On The Insider Security of MLS

Joël Alwen[1], Daniel Jost[2]*[0000−0002−6562−9665], and Marta Mularczyk[1]**

[1] AWS Wickr, New York, USA
{alwenjo,mulmarta}@amazon.com
[2] New York University, New York, USA
daniel.jost@cs.nyu.edu

Abstract. The *Messaging Layer Security* (MLS) protocol is an open standard for end-to-end (E2E) secure group messaging being developed by the IETF, poised for deployment to consumers, industry, and government. It is designed to provide E2E privacy and authenticity for messages in long-lived sessions whenever possible, despite the participation (at times) of malicious insiders that can adaptively interact with the PKI at will, actively deviate from the protocol, leak honest parties' states, and fully control the network. The core of the MLS protocol (from which it inherits essentially all of its efficiency and security properties) is a *Continuous Group Key Agreement* (CGKA) protocol. It provides asynchronous E2E *group management* by allowing group members to agree on a fresh independent symmetric key after every change to the group's state (e.g. when someone joins/leaves the group).

In this work, we make progress towards a precise understanding of the insider security of MLS (Draft 12). On the theory side, we overcome several subtleties to formulate the first notion of insider security for CGKA (or group messaging). Next, we isolate the core components of MLS to obtain a CGKA protocol we dub *Insider Secure TreeKEM* (ITK). Finally, we give a rigorous security proof for ITK. In particular, this work also initiates the study of insider secure CGKA and group messaging protocols. Along the way we give three new (very practical) attacks on MLS and corresponding fixes. (Those fixes have now been included into the standard.) We also describe a second attack against MLS-like CGKA protocols proven secure under all previously considered security notions (including those designed specifically to analyze MLS). These attacks highlight the pitfalls in simplifying security notions even in the name of tractability.

1 Introduction

1.1 Background and Motivation

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

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

** Research supported by the Zurich Information Security and Privacy Center (ZISC). Work partially done while at ETH Zurich, Switzerland.

CGKA protocols are designed to be truly practical even when used over an adversarial network by large groups of uncoordinated parties with little, if any, common points of trust.

CGKA protocols should be end-to-end (E2E) secure and use *asynchronous* communication (in contrast to older, highly interactive, Dynamic Group Key Agreement protocols). That is, no assumptions are made about when or for how long parties are online. Instead, an (untrusted) network is expected only to buffer packets for each party until they come online again. As a consequence, all actions a party might wish to take must be performed non-interactively. Moreover, protocols cannot rely on specially designated parties (like the group managers in broadcast encryption). To achieve E2E security, protocols shouldn't rely on trusted third parties including the PKI that distributes long and short term public keys.[3]

Intuitively, CGKA protocols encapsulate the cryptographic core necessary to build higher-level distributed E2E secure group applications like secure messaging (not unlike how Key Encapsulation captures the core of Public Key Encryption). Any change to a group's state (e.g. parties joining/leaving) initiates a new *epoch* in a CGKA session. Each epoch E is equipped with its own uniform and independent epoch key k_E, called the *application secret* of E, which can be derived by all group members in E. The term "application secret" reflects the expectation that k_E will be used by a higher-level cryptographic application during E.[4] For example, k_E might seed a key schedule to derive (epoch specific) symmetric keys and nonces, allowing group members in E to use authenticated encryption for exchanging private and authenticated messages during E.

The Messaging Layer Security Protocol. Probably the most important family of CGKA protocols today is TreeKEM. An initial version was introduced in [34]. It was soon followed by a more precise description in [18] and the improved version [14]. Another major revision came with the introduction of the "propose-and-commit" paradigm [17]. The product of this evolution (implicitly) makes up most of the cryptographic core of the latest draft (Draft 12) of the *Messaging Layer Security* (MLS) protocol [13]. It is this most recent version which is the main focus of this work.

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

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

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

Insider Security. Intuitively, MLS is designed to provide security whenever possible in the face of a weak PKI and despite potential participation by malicious insiders with very powerful adaptive capabilities. These include full control of the network and repeatedly leaking the local states of honest users and even choosing their random coins[5]. However, thus far it has remained open how to formally capture (let alone analyze) such a security notion for CGKA/group messaging. Instead, simplified security models have been used to analyze (various versions of) TreeKEM. See Sec. 1.3 for a thorough discussion. Most critically, none of these models let the adversary deliver *arbitrary* packets; a very natural capability for a real-world attacker controlling the network. Further, they do not let the adversary register public keys in the PKI (let alone without proving knowledge of the corresponding secret keys) or choose all random coins of corrupt parties.

1.2 Our Contribution

New Security Model. To further our understanding of how MLS behaves against such insider attacks, we first precisely define insider security of CGKA. In our new model, the adversary has all above-mentioned capabilities available to malicious insiders. Our notion captures correctness as well as the following security goals: security of epoch secret keys, authenticity and agreement on group state. Formally, our model extends the notions in [8] to capture a more accurate and untrusted PKI (solving an open problem from [8]). E.g., in our model the adversary can register arbitrary (even long term) public keys on behalf of parties and without proving knowledge of corresponding secrets. Of course, security is degraded for epochs in which such keys are used but, crucially, only those.

We note that our notion can be used to analyze different CGKA protocols and compare their security guarantees. We believe that it should be directly applicable to (propose/commit versions of) protocols like [10,6,29]. Further, in a subsequent work [28] the authors use it (after small modification) to prove that their protocol enjoys the same security as TreeKEM.

Security of TreeKEM. Second, we isolate the core features of the full MLS protocol, Draft 12 (the most recent draft at the time of writing) sufficient for realizing an insider secure CGKA protocol. We call the result *Insider Secure TreeKEM* (ITK). Specifically, ITK augments TreeKEM with message authentication, tree-signing, confirmation keys and small parts of MLS's key schedule.

Third, we prove that ITK is secure in our model. Our analysis unveiled three new (and quite practical) attacks on MLS Draft 10. All attacks require the capabilities of malicious insiders, and hence they are outside the models used so far to analyze MLS, which explains why they went unnoticed until now. We proposed fixes for each of the three attacks. They have since been incorporated into to the IETF standard (in Draft 11) and are already reflected in ITK. In summary, the result of the attacks are as follows:

[5] We stress that adversarially chosen coins can lead to real-world attacks, see e.g. [22].

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

The first attack is the most interesting, since it relies on the flawed design of the so-called tree-signing mechanism, adopted due to a lack of (even intuitive) clarity around what it *should* do (which lead to differing constructions being proposed and significant debate on the topic within the MLS working group, e.g. [32,1,35]). This work finally elucidates what is the goal of tree signing.

Justifying the New Model. Finally, to justify our model and the importance of formally capturing the *complete* adversarial capabilities against which CGKA protocols intend to defend, we formally prove the following: First, for each of the three fixes for the above mentioned attacks, ITK modified to undo the fix is *not* secure in our model. Second, we observe that all previous analyses of CGKA protocols (including TreeKEM and others) in simplified models assumed CPA security of the encryption schemes they use, implying that this is sufficient (see e.g. [6,7,10]). We show that this is an oversimplification by demonstrating a practical attack on ITK modified to use a particular (contrived) CPA secure scheme, resulting in malicious insiders being able to compute epoch secrets after having been removed from the group. Again, we show that the above modification of ITK is not secure in our model. (Fortunately, as implied by [5], the PKE used in MLS is indeed CCA secure which we show to be sufficient.)

1.3 Related Work

Analyses of MLS. A summary is given in Table 1. The research on CGKA was initiated with the introduction of the Asynchronous Ratcheting Tree (ART) protocol by Cohn-Gordon et al. in [24]. ART later was adopted as part of MLS Draft 1, before being replaced by TreeKEM as part of Draft 2. TreeKEM based MLS has been analyzed in the computational setting (using the game-based approach) in the works [6,7,21]. The work by [6] analyzed the TreeKEM portion of MLS and, to this end, coined the respective CGKA abstraction. On the other hand, [7] considers the full MLS protocol and, importantly, validates the soundness of the CGKA abstraction as an intermediate building block.

In contrast to this work, [6,7] however used simplified security models. In [6] the adversary is forced to deliver packets in the same order to all parties and learns nothing about the coins of parties she has compromised. Meanwhile, [7] permits arbitrary packet delivery scheduling and leaks the random coins of corrupt parties but still does not allow the adversary to choose corrupt parties'

	MLS Version	Part Analyzed	Adversarial Model	Considers Group Splits	Framework
[24]	Draft 1 (ART)	CGKA in static groups	active	yes	part game-based, part symbolic
[6]	Draft 6	CGKA	passive	no	game-based
[19]	Draft 7	Messaging	insider	yes	symbolic
[7]	Draft 11	Messaging	semi-active	yes	game-based
[21]	Draft 11	Key derivation	insider	n/a	game-based
[25]	Draft 11	Multi-group messaging	n/a	n/a	n/a
this work	Draft 12	CGKA	insider	yes	UC

Table 1: Related work: Analyses of MLS.

coins. Neither model allows fully active attacks. In [6], the adversary cannot modify/inject packets at all while in [7] she may only deliver modified/injected packets to an honest party if the party will reject the packet.

