Continuous Group Key Agreement with Active Security

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

[1] Wickr, jalwen@wickr.com
[2] IOHK, sandro.coretti@iohk.io
[3] ETH Zurich, Switzerland, {dajost, mumarta}@inf.ethz.ch

Abstract. A *continuous group key agreement* (CGKA) protocol allows a long-lived group of parties to agree on a continuous stream of fresh secret key material. CGKA protocols allow parties to join and leave mid-session but may neither rely on special group managers, trusted third parties, nor on any assumptions about if, when, or for how long members are online. CGKA captures the core of an emerging generation of highly practical end-to-end secure group messaging (SGM) protocols.

In light of their practical origins, past work on CGKA protocols have been subject to stringent engineering and efficiency constraints at the cost of diminished security properties. In this work, we somewhat relax those constraints, instead considering progressively more powerful adversaries.

To that end, we present 3 new security notions of increasing strength. Already the weakest of the 3 (*passive* security) captures attacks to which all prior CGKA constructions are vulnerable. Moreover, the 2 stronger (*active* security) notions even allow the adversary to use parties' exposed states combined with full network control to mount attacks. In particular, this is closely related to so-called *insider attacks* which involve malicious group members actively deviating from the protocol. Although insiders are of explicit interest to practical CGKA/SGM designers, our understanding of this class of attackers is still quite nascent. Indeed, we believe ours to be the first security notions in the literature to precisely formulate meaningful guarantees against (a broad class of) insiders.

For each of the 3 new security notions we give a new CGKA scheme enjoying sub-linear (potentially even logarithmic) communication complexity in the number of group members (on par with the asymptotics of state-of-the-art practical constructions). We prove each scheme *optimally* secure, in the sense that the only security violations possible are those necessarily implied by correctness.

1 Introduction

1.1 Overview and Motivation

A *continuous group key agreement* (CGKA) protocol allows a long-lived dynamic group to agree on a continuous stream of fresh secret group keys. In

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

CGKA new parties may join and existing members may leave the group at any point mid-session. In contrast to standard (dynamic) GKA, the CGKA protocols are *asynchronous* in that they make no assumptions about if, when, or for how long members are online.[4] Moreover, unlike, say, broadcast encryption, the protocol may not rely on a (trusted) group manager or any other designated party. Due to a session's potentially very long life-time (e.g., years), CGKA protocols must ensure a property called *post-compromise forward security (PCFS)*. PCFS strengthens the two standard notions of *forward security (FS)* (the keys output must remain secure even if some party's state is compromised in the future) and *post-compromise security (PCS)* (parties recover from state compromise after exchanging a few messages and the keys become secure again) in that it requires them to hold *simultaneously*.

The first CGKA protocol was introduced by Cohn-Gordon et al. in [15] although CGKA as a (term and) generic stand-alone primitive was only later introduced by Alwen et al. in [4]. To motivate the new primitive [4] puts forth the intuition that CGKA abstracts the cryptographic core of an "MLS-like" approach to SGM protocol design in much the same way that CKA (the 2-party analogue of CGKA) abstracts the asymmetric core of a double-ratchet based 2-party secure messaging protocol [1]. Indeed, MLS's computational and communication complexities, support for dynamic groups, it's asynchronous nature, trust assumptions and it's basic security guarantees are naturally inherited from the underlying TreeKEM CGKA sub-protocol. Finally, we believe that the fundamental nature of key agreement and the increasing focus on highly distributed practical cryptographic protocols surely allows for further interesting applications of CGKA beyond SGM.

In [4] the authors analyzed (a version of) the *TreeKEM* CGKA protocol [12]; the core cryptographic component in the *scalable* end-to-end *secure group messaging (SGM)* protocol *MLS*, currently under development by the eponymous Messaging Layer Security working group of the IETF [10].

An SGM protocol is an asynchronous (in the above sense) protocol enabling a dynamic group of parties to privately exchange messages over the Internet. While such protocols initially relied on a service provider acting as a trusted third party, nowadays end-to-end security is increasingly the norm and provider merely act as untrusted delivery services. SGM protocols are expected to provide PCFS for messages (defined analogously to CGKA).[5] The proliferation of SGM protocols in practice has been extensive with more than 2 billion users today.

For both CGKA and SGM, the main bottleneck in scaling to larger groups is the communication and computational complexity of performing a group op-

[4] Instead, the protocol must allow parties that come online to immediately derive all new key material agreed upon in their absence simply by locally processing all protocol messages sent to the group during the interim. Conversely, any operations they wish to perform must be implemented non-interactively by producing a single message to be broadcasted to the group.

[5] As for CGKA, PCFS is *strictly* stronger than the "non-simultaneous" combination of FS and PCS. That is, there are protocols that individually satisfy FS and PCS, but not PCFS [4].

eration (e.g. agree on a new group key, add or remove a party, etc.). Almost all protocols, in particular all those used in practice today, have complexity $\Omega(n)$ for groups of size n (e.g. [24, 20] for sending a message and [26] for removing a party). This is an unfortunate side effect of them being built black-box on top of 2-party secure messaging (SM) protocols. The first (CGKA) protocol to break this mold, thereby achieving "fair-weather" complexity of $O(\log(n))$, is the ART protocol of [15]. Soon to follow were the TreeKEM family of protocols including those in [12, 4, 2] and their variations (implicit) in successive iterations of MLS. By *fair-weather* complexity we informally mean that the cost of the next operation in a session can range from $\Theta(\log(n))$ to $\Theta(n)$ depending on the exact sequence of preceding operations. However, under quite mild assumptions about the online/offline behaviour of participants, the complexity can be kept in the $O(\log(n))$ range.

