     parse (hash, cipher) ← e
     for (kp, sk) ∈ kbs do
        if hash = Hash(kp) then
           v.sk ← sk
           req v.kp() = kp
           parse (joinerSecret, pathSecret) ← PKE.dec(sk, cipher)
  req joinerSecret ≠ ⊥
  if pathSecret ≠ ϵ then
     v ← γ.τ.lca(γ.leafIdx, senderIdx)
     while v ≠ ⊥ do
        nodeSecret ← HKDF.Expand(pathSecret, 'node')
        (sk, v.sk) ← PKE.kg(nodeSecret)
        req v.sk = sk
        pathSecret ← HKDF.Expand(pathSecret, 'path')
        v ← v.parent
  (γ, confKey) ← *derive-epoch-keys(γ, joinerSecret)
  req *vrf-conf-tag(γ, confKey, confTag)
  return (γ.τ.roster(), γ.τ.leaves[senderIdx].id)

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

Fig. 10: 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(γ)
  γ' ← γ.clone()
  γ'.epoch ← γ'.epoch + 1
  γ'.pendUp[∗], γ'.pendCom[∗] ← ⊥
  return γ'

helper *rekey-path(γ', id, spk)
  directPath ← γ'.τ.directPath(γ'.leafIdx)
  coPath ← γ'.τ.coPath(γ'.leafIdx)
  updatePathNodes ← ()
  pathSecrets[∗] ← ⊥
  leafSecret ←$ {0,1}^κ
  leafNodeSecret ← HKDF.Expand(leafSecret, 'node')
  pathSecret ← HKDF.Expand(leafSecret, 'path')
  for (v, c) ∈ zip(directPath, coPath) do
    pathSecrets[v] ← pathSecret
    nodeSecret ← HKDF.Expand(pathSecret, 'node')
    (v.pk, v.sk) ← PKE.kg(nodeSecret)
    encPathSecrets ← ()
    for t ← c.resolution() do
      encPathSecrets +← PKE.enc(t.pk, pathSecret)
    updatePathNodes +← (v.pk, encPathSecrets)
    pathSecret ← HKDF.Expand(pathSecret, 'path')
  commitSecret ← pathSecret
  γ'.τ.mergeLeaves(γ'.leafIdx)
  γ' ← *set-parent-hash(γ', γ'.leafIdx)
  try ssk ← *fetch-ssk-if-nec(γ', spk)
  v ← γ'.τ.leaves[γ'.leafIdx]
  (kp, sk) ← gen-kp(id, spk, ssk, v.parentHash; leafNodeSecret)
  v.assignKp(kp)
  v.sk ← sk
  γ' ← *set-tree-hash(γ')
  updatePath ← (kp, updatePathNodes)
  return (γ', commitSecret, updatePath, pathSecrets)

helper *apply-rekey(γ', senderIdx, updatePath)
  parse (kp, updatePathNodes) ← updatePath
  directPath ← γ'.τ.directPath(senderIdx)
  coPath ← γ'.τ.coPath(senderIdx)
  lca ← γ'.τ.lca(γ'.leafIdx, senderIdx)
  for (v, c, updatePathNode)
            ∈ zip(directPath, coPath, updatePathNodes) do
    parse (v.pk, encPathSecrets) ← updatePathNode
    if v = lca then
      r ← c.resolvent(γ'.τ.leaves[γ'.leafIdx])
      i ← c.resolution().indexofr
      pathSecret ← PKE.dec(r.sk, encPathSecrets[i])
    if pathSecret ≠ ⊥ then
      nodeSecret ← HKDF.Expand(pathSecret, 'node')
      (pk, v.sk) ← PKE.kg(nodeSecret)
      req v.pk = pk
      pathSecret ← HKDF.Expand(pathSecret, )
  commitSecret ← pathSecret
  γ'.τ.mergeLeaves(senderIdx) γ' ←
  *set-parent-hash(γ', senderIdx)
  v ← γ'.τ.leaves[senderIdx]
  try γ' ← *validate-kp(γ', kp, v.id, v.parentHash)
  v.assignKp(kp)
  γ' ← *set-tree-hash(γ')
  return (γ', commitSecret)

helper *apply-props(γ, γ', p⃗)
  upd, rem, add ← ()
  for p ∈ p⃗ do
    try (senderIdx, P) ← *unframeProp(γ, p)
    id_s ← γ.τ.leaves[senderIdx].id
    parse (type, val) ← P
    if type = 'upd' then
      req (id_s, ∗) ∉ upd ∧ rem = () ∧ add = ()
      try γ' ← *validate-kp(γ', val, id_s, ϵ)
      γ'.τ.leaves[senderIdx].assignKp(val)
      γ'.τ.blankPath(senderIdx)
      if senderIdx = γ.leafIdx then
        parse (ssk, sk) ← γ.pendUp[p]
        γ'.τ.leaves[γ.leafIdx].sk ← sk
        γ'.ssk ← ssk
      spk ← γ'.τ.leaves[senderIdx].spk
      upd +← (id_s, 'upd'-spk)
    else if type = 'rem' then
      id_t ← γ.τ.leaves[val].id
      req val ≠ senderIdx ∧ γ'.τ.leaves[val] ≠ ⊥
              ∧ γ'.τ.leaves[val].inuse() ∧ (id_t, ∗) ∉ upd ∧ add = ()
      γ'.τ.leaves[val].blank()
      γ'.τ.leaves[val].blankPath(val)
      rem +← (id_s, 'rem'-id_t)
    else if type = 'add' then
      parse (id_t, ∗, spk, ∗, ∗) ← val
      req id_t ∉ γ'.τ.roster()
      try γ' ← *validate-kp(γ', val, id_t, ϵ)
      newIdx ← γ'.τ.addLeaf()
      γ'.τ.leaves[newIdx].assignKp(val)
      γ'.τ.unmergeLeaf(newIdx)
      add +← (id_s, 'add'-id_t-spk)
    else
      return ⊥
  return (γ', upd, rem, add)

helper *welcome-msg(γ, γ', add, joinerSecret, pathSecrets, confTag)
  groupInfoTBS ← (γ'.groupId, γ'.epoch, γ'.treeHash,
      γ'.confTransHash, γ'.interimTransHash,
      γ'.τ.public(), confTag, γ'.leafIdx)
  sig ← Sig.sign(γ'.ssk, groupInfoTBS)
  groupInfo ← (groupInfoTBS, sig)
  encGroupSecrets ← ()
  for (∗, 'add'-id_t-spk_t) ∈ add do
    leafIdx_t ← γ'.τ.leafof(id_t)
    v_t ← γ'.τ.leaves[leafIdx_t]
    lca ← γ'.τ.lca(γ'.leafIdx, leafIdx_t)
    encGroupSecret ← PKE.enc(v_t.pk, (joinerSecret, pathSecrets[lca]]))
    encGroupSecrets +← (Hash(v_t.kp), encGroupSecret)
  w ← (encGroupSecrets, groupInfo)
  return (γ', w)

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

Fig. 11: The helper methods related to creating and processing the commit and welcome messages.