Further, [21] focuses exclusively on the pseudorandomness of secrets produced by the key derivation process in MLS. So, unlike other works, they do not consider the general effects malformed protocol packets can have (e.g. as part of an arbitrary active attack). Instead they focus only on a specific set of effects such packets could have on the key derivation mechanism in MLS. (So for example, they make no statements about authenticity.) In contrast to the other two works, they also only allow for a limited type of adaptivity where adversaries must leak secrets at the moment they are first derived and no later. On the other hand, [21] considers a more fine-grained leakage model where secrets can be individually leaked rather than the whole local state of the victim at once. Finally, the recent work of [25] considers the 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 compromise. They present intuitive deficiencies of MLS-style constructions but they do not define a formal security model.

Complementing the above line of work, the paper [19] analyzed the insider security of TreeKEM as of Draft 7 in the symbolic setting (in the sense of Dolev-Yao). Their model covers most intuitive adversary's abilities and security properties considered in this work. Actually, they even consider a slightly more fine-grained corruption model that allows the adversary to corrupt individual keys held by parties. It is noteworthy, however, that [19] analyze a version of TreeKEM that does not yet have any tree-signing mechanism. Consequently, they find an attack on TreeKEM Draft 7 (that would also appear in our insider security model) and proposes a strong version of tree signing (aka. "tree-hash based parent hash") that prevents it. Unfortunately, that scheme soon became unworkable (i.e. not correct) as it conflicts with new mechanisms in subsequent drafts of TreeKEM, namely truncation and unmerged leaves. Thus, Draft 9 TreeKEM/MLS adopted a different, more efficient version of tree signing. In this work, we show however that the latter version is too weak and propose a new tree-signing mechanism providing the desired security.

	Protocol	Approach to Improving Efficiency	Adversarial Model	Considers Group Splitting	Framework
[10]	Tainted TreeKEM	Geared to a setting with administrators	passive	yes	game-based
[36]	Causal TreeKEM	Concurrency (static groups, no PCS)	passive	yes	game-based
[20]	Concurrent Group Ratcheting	Concurrency (static groups)	passive synchronous	no	game-based
[4]	CoCoA	Concurrency and partial views of the group state (*)	passive	yes	game-based
[3]	DeCAF	Concurrency and partial views of the group state (faster PCS than CoCoA)	passive	yes	game-based
[28]	CmPKE	Server-aided CGKA: better bandwidth	insider	yes	UC
[9]	SAIK	Server-aided CGKA: better bandwidth	active	yes	UC
[2]	Grafting Key Trees	Utilize multiple overlapping groups	n/a	n/a	n/a

	Protocol	Security Goal	Adversarial Model	Considers Group Splitting	Framework
[6]	RTreeKEM	Stronger PCFS	passive	no	game-based
[8]	Optimally Secure	Best-possible security	active	yes	UC
[27]	Membership Private ART	Hiding group roster and message senders	n/a	n/a	n/a

(*) Partial views means that parties fetch parts of their state on demand from an untrusted server.

Table 2: Related work: Other CGKA protocols. Top: protocols that improve efficiency over TreeKEM. Bottom: protocols that improve security.

Other CGKA Protocols. Numerous alternative CGKA protocols have been considered, in various security models, as summarized in Table 2. First, the Tainted TreeKEM protocol [10] exhibits a different complexity profile than the TreeKEM protocol, optimized for groups with a small set of "administrators" (i.e., parties making changes to the group roster). It was shown to enjoy the same security as TreeKEM, Draft 7, at least with regards to adaptive but passive adversaries.

Another line of research aims for better efficiency than that of TreeKEM. First, the works [36,20,4,3] achieve this by supporting (to various degrees) concurrent changes to the group state. Further, the works [28,9] proposed a different communication model of CGKA: Instead of an untrusted broadcast channel, they consider a more general (untrusted) delivery service that processes the messages and delivers to each party only the part it needs, greatly improving bandwidth. We note that [9] uses a simplified security model based on [8], while [28] uses the model proposed in this work. Finally, the work [2] introduced new techniques to accommodate for multiple intersecting groups, which may enable to get better efficiency than running several CGKAs in parallel, partially remedying the issues uncovered by [25]. (They do not specify a CGKA protocol.)

From a different angle, various constructions aim to improve on the security guarantees of TreeKEM and MLS. First, the RTreeKEM construction of [6] improved on the forward secrecy properties of the TreeKEM family of protocols, albeit by making use of non-standard (but practically efficient) cryptographic components. Further, the three CGKA protocols in [8] eschew the constraint of practical efficiency to instead focus on exploring new mechanisms for achieving the increasingly stringent security notions introduced in that work. In particular, they introduce two notions of so-called robustness for CGKA. A *weakly robust* CGKA ensures that if a honest party in epoch E accepts an arbitrary packet

p, then all other honest parties in epoch E either end up in the same state as that party or reject p. In a *strongly robust* CGKA it is further guaranteed that then all other parties currently in E will accept p. Note that neither ITK nor MLS (as a whole) are strongly robust.[6] A variant of strong robustness has also been considered by [26] who propose efficient zero-knowledge proofs with which a group member can prove to the delivery server that his message is well formed. They observe that in case the server behaves honestly, this allows the server to prevent group splitting attacks (a type of denial-of-service) caused by malicious insiders. Albeit, they do not introduce or analyze a full CGKA or messaging protocol. Finally, [27] presented CGKA with novel membership hiding properties. However, no security definitions are given.

1.4 Outline of the Rest of the Paper

We define insider secure CGKA in Sec. 3. In Sec. 4, we specify the ITK protocol. We then formalize the exact security properties achieved by ITK and sketch the respective security proof in Sec. 5. The four attacks are described in Sec. 6.

2 Preliminaries

2.1 Notation

We use $v \leftarrow x$ to denote assigning the value x to the variable v and $v \leftarrow\!\!\$\ S$ to denote sampling an element u.a.r. from a set S. If V denotes a variable storing a set, then we write $V \mathrel{+}\!\!\leftarrow x$ and $V \mathrel{-}\!\!\leftarrow x$ as shorthands for $V \leftarrow V \cup \{x\}$ and $V \leftarrow V \setminus \{x\}$, respectively. We further make use of associative arrays and use $A[i] \leftarrow x$ and $y \leftarrow A[i]$ to denote assignment and retrieval of element i, respectively. Additionally, we denote by $A[*] \leftarrow v$ the initialization of the array to the default value v. Further, we use the following keywords: **req** *cond* denotes that if the condition *cond* is false, then the current function unwinds all state changes and returns $\perp$. **assert** *cond* is used in the description of functionalities to validate inputs of the simulator. It means that if *cond* is false, then the given functionality permanently halts, making the real and ideal worlds trivially distinguishable.

2.2 Universal Composability

We use the Universal Composability (UC) framework [23].

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 [8].[7] In a nutshell, this corruption model allows the adversary to repeatedly corrupt parties by sending

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

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

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 (CorrRand, b) sets the party's rand-corrupted flag to b. If b is set, the party's randomness-sampling algorithm is replaced by the adversary providing the coins instead. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly.

Restricted Environments. In order to avoid the so-called commitment problem caused by adaptive corruptions in simulation-based frameworks, we restrict the environment not to corrupt parties at certain times. (This roughly corresponds to ruling out "trivial attacks" in game-based definitions. In simulation-based frameworks, such attacks are no longer trivial, but security against them requires strong cryptographic tools and is not achieved by most protocols.) To this end, we use the technique used in [8] (based on prior work by Backes et al. [12] and Jost et al. [30]) and consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment $\mathcal{Z}$ is admissible or not is defined as part of the ideal functionality $\mathcal{F}$: The functionality can specify certain boolean conditions, and $\mathcal{Z}$ is then called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such condition when interacting with $\mathcal{F}$.

3 Insider-Secure Continuous Group Key Agreement

This section defines security of CGKA protocols. For better readability, we skip some less crucial details. We refer to the full version [11] for the precise definition.