The Security of CGKA. To achieve PCFS, TreeKEM (and thus MLS) allows a party to perform an "update" operation. These refresh the parties state so as to heal in case of past compromises but come at the price of necessitating a broadcast to the group. The current design of MLS (and, consequently, the analysis by [4]) does not prevent attackers from successfully forging communication from compromised parties in the time period *a state compromise of the party and their next update*. The assumption that attackers won't attempt such forgeries — henceforth referred to as the *cannot-inject assumption (CIA)* — prevents adversaries from, say, *destroying* the group's state by sending maliciously crafted broadcasts. Thus it is closely related to *insider security*, i.e., security against group members who actively deviate from the prescribed protocol, which has hitherto been a mostly open problem and remains an ongoing concern for the MLS working group.[6]

A second assumption that underlies prior work on secure group messaging is the *no-splitting assumption (NSA)*: When multiple parties propose a change to the group state simultaneously, the delivery service (and, hence, the attacker) is assumed to mediate and choose the change initiated by *one* of the parties and deliver the corresponding protocol message to all group members. This (artificially) bars the attacker from splitting the group into subgroups (unaware of each other) and thereby potentially breaking protocol security. As such, the NSA represents a serious limitation of the security model.

Contributions. At a high level, this paper makes 2 types of contributions: (1) we introduce *optimal* CGKA security definitions that avoid the CIA and the NSA, where "optimality" requires that each produced key be secure unless it can be *trivially* — due to the correctness of the protocol — computed using the information leaked to the attacker via corruption; (2) we provide protocols satisfying the proposed definitions. These contributions are discussed in Sections 1.2 and 1.3, respectively.

[6] Note that the Signal [24] 2-party SM protocol is not secure without CIA.

1.2 Defining Optimally Secure CGKA

Overview. This work proposes the first security definitions in the realm of secure group messaging that do not impose any unrealistic restrictions on adversarial capabilities. The definitions allow the adversary to control the communication network, including the delivery service, as well as to corrupt parties by leaking their states and/or controlling their randomness. Furthermore, two settings, called the *passive setting* and the *active setting*, are considered: The passive setting only makes the CIA (but not the NSA) and hence corresponds to a passive network adversary (or authenticated channels). It should be considered a stepping stone to the active setting, where attackers are limited by neither CIA nor NSA. We note that [15, 2, 3] have also considered the setting without the NSA.

While the active setting does not, per se, formally model malicious parties, it does allow the adversary to send arbitrary messages on behalf of parties whose states leaked.[7] Thus, the new security definition goes a long way towards considering the insider attacks mentioned above.

Flexible security definitions. The security definitions in this work are flexible in that several crucial parts of the definitions are generic. Most importantly, following the definitional paradigm of [3], they are parameterized by a so-called *safety predicate*, encoding which keys are expected to be secure in any given execution. *Optimal* security notions are obtained if the safety predicate marks as insecure only those keys that are trivially computable by the adversary due to the correctness of the protocol. While the constructions in this work all achieve optimal security (in different settings), sub-optimal but meaningful security notions may also be of interest (e.g., for admitting more efficient protocols) and can be obtained by appropriately weakening the security predicate.

History Graphs. The central formal tool used to capture CGKA security are so-called *history graphs*, introduced in [3]. A history graph is a symbolic representation of the semantics of a given CGKA session's history. It is entirely agnostic to the details of a construction and depends only on the high-level inputs to the CGKA protocol and the actions of the adversary.

More concretely, a history graph is an annotated tree, in which each node represents a fixed group state (including a group key). A node v is annotated with (the semantics of) the group operations that took place when transitioning from the parent node to v, e.g., "Alice was added using public key epk. Bob was removed. Charlie updated his slice of the distributed group state." The node is further annotated to record certain events, e.g., that bad randomness was used in the transition or that parties' local states leaked to the adversary while they are in group state v. To this end, for each party the history graph maintains a pointer indicating which group state the party is (meant) to be in.

[7] For example, the adversary is allowed to "bypass" the PKI and add new members with arbitrary keys.

Active case: dealing with injections. Probably the greatest challenge in defining security for the *active* setting is how to sensibly model injected messages in a way that maintains consistency with a real world protocol, yet provides interesting security guarantees. In more detail, by using the leaked protocol state of a party and fixing their randomness, the attacker can "run ahead" to predict the exact protocol messages a party will produce for future operations. In particular, it may use an injection to invite new members to join the group at a future history graph node which does not even exist yet in the experiment. Yet, an existing member might eventually catch up to the new member at which point their real world protocols will have consistent states (in particular, a consistent group key).

More fundamentally, the security definition can no longer rely on 2 assumptions which have significantly simplified past security notions (and proofs) for CGKA. Namely, (A) that injections are never accepted by their receiver and (B) that each new protocol message by an honest party always defines a fresh group state (i.e. history graph node).

Hence, to begin modeling injections, we create new "adversarial" history graph nodes for parties to transition to when they join a group by processing an injected message. This means that, in the active setting, the history graph is really a forest, not a tree. We restrict our security experiment to a single "Create Group" operation so there is (at most) 1 tree rooted at a node *not* created by an injection. We call this tree the *honest group* and it is for this group that we want to provide security guarantees.

The above solution is incomplete, as it leaves open the question of how to model delivery of injected protocol messages to members already in a group (honest or otherwise). To this end, the functionality relies on 2 reasonable properties of a protocol:

1. Protocol messages are unique across the whole execution and can be used to identify nodes. This means that any pair of parties that accept a protocol message will agree on all (security relevant) aspects of their new group states, e.g., the group key and group membership.
2. Every protocol message w welcoming a new member to a group in state (i.e., node) v_i must uniquely identify the corresponding protocol message c updating existing group members to v_i.