Protocol ITK : Confirmation-Tag

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

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

Protocol ITK : Tree-Hash

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

helper *parent-hash(v)
  return Hash(v.pk, v.unmergedLvs, v.parentHash)

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

helper *tree-hash(v)
  if v.isleaf then
    return Hash(v.leafIdx, v.kp())
  else
    leftHash ← *tree-hash(v.lchild)
    rightHash ← *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(γ, γ', senderIdx, C, sig)
  commitContent ← (γ.groupId, γ.epoch, senderIdx, 'commit', C, sig)
  γ'.confTransHash ← Hash(γ.interimTransHash, commitContent)
  return γ'

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

Protocol ITK : Key-Schedule

```
helper *derive-keys(γ, γ', commitSecret)
  joinerSecret ← HKDF.Extract(γ.initSecret, commitSecret)
  (γ', confKey) ← *derive-epoch-keys(γ', joinerSecret)
  return (γ', confKey, joinerSecret)

helper *derive-epoch-keys(γ', joinerSecret)
  s ← HKDF.Expand(γ.joinerSecret, 'member')
  memberSecret ← HKDF.Extract(s, 0)
  e ← HKDF.Expand(memberSecret, 'epoch')
  epSecret ← HKDF.Extract(e, γ'.groupCtxt())
  confKey ← HKDF.Expand(epSecret, 'confirm')
  γ'.appSecret ← HKDF.Expand(epSecret, 'app')
  γ'.membKey ← HKDF.Expand(epSecret, 'membership')
  γ'.initSecret ← HKDF.Expand(epSecret, 'init')
  return (γ', confKey)
```

Protocol ITK : Setup Interaction

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

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

Fig. 12: Various helper methods for the protocol ITK.

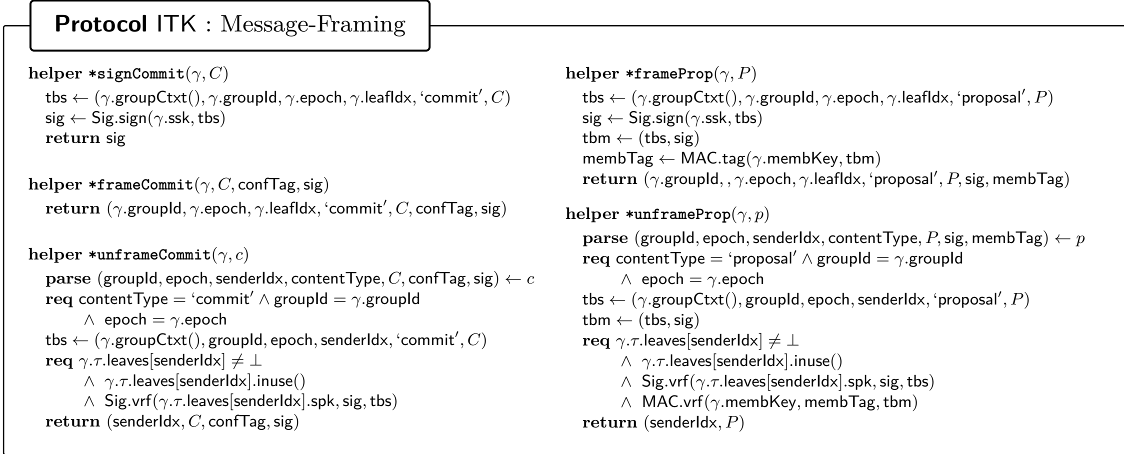

Fig. 13: The helper methods related to message framing.

Process. Consider an input $(\texttt{Process}, c, \vec{p})$. If the party created this commit message c (and the proposals match), then the protocol can simply retrieve the new epoch's state from $\gamma.\mathsf{pendCom}$. 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. 11). 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 $(\texttt{Join}, w)$, 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. To this end `*vrf-tree-state` (see Fig. 11) first validates the parent hashes using the following invariant:

> **Invariant:** For each non-blank internal node v, there exists exactly one non-blank child node, whose parent hash refers to v.

See Fig. 14 for an explanation of the tree-signing mechanism and the invariant in particular. 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.

4.5 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.

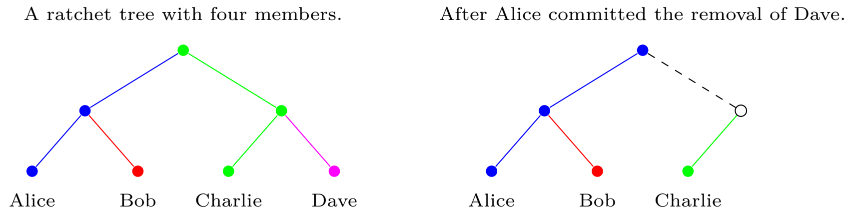

Fig. 14: An example of the tree-signing mechanism. In the left configuration, the keys in blue, red, green, and purple nodes have been sampled and certified by Alice, Bob, Charlie, and Dave, respectively. When Alice now commits the removal of Dave, first Dave's direct path gets blanked (including the root). Then Alice re-keys her direct path. Her leaf now contains a signature on the hash of the internal node shared between her and Bob, which in turn includes a hash of the new root, thus 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 is always bound by its child sampled by the same party, ultimately certified by the corresponding leaf node.

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

No membership MAC on commits. ITK uses an explicit MAC to ensure the authenticity of proposals. The MLS protocol also includes the MAC for commits (when using MLSPlaintext) [9]. For the security properties considered in this work this inclusion is redundant 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.)

5 Security of ITK

5.1 The Safety Predicate for ITK

The predicate, formally stated in Fig. 15, is defined using recursive deduction rules $\textbf{know}(c, \mathsf{id})$ and $\textbf{know}(c, \text{'epoch'})$, 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:

- $\textbf{know}(c, \mathsf{id})$ 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. 15). 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 $\textbf{know}(c, \mathsf{id})$ is implied by $\textbf{know}(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})$ (condition b)). If a child c' of c is not of this type, then it is implied by $\textbf{know}(c', \mathsf{id})$ (condition c)).

Predicate safe

Knowledge of parties' secrets.