3.1 Overview

Security via Idealized Services. We model security and correctness of CGKA in the Universal Composability (UC) framework [23]. At a high level, this means that a CGKA protocol is secure if no efficient environment $\mathcal{Z}$ can distinguish between the following two experiments: First, in the real world experiment, $\mathcal{Z}$ interacts with an instance of the CGKA protocol. It controls all parties, i.e., it chooses their inputs and receives their outputs and the adversary, i.e., it corrupts parties. Second, in the ideal world experiment, $\mathcal{Z}$ interacts with an ideal *CGKA functionality* $\mathcal{F}_{\text{CGKA}}$ and a simulator $\mathcal{S}$. $\mathcal{F}_{\text{CGKA}}$ represents the idealized "CGKA service" a CGKA protocol should provide and is secure by design (like a trusted third party). $\mathcal{S}$ translates the real-world adversary's actions into corresponding ones in the ideal world. Since $\mathcal{F}_{\text{CGKA}}$ is secure by definition, this implies that the real-world execution cannot exhibit any attacks either. Readers not familiar with UC should think of $\mathcal{Z}$ as the adversary attacking the protocol.

In our model, analogous to [8], whenever $\mathcal{Z}$ instructs a party to perform some group operation (e.g. adding a new member) $\mathcal{F}_{\text{CGKA}}$ simply hands back an idealized protocol message to that party — it is then up to $\mathcal{Z}$ to deliver those protocol messages to the other group members, thus not making any assumptions on the underlying network or the architecture of the delivery service.

The Attack Model. In this work, we consider a powerful adversary that (a) fully controls the network (i.e., the delivery service), and (b) potentially colludes with malicious insiders. The former is captured by having $\mathcal{Z}$ (i.e., the attacker) deliver packets. The latter is captured by giving the adversary the following abilities: to register arbitrary PKI keys on behalf of any party, to repeatedly leak parties' states and to choose randomness used by parties. The first attack vector is reflected in our PKI functionalities in Sec. 3.2. The latter two vectors are reflected in our choice of UC corruption model described in Sec. 2.

We remark that additionally considering a model with malicious insider attacker but an honest delivery infrastructure is an interesting open problem. It appears, however, that in case of MLS an honest delivery server cannot prevent most of a malicious insider's attacks such as group-splitting attacks (see below).

Security Guarantees. Our model captures the following security properties: consistency, confidentiality and authenticity. They are reflected in different aspects of the ideal functionality $\mathcal{F}_{\text{CGKA}}$. We note that $\mathcal{F}_{\text{CGKA}}$ maintains a symbolic representation of the group's evolution, including corruptions, in the form of a history graph [7], where nodes represent epochs.

Intuitively, consistency means that all parties in the same epoch agree on the group state, including e.g. the history of the group's evolution. This is formalized by $\mathcal{F}_{\text{CGKA}}$ outputting consistent information to all parties in the same node of the graph. $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **safe** which identifies confidential epochs, i.e., ones for which the adversary must have no information about its group secret, for a given CGKA protocol and graph. For each confidential epoch, $\mathcal{F}_{\text{CGKA}}$ chooses a random and independent secret (and outputs it to parties who decide to fetch it) while for other epochs the key is arbitrary, i.e., chosen by the simulator. Authenticity for a party A and epoch E holds if $\mathcal{Z}$ cannot inject messages on behalf of A in E. $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **inj-allowed** which decides whether messages can be injected on behalf of the party.

The PKI. In the real-world experiment, the parties execute the protocol that furthermore interacts with the (untrusted) PKI. The latter is modeled as two UC functionalities: Authentication Service (AS) which manages long-term identity keys and Key Service (KS) that allows parties to upload single-use key packages, used by group members to non-interactively add them to the group (see Sec. 3.2 for details). Our model is agnostic to how these functionalities are realized, as long as the behavior we describe is implemented.

The primary interaction with the PKI is not group specific and, thus, it is assumed to be handled by the higher-level protocol embedding CGKA. Intuitively, this means that the protocol requires that the environment registers all keys necessary for a given group operation before performing it. As the PKI management is exposed to the environment in the real world, those operations also need to be available in the ideal world. We achieve this by having "ideal-world variants" of the AS and KS, which should be thought of as part of $\mathcal{F}_{\text{CGKA}}$. The ideal AS records which keys have been exposed, which is then used to define the predicates. The actual keys in the ideal world do not convey any particular meaning beyond serving as identifiers.

Group-Splitting Attacks. The following attack is inherent to any CGKA protocol: A malicious delivery service seletively forwards different packets to different group members, causing them to have inconsistent views of the group's evolution. Such members will never end up in the same epoch again (and so they will not be able to communicate), as this would contradict the consistency property. Our ideal functionality $\mathcal{F}_{\text{CGKA}}$ accounts for this with the history graph forming a tree, with different branches representing different partitions.

We remark that there is another type of splitting attack where (the delivery service may be honest but) a malicious insider creates a message that is accepted by some but not other members of the group. (Note that all parties accepting the commit will end up in a consistent state.) MLS does not prevent this attack, and this is reflected in our model. We note that the only way to prevent such attacks that we are aware of relies on zero-knowledge proofs [8,26] which are not widely implemented primitives MLS is constrained to use.

On the Choice of UC Security. First, the UC framework lends itself well to strong and comprehensive security definitions. Indeed, UC definitions naturally gravitate towards strongest possible guarantees. In fact, formalizing weak guarantees typically takes extra effort: Each of a protocol's weaknesses must be explicitly accounted for by providing the simulator all the necessary capabilities to emulate the effect when interacting with the ideal functionality. In contrast, game-based notions lend themselves well to simple definitions that focus on the core of a problem — potentially deliberately ignoring certain attack vectors (such as active attacks in many of the secure group-messaging work) for simplicity.

Second, the UC framework provides plenty of useful conventions and building blocks, such as the interaction with complex setup functionalities. Third, the UC framework allows us to directly formalize the *guarantees*, independent of the concrete scheme. For instance, when an active attacker can inject messages, we care about the potential effects and not so much about which exact bit-string the attacker might craft has which effect — which is handled by the simulator in our UC-based notion. (Game-based formalizations, such as [7], often circumvent this by augmenting the *primitive* to output additional information specially needed for formalizing the game, such as the interpretation of a given message.)

3.2 PKI Setup

In general, we model fully untrusted PKI, where the adversary can register arbitrary keys for any party (looking ahead, security guarantees degrade if such keys are used in the protocol). This especially models insider attacks.[8] All functionalities are formally defined in the full version [11].

[8] 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.

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

- A party id can check if a pair $(\mathsf{id}', \mathsf{spk}')$ is registered.
- id can register a new key. In this case, $\mathcal{F}_{\text{AS}}$ generates a pair $(\mathsf{spk}, \mathsf{ssk})$ (the key-generation algorithm is a parameter), sends spk to id and registers $(\mathsf{id}, \mathsf{spk})$. spk can be later retrieved at any time and then deleted.[9] If id's randomness is corrupted, the adversary provides the key-generation randomness.
- The adversary can register an arbitrary pair $(\mathsf{id}, \mathsf{spk})$.
- When a party's state is exposed, all secret keys it has generated but not deleted yet are leaked to the adversary.

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

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

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

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

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

3.3 Interfaces of the CGKA Functionality

This section explains the inputs to $\mathcal{F}_{\text{CGKA}}$, which defines the syntax of CGKA.

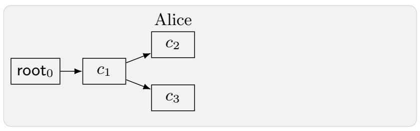
Proposals and commits. ITK is a so-called propose-and-commit variant of a CGKA, where current group members can propose to *add* new members, *remove* existing ones, or *update* their own key material (for PCS) by sending out a corresponding *proposal message*. The proposals do not affect the group state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

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

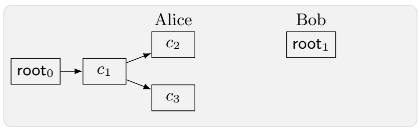
Formal syntax. The functionality accepts the following inputs (for simplicity, we treat the party's identity id as implicitly known to the protocol):

- **Group Creation:** $(\mathtt{Create}, \mathsf{spk})$ creates a new group with id being the single member, using the signing public key spk. (This input is only allowed once.)
- **Add, Remove Proposals:** $p \leftarrow (\mathtt{Propose}, \mathsf{add}\text{-}\mathsf{id}_t)$ (resp., $p \leftarrow (\mathtt{Propose}, \mathsf{rem}\text{-}\mathsf{id}_t)$) proposes to add (resp., remove) the party id_t. It outputs a proposal p, or $\perp$ if either id is not in or id_t already in (resp., not in) the group.
- **Update Proposal:** $p \leftarrow (\mathtt{Propose}, \mathsf{up}\text{-}\mathsf{spk})$ 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:** $(c, w) \leftarrow (\mathtt{Commit}, \vec{p}, \mathsf{spk})$ commits the vector of proposals $\vec{p}$ and outputs the commit message c and the (optional) welcome message w. The operation optionally updates the signing public key of the committer.
- **Process:** $(\mathsf{id}_c, \mathsf{propSem}) \leftarrow (\mathtt{Process}, c, \vec{p})$ processes the message c committing proposals $\vec{p}$ and advances id to the next epoch.[10] It outputs the committer id_c and a vector conveying the semantics of the applied proposals.
- **Join:** $(\mathsf{roster}, \mathsf{id}_c) \leftarrow (\mathtt{Join}, w)$ 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:** $K \leftarrow \mathtt{Key}$ queries the current application secret. This can only be queried once per epoch by each group member (otherwise returning $\perp$).