The net result is that we can now reasonably model meaningful expectations for how a protocol handles injections. In particular, suppose an existing group member id_1 at a node v_1 accepts an injected protocol message c. If another party id_2 already processed c, then we simply move id_1 to the same node as id_2. Otherwise, we check if c was previously assigned to a welcome message w injected to some id_3. If so, we can safely attach the node v_3 created for w as a child of v_1 and transition id_1 to v_3. With the two properties above, we can require that id_1 and id_2 (in the first case) or id_1 and id_3 (the second case) end up in consistent states.

Finally, if neither c nor a matching w has appeared before then we can safely create a fresh "adversarial" node for id_1 as a child of v_1. We give no guarantees for keys in adversarial nodes (as secrecy is anyway inherently lost). Still, we require that they do not affect honest nodes.

Composable and simulation-based security. This work formalizes CGKA security by considering appropriate functionalities in the UC framework [13].Since universal composition is an extremely strong guarantee and seems to be impossible for CGKA in the standard model (for reasons similar to the impossibility of UC-secure key exchange under adaptive corruptions [18]), this work also considers a weaker definition in which, similarly to [7] and [23], the environment is constrained to not perform corruptions that would cause the so-called *commitment problem*. In particular, the weaker statement is still (at least) as strong as a natural game-based definition (as used by related work) that would exclude some corruptions as "trivial wins." In other words, restricting the environment only impacts composition guarantees, which are not the main aspect of this work. Nevertheless, we believe that our statements are a solid indication for multi-group security (see th full version [5] for more discussion).

A simulation-based security notion also provides a neat solution for deciding how adversarially injected packets should affect the history graph. That is, it provides a clean separation of concerns, dealing with the protocol specific aspects in the simulator, while keeping the definition protocol independent.

CGKA functionalities. As mentioned above, the approach taken in this work is to formalize CGKA security in the UC framework via ideal CGKA functionalities, which maintain the history graph as the session evolves. A reader familiar with the use of UC security (in the context of secure multi-party computation) might expect passive and active security to be captured by considering protocol executions over an authenticated and an insecure network, respectively.

As we strive to treat CGKA as a primitive, however, and not directly enforce how it is used, we design our CGKA UC functionalities as "idealized CGKA services" (much in the way that PKE models an idealized PKE service in [13, 14]) instead. Thus, they offer the parties interfaces for performing all group operations, but then simply hand out the corresponding idealized protocol message back to the environment. The attacker gets to choose an arbitrary string to represent the idealized protocol message that would be created for that same operation in the real world. This encodes that no guarantees are made about protocol messages beyond their semantic effects as captured by the history graph.

Just as for PKE, this approach further means that it is up to the environment to "deliver" the idealized messages from the party that initiated an operation to all other group members. This carries the additional benefit that it allows to formalize correctness, whereas typical UC definitions often admit "trivial" protocols simply rejecting all messages (with the simulator not delivering them in the ideal world).

The passive setting is then modeled by restricting the environment to only deliver messages previously chosen to represent a group operation. Meanwhile, in the active setting, the restriction is dropped instead allowing injections; that is delivery of new messages.

Relation to full insider security. We model active corruptions as leaking a party's state, intercepting all their communication, and injecting arbitrary messages on there behalf. While this allows the adversary to emulate the party in essentially every respect it does leave one last capability out of reach to the adversary; namely interactions with the PKI. A malicious insider might, say, register malformed or copied public keys as their own in the PKI. In contrast, an active adversary may not register keys even on behalf of corrupt parties. At most they can leak the secret components of honestly generated key pairs.

While one might conceivably implement such strong PKI in certain real world settings we believe that closing this gap remains an important open problem for future work.

1.3 Protocols with Optimal Security

Overview. We put forth three protocols, all with the same (fair-weather) asymptotic efficiency as the best CGKA protocols in the literature.

Interestingly, even in the passive case, optimal security is not achieved by any existing protocol — not even inefficient solutions based on pairwise channels. Instead, we adapt the "key-evolving" techniques of [21] to the group setting to obtain Protocol P-Pas enjoying optimal security for the passive setting; i.e., against passive but adaptive adversaries.[8]

Next, we augment P-Pas to obtain two more protocols geared to the active setting and meeting incomparable security notions. Specifically, Protocol P-Act provides security against both active and adaptive adversaries but at the cost of a slightly less than ideal "robustness" guarantees. More precisely, the adversary can use leaked states of parties to inject messages that are processed correctly by some parties, but rejected by others.

Meanwhile, the protocol P-Act-Rob uses non-interactive zero-knowledge proofs (NIZKs) to provide the stronger guarantee that if one party accepts a message, then all other parties do but therefore only against active but static adversaries.

For protocols P-Pas and P-Act we prove security with respect to two models. First, in a relaxation of the UC framework with restricted environments (this notion achieves restricted composition and is analogous to game-based notions), we prove security in the non-programmable random oracle model. Second, we prove full UC security in the programmable random oracle model. For the third protocol P-Act-Rob, we consider the standard model, but only achieve semi-static security (the environment is restricted to commit ahead of time to certain information — but not to all inputs).

Techniques used in the protocols. Our protocol P-Pas for the passive setting is an adaptation of the TTKEM protocol, a variant of the TreeKEM protocol introduced in [2] which we have adapted to the propose-and-commit syntax of MLS (draft 9). Next we use hierarchical identity based encryption (HIBE) in

[8] We do place some restrictions on their adaptivity described bellow in the paragraph on the commitment problem.

lieu of regular public-key encryption and ensures that *all* keys are updated with every operation. This helps in avoiding group-splitting attacks, as it ensures that different subgroups use keys for different HIBE identities.

In the active setting, there are two difficulties to solve. First, to prevent injecting messages from uncorrupted parties, we use key-updating signatures [21] that prevent injections using state from another subgroup after a split.