$\textsf{know}(c, \textsf{id}) \iff$

a) // id's state leaks directly e.g. via corruption (see below):
$*\textsf{state-directly-leaks}(c, \textsf{id}) \lor$

b) // know state in the parent:
$(\textsf{Node}[c].\textsf{par} \neq \bot \land \neg *\textsf{secrets-replaced}(c, \textsf{id}) \land \textsf{know}(\textsf{Node}[c].\textsf{par}, \textsf{id})) \lor$

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

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

a) // id has been exposed in c:
$(\textsf{id}, *) \in \textsf{Node}[c].\textsf{exp} \lor$

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

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

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

a) // id is the sender of c and c was injected or generated with bad randomness
$(\textsf{Node}[c].\textsf{orig} = \textsf{id} \land \textsf{Node}[c].\textsf{stat} \neq \textsf{good}) \lor$

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

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

$*\textsf{secrets-replaced}(c, \textsf{id}) \iff \textsf{Node}[c].\textsf{orig} = \textsf{id} \lor \exists p \in \textsf{Node}[c].\textsf{pro} :$
$\textsf{Prop}[p].\textsf{act} \in \{\textsf{add-id-}*, \textsf{rem-id}\} \lor (\textsf{Prop}[p].\textsf{act} = \textsf{up-}* \land \textsf{Prop}[p].\textsf{orig} = \textsf{id})$

Knowledge of epoch secrets.

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

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

a) // orphan root with a corrupted signature public key:
$(\textsf{Node}[c].\textsf{par} = \bot \land (*, \textsf{spk}) \in \textsf{Node}[c].\textsf{mem} \land \textsf{spk} \in \textsf{Exposed}) \lor$

b) // commit to an add proposal that uses an exposed key package:
$(\exists p \in \textsf{Node}[c].\textsf{pro} : \textsf{Prop}[p].\textsf{act} = \textsf{add-id-spk} \land \textsf{spk} \in \textsf{Exposed}) \lor$

c) // secrets encrypted in the welcome message under an exposed leaf key
$*\textsf{leaf-welcome-key-reuse}(c) \lor$

d) // know necessary info to traverse the edge:
$(\textsf{know}(c, *) \land (c = \textsf{root}_* \lor \textsf{know}(\textsf{Node}[c].\textsf{par}, \text{'epoch'})))$

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

Safe and can-inject.

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

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

Fig. 15: The safety predicate for the CGKA functionality, defined in Fig. 3, reflecting the sub-optimal security of the ITK protocol.

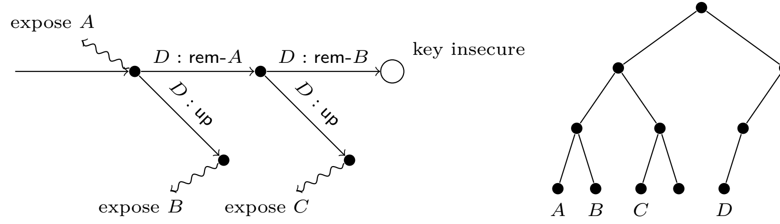

Fig. 16: An execution illustrating that many simultaneous corruptions leak information that cannot be deduced from any single corruption alone: the history graph (left) and the ratchet tree at the beginning (right). Exposing A reveals the initSecret. Exposing B reveals secrets on his direct path (untouched by D's update), which, together with the initSecret from A, allow to process D's commit removing A. In particular, this reveals the next initSecret, which, together with C's secrets, allows to process the commit removing B. Notice that states of A and B cannot be used to process the last commit, since afterwards they are not group members.

- **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 ¬**safe**(c) and **know**$(c, \text{'epoch'})$ is that the application secret is not leaked if id is exposed in c after outputting it.

Let us highlight some aspects of **safe**, which make security of ITK suboptimal:

- *Weak FS.* The fact that **know**(c, id) is implied by knowledge of id's secrets in a child of c is not inherent. For instance, RTreeKEM [1] improves this with almost no efficiency loss.
- *One encryption key per key package.* When adding a party id_t, ITK uses the same key pair for the leaf of id_t and to encrypt secrets in the welcome message to id_t. As a result, if id_t is exposed before it updates its leaf, the key can be used to decrypt epoch secrets from the welcome message, effectively removing the FS added by initSecret. This is reflected by condition c) of ***can-traverse**.
- *Weak protection against bad randomness.* In ITK, the effect of updating or committing with bad randomness is essentially the same as corrupting the sender. This is reflected in conditions a) and b) of ***secrets-injected**. This could however be (partly) mitigated by a so-called randomness pool, mixing the freshly sampled randomness with the secret state. Then, bad randomness would not compromise a previously secure state but just hamper PCS.
- *Signature schemes not resistant to bad randomness.* ITK supports ECDSA signatures, which leak the secret key in case randomness is reused. Accordingly, the signature key spk is marked by $\mathcal{F}_{\text{CGKA}}$ as corrupted each time an action that may use it is executed with bad randomness.

Remark 2. Previous works [2, 1] 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 simulatneous exposures may leak information that is not leaked by any of the exposures alone, as illustrated in Fig. 16.

5.2 Security Statement

In Section 6 we prove the following theorem.

Theorem 1. *Assuming that* PKE *is IND-CCA secure, and that* Sig *is EUF-CMA secure, the* ITK *protocol securely realizes* $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}}, \mathcal{F}_{\text{CGKA}})$*, where* $\mathcal{F}_{\text{CGKA}}$ *uses the predicate* ***safe*** *from Fig. 15, in the* $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where calls to* HKDF.Expand*,* HKDF.Extract *and* MAC *functions are replaced by calls to the global random oracle* $\mathcal{G}_{\text{RO}}$*.*

Remark 3 (Modeling HKDF *and* MAC *as Random Oracle).* Our proof uses techniques of [2], which require that HKDF.Expand, usually modeled as a PRF, is instead modeled as a (non-programmable but observable) random oracle. Modeling HKDF.Extract as a random oracle as well is necessary to extend the techniques of [2] to account for mixing the init secret into the key schedule. We note, however, that the standard security requirement for HKDF.Extract [25] is anyway not applicable to the way MLS uses it.[24]

Finally, we need non-standard assumptions on the MAC: extractability (a tag proves knowledge of the corresponding message and key) and collision-resistance (it is hard to produce two message-key pairs with the same tag). Both assumptions hold if MAC is modeled as a random oracle (and it is unlikely to achieve extractability under weaker assumptions).[25]

6 Proof of Theorem 1: Security of ITK

We introduce the simulator gradually, with a sequence of hybrids.

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 the protocol.

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

Hybrid 3. This change concerns encrypting welcome messages. Whenever a party id commits, and **safe** is true in the resulting commit node, the simulator $\mathcal{S}_3$ replaces the ciphertexts in the welcome message by encryptions of 0 and computes the initial state of the new members using the information that would be encrypted directly.