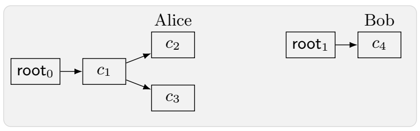
[10] For simplicity, we require that the higher-level protocol that buffers proposals also finds the list p matching c. This is without loss of generality, since ITK uses MLSPlaintext for sending proposals, and c includes hashes of proposals in $\vec{p}$.



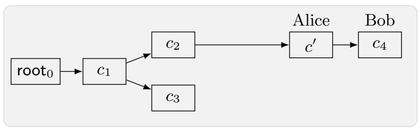
(a) The passive case. Alice processes c_1 and c_2.



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



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



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

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

3.4 History Graph

The functionality $\mathcal{F}_{\text{CGKA}}$ uses history graphs to symbolically represent a group's evolution. A history graph is a labeled directed graph. It has two types of nodes: *commit* and *proposal nodes*, representing all executed commit and propose operations, respectively. Note that each commit node represents an epoch. The nodes' labels, furthermore, keep track of all the additional information relevant for defining security. In particular, *all nodes* store the following values:

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

Proposal nodes further store the following value:

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

Commit nodes further store the following values:

- pro: the ordered list of committed proposals;
- mem: the list of group members and their signature public keys;
- key: the group key;

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

3.5 Details of the CGKA Functionality

This section presents a simplified version of $\mathcal{F}_{\text{CGKA}}$. Compared to the precise definition in the full version [11], we skip some less relevant border cases and details. A pseudo-code description is in Figs. 2 to 4 and an example history graph built by $\mathcal{F}_{\text{CGKA}}$ is in Fig. 1. We next build some intuition about how $\mathcal{F}_{\text{CGKA}}$ works.

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

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

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

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

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

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

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

Functionality $\mathcal{F}_{\text{CGKA}}$: Propose and Commit

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

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

Fig. 2: $\mathcal{F}_{\text{CGKA}}$: initialization, propose and commit. Parts related to injections and randomness corruptions are marked by comments containing [Inj] and [RndCor], respectively.

Functionality $\mathcal{F}_{\text{CGKA}}$: Process and Join

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

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

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

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

```
Input ($\texttt{CorrRand}$, id, $b$), $b \in \{$good, bad$\}$ from adv.
    RndCor[id] ← $b$

Input $\texttt{Key}$ from id
    // Only possible if id has the key.
    req Ptr[id] ≠ ⊥ ∧ HasKey[id]
    // Set the key if id is the first party fetching it in
    its node. (Guarantees consistency across parties.)
    if Node[Ptr[id]].key = ⊥ then
        if safe(Ptr[id]) then
            Set key to a fresh random key and chall to
            true.
        else
            Let the adversary choose key and set chall
            to false.
    // id should remove the key from his state
    HasKey[id] ← false
    return Node[Ptr[id]].key
```

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

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

helper *require-correctness($\text{'comm'}, \mathsf{id}, c, \vec{p}$)
Returns `true` if a) c and each $p \in \vec{p}$ identifies a node with $\mathsf{stat} \neq \mathsf{adv}$, and b) $\mathsf{Ptr}[\mathsf{id}] = \mathsf{Node}[c].\mathsf{par}$, and c) $\vec{p} = \mathsf{Node}[c].\mathsf{pro}$.

helper *require-correctness($\text{'proc'}, \mathsf{id}, \vec{p}, \mathsf{spk}$)
Returns `true` if ***usable-spk**($\mathsf{id}, \mathsf{spk}$) and $\forall p \in \vec{p}$: $\mathsf{Prop}[p] \neq \perp$ and the vector can be committed by id (in its current node) according to MLS spec.

helper *require-correctness($\text{'prop'}, \mathsf{id}, \mathsf{act}$)
Returns `true` if $\mathsf{act} = \mathsf{up\text{-}spk}$ and ***usable-spk**($\mathsf{id}, \mathsf{spk}$) or if $\mathsf{act} = \mathsf{rem\text{-}id}_t$ and removing id_t is allowed according to MLS spec.

helper *usable-spk($\mathsf{id}, \mathsf{spk}$)
Returns `true` if if either spk is id's current spk, or id has the secret key according to $\mathcal{F}_{\text{AS}}^{\text{IW}}$.

helper *members($C, \mathsf{id}, \vec{p}, \mathsf{spk}$)
Computes the member set after id, currently in C, calls commits with inputs $\vec{p}$ and spk, according to MLS spec. For each member, the set contains a tuples $(\mathsf{id}', \mathsf{spk}')$, indicating the member's identity and his identity key.

helper *can-join(C, id)
Returns `true` if $C.\mathsf{pro}$ adds id with spk and, according to $\mathcal{F}_{\text{KS}}^{\text{IW}}$, id has a secret key for some key-package registered together with spk.

helper *output-process(C)
Computes committer id_c and proposal semantics $\mathsf{propSem}$, returned by Process when transitioning into C.

helper *output-join(C)
Computes roster and committer id_c, returned when joining into C.

helper *consistent-nodes(N, N')
Returns `true` if all values in proposal or commit nodes N and N' except status match.

helper *auth-invariant
Returns `true` if there is no proposal or commit node s.t. $\mathsf{stat} = \mathsf{adv}$ and **inj-allowed**($\mathsf{par}, \mathsf{id}$) is false, where par is the node's parent.s

helper *cons-invariant
Returns `true` if HG has no cycles, each id is in the member set of $\mathsf{Ptr}[\mathsf{id}]$ and for each non-root c, the parent of each p in c's pro vector is c's parent.

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

the output of process). Note that we also guarantee correctness — if the input of process is an honest message c generated by $\mathcal{F}_{\text{CGKA}}$, then the adversary cannot make the commit fail.

A more challenging scenario is when the environment injects a welcome message w'. Now there are two possibilities. First, w' could lead to an existing node. In this case, $\mathcal{F}_{\text{CGKA}}$ asks the adversary to provide the node c and records that w' leads to it. We require agreement — any party subsequently joining using w' transitions to c. However, in general, we cannot expect that the adversary (i.e., simulator), given an arbitrary w' computed by the environment, can come up with the whole commit message c' and its position in the history graph.[11] Therefore, in this case $\mathcal{F}_{\text{CGKA}}$ creates a detached root, identified by a unique label $\mathsf{root}_{\mathsf{rootCtr}}$, where $\mathsf{rootCtr}$ is a counter. If at some later point, e.g. after an additional commit by the newly joined party, the environment injects c' corresponding to w', then the root is attached and re-labeled as c'. This scenario is depicted in Figs. 1b to 1d. We require consistency — when creating a detached root, the adversary chooses the member set, but when it is attached, we check that it matches the new parent.

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

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

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

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

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.[12]

4 The Insider-secure TreeKEM Protocol

This section provides a (high-level) description of the Insider-Secure TreeKEM (ITK) protocol. A formal description of the protocol can be found in the full version [11].

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

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

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

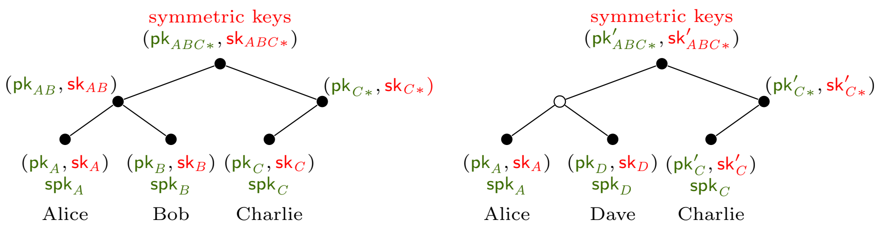

symmetric keys
$(\mathsf{pk}_{ABC*}, \mathsf{sk}_{ABC*})$
$(\mathsf{pk}_{AB}, \mathsf{sk}_{AB})$
$(\mathsf{pk}_{C*}, \mathsf{sk}_{C*})$
$(\mathsf{pk}_A, \mathsf{sk}_A)$ spk_A Alice
$(\mathsf{pk}_B, \mathsf{sk}_B)$ spk_B Bob
$(\mathsf{pk}_C, \mathsf{sk}_C)$ spk_C Charlie
symmetric keys
$(\mathsf{pk}'_{ABC*},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

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

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.