Second, we have to ensure that the adversary cannot use leaked secrets (including signing keys) to craft a message that processed by two parties makes them transition to incompatible states. In other words, a message should prove to a party that any other party processing it ends up in a compatible state. A natural attempt to solve this would be a generic compiler inspired by GMW [19], where the committer provides a non-interactive zero knowledge (NIZK) proof that it executed the protocol correctly. Unfortunately, the GMW approach requires each part to commit to the whole randomness at the beginning of the protocol.[9] which is incompatible with PCS, since healing from corruption requires fresh randomness.

Hence, we instead propose two non-black-box modifications of P-Pas. First, th protocol P-Act uses a simple solution based on a hash function. The mechanism guarantees that all partitions that accept a message also end up with a consistent state. However, parties may not agree on whether to accept or reject the injection. So our second protocol P-Act-Rob implements the consistency using a NIZK proof attached to each message proving its consistency. As a price, we can no longer model a key part of the consistency relation via a random oracle which means our proof technique for adaptive adversaries no longer applies. Thus, for P-Act-Rob, we only prove a type of static security.

1.4 Related Work

2-party ratcheting. 2-party Ratcheting is a similar primitive to CKA (the 2-party analogue of CGKA), both originally designed with secure messaging protocols in mind. (Indeed, the terms are sometimes used interchangeably.)

Ratcheting was first investigated as a stand-alone primitive by Bellare et al. [11]. That work was soon followed by the works of [25] and [21] who considered active security for Ratcheting (the later in the context of an SM protocol). In particular, the work of Poettring and Rösler [25] can be viewed as doing for Ratcheting what our work does for the past CGKA results. In contrast, [16, 22] looked at strong security notions for Ratcheting achievable using practically efficient constructions, albeit at the cost of losing message-loss resilience. In recent work, Balli et al. [8] showed that for such strong security notions imply a weak version of HIBE. Two-party continuous key agreement (CKA) was first defined in [1] where it was used build a family of SM protocols generalizing Signal's messaging protocol [24].

[9] The NIZK is with respect to th committed randomness. The randomness is sampled jointly using an MPC protocol.

CGKA. In comparison to the 2-party primitives, SGM and CGKA have received less attention. In practice, SGM protocols make black-box use of 2-party SM (or at least 2-party Ratcheting) which results in $\Omega(n)$ computational and communication complexity in the group size n for certain operations[17, 20, 24, 26]. The first CGKA with logarithmic fair-weather complexity (defined above) was introduced ART protocol by Cohn-Gordon et al. in [15]. This was soon followed by (several variant of) the TreeKEM CGKA [12]. The RTreeKEM (for "re-randomized TreeKEM") introduced and analyzed in [3] greatly improves the FS properties of TreeKEM and ART. However, security is only proven using both the CIA and NSA and results in a quasi-polynomial loss for adaptive security. Meanwhile, the TTKEM construction (i.e. "Tainted TreeKEM") in [2] has the first adaptive security proof with *polynomial* loss and only uses the CIA (although it does not achieve optimal security). Finally, the CGKA in the current MLS draft [9] represents a significant evolution of the above constructions in that it introduces the "propose and commit" paradigm used in this work and in [4]. Our construction build on TTKEM, RTreeKEM and the propose-and-commit version of TreeKEM.

Modeling CGKA. From a definitional point of view, we build on the history graph paradigm of [3]. That work, in turn, can be seen as a generalization of the model introduced by Alwen et al. [4]. To avoid the commitment problem we adopt the restrictions of environments by Backes et al. [7] to the UC framework. A similar approach has also been used by Jost, Maurer, and Mularczyk [23] in the realm of secure messaging.

2 Continuous Group Key Agreement

2.1 CGKA Schemes

A CGKA scheme aims at providing a steady stream of shared (symmetric) secret keys for a dynamically evolving set of parties. Those two aspects are tied together by so-called *epochs*: each epoch provides a (fresh) group key to a (for this epoch) fixed set of participants. CGKA schemes are *non-interactive* — a party creates a new epoch by broadcasting a single message, which can then be processed by the other members to move along. Rather than relying on an actual broadcast scheme, CCKA schemes however merely assume an untrusted (or partially trusted) delivery service. As multiple parties might try to initiate a new epoch simultaneously, the delivery service's main job is to determine the successful one by picking an order. As a consequence, a party trying to initiate a new epoch itself cannot immediately move forward to it but rather has to wait until its message is confirmed by the delivery service. For simplicity, we assume that the party then just processes it the same way as any other member.

Evolving the member set: add and remove proposals. During each epoch, the parties propose to add or remove members by broadcasting a corresponding *proposal.* To create a new epoch, a party then selects an (ordered) list thereof to be applied. We say that the party *commits* those proposals, and thus call the respective message the *commit message* and the creator thereof the committer.

Group policies. A higher-level application using a CGKA scheme may impose various restrictions on who is allowed to perform which operations (e.g. restricting commits to administrators or restricting valid proposal vectors within a commit). In this work, we consider a very permissive setting. It is easy to see that any result in the permissive setting carries over to a more restrictive setting.

PKI. CGKA schemes in many aspects represent a generalization of non-interactive key exchange (NIKE) to groups. Indeed, adding a new member must be possible without this party participating in the protocol. Rather, the party should be able to join the group by receiving a single *welcome message* that was generated alongside the commit message. Hence, CGKA schemes rely on a PKI that provides some initial key material for new members. This work assumes a simple PKI functionality for this purpose, described in Section 3.

State compromises and forward security. CGKA schemes are designed with exposures of parties' states in mind. In particular, they strive to provide FS: exposing a party's state in some epoch should not reveal the group keys of past epochs. This also implies, that once removed, a party's state should reveal nothing about the group keys.

Post-compromise security and update proposals. In addition, CGKA schemes should also provide PCS. For this, parties regularly send *update proposals*, which roughly suggest removing the sender and immediately adding him with a fresh key (analogous to the one from PKI). In addition, the committer always implicitly updates himself.