Hybrid 4. This change concerns the confidentiality of epoch secrets. $\mathcal{F}_{\text{CGKA}}$ uses the original **safe** predicate. The simulator $\mathcal{S}_4$ sets only those application secrets for which **safe** is false. In addition, $\mathcal{S}_4$ replaces the membership keys for nodes c with **safe**$(c) =$ `true` by random values, independent of the protocol.

Hybrid 5. This change concerns authenticity. $\mathcal{F}_{\text{CGKA}}$ uses the original **inj-allowed**. The simulator remains the same. This is the ideal world.

On a high level, to argue indistinguishability of Hybrids 1 and 2, we use the fact that the group context is included in all messages during framing. Indistinguishability of Hybrids 2 and 3 follows easily from IND-CCA security of PKE. (Note that as soon as a party with exposed key package is added, **safe** is false, and in particular the condition b) of ***can-traverse** is true), and for Hybrids 4 and 5 we use unforgeability of signatures and the MAC (note that the MAC key is already random in Hybrid 4). These proofs are rather standard and can be found in Section 6.3. The most challenging part of the proof is showing that indistinguishability of Hybrids 3 and 4 follows from IND-CCA of PKE, and this is what we focus on first, in Sections 6.1 and 6.2.

In general, our proof follows the strategy of [2], where the authors prove security of (Tainted) TreeKEM in the passive setting, assuming IND-CPA security of PKE. The major difficulty in the reduction is that a TreeKEM execution creates "encryption chains", where a public key is used to encrypt the next secret key, the last key is used to encrypt a challenge, and the adversary can adaptively choose which keys to corrupt. In order to tame the adaptivity, the authors of [2] use the intermediate notion of Generalized Selective Decryption (GSD), which is, roughly, the standard IND-CPA game, which additionally allows the adversary to create encryption chains. They prove that TreeKEM security is implied by (slightly modified) GSD security of PKE, and then that GSD

[24] The standard notion is that of a computational extractor. This does not imply security of MLS, which requires that the extractor's output looks random even if the salt (used for the init secret) is secret and the input (used for the commit secret) is fixed. See also Appendix A.

[25] Extractability is needed for the confirmation tag to prove knowledge of the key schedule in the next epoch. Collision resistance is needed so that the adversary cannot produce a proposal accepted by parties in 2 different epochs (recall that a proposal includes a MAC over the transcript hash, which uniquely identifies an epoch, but not the transcript hash itself).

is implied by IND-CPA, where the latter reduction is in the (non-programmable but observable) ROM.[26]

In the following, we build on their proof to reduce distinguishing of Hybrids 3 and 4. In particular, in Section 6.1 we introduce a GSD game modified to fit ITK executions and adapt the proof of [2] to show that our modified GSD security is implied by IND-CCA. Then, in Section 6.2 we show that GSD security implies indistinguishability of the hybrids (the reduction uses a lemma implied by unforgeability of the signature scheme used to sign parentHash in leaves).

6.1 Modified GSD Security

This section first explains the Generalized Selective Decryption (GSD) security notion for public-key encryption, modified to include additional capabilities of the adversary, given to it in ITK executions.[27] Then, we it proves that the modified GSD security is implied by IND-CCA security in (observable) ROM, by extending the proof of [2].

GSD security is formalized by the game defined in Fig. 17. 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. 18.

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 ¬**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 3. *Let* $\mathrm{Adv}^{\mathrm{GSD}}_{\mathsf{PKE},\mathcal{A}} \coloneqq 2\Pr[\mathrm{GSD}_{\mathsf{PKE},\mathcal{A}} = \mathtt{true}] - 1$ *denote the advantage of* $\mathcal{A}$ *against the game defined in Fig. 17. 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*.*

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

Proof. The proof is adapted from [2]. 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. 17

[26] Proving GSD security turns out very challenging and no polynomial reduction in the standard model is known. Before [2], no efficient reduction was known, even in ROM.

[27] [2] already modifies the standard GSD notion, allowing for keys to be derived via hash chains, like in an ITK commit. We additionally allow the hash to take two inputs, like in computing the joinerSecret, and add a decryption oracle.

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 (a) (V,E) acyclic and
(b) u^* sink and
(c) $\neg\mathbf{gsd\text{-}exp}(u^*)$
then return $b = b'$
else return `false`

Oracle Chal(u)

req $u^* = \bot$
$u^* \leftarrow u$
if $b = 0$ **then return** s_u
else return s'

Oracle Hash(u, v, lbl)

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

Oracle Enc(u, v)

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

Oracle Dec(u, c)

req $s_u \neq \bot \wedge u$ not a sink
// keys must not be used for sinks (possible challenges)
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 $u \in \mathsf{Corr}$
$\vee\ \exists (v, u, *) \in E : \mathbf{gsd\text{-}exp}(v)$
$\vee\ \exists ((v, v'), u, *) \in E : \mathbf{gsd\text{-}exp}(v) \wedge \mathbf{gsd\text{-}exp}(v')$

Fig. 17: The GSD game, modified to explain ITK executions.

without the Hash, Join-Hash and Dec oracles. This proof solves the main technical challenges, and we refer the reader to [2] for the details. Then, [2] 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 [2] 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.

Decryption. While [2] 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 [2]. 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, [2] 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 [2] 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 [2] 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 $\mathcal{A}_{\mathsf{gsd}}^{E}$ that triggers E, [2] defines $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ that does not trigger E roughly as follows. $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ simulates the experiment for $\mathcal{A}_{\mathsf{gsd}}^{E}$ using its oracles, and halts as soon as E turns true (it can realize that E is true by checking each RO query of $\mathcal{A}_{\mathsf{gsd}}^{E}$ 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 $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$'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 $\mathcal{A}_{\mathsf{gsd}}^{E}$'s RO queries. If s^* is the real seed (and the guess for v^* is correct), then it is queried to the RO and $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ outputs 0. Else, if s^* is random, then it is independent of $\mathcal{A}_{\mathsf{gsd}}^{E}$'s view, so with high probability it is not queried and, accordingly, $\mathcal{A}_{\mathsf{gsd}}^{\neg E}$ outputs 1 (when E is triggered for a different node, or $\mathcal{A}_{\mathsf{gsd}}^{E}$ 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 $\mathcal{A}_{\mathsf{gsd}}^{E}$'s game), such query would trigger E. □

6.2 Indistinguishability of Hybrids 3 And 4

To prove inditinguishability, we define two sequences of hybrids: $\mathcal{H}_i^{\mathsf{appSecret}}$ and $\mathcal{H}_i^{\mathsf{membKey}}$ for $i \in [N]$. $\mathcal{H}_i^{\mathsf{appSecret}}$ is the same as Hybrid 3, except the first i application secrets chosen by $\mathcal{F}_{\text{CGKA}}$ are sampled as in Hybrid 4, i.e. they are random if **safe** is true. $\mathcal{H}_i^{\mathsf{membKey}}$ is the same as $\mathcal{H}_N^{\mathsf{appSecret}}$, except the first i membership keys used by the simulator are as in Hybrid 4, i.e. they are random if **safe** is true. In the following, we show that $\mathcal{H}_{i-1}^{\mathsf{appSecret}}$ and $\mathcal{H}_i^{\mathsf{appSecret}}$ are indistinguishable. The proof for $\mathcal{H}_i^{\mathsf{membKey}}$ is analogous.