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

A *remove* proposal represents removing from τ all keys known to the removed party (see Fig. 5). That is, its leaf is cleared, and all keys in its *direct path* — i.e., the path from the party's leaf to the root — are *blanked*, meaning that all their labels are cleared. This is followed by shrinking the tree by removing unneeded leaves from the right side of the tree. Note that until a blanked node gets a new key pair assigned (as explained shortly), in order to encrypt to the respective subgroup one has to encrypt to the node's children instead (and recursing if either child is blanked as well). The minimal set of non-blanked nodes covering a given subgroup is called the subgroup's *resolution*.

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

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

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

To minimize the number of secret keys needed to be communicated as part of the commit message, the committer samples the fresh key pairs along the path by "hashing up the tree". That is, the committer derives a sequence of *path secrets* s_i, one for each node on the path, where s_0 for the leaf is random and s_{i+1} is derived from s_i using the $\mathsf{HKDF.Expand}$ function. 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. All messages intended for existing group members — commit messages and proposals — are subject to *message framing*, which binds them to the group and epoch, indicates the sender, and protects the message's authenticity. The sender first signs the group identifier, the epoch, his leaf index, and the message using his private signing key. This in particular prevents impersonation by another (malicious) group member.

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

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

Finally, ITK uses a mechanism called *tree signing* to achieve a certain level of insider security. We discuss this aspect in detail in Sec. 6.3.

Remark 1 (Simplifications and Deviations). While ITK closely follows the IETF MLS protocol draft, there are some small deviations as well as some omissions. In particular, our model assumes a fixed protocol version and ciphersuite, and omits features such as advanced meta-data protection, external proposals and commits, exporters, preshared keys, as well as extensions. We discuss those deviations and their implications on our results in more detail in the full version [11].

5 Security of ITK

Security of ITK is expressed by the predicates **safe**(c, id) and **inj-allowed**(c, id), where c is a commit message identifying a history graph node and id is a party. The predicates are formally stated in Fig. 6. They are defined using recursive deduction rules **know**(c, id) and **know**$(c, \text{`epoch'})$, indicating that the adversary knows id's secrets (such as the leaf secret), and that it knows the epoch secrets (such as the init secret), respectively. In more detail:

- **know**(c, id) consists of three conditions, the last two being recursive. Condition a) is true if id's secrets in c are known to the adversary because they leaked as part of an exposure or were injected by the adversary in id's name (due to many attack vectors, this can happen in many ways, see Fig. 6). The conditions b) and c) reflect that in ITK only commits sent by or affect id (id updates, is added, or removed) are guaranteed to modify all id's secrets. If c is not of this type, then **know**(c, id) is implied by **know**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{id})$ (condition b)). If a child c' of c is not of this type, then it is implied by **know**(c', id) (condition c)).
- **know**$(c, \text{`epoch'})$ takes into account the fact that ITK derives epoch secrets using the initSecret from the previous epoch, and hence achieves slightly better FS compared to parties' individual secrets.
  In particular, the adversary knows the epoch secrets in c only if it corrupted a party in c, or knows the epoch secrets in c's parent and knows individual secret of some party id in c. The latter condition allows the adversary to process c using id's protocol and is formalized by the ***can-traverse** predicate.
- The only difference between $\neg$**safe**(c) and **know**$(c, \text{`epoch'})$ is that the application secret is not leaked if id is exposed in c after outputting it.

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

Theorem 1. *Assuming that* PKE *is* IND-CCA *secure, and that* Sig *is* EUF-CMA *secure, then the* ITK *protocol securely realizes* $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}}, \mathcal{F}_{\text{CGKA}})$ *in the* $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where* $\mathcal{F}_{\text{CGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Fig. 6 and calls to* HKDF.Expand*,* HKDF.Extract *and* MAC *functions are replaced by calls to the global random oracle* $\mathcal{G}_{\text{RO}}$*.*

Proof (Sketch). We here provide the high level proof idea; the complete proof is presented in the full version [11]. The proof proceeds in three steps. The first step is to show that various consistency mechanisms, such as MACing the group context, guarantee consistency of the distributed group state. More precisely, the real world (Hybrid 1) is indistinguishable from the following Hybrid 2: The experiment includes a modified CGKA functionality, $\mathcal{F}_{\text{CGKA}}^{\texttt{real}}$, which differs from $\mathcal{F}_{\text{CGKA}}$ in that it uses **safe** = `false` and **inj-allowed** = `true`. The functionality interacts with the trivial simulator who sets all keys and messages according to the protocol. The second step is to show that IND-CCA of the PKE scheme guarantees confidentiality: Hybrid 2 is indistinguishable from Hybrid 3 where

Predicate safe

Knowledge of parties' secrets.

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

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

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

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

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

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

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

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

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

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

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

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

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

Knowledge of epoch secrets.

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

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

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

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

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

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

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

Safe and can-inject.

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

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

Fig. 6: The safety and injectability predicates for the CGKA functionality reflecting the sub-optimal security of the ITK protocol.

application and membership secrets in safe epochs are random, i.e. the original **safe** is restored. The final step is to show that unforgeability of the MAC and signature schemes implies that Hybrid 3 is indistinguishable from the ideal world, where the original **inj-allowed** is restored as well. (Considering confidentiality before integrity, while somewhat unusual, is necessary, because we must first argue secrecy of MAC keys. We note that IND-CPA would be anyway insufficient, because some injections are inherently possible.)

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

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

Unfortunately, already the passive setting turns out challenging for the following reason: The path secrets in a (safe) commit c are encrypted under public keys created in another commit c', which contains encryptions of the corresponding secret keys under public keys created in another commit c'', and so on. Moreover, the keys are related by hash chains (of path secrets). Even worse, the environment can adaptively choose who to corrupt, revealing some subset of the secret keys, which mean that we cannot simply apply the hybrid argument to replace encryptions of secret keys by encryptions of zeros.[13]

To tackle adaptivity and related keys, we adapt the techniques of [33,10]. Namely, we define a new security notion for PKE, called (modified) Generalized Selective Decryption (GSD),[14] which generalizes the way ITK uses PKE together with the hash function to derive its secrets. Roughly speaking, the GSD game creates a graph, where each node stores a secret seed. The adversary can instruct the game to 1) create a node with a random seed, 2) create a node v where the seed is a hash of the seed of another node u, 3) use a (different) hash of the seed in a node u to derive a key pair, use the public key to encrypt the seed in a node v and send the public key and ciphertext to the adversary. Each of the actions 2) and 3) creates an edge (u, v) to indicate their relation. Moreover, the adversary can adaptively corrupt nodes and receive their seeds. For the challenge of the game, she receives either a seed from a sink node or a random value. (See the full proof for a precise definition.)[15] It remains to be shown that 1) GSD security

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

[14] GSD was first defined for symmetric encryption [33] and then extended to prove security of TreeKEM [10]. Our notion is an extension of [10].

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

implies secrecy of ITK keys, and 2) IND-CPA security implies GSD security. The latter proof is adapted from [10], so we now focus on 1).

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

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

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

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

is the commit secret due to *ctxt* being the same. Hence, the joiner secret of c is the same as the one extracted from c'. □

6 Insider Attacks

We first discuss three insider attacks on the design of MLS Draft 10 (as it stood prior to applying the fixes proposed as part of this work). Each is practical, yet violates the design goals of MLS. Next, we present an insider attack on MLS made possible when its ciphersuite is replaced by a weaker one that still meets assumptions deemed sufficient in previous analyses. Together these attacks highlight the limitations of prior security notions.

6.1 An Attack on Authenticity in Certain Modes

MLS supports two wire formats for packets: MLSCiphertext, meant to provide extra metadata protection by applying an extra layer of authenticated symmetric encryption, and MLSPlaintext, allowing for additional server-assisted efficiency improvements. As part of our analysis, we realized that an MLSCiphertext (unintentionally) provides stronger authentication guarantees than an MLSPlaintext: Forging the latter requires only signature keys of a group member while the former also requires knowing the current epoch's key. This results in weaker than expected PCS since signature keys will be rotated much less frequently than epoch keys: Despite a party having issued an update proposal or a commit the adversary may, thus, still be able to forge certain types of messages, such as proposals.