2.2 CGKA Syntax

A *continuous group key-agreement scheme* is a tuple of algorithms $\mathsf{CGKA} = (\mathsf{kg}, \mathsf{create}, \mathsf{join}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{commit}, \mathsf{proc}, \mathsf{key})$ with the following syntax. To simplify notation, we assume that all algorithms implicitly know ID of the party running them.

- **Group Creation:** $\gamma \leftarrow \mathsf{create}()$ takes no input and returns a fresh protocol state for a group containing only the user party running the algorithm. In particular, this represents the first *epoch* of a new session.[10]
- **Key Generation:** $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{kg}()$ samples a fresh public/secret key pair (which will be sent to the PKI).
- **Add Proposal:** $(\gamma', p) \leftarrow \mathsf{add}(\gamma, \mathsf{id}_t, \mathsf{pk}_t)$ proposes adding a new member to the group. On input a protocol state, identity of the new member and his public key (generated by kg), it outputs an updated state and *add proposal message*.

[10] To create a group, a party adds the other members using individual add proposals.

- **Remove Proposal:** $(\gamma', p) \leftarrow \mathsf{rem}(\gamma, \mathsf{id}_t)$ proposes removing a member from the group. On input a protocol state and identity, it outputs an updated state and *remove proposal message*.
- **Update Proposal:** $(\gamma', p) \leftarrow \mathsf{upd}(\gamma)$ proposes updating the member's key material. It outputs an updated state and an *update proposal message*.
- **Join A Group:** $(\gamma', \mathsf{roster}, \mathsf{id}_i) \leftarrow \mathsf{join}(\mathsf{sk}, w)$ allows a party with secret key sk (generated by kg) to join a group with a welcome message w. The outputs are: an updated protocol state, a group *roster* (i.e. a set of IDs listing the group members), and the ID of the inviter (i.e. the party that created the welcome message).
- **Commit:** $(\gamma, c, w) \leftarrow \mathsf{commit}(\gamma, \vec{p})$ applies (a.k.a. *commits*) a vector of proposals to a group. The output consists of an updated protocol state, *commit message* and a (potentially empty) *welcome message* (depending on if any add proposal messages where included in $\vec{p}$).[11]
- **Process:** $(\gamma', \mathsf{info}) \leftarrow \mathsf{proc}(\gamma, c, \vec{p})$ processes an incoming commit message and the corresponding proposals to output a *commit info message* info and an updated group state which represents a new epoch in the ongoing CGKA session. The commit info message captures the semantics of the processed commit and it has the form:

  $$\mathsf{info} = (\mathsf{id}, (\mathsf{propSem}_1, \ldots, \mathsf{propSem}_z))$$

  where id is the ID sender of the commit message the vector conveys the semantics of the committed add and remove proposals via triples of the form $\mathsf{propSem} = (\mathsf{id}_s, \mathsf{op}, \mathsf{id}_t)$. Here, id_s denotes the identity of the proposal's sender, $\mathsf{op} \in \{\texttt{"addP"}, \texttt{"remP"}\}$ is the proposal's type and id_t is the identity of the proposal's target (i.e. the partying being added or removed).
- **Get Group Key:** $(\gamma', K) \leftarrow \mathsf{key}(\gamma)$ outputs the current group key for use by a higher-level application, and deletes it from the state.

3 UC Security of CGKA

This section outlines the basic UC security statements of CGKA schemes we use throughout the remaining part of this work. The concrete functionalities $\mathcal{F}_{\text{CGKA-AUTH}}$ and $\mathcal{F}_{\text{CGKA}}$, formalizing the guarantees in the passive and active setting, are then introduced in Sections 4 and 5, respectively.

The CGKA Functionalities. This paper captures security of CGKA schemes by comparing the UC protocol based on CGKA to an ideal functionality. Recall that we model the functionalities as idealized "CGKA services". For example, when a party wishes to commit proposals, it has to input (an idealized version of) those proposals $\vec{p}$ to the functionality. The functionality then outputs an idealized control message c (and potentially a welcome message w), chosen by

[11] For simplicity, we do assume a global welcome message sent to all joining parties, rather than individual ones (which could result in lower overall communication).

the simulator. The functionality does not concern itself with the delivery of control messages c; this must be accomplished by a higher-level protocol.

Our functionalities encode the following basic assumptions: (1) Only group members allowed to create proposals and commit to sequences thereof. (2) We require that every proposal individually makes sense, i.e., a party is only allowed to propose to remove or add a party that is currently in, respectively not in the group. When committing to a sequence of proposals where some are no longer applicable (e.g., due to first including a removal proposal and then one that updates the same party) the offending one is ignored (here the update). More restrictive policies can of course be enforced by the higher-level application making use of the CGKA functionality.

Finally, to simplify definitions, the functionality identify epochs by the control messages c creating them.

PKI. CGKA protocols rely on a service that distributes so-called key bundles used to add new members to the group. (Using the syntax of Section 2, a key bundle is the public key output by kg.) In order not to distract from the main results, this work uses a simplified PKI service that generates one key pair for each identity, making the public keys available to all users. This guarantees to the user proposing to add someone to the group that the new member's key is available, authentic, and honestly generated.

Our PKI is defined by the functionality $\mathcal{F}_{\text{PKI}}$, and our CGKA protocols are analyzed in the $\mathcal{F}_{\text{PKI}}$-hybrid model. Concretely, $\mathcal{F}_{\text{PKI}}$ securely stores key bundle secret keys until fetched by their owner. For a formal description of $\mathcal{F}_{\text{PKI}}$ is presented in the full version [5]. We there also discuss the rationale of our PKI model and how it relates to how comparable PKI are thought of in practice.