Assume that an environment $\mathcal{Z}$ has a non-negligible advantage in distinguishing between hybrids $\mathcal{H}_{i-1}^{\mathsf{appSecret}}$ and $\mathcal{H}_i^{\mathsf{appSecret}}$, 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 $\mathcal{F}_{\text{CGKA}}$, $\mathcal{F}_{\text{KS}}^{\text{IW}}$, $\mathcal{F}_{\text{AS}}^{\text{IW}}$ and the simulator. In particular, $\mathcal{A}$ executes the code of all these functionalities and the simulator as defined in $\mathcal{H}_{i-1}^{\mathsf{appSecret}}$, 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 $\mathsf{memberSecret}$ and $\mathsf{appSecret}$. 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 $\text{Enc}(u, 0)$ (the special vertex 0 is only used here).

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 $\mathcal{F}_{\text{KS}}^{\text{IW}}$ and $\mathcal{F}_{\text{AS}}^{\text{IW}}$, 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 $\mathcal{F}_{\text{KS}}^{\text{IW}}$, $\mathcal{A}$ 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 $\mathsf{add\text{-}id}_t$ from id. Recall that whenever the protocol requests a key package for id_t from $\mathcal{F}_{\text{KS}}^{\text{IW}}$, $\mathcal{Z}$ 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. 18 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 $u_{\mathsf{ctr}}, u_{\mathsf{ctr}}{+}1, \ldots$ and u_{ctr} is incremented). The vertices are created as follows: for each $i \in [n-1]$, query $\text{Hash}(u_i, u_{i+1}, \mathsf{path})$. Then,

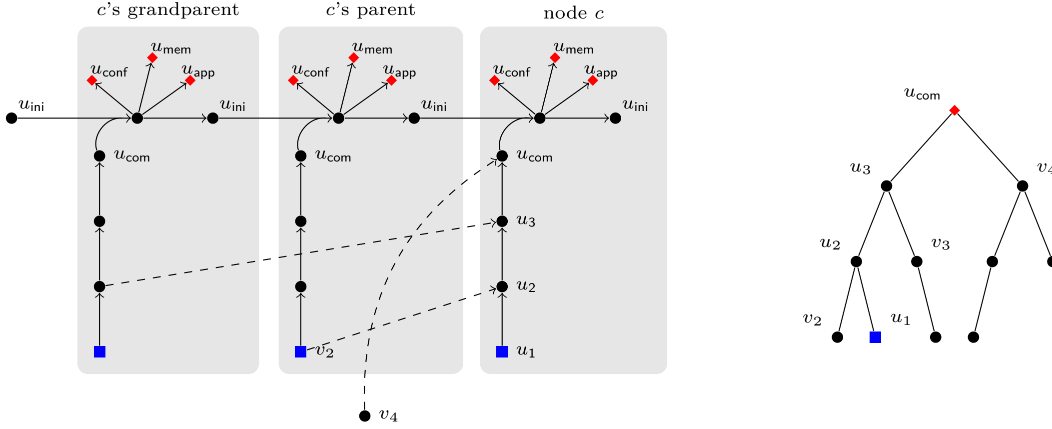

Fig. 18: An example commit creating a node c: (right) the ratchet tree in c and its parent, (left) the corresponding GSD graph created by the reduction $\mathcal{A}$. The sinks and sources are marked by ◆ and ■, respectively. The continuous and dashed edges denote hash and encryption edges, respectively. 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 joinerSecret obtained by combining commitSecret and previous 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).

query Join-Hash($u_{\mathsf{par\text{-}ini}}, u_{\mathsf{n}}, .$), where $(\mathsf{gsd}, u_{\mathsf{par\text{-}ini}})$ is stored in the initSecret of Ptr[id]. For $\mathsf{lbl} \in \mathsf{app}, \mathsf{mem}, \mathsf{ini}, \mathsf{conf}$, query 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 memberSecret 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:* If for some new member id_t, the leaf key secret is not of the form $(\mathsf{gsd}, *)$, $\mathcal{A}$ corrupts the vertex u_{joi} and uses the revealed joinerSecret to compute the ciphertexts. Else, it outputs an encryption of 0. (Note that if **safe** is true for the new commit node, then in Hybrid 3 the welcome message is replaced by an encryption of 0.)

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.

Key in node c. $\mathcal{A}$ 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.

Expose id. 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_{app})) is false, and hence $\mathcal{A}$ can win the game by outputting whatever $\mathcal{Z}$ outputs.

Recalling the definition of **gsd-exp** (Fig. 17) and the GSD graph created by $\mathcal{A}$ (Fig. 18), **gsd-exp**((c, u_{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 $\neg\mathbf{safe}(c)$.
(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. 18) 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**$((c, u))$,
     (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. 19).
   - 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.[28] 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.

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

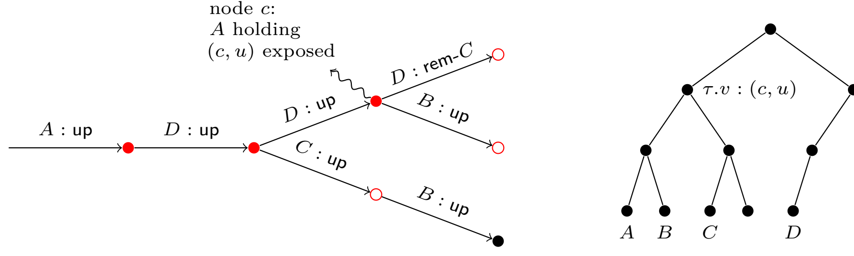

Fig. 19: An illustration for the proof that **gsd-exp**$((c,u))$ implies **know**$(c,*)$: the history graph (left) and the ratchet tree in the exposed node c (right). The history graph subtree G is marked by • and the leafs are marked by ○. In the root of G, the committer A inserts (c,u) as one of the 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.

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

Injected proposals. $\mathcal{A}$ 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 $\perp$.

Applying proposals. 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.

Commit from id. $\mathcal{A}$ 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 $\perp$.