Theorem 2. *The* $\mathsf{ITK}_{\mathsf{Atk\text{-}1}}$ *protocol, which behaves like the* ITK *protocol but does not include membership tags, does* not *securely realize* $(\mathcal{F}_{\mathrm{AS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{KS}}^{\mathrm{IW}}, \mathcal{F}_{\mathrm{CGKA}})$ *in the* $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}}, \mathcal{G}_{\mathrm{RO}})$*-hybrid model when* $\mathcal{F}_{\mathrm{CGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Fig. 6.*

Proof (Sketch). Let $\mathcal{S}$ be an arbitrary simulator and consider the following environment $\mathcal{Z}$ that initially sets up a group consisting of three parties A, B, and C in the same group state. In this state, $\mathcal{Z}$ then corrupts party A, hence learns its signing key ssk_A. Then, $\mathcal{Z}$ instructs A to issue a commit message c with an empty list of proposals and the old spk_A. (This causes A to update its ephemeral key and resample the compromised path in the ratchet tree, but keep its long-term signing key.) Now, $\mathcal{Z}$ crafts a proposal message p^* that removes C on behalf of A, according to the (modified) protocol $\mathsf{ITK}_{\mathsf{Atk\text{-}1}}$. Note that all the included values are public and thus known to the environment, and $\mathcal{Z}$ can sign the proposal using the leaked ssk_A. (Important: note that the environment does *not* instruct A do create such a proposal command, but forges it!) Finally, $\mathcal{Z}$ instructs B to commit to this proposal p^* and lets B process the respective commit message c^*. If B accepts and outputs the correct semantics for p^*, then $\mathcal{Z}$ returns 1, otherwise it returns 0.

It is easy to see that $\mathcal{Z}$ outputs 1 when interacting with the hybrid world as p^* is a valid proposal created identically to how the honest party A would. Now consider the ideal world functionality and observe that after A issues the commit c_2 all parties are in the same state, which is further marked as good, i.e., with $\mathsf{stat} = \mathsf{good}$ for it is created by an honest party with good randomness. We now observe that functionality's authenticity invariant will fail at the end of B committing p^*, as **inj-allowed**(c, A) (whether the adversary can inject on behalf of A) in the parent state (the one created by A's second commit) as **know**$(c, \text{‘}\mathsf{epoch}\text{'})$ will return false indicating that the adversary does not know the symmetric key of said state. Hence, when interacting with the ideal functionality the authenticity invariant prevents B from successfully committing to the proposal p^*, causing $\mathcal{Z}$ to return 0. □

To bring the authenticity guarantees in line, we proposed adding a MAC to MLSPlaintexts [15].

6.2 Breaking Agreement

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

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

In our security model this shows up as a break on the notion of a group state, as formalized by the history graph nodes. That is, in our model each history graph node is supposed to correspond to a well-defined and consistent group state. The way the transcript hash used to be computed violated this property, as on the one hand parties had the same key and could exchange messages (same state) while on the other hand parties would no longer be able to process each other's commit messages (different states). In particular, when processing two such related commit messages c and c' that only differ in the signature, in the ideal functionality $\mathcal{F}_{\text{CGKA}}$ the parties end up in two distinct states. Yet, in the real world execution the parties would still accept each other's proposals, which in $\mathcal{F}_{\text{CGKA}}$ is ruled out by the consistency invariant.

Theorem 3. *Assume the signature scheme* Sig *does not have unique signatures (this strong property is not achieved by the schemes used by MLS). Then, the* $\mathsf{ITK}_{\mathsf{Atk\text{-}2}}$ *protocol, which behaves like* ITK *using* Sig *but does not include the* *package's signature into the confirmed transcript hash, does not securely realize* $(\mathcal{F}_{\text{AS}}^{\text{IW}}, \mathcal{F}_{\text{KS}}^{\text{IW}}, \mathcal{F}_{\text{CGKA}})$ *in the* $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}}, \mathcal{G}_{\text{RO}})$*-hybrid model when* $\mathcal{F}_{\text{CGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Fig. 6.*

Proof (Sketch). Let $\mathcal{S}$ be an arbitrary simulator and consider the following environment $\mathcal{Z}$ that initially sets up a group consisting of parties A, B, and C that are in the same consistent state, as in the previous proof. Then, the environment acts as a malicious insider A sending semi-inconsistent commit messages to B and C. To this end, it corrupts party A and learns ssk_A. Afterwards it computes a commit message c_1 (to an empty proposal list) and another one c_1' by first copying c_1 and then replacing the signature by a different valid one. It delivers c_1 to B and c_1' to C. Finally, $\mathcal{Z}$ instructs B to create a proposal p that removes A from the group. Moreover, instruct both B and C to first commit to this proposal (creating commit messages c_2 and c_2', respectively) and have each of the parties process their own commit message. If both parties successfully process their commit messages, $\mathcal{Z}$ outputs 1, and 0 otherwise.

It is easy to see that when interacting with the hybrid world both B and C successfully process their own commits, as the interim transcript hash does not affect the proposal p, making it valid for both B and C whose views agree in everything but the interim transcript hash. In the ideal world, however, p is associated with B's node and as a result cannot be committed to by C, as enforced by the consistency invariant. (In our model two different ciphertexts c_1 and c_1' cannot point to the same node.) As a result, $\mathcal{Z}$ outputs 0 when interacting with the ideal world. $\square$

Our fix that moves the signature into the confirmed transcript hash has been incorporated into MLS [16].

6.3 Inadequate Joiner Security (Tree-Signing)

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

Surprisingly, we can show that the initial tree signing mechanism introduced in MLS Draft 9 does not achieve this guarantee. Rather, it achieves something much weaker: A joiner ends up in a secure epoch once all members with the following types of long-term signature keys have been removed: (a) corrupt keys and (b) keys used in a different epoch that includes a key of type (a). We believe this to be an unexpectedly weak guarantee. In particular, it means that malicious insiders can read messages after being removed.[16]

[16] It also seems to contradict the (informal) notion of the "tree-invariant" often cited on the MLS mailing list.

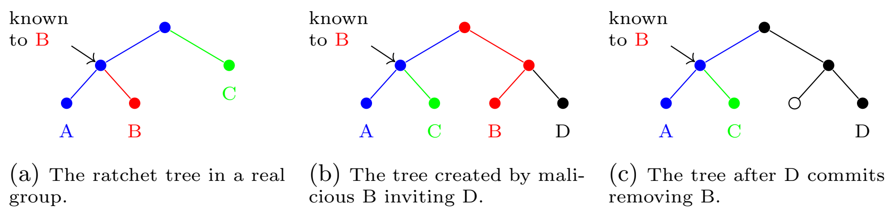

known to B
A
B
C
(a) The ratchet tree in a real group.
known to B
A
C
B
D
(b) The tree created by malicious B inviting D.
known to B
A
C
D
(c) The tree after D commits removing B.

Fig. 7: The attack on the tree signing of $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$.

The attack on tree-signing. We call ITK using the tree-signing mechanism from MLS Draft 9 $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$. We next present a simple and highly practical attack against $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$. It results in groups with epochs containing no keys of type A) yet for which the epoch key is easy to compute by the malicious insiders.

We first recall the tree signing of $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$. It works by storing in each ratchet-tree node v a value $v.\mathsf{parentHash}$ computed as follows.

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

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

Intuitively, the issue is, however, that committers only attest to the key pairs they (honestly) generated, but *not* to which parties they informed of the secret keys. This allows a malicious insider to create his own ratchet tree, where they knows secrets of nodes that are not on his direct path. Therefore, removing them from the fake group doesn't cause removal of every key they knows, breaking Invariant (2) of the protocol.

Theorem 4. *The* $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$ *protocol, that behaves like* ITK *but with the MLS Draft 9 tree-signing mechanism, does* not *securely realize* $(\mathcal{F}^{\mathrm{IW}}_{\mathrm{AS}}, \mathcal{F}^{\mathrm{IW}}_{\mathrm{KS}}, \mathcal{F}_{\mathrm{CGKA}})$ *in the* $(\mathcal{F}_{\mathrm{AS}}, \mathcal{F}_{\mathrm{KS}}, \mathcal{G}_{\mathrm{RO}})$*-hybrid model when* $\mathcal{F}_{\mathrm{CGKA}}$ *uses the predicates* **safe** *and* **inj-allowed** *from Fig. 6.*

Proof (Sketch). The attack is illustrated in Fig. 7. Assume that the environment $\mathcal{Z}$ sets up a group with a group creator A adding parties B and C (in this order), leading in the hybrid world to ratchet tree depicted in Fig. 7a. In this state, the adversary corrupts party B, which henceforth is assumed to be malicious, while A and C are never corrupted and, thus, honest. In the following $\mathcal{Z}$ acts on behalf of the corrupted B and builds the fake ratchet tree from Fig. 7b, meaning $\mathcal{Z}$ swaps parties B and C (their public keys), then adds party D on behalf of B to the group, outputting a respective welcome message w using B's leaked signing key. Crucially, we observe that the ratchet tree from Fig. 7b represents a valid one that D will accept: In Fig. 7a C's leaf signature only attested to C's leaf key (the green one) as the parent hash field is empty. Second, A's leaf signature does not attests to B's leaf key (but only the blue ones) as the parent hash only includes the nodes on A's direct path to the root. Third, $\mathcal{Z}$ can re-key B's new path and attest to the fresh keys (the red ones) using the leaked signing key.