CGKA as a UC Protocol. In order to assess the security of CGKA scheme as defined in Section 2 relative to an ideal functionality, the CGKA scheme is translated into a CGKA *protocol* where a user id accepts the following inputs:

- $\mathtt{Create}$**:** If the party is the designated group creator,[12] then the protocol initializes γ using $\mathsf{create}()$.
- $(\mathtt{Propose}, \mathsf{act}), \mathsf{act} \in \{\mathsf{up}, \mathsf{add}\text{-}\mathsf{id}_t, \mathsf{rem}\text{-}\mathsf{id}_t\}$**:** If id is not part of the group, the protocol simply returns $\perp$. Otherwise, it invokes the corresponding algorithm add, rem, or upd using the currently stored state γ. For add, it first fetches pk_t for id_t from $\mathcal{F}_{\text{PKI}}$. The protocol then outputs p to the environment, and stores the updated state γ' (deleting the old one).
- $(\mathtt{Join}, w)$**:** If id is already in the group, the protocol returns $\perp$. Otherwise, it fetches sk and fresh randomness r from $\mathcal{F}_{\text{PKI}}$, invokes $\mathsf{join}(\mathsf{sk}, w; r)$, stores γ, and outputs the remaining results (or an error $\perp$).
- $(\mathtt{Commit}, \vec{p})$ and $(\mathtt{Process}, c, \vec{p})$ and $\mathtt{Key}$**:** If id is not part of the group, the protocol returns $\perp$. Otherwise, it invokes the corresponding algorithm using the current γ, stores γ', and outputs the remaining results (or $\perp$) to the environment.

[12] Formally, the creator is encoded as part of the SID; upon calling $\mathtt{Create}$, a party checks whether it is the designated one, and otherwise just ignores the invocation.

Modeling Corruptions. We start with the (non-standard for UC but common for messaging) corruption model with both continuous state leakage (in UC terms, transient passive corruptions) and adversarially chosen randomness (this resembles the semi-malicious model of [6]). Roughly, we model this in UC as follows. The adversary repeatedly corrupts parties by sending them two types of corruption messages: (1) a message `Expose` causes the party to send its entire state to the adversary (once), (2) a message (`CorrRand`, b) sets the party's rand-corrupted flag to b. If this flag is set, the party's randomness-sampling algorithm is replaced by asking the adversary to choose the random values. Ideal functionalities are activated upon corruptions and can adjust their behavior accordingly. We give a formal description of the corruption model in the full version [5].

Restricted Environments. Recall that in the passive setting we assume that the adversary does not inject messages, which corresponds to authenticated network. However, with the above modeling, one obviously cannot assume authenticated channels. Instead, we consider a weakened variant of UC security, where statements quantify over a restricted class of *admissible* environments, e.g. those that only deliver control messages outputted by the CGKA functionality, and provide no guarantees otherwise. Whether an environment is admissible or not is defined by the ideal functionality $\mathcal{F}$. Concretely, the pseudo-code description of $\mathcal{F}$ can contain statements of the form **req** *cond* and an environment is called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such *cond* when interacting with $\mathcal{F}$. See the full version [5] for a formal definition.

Apart from modeling authenticated channels, we also use this mechanism to avoid the so-called commitment problem (there, we restrict the environment not to corrupt parties at certain times, roughly corresponding to "trivial wins" in the game-based language). We always define two versions of our functionalities, with and without this restriction.

4 Security of CGKA in the Passive Setting

The history graph. CGKA functionalities keep track of group evolution using so-called *history graphs* (cf. Fig. 1), a formalism introduced in [3]. The nodes in a history graph correspond either to group creation, to commits, or to proposals. Nodes of the first two categories correspond to particular group states and form a tree. The root of the tree is a (in fact, the only) group-creation node, and each commit node is a child of the node corresponding to the group state from which it was created. Similarly, proposal nodes point to the commit node that corresponds to the group state from which they created.

Any commit node is created from a (ordered) subset of the proposals of the parent node; which subset is chosen is up to the party creating the commit. Observe that it is possible for commit nodes to "fork," which happens when parties simultaneously create commits from the same node.

For each party, the functionality also maintains a pointer $\mathsf{Ptr}[\mathsf{id}]$ indicating the current group state of the party. This pointer has two special states: before

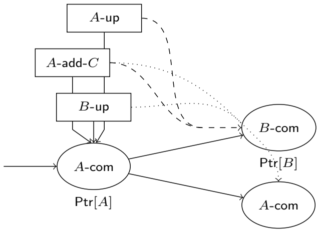

Fig. 1: A graphical representation of a history graph with three commit nodes (circles) and proposal nodes (rectangles), respectively.

joining the pointer is set to fresh and after leaving the group to removed. Note that a party's pointer does not move upon creation of a new commit node. Rather, the pointer is only moved once the corresponding control message is input by the party. This models, e.g., the existence of a delivery service that resolves forks by choosing between control messages that correspond to nodes with the same parent.[13]

CGKA functionalities identify commit resp. proposal nodes by the corresponding (unique) control messages c resp. proposal messages p (chosen by the simulator). The arrays $\mathsf{Node}[\cdot]$ resp. $\mathsf{Prop}[\cdot]$ map control messages c resp. proposal messages p to commit resp. proposal nodes. Moreover, for a welcome message w, array $\mathsf{Wel}[w]$ stores the commit node to which joining the group via w leads. Nodes in the history graph store the following values:

- orig: the party whose action created the node
- par: the parent commit node
- stat $\in \{\mathsf{good}, \mathsf{bad}\}$: a status flag indicating whether secret information corresponding to the node is known to the adversary (e.g., by having corrupted its creator or the creator having used bad randomness).

Proposal nodes further store the following value:

- lbl $\in \{\mathsf{up}, \mathsf{add\text{-}id}', \mathsf{rem\text{-}id}'\}$: the proposed action

Commit nodes further store the following values:

- pro: the ordered list of committed proposals,
- mem: the group members,
- key: the group key,
- chall: a flag set to `true` if a random group key has been generated for this node, and to `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 key leaked.