Commits injected to process. 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 joinerSecret, which in turn is computed as joinerSecret = Hash(initSecret, commitSecret, .), modeling HKDF as a RO. Moreover, observe that commitSecret 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 joinerSecret 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 (confKey, confTransHash), and $\mathcal{A}$ can extract confKey. Analogously, as confKey is

derived by hashing $\mathsf{joinerSecret}$ with an appropriate label, it can extract $\mathsf{joinerSecret}$ (of c) from the queries as well.

3. Now consider two cases. First, if $\mathsf{initSecret}$ is the same in c and c', then $\mathsf{joinerSecret} = \mathsf{Hash}(\mathsf{initSecret}, \mathsf{commitSecret}, .)$ is the same in c and c' (by $\mathsf{commitSecret}$ being the same).
Second, if $\mathsf{initSecret}$ is different, then $\mathsf{joinerSecret}$ in c is the hash of an honestly generated $\mathsf{commitSecret}$ with a different $\mathsf{initSecret}$. Since the protocol only uses $\mathsf{commitSecret}$ once with the correct $\mathsf{initSecret}$, the only way for $\mathcal{Z}$ to compute the joiner is to query the RO, and $\mathsf{commitSecret}$ can be extracted from the queries. Now $\mathcal{A}$ corrupts $\mathsf{initSecret}$ in c' and combines it with $\mathsf{commitSecret}$ to compute $\mathsf{joinerSecret}$. Note that this does not affect u^* being a valid challenge, since the node corresponding to $\mathsf{commitSecret}$ (of c') is not exposed.
In either case, $\mathcal{A}$ can now compute $\mathsf{appSecret}$ 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 $w = (\mathsf{encGroupSecrets}, \mathsf{groupInfo})$, $\mathcal{A}$ does as follows.

1. *Join to an existing node.* If there exists a node c with $\mathsf{confTransHash}$ matching that in $\mathsf{groupInfo}$, $\mathcal{A}$ searches for a key package kp such that $\mathsf{SK}[\mathsf{id}, \mathsf{kp}] \neq \bot$ and $\mathsf{Hash}(\mathsf{kp})$ matches an entry $e \in \mathsf{encGroupSecrets}$.
If e is copied from a welcome message generated by $\mathcal{A}$ while creating a commit node c and $\mathsf{SK}[\mathsf{id}, \mathsf{kp}]$ is a GSD node, $\mathcal{A}$ moves id to c. Else, it uses either the secret in $\mathsf{SK}[\mathsf{id}, \mathsf{kp}]$ 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 $\mathsf{groupInfo}$ and the ratchet tree set to the public part of τ from $\mathsf{groupInfo}$. 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 $\bot$).
3. *Join a new node: decrypt the secrets.* $\mathcal{A}$ searches for a key package kp such that $\mathsf{SK}[\mathsf{id}, \mathsf{kp}] \neq \bot$ and $\mathsf{Hash}(\mathsf{kp})$ matches an entry $e \in \mathsf{encGroupSecrets}$ 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 $\mathsf{SK}[\mathsf{id}, \mathsf{kp}]$ is a GSD node. To this end, observe that $\mathcal{A}$ could then use $\mathsf{confTag}$ from w to compute $\mathsf{appSecret}$ in c and win the GSD game as follows. Recall that $\mathsf{confTag} = \mathsf{Hash}(\mathsf{confKey}, \mathsf{confTransHash})$, where $\mathsf{confKey}$ is derived from $\mathsf{joinerSecret}$ id decrypts and $\mathsf{confTransHash}$ is taken from w. Since e is copied, $\mathsf{joinerSecret}$ used (implicitly) for the tag is the same as in c. On the other hand, $\mathsf{confTransHash}$ in w and c differ (else, id would have joined to c). Hence, $\mathsf{joinerSecret}$ inn c can be extracted from $\mathcal{Z}$'s RO queries and used to compute $\mathsf{appSecret}$ in c.
Otherwise, $\mathcal{A}$ can obtains the encrypted $\mathsf{joinerSecret}$ and $\mathsf{pathSecret}$ using the stored secret or the Dec oracle. It updates ratchet tree secrets to those derived from $\mathsf{pathSecret}$ (if any secret key was set to a GSD node, $\mathcal{A}$ uses $\mathsf{pathSecret}$ to win the game), and computes the epoch secrets from $\mathsf{joinerSecret}$.

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 $\textbf{gsd-exp}((c, u_{\mathsf{app}}))$ 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 $\textbf{gsd-exp}((c, u_{\mathsf{com}}))$ implies $\mathbf{know}(c, *)$. We modify the proof of (c) as follows.

– $\textbf{gsd-exp}((c, u_i))$ 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 $\textbf{gsd-exp}((c, u))$,
  (c) The secret in $\tau.v$ is $\bot$.

  We show that all cases imply $\mathbf{know}(c, *)$.

– *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 $\bot$ 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 2 cases correspond exactly to 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), and in particular condition 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.

<u>PROPOSAL FROM id.</u> $\mathcal{A}$ 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 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.

<u>COMMIT FROM id.</u> 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 commitSecret, 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 joinerSecret, which is a hash of the current initSecret and the freshly computed commitSecret: If initSecret stores a GSD node u, $\mathcal{A}$ queries Hash with input $(u, u_{\mathsf{ctr}}, \mathsf{commitSecret})$, corrupts u_{ctr}, sets joinerSecret to the result and increments u_{ctr}. Else, if initSecret stores a value, $\mathcal{A}$ computes joinerSecret itself.
3. Using joinerSecret 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 joinerSecret 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, commitSecret corresponds to a GSD node with **gsd-exp** false, and hence **gsd-exp** is false for the joinerSecret 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.

6.3 The Rest of the Hybrids

Claim. Hybrids 1 and 2 are indistinguishable.

<u>Proof</u>: 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 $(\texttt{Propose}, \mathsf{id}, \mathsf{add})$ 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 $\mathcal{F}_{\text{CGKA}}$ $ack = \texttt{false}$. Else, it sends $(p, \mathsf{spk}_t, \texttt{true})$, 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 membTag — a MAC, modeled as a random oracle,[29] over groupCtxt, which includes confTransHash (c.f. framing in Fig. 13).[30]

COMMITS. $\mathcal{S}_2$ computes the packets c and w according to ITK and sets $ack = \texttt{false}$ if $c = \perp$. If $ack = \texttt{true}$, it first checks if c corresponds to a detached root — if $\mathsf{Node}[c] = \perp$ and there exists a w such that $\mathsf{Wel}[w] = \mathsf{root}_{rt}$ and confTransHash in w (included as a part of 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).