The environment then delivers w to D, joining them to the fake group, and afterwards $\mathcal{Z}$ instructs D remove B, i.e., to propose, commit, and then process the respective commit message c'. Finally, $\mathcal{Z}$ queries D's group key key and also computes the expected group key key' by taking D's commit message and using the secret key known to $\mathcal{Z}$ marked in Fig. 7c and perform the same computation C would in the $\mathsf{ITK}_{\mathsf{Atk\text{-}3}}$ protocol. If $\mathsf{key} = \mathsf{key}'$, then $\mathcal{Z}$ outputs 1 and 0 otherwise.

It remains to convince ourselves that $\mathcal{Z}$ distinguishes with non-negligible probability for any simulator $\mathcal{S}$. It is easy to see that when interacting with the hybrid world $\mathcal{Z}$ outputs 1. Finally, consider the ideal-world. We argue that $\textbf{safe}(c') = \texttt{true}$ meaning that the functionality outputs an independent and u.a.r. key and, thus, $\mathcal{Z}$ outputs 0 with overwhelming probability. First, it is easy to see that $\mathcal{S}$ has to join D to a detached root as no other group state matches, e.g,. none has D as a member. Next, observe that D has not been corrupted implying that the node created by D's commit is marked with $\mathsf{Node}[c'].\mathsf{stat} = \mathsf{good}$ and has no direct exposures, i.e., $\mathsf{Node}[c'].\mathsf{exp} = \varnothing$. As a result, we have $\textbf{safe}(c') = \neg\textbf{*can-traverse}(c')$ while $\textbf{*can-traverse}(c') = \texttt{false}$ as clearly only case (d) might apply but $\textbf{know}(c', \mathsf{id}) = \texttt{false}$ for all $\mathsf{id} \in \{A, B, C, D\}$ for the following reasons: First, A, C, and D have never been corrupted, in particular implying $\textbf{*state-directly-leaks}(c', \mathsf{id}) = \texttt{false}$ and $\textbf{know}(\mathsf{root}_1, \mathsf{id}) = \texttt{false}$, where root_1 denotes the detached root to which D joined. Second, for B, observe that $\textbf{*state-directly-leaks}(c', B) = \texttt{false}$ as $B \notin \mathsf{Node}[c'].\mathsf{mem}$ and $B \notin \mathsf{Node}[c'].\mathsf{exp}$ while $\textbf{*secrets-replaced}(c', B) = \texttt{true}$ as B has been removed from that state. Thus, we can deduce that $\textbf{safe}(c') = \texttt{true}$, concluding the proof. □

Fixing tree signing. In essence, we can prevent the attack by modifying the parent hash such that committers attest to the key pairs they generated *and* to which parties were informed about the secret keys. We can achieve this by computing the parent hash $v.\mathsf{parentHash}$ as $\mathsf{Hash}(w.\mathsf{pk}, w.\mathsf{parentHash}, w.\mathsf{memberCert})$ where w is v's parent and $\mathsf{memberCert}$ attests to the set of parties informed about the $w.\mathsf{sk}$. It is left to find a good candidate for $\mathsf{memberCert}$; one that is secure and easy to compute. We next discuss 3 candidates for $\mathsf{memberCert}$.

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

The second candidate, called *the tree parent hash*, has been initially considered for MLS [35]. It basically sets $w.\mathsf{memberCert}$ to the tree hash of $v.\mathsf{sibling}$ with the unmerged leaves omitted (recall that ITK computes the tree hash as the

Merkle hash of the ratchet tree). Observe that the tree hash binds strictly more than the leaf hash. The tree hash would be more straightforward to compute. Unfortunately, it is not workable due to other mechanisms of MLS.[17]

Therefore, we propose a new candidate called *the resolution parent hash*. It improves upon the leaf hash in 2 ways: it is more implementation-friendly and it has slightly better deniability properties.[18] The resolution hash sets $\mathsf{memberCert}$ to the PKE public keys of nodes in $u.\mathsf{origChildResolution}$ where $u.\mathsf{origChildResolution}$ is the resolution of u with the unmerged leaves of $u.\mathsf{parent}$ omitted. Observe that $u.\mathsf{origChildResolution}$ is the resolution of u at the time the last committer in the subtree of v generated the key pair of w.

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

6.4 IND-CPA Security Is Insufficient

Many prior analysis of MLS only assume IND-CPA security of the PKE scheme it uses. However, there are PKE schemes that are IND-CPA secure but that make MLS clearly insecure against active attackers — despite MLS employing signatures and MACs to protect authenticity — highlighting the inadequacies of those works' simplified security models to account for all relevant aspects (and the danger of analyzing too piecemeal protocols without considering their composition in general).

Consider the protocol $\mathsf{ITK}_{\mathsf{cpa}}$ which behaves like ITK but replaces its PKE scheme with PKE^*. PKE^* is IND-CPA secure and has the following property: a ciphertext ctx containing a message m can be modified into ctx_i, s.t. decrypting ctx_i outputs $\perp$ if and only if the i-th bit of m is 0, and otherwise decrypting ctx_i outputs m.[19] The following attack shows that $\mathsf{ITK}_{\mathsf{cpa}}$ is clearly insecure in the setting with active attackers. In particular, a malicious insider can decrypt messages after being removed from the group. Let κ denote the length of a path secret used by MLS. The attack proceeds as follows:

1. An honest execution leads to an epoch E_1 where the group has $N = 4\kappa$ members $P_1, \ldots, P_N$, ordered according to their leaves from left to right. Further, the ratchet tree has no blanks.

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

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

[19] PKE^* can be easily obtained as a straightforward adaptation of the artificial symmetric encryption scheme by Krawczyk [31] (used to show that the authenticate-then-encrypt paradigm is not secure in general) to the public key setting.

2. The adversary corrupts P_1 and P_N.
3. P_1 (honestly) sends a commit c_1, creating an epoch E_2. P_{N-1} transitions to E_2, and sends a commit c_2 that removes P_N, creating epoch E_3.
The expectation is that E_3 is secure due to PCS and removing all corrupted members. The adversary will next compute group key in E_3.
4. The adversary has the following information: P_1's signing key ssk_1 (the same in all epochs), the secret key sk of the right child of the root in E_1 (corrupted P_N knows sk), the init secret in E_1 and the ciphertexts *ctxRoot* and *ctxLchild* encrypting P_1's two last path secrets in c_1.
The adversary shouldn't know the path secret s encrypted in ctxLchild, since this breaks the tree invariant. He will next learn s it bit by bit.
5. The members who will decrypt *ctxLchild* are $P_{\kappa+1}$ to $P_{2\kappa}$. For $i = 1$ to κ, the adversary injects to $P_{\kappa+i}$ the packet c_1 modified as follows:
   (a) Replace *ctxLchild* by $ctxLchild_i$ obtained using the PKE^* property.
   (b) Update the confirmation tag accordingly: 1) Decrypt *ctxRoot* using sk. The result is the next path secret s' after s. 2) Use s' to compute the commit secret. 3) Compute the new key schedule using the init secret in E_1 and the commit secret from 2). 4) Compute the tag.
   (c) Update the signature using ssk_1.
6. Clearly, if $P_{\kappa+i}$ accepts, then the i-th bit of s is 0, else 1.
Now the adversary uses s to compute the key in E_3.
7. Using s, the adversary derives the secret key for the left child of the root in E_2. Since this node is in the copath of P_{N-1}, the adversary can use it to decrypt the commit secret from c_2. The adversary then computes the init secret in E_2 by honestly running P_N's protocol and mixes it with the commit to derive the key schedule in E_3.

Clearly, however, the **safe** predicate of our $\mathcal{F}_{\text{CGKA}}$ functionality considers the resulting key from epoch E_3 as secure. Hence, we get the following result.