[13] Note, however, that such behavior is not imposed by the functionality; it is entirely possible that group members follow different paths.

The CGKA functionality $\mathcal{F}_{\text{CGKA-AUTH}}$. The remainder of this section introduces and explains functionality $\mathcal{F}_{\text{CGKA-AUTH}}$, which deals with *passive* network adversaries, i.e., adversaries who do not create their own control messages (nor proposals) and who deliver them in the correct order. It is described in Fig. 2 with some bookkeeping functions outsourced to Fig. 3.

Interaction with parties. The inputs with which uncorrupted parties interact with $\mathcal{F}_{\text{CGKA-AUTH}}$ are described first; the boxed content in Fig. 2 is related to corruption and described later. Initially, the history graph is empty and the only possible action is for a designated party $\mathsf{id}_{\mathsf{creator}}$ to create a new group with itself in it.

The input `Propose` allows parties to create new proposals. $\mathcal{F}_{\text{CGKA-AUTH}}$ ensures that only parties that are currently in the group can create proposals (line [a]). Recall that the proposal identifier p is chosen by the simulator (line [b]) but guaranteed to be unique (line [c]). The identifier is returned to the calling party.

Parties create new commits using the input `Commit`. As part of the input, the calling party has to provide an ordered list of proposals to commit to. All proposals have to be well-defined, belong to the party's current commit node, and are valid with respect to its member set (line [d]). Moreover, a party is not allowed to commit to a proposal that removes the party from the group (line [e]). Once more, the simulator chooses the identifier c for the commit, and, if a new party is added in one of the proposals, the attacker also choses the welcome message w (line [b]). Both c and w must be unique (line [c]).

A current group member can move their pointer to a child node c of their current state by calling $(\texttt{Process}, c, \vec{p})$ (in case the proposals $\vec{p}$ in c removes the group member, their pointer is set to $\perp$ instead). The functionality ensures a party always inputs the correct proposal array (line [d]). Moreover, it imposes *correctness*: while the simulator is notified of the action (line [f]), the pointer is moved to c and the helper **get-output-process** returns the proposals true interpretations irrespective of the simulator's actions.

A new member can join the group at node $\mathsf{Wel}[w]$ via $(\texttt{Join}, w)$. The value $\mathsf{Wel}[w]$ must exist and correspond to a commit node for which the calling party is in the group (line [g]).

Finally, `Key` outputs the group key for the party's current node. The keys are selected via the function **set-key**(c), which either returns a random key or lets the simulator pick the key if information about it has been leaked due to corruption or the use of bad randomness (see below).

Corruptions and bad randomness. Generally, keys provided by $\mathcal{F}_{\text{CGKA-AUTH}}$ are always uniformly random and independent unless the information the adversary has obtained via corruption would trivially allow to compute them (as a consequence of protocol correctness). In order to stay on top of this issue, the functionality must do some bookkeeping, which is used by the predicate **safe** to determine whether a key would be known to the adversary.

First, when a party id is exposed via $(\texttt{Expose}, \mathsf{id})$, the following from id's state that becomes available to the adversary:

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

The group creator $\mathsf{id}_{\mathsf{creator}}$ is encoded as part of sid. The functionality is parameterized in:

- the predicate **safe**, specifying which keys are leaked via corruptions
- the flag **restrict-corruptions**, denoting whether it provides full adaptive security.

```
Initialization
    $\mathsf{Ptr}[\cdot] \leftarrow \mathsf{fresh}$
    $\mathsf{Node}[\cdot], \mathsf{Prop}[\cdot], \mathsf{Wel}[\cdot] \leftarrow \bot$
    $\boxed{\mathrm{RndCor}[\cdot], \mathsf{RndPool}[\cdot] \leftarrow \mathsf{good}}$
    $\mathsf{HasKey}[\cdot] \leftarrow \mathtt{false}$

Inputs from $\mathsf{id}_{\mathsf{creator}}$
Input Create
    if $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] \neq \mathsf{fresh}$ then return $\bot$
    $\boxed{\mathsf{stat} \leftarrow \textbf{rand-stat}(\mathsf{id}_{\mathsf{creator}})}$
    $\mathsf{Node}[\epsilon] \leftarrow \textbf{create-root}(\mathsf{id}_{\mathsf{creator}}, \boxed{\mathsf{stat}})$
    $\mathsf{HasKey}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \mathtt{true}$
    $\mathsf{Ptr}[\mathsf{id}_{\mathsf{creator}}] \leftarrow \epsilon$

Inputs from a party id

Input $(\mathtt{Propose}, \mathsf{act}), \mathsf{act} \in \{\mathsf{up}, \mathsf{add}\text{-}\mathsf{id}', \mathsf{rem}\text{-}\mathsf{id}'\}$
 a: if $\mathsf{Ptr}[\mathsf{id}] \in \{\mathsf{fresh}, \mathsf{removed}\}$ then return $\bot$
 b: Send $(\mathtt{Propose}, \mathsf{id}, \mathsf{act})$ to the adversary and receive $p$.
 c: assert $\mathsf{Prop}[p] = \bot$
    $\boxed{\mathsf{stat} \leftarrow \mathsf{good}}$
    $\boxed{\textbf{if } \mathsf{act} = \mathsf{up} \textbf{ then}}$
        $\boxed{\mathsf{stat} \leftarrow \textbf{rand-stat}(\mathsf{id})}$
    $\mathsf{Prop}[p] \leftarrow \textbf{create-prop}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \mathsf{act}, \boxed{\mathsf{stat}})$
    return $p$

Input $(\mathtt{Commit}, \vec{p})$
 a: if $\mathsf{Ptr}[\mathsf{id}] \in \{\mathsf{fresh}, \mathsf{removed}\}$ then return $\bot$
 d: req $\forall p \in \vec{p} : (\mathsf{Prop}[p] \neq \bot \wedge \textbf{valid-proposal}(c, p))$
    $\mathsf{mem} \leftarrow \textbf{members}(\mathsf{Ptr}[\mathsf{id}], \vec{p})$
 e: req $\mathsf{id} \in \mathsf{mem}$
 b: Send $(\mathtt{Commit}, \mathsf{id}, \vec{p})$ to the adversary and receive $(c, w)$.
 c: assert $\mathsf{Node}[c] = \bot$
    $\boxed{\mathsf{stat} \leftarrow \textbf{rand-stat}(\mathsf{id})}$
    $\mathsf{Node}[c] \leftarrow \textbf{create-child}(\mathsf{Ptr}[\mathsf{id}], \mathsf{id}, \vec{p}, \mathsf{mem}, \boxed{\mathsf{stat}})$
    assert $w \neq \bot$ iff $(\mathsf{mem} \setminus \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}) \neq \varnothing$
    if $w \neq \bot$ then
 c:     assert $\mathsf{Wel}[w] = \bot$
        $\mathsf{Wel}[w] \leftarrow c$
    return $(c, w)$

Input Key
 a: if $\mathsf{Ptr}[\mathsf{id}] \in \{\mathsf{fresh}, \mathsf{removed}\} \vee \neg\mathsf{HasKey}[\mathsf{id}]$ then
        return $\bot$
    if $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key} = \bot$ then
        $\textbf{set-key}(\mathsf{Ptr}[\mathsf{id}])$
    $\mathsf{HasKey}[\mathsf{id}] \leftarrow \mathtt{false}$
    return $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{key}$

Input $(\mathtt{Process}, c, \vec{p})$
 a: if $\mathsf{Ptr}[\mathsf{id}] \in \{\mathsf{fresh}, \mathsf{removed}\}$ then return $\bot$
 d: req $\mathsf{Node}[c] \neq \bot \wedge \mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}]$
        $\wedge\ \mathsf{Node}[c].\mathsf{pro} = \vec{p}$
 f: Send $(\mathtt{Process}, \mathsf{id}, c, \vec{p})$ to the adversary.
    if $\exists p \in \vec{p} : \mathsf{Prop}[p].\mathsf{act} = \mathsf{rem}\text{-}\mathsf{id}$ then
        $\mathsf{Ptr}[\mathsf{id}] \leftarrow \mathsf{removed}$
    else
        $\mathsf{Ptr}[\mathsf{id}] \leftarrow c$
        $\boxed{\textbf{rand-stat}(\mathsf{id})}$
        $\mathsf{HasKey}[\mathsf{id}] \leftarrow \mathtt{true}$
    return $\textbf{get-output-process}(c)$

Input $(\mathtt{Join}, w)$
    if $\mathsf{Ptr}[\mathsf{id}] \notin \{\mathsf{fresh}, \mathsf{removed}\}$ then return $\bot$
    $c \leftarrow \mathsf{Wel}[w]$
 g: req $c \neq \bot \wedge \mathsf{Node}[c] \neq \bot \wedge \mathsf{id} \in \mathsf{Node}[c].\mathsf{mem}$
    Send $(\mathtt{Join}, \mathsf{id}, w)$ to the adversary
            and receive $ack$.
    if $\mathsf{Ptr}[\mathsf{id}] = \mathsf{fresh} \vee ack$ then
        $\mathsf{Ptr}[\mathsf{id}] \leftarrow c$
        $\boxed{\textbf{rand-stat}(\mathsf{id})}$
        $\mathsf{HasKey}[\mathsf{id}] \leftarrow \mathtt{true}$
        return $\textbf{get-output-join}(c)$
    else
        return $\bot$

Corruptions

Input $(\mathtt{Expose}, \mathsf{id})$
    if $\mathsf{Ptr}[\mathsf{id}] \in \{\mathsf{fresh}, \mathsf{removed}\}$ then
        return
    $\mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{exp} \leftarrow \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{exp}$
                    $\cup \{(\mathsf{id}, \mathsf{HasKey}[\mathsf{id}])\}$
    $\textbf{update-status-after-expose}(\mathsf{id})$
    $\mathsf{RndPool}[\mathsf{id}] \leftarrow \mathsf{bad}$
    if restrict-corruptions then
        req $\forall c$, if $\mathsf{Node}[c].\mathsf{chall} = \mathtt{true}$ then $\textbf{safe}(c)$
    else
        Send to the adversary
                $\{(c, \mathsf{Node}[c].\mathsf{key}) : \neg\textbf{safe}(c)\}$.

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

Fig. 2: The ideal CGKA functionality for the passive setting. The behavior related to corruptions is marked in $\boxed{\text{boxes}}$. The helper functions are defined in Fig. 3 and the optimal predicate **safe** used in this paper is defined in Fig. 4.

Helper Functions

```
helper create-child($c$, id, $\vec{p}$, mem, stat)
    return new node with par ← $c$, orig ← id, pro ← $\vec{p}$,
        mem ← mem, stat ← stat.

helper create-root(id, stat)
    return new node with par ← $\perp$, orig ← id, pro ← (),
        mem ← {id}, stat ← stat.

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

helper members($c$, $\vec{p}$)
    $(G, \cdot)$ ← apply-proposals($c$, $\vec{p}$)
    return $G$

helper get-output-process($c$)
    $(\cdot$, propSem$)$ ← apply-proposals($c$, Node[$c$].pro)
    return (Node[$c$].orig, propSem)

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

helper apply-proposals($c$, $\vec{p}$)
    $G$ ← Node[$c$].mem; $P$ ← ()
    for $p \in \vec{p}$ do
        if Prop[$p$].act = add-id′ $\wedge$ id′ $\notin G$ then
            $P \leftarrow P \parallel$ (Prop[$p$].orig, Prop[$p$].act)
            $G \leftarrow G \cup$ {id