PROCESS. $\mathcal{S}_2$ executes the protocol to check if the receiver would accept the inputs and sends $ack = \texttt{false}$ 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 $\mathsf{Node}[c] = \perp$, $\mathcal{S}_2$ retrieves orig' and spk' from c (the latter can be found in the committer's key package in the 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 $(\texttt{Process}, c, \vec{p})$. 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.

JOIN. To process an injected welcome message $w = (\mathsf{encGroupSecrets}, \mathsf{groupInfo})$ (non-injected messages are handled by $\mathcal{F}_{\text{CGKA}}$ without interaction with $\mathcal{S}_2$), $\mathcal{S}_2$ searches for a node c' with confTransHash matching that in groupInfo. If such node exists, it sends c' to $\mathcal{F}_{\text{CGKA}}$. Else, it sends to $\mathcal{F}_{\text{CGKA}}$ the sender orig' and the group members mem' retrieved from groupInfo (and $\mathcal{F}_{\text{CGKA}}$ creates a new detached root). ■

Claim. Assuming PKE is IND-CCA secure, Hybrids 2 and 3 are indistinguishable.

Proof: We use the standard hybrid argument, where in the i-th hybrid, the first i welcome messages generated for commits with **safe** true, are real, and the rest are random. The (multi-user) IND-CCA adversary $\mathcal{A}$, given an environment $\mathcal{Z}$ distinguishing hybrids $i-1$ and i, embeds its challenge in the i-th welcome message and generates the rest of the welcome messages itself. It uses the decryption oracle to deal with injected welcome messages.

[29] 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).

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

Finally, due to the restriction on $\mathcal{Z}$, $\mathcal{A}$ never has to reveal the secrets encrypted in its welcome message, or the secret key that can be used to decrypt it. The reason is that $\mathcal{A}$ would only have to do this if a party id_t added to a node c with **safe** true, and then corrupted while holding the decryption key for the welcome message. The latter is true iff id_t has not processed its welcome message yet, or if it has not updated its leaf key after joining. The former is disallowed by the "for each" loop in `Expose`, and the latter — by the ***leaf-welcome-key-reuse** condition in **safe**. ■

Claim. Assuming Sig and MAC are EUF-CMA secure, Hybrids 4 and 5 are indistinguishable.

Proof: Since the memberSecret is replaced by an independent random string in Hybrid 4, this proof is straightforward. ■

7 Different Tree-Signing Methods

In this section, we first explain why the tree-signing mechanism used by the ITK protocol provides unexpectedly weak security, and then show that the expected level of security is achieved by the protocol ITK*, which uses an alternative tree-signing mechanism. The tree signing of ITK* was discussed once on the MLS mailing list [30], but it was rejected for worse deniability (we note that deniability is outside the scope of this work).

7.1 Tree Signing of ITK is Suboptimal

Recall that tree signing of ITK works by including in each leaf a signature on the leaf's parentHash. This way, each (honest) committer, say Alice, certifies that she (honestly) generated a number of keys above her leaf. Intuitively, the goal is to allow a party, say Dave, invited to a (fake) group by a malicious insider, say Bob, to meaningfully remove Bob. Unfortunately, this is not necessarily the case, as illustrated by the attack in Fig. 20. Intuitively, the problem is that Alice's signature certifies that the key pairs were (honestly) generated by her, but not that the set of parties who know the secret keys (for Invariant (2) of ITK) matches the one in Alice's ratchet tree. This allows Bob to create a fake 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 a refresh of every key he knows.

7.2 Alternative Tree-Signing: ITK*

In essence, ITK* fixes the problem of ITK by signing not only public keys on the direct path, but also the identities of all parties the secret keys were revealed to.[31] This is achieved by, first, storing in each node the treeHash of its subtree. Then, the order of computing tree and parent hashes is reversed and treeHash is included in the (signed) parentHash.

More specifically, the following modifications are made compared to ITK. First, ITK* computes treeHash from the bottom up followed by the parentHash from the top back down again. For ratchet tree $\gamma.\tau$ and node $v \in \gamma.\tau$ we write $v.\mathsf{data} := (v.\mathsf{nodeIdx}, v.\mathsf{pk}, v.\mathsf{unmergedLvs})$ to denote a (unique prefix free encoding of) all public labels of node v except for the parentHash label and, at a leaf, the signature. We recursively define parentHash and treeHash as follows:

if $v.\mathsf{isleaf}$ **then**
  $v.\mathsf{treeHash} \leftarrow \mathsf{Hash}(v.\mathsf{data})$
else
  $v.\mathsf{treeHash} \leftarrow \mathsf{Hash}(v.\mathsf{data}, v.\mathsf{lchild}.\mathsf{treeHash}, v.\mathsf{rchild}.\mathsf{treeHash})$

[31] One reason this hurts deniability is because MLS allows that the public part of the ratchet tree is communicated to joining members not as part of the welcome message, but via an untrusted server (this increases efficiency in certain scenarios, e.g. it decreases the amount of data sent by the committer). Signing parties' identities (with undeniable signing keys) means that now they can no longer deny having being part of some group together with that party.

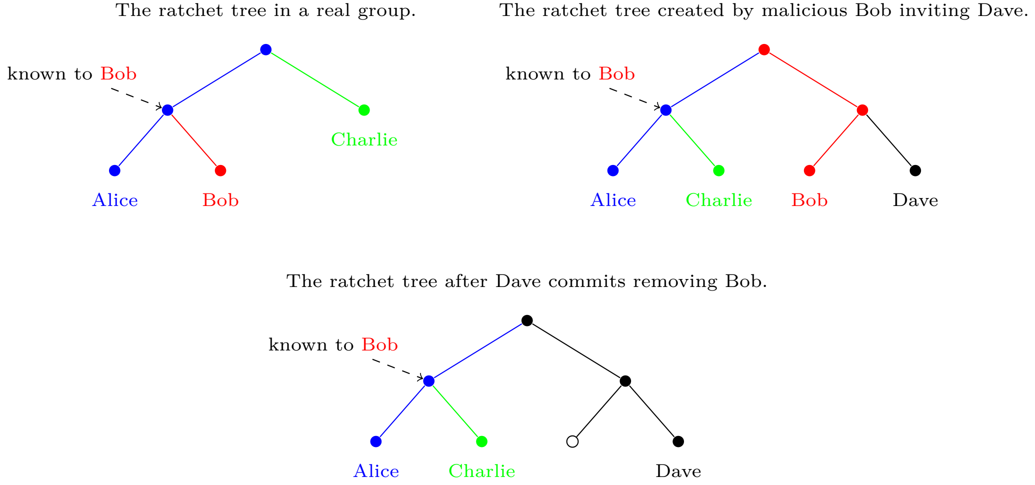

Fig. 20: An attack showing that tree signing of ITK is suboptimal. Alice and Charlie are honest, and Bob is a malicious insider. The keys in blue, red and green nodes are certified by Alice's, Bob's and Charlie's signatures, respectively. Bob copies two lowest nodes on Alice's direct path and Charlie's leaf, and builds the ratchet tree for a fake group, where he invites Dave (note that the copied signatures are still valid for sub-paths). After Dave removes Bob, the group contains only honest parties. However, the new application secret is known to Bob, because, due to the invariant (2) of ITK in the real group, he knows some secret key generated by Alice.

if v.isroot **then**
  v.parentHash $\leftarrow 0$
else
  v.parentHash $\leftarrow$ Hash(v.parent.treeHash, v.parent.parentHash)

That is, to avoid circular dependencies the parent hash is no longer included in the tree hash.