Theorem 5. *The $\mathsf{ITK}_{\mathsf{cpa}}$ protocol that behaves like ITK does* not *securely realize $(\mathcal{F}^{\text{IW}}_{\text{AS}}, \mathcal{F}^{\text{IW}}_{\text{KS}}, \mathcal{F}_{\text{CGKA}})$ in the $(\mathcal{F}_{\text{AS}}, \mathcal{F}_{\text{KS}}, \mathcal{G}_{\text{RO}})$-hybrid model when $\mathcal{F}_{\text{CGKA}}$ uses the predicates* **safe** *and* **inj-allowed** *from Fig. 6.*

Proof. We show that for every simulator $\mathcal{S}$, there exists an environment $\mathcal{Z}$ that has non-negligible advantage in distinguishing the ideal world from the real world with $\mathsf{ITK}_{\mathsf{cpa}}$. Let $\mathcal{S}$ be any simulator. The environment $\mathcal{Z}$ executes the attack described above, i.e., it gives appropriate instructions to honest parties and performs the adversary's attacks. Let key' denote the group key computed at the end by the adversary. $\mathcal{Z}$ fetches the group key key in E_3 (via the `Key` query to say P_5). If $\mathsf{key} = \mathsf{key}'$, it outputs 1 else 0.

We will show that **safe** is true in E_3. Given this, we can conclude the proof with the following observations: Clearly, in the real world, $\mathcal{Z}$ always outputs 1 (for simplicity we assume perfect correctness). In the ideal world, since **safe** is true, key is chosen by $\mathcal{F}_{\text{CGKA}}$ random and independent of $\mathcal{S}$. Since key' is computed by $\mathcal{Z}$ only from information given to $\mathcal{S}$, this means that with overwhelming probability $\mathsf{key} \neq \mathsf{key}'$, and hence $\mathcal{Z}$ outputs 0.

It remains to show that **safe** is true. Informally, the only corruptions are of P_1 and P_N in E_1. Transitioning to E_1 "heals" from P_1's corruption, since this is an honest commit from them, and transitioning to E_3 heals from P_N's corruption, since they are removed.

Formally, we will show that **know** is false for all parties in E_3. This will mean that ***can-traverse**(c_2) = `false` (by inspection, all other conditions that can make it true do not occur). Hence, **safe** is true in E_3.

Observe that **know** can only be true for P_1 and P_N, as ***state-directly-leaks** is only true for these parties in E_1. First, ***secrets-replaced**(c_1, P_1) is true, since None$[c_1]$.orig $= P_1$. Therefore, **know**(c_1, P_1) = `false` and by recursion **know**(c_2, P_1) = `false`. Second, ***secrets-replaced**(c_2, P_2) is true, since it includes a proposal with act = rem-P_N. Therefore, **know**(c_1, P_1) = `false`. □

References

1. Messagying layer security (mls) wg - meeting minutes for interim 2020-1 (January 2020), https://datatracker.ietf.org/doc/minutes-interim-2020-mls-01-202001110900/
2. Alwen, J., Auerbach, B., Baig, M.A., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Grafting key trees: Efficient key management for overlapping groups. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part III. LNCS, vol. 13044, pp. 222–253. Springer, Heidelberg (Nov 2021). https://doi.org/10.1007/978-3-030-90456-2_8
3. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K.: DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559 (2022), https://eprint.iacr.org/2022/559
4. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: CoCoA: Concurrent continuous group key agreement. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 815–844. Springer, Heidelberg (May / Jun 2022). https://doi.org/10.1007/978-3-031-07085-3_28
5. Alwen, J., Blanchet, B., Hauck, E., Kiltz, E., Lipp, B., Riepel, D.: Analysing the HPKE standard. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part I. LNCS, vol. 12696, pp. 87–116. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-77870-5_4
6. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020). https://doi.org/10.1007/978-3-030-56784-2_9
7. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 1463–1483. ACM Press (Nov 2021). https://doi.org/10.1145/3460120.3484820
8. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020). https://doi.org/10.1007/978-3-030-64378-2_10

9. Alwen, J., Hartmann, D., Kiltz, E., Mularczyk, M.: Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456 (2021), https://eprint.iacr.org/2021/1456
10. Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy, S&P. pp. 268–284 (2021). https://doi.org/10.1109/SP40001.2021.00035, full version: https://eprint.iacr.org/2019/1489
11. Alwen, J., Jost, D., Mularczyk, M.: On the insider security of mls. Cryptology ePrint Archive, Paper 2020/1327 (2020), https://eprint.iacr.org/2020/1327, full version of this paper.
12. Backes, M., Dürmuth, M., Hofheinz, D., Küsters, R.: Conditional reactive simulatability. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 424–443. Springer, Heidelberg (Sep 2006). https://doi.org/10.1007/11863908_26
13. Barnes, R., Beurdouche, B., , Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: The messaging layer security (mls) protocol (draft-ietf-mls-protocol-12). Tech. rep., IETF (Mar 2020), https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/12/
14. Barnes, R.: Subject: [MLS] Remove without double-join (in TreeKEM). MLS Mailing List (06 August2018 13:01UTC), https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik
15. Barnes, R.: MLS Protocol Pull Requests #396: Authenticate group membership in MLSPlaintext (18 August 2020), https://github.com/mlswg/mls-protocol/pull/396
16. Barnes, R.: MLS Protocol Pull Requests #416: Inlclude the signature in the confirmation tag (18 August 2020), https://github.com/mlswg/mls-protocol/pull/416
17. Barnes, R.: Subject: [MLS] Proposal: Proposals (was: Laziness). MLS Mailing List (22 August 2019 22:17UTC), https://mailarchive.ietf.org/arch/msg/mls/5dmrkULQeyvNu5k3MV_sXreybj0/
18. Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups (May 2018), http://prosecco.inria.fr/personal/karthik/pubs/treekem.pdf, published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8
19. Bhargavan, K., Beurdouche, B., Naldurg, P.: Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris (Dec 2019), https://hal.inria.fr/hal-02425229
20. Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Heidelberg (Nov 2020). https://doi.org/10.1007/978-3-030-64378-2_8
21. Brzuska, C., Cornelissen, E., Kohbrok, K.: Security analysis of the mls key derivation. In: 2022 IEEE Symposium on Security and Privacy, S&P. pp. 595–613. IEEE Computer Society, Los Alamitos, CA, USA (may 2022). https://doi.org/10.1109/SP46214.2022.00035, https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00035
22. Bushing, Marcan, Segher, Sven: Console hacking 2010 — PS3 epic fail. In: 27th Chaos Communication Congress — 27C3 (2010), https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html

23. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd FOCS. pp. 136–145. IEEE Computer Society Press (Oct 2001). https://doi.org/10.1109/SFCS.2001.959888
24. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 1802–1819. ACM Press (Oct 2018). https://doi.org/10.1145/3243734.3243747
25. Cremers, C., Hale, B., Kohbrok, K.: The complexities of healing in secure group messaging: Why cross-group effects matter. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021. pp. 1847–1864. USENIX Association (Aug 2021)
26. Devigne, J., Duguey, C., Fouque, P.A.: MLS group messaging: How zero-knowledge can secure updates. In: Bertino, E., Shulman, H., Waidner, M. (eds.) ESORICS 2021, Part II. LNCS, vol. 12973, pp. 587–607. Springer, Heidelberg (Oct 2021). https://doi.org/10.1007/978-3-030-88428-4_29
27. Emura, K., Kajita, K., Nojima, R., Ogawa, K., Ohtake, G.: Membership privacy for asynchronous group messaging. Cryptology ePrint Archive, Report 2022/046 (2022), https://eprint.iacr.org/2022/046
28. Hashimoto, K., Katsumata, S., Postlethwaite, E., Prest, T., Westerbaan, B.: A concrete treatment of efficient continuous group key agreement via multi-recipient pkes. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1441–1462 (2021)
29. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2_6
30. Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019, Part II. LNCS, vol. 11892, pp. 180–210. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-36033-7_7
31. Krawczyk, H.: The order of encryption and authentication for protecting communications (or: How secure is SSL?). In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 310–331. Springer, Heidelberg (Aug 2001). https://doi.org/10.1007/3-540-44647-8_19
32. Miller, M.A.: Messaging layer security (mls) wg - meeting minutes for ietf105 (August 2019), https://datatracker.ietf.org/doc/minutes-105-mls/
33. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 21–40. Springer, Heidelberg (Feb 2007). https://doi.org/10.1007/978-3-540-70936-7_2
34. Rescorla, E.: Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List (03 May 2018 14:27UTC), https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no
35. Sullivan, N.: Subject: [MLS] Virtual interim minutes. MLS Mailing List (29 January 2020 21:39UTC), https://mailarchive.ietf.org/arch/msg/mls/ZZAz6tXj-jQ8nccf7SyIwSnhivQ/
36. Weidner, M.: Group messaging for secure asynchronous collaboration. MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann (2019), https://mattweidner.com/acs-dissertation.pdf