Second, observe that to implement this in ITK*, we need to tweak the gen-kp abstraction for `*rekey-path` a bit, separating the sampling of the keys from the signing process: The signature includes the updated parent hash, which depends on the updated tree hashes, which in turn depend on the new public key kp. In short, we first have to sample the new key pair, then invoke `*set-tree-hash` and `*set-parent-hash`, and finally prepare and sign the key package.

7.3 Security of ITK*

ITK* provides the security guarantees one would intuitively expect from tree signing — removing all parties using corrupt signing keys results in a secure epoch, even if the group is fake. Formally, we show the following theorem.

Theorem 3. *Let $\mathcal{F}^*_{\text{CGKA}}$ be analogous to $\mathcal{F}_{\text{CGKA}}$ but with an improved safe predicate* **safe***, *where the condition b) of* ***state-directly-leaks** *(cf. Fig. 15) is replaced by*

b*) $\exists c_a : \text{*ancestor}(c_a, c) \land \text{Node}[c_a].\text{par} = \bot \land (\text{id}, \text{spk}) \in \text{Node}[c].\text{mem} \land \text{spk} \in \text{Exposed}$.

Assuming that PKE *is IND-CCA secure, and that* Sig *is EUF-CMA secure, the* ITK* *protocol then 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 calls to* HKDF.Expand, HKDF.Extract *and* MAC *functions are replaced by calls to the global random oracle $\mathcal{G}_{\text{RO}}$.*

Recall that **safe*** returns false, i.e., epoch c is not secure, whenever ***state-directly-leaks** returns true. Intuitively, the new condition in can be understood of as follows. The first two clauses ensure that c is the result of an adversarially created session. The last two clauses check whether c contains any exposed keys. While we could omit the first two clauses to still obtain a correct theorem statement it would also be a needlessly weak one. That is because for honestly generated sessions we are able to show that, under the right conditions, even an epoch c containing exposed keys can still be secure. Thus we handle such epochs using other conditions in the safe predicate.

Proof. Notice that the only difference between **safe** and the new **safe*** concerns how detached history-graph nodes are handled so it suffices to consider these. Specifically, for detached node c **safe*** only guarantees that the epoch doesn't include an exposed signing key spk. We argue 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 in the epoch's ratchet tree τ contains a leaked key pk. Suppose, for the sake of contradiction, that node $\tau.v$ does contain a leaked key pk. 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. As that contradicts **safe*** being true we take v to be an internal node instead.

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 **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 no 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 each public key at an internal node of τ must be included in parent hash value that is then signed (along with other data) at some leaf in τ. Let id_s denote the party whose leaf in $\tau.v$ includes a signature authenticating pk. (We can assume that id_s is in the subtree rooted at $\tau.v$ as otherwise no party would have accepted the commit and or welcome message leading to this epoch.)

The signature in id_s's leaf includes not just pk but also the complete tree hash at $\tau.v$. Once again, we can assume that this tree hash matches the data in the ratchet tree for that subtree since otherwise no party would ever reach epoch c. By assumption id_s's signature public key spk_s is not exposed so pk was generated honestly (and with good randomness) by id_s as part of some commit message c_s (possibly for a different session). However (due the unforgeability of the signature scheme) we can conclude that the ratchet tree for epoch c_s must have contained an identical subtree as the subtree of c rooted at $\tau.v$ (except for the parent hash values and signatures the leaves).

Now in the proof of Theorem 1, we already argued that for pk to have been leaked it must be that the subtree in epoch c_s rooted at the node with pk contained a party id_e with an exposed spk_e. But since the two subtrees are the same, this implies that τ in epoch c also contains an exposed key which is a contradiction to **safe*** returning true.

□

References

[1] 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.

[2] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. Cryptology ePrint Archive, Report 2019/1489, 2019. https://eprint.iacr.org/2019/1489.

[3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging. Private communication, 2020.

[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.

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

[5] 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.

[6] R. Barnes, B. Beurdouche, , J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol (draft-ietf-mls-protocol-latest). Technical report, IETF, Oct 2020. https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html.

[7] 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.

[8] Richard Barnes. Subject: Re: [MLS] Include signature in the confirmed transcript hash? MLS Mailing List, 17 September 2020 22:22UTC. https://mailarchive.ietf.org/arch/msg/mls/w0XK93yXZ_dFzApbxroKJggxmZk/.

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

[10] 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.

[11] Richard Barnes. Subject: Re: [MLS] MLSPlaintext packets aren't authenticated using symmetric key schedule. MLS Mailing List, 18 August 2020 15:26UTC. https://mailarchive.ietf.org/arch/msg/mls/MO7syaR7pS_z-dcXTNoiN73WiQ8/.

[12] 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/.

[13] 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.

[14] 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.

[15] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. Cryptology ePrint Archive, Report 2020/1171, 2020. https://eprint.iacr.org/2020/1171.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178. IEEE Computer Society, 2016.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Ralf Küsters and Max Tuengerthal. Composition theorems without pre-established session identifiers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 41–50. ACM Press, October 2011.

[27] E. Omara, B. Beurdouche, E. Rescorla, S. Inguva, A. Kwon, and A. Duric. The messaging layer security (mls) architecture (draft-ietf-mls-architecture-05). Technical report, IETF, Jul 2020. `https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/`.

[28] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, pages 51–59. ACM, 1991.

[29] 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`.

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

[31] 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: Cryptographic Primitives

We introduce the basic cryptographic primitives used throughout this work.

Signature Scheme. A signature scheme is a tuple of PPT algorithms $\mathsf{Sig} := (\mathsf{Sig.kg}, \mathsf{Sig.sign}, \mathsf{Sig.vrf})$. 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 $\mathsf{PKE} := (\mathsf{PKE.kg}, \mathsf{PKE.enc}, \mathsf{PKE.dec})$. 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 $\mathsf{MAC} := (\mathsf{MAC.tag}, \mathsf{MAC.vrf})$. 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 [23].

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 [25].

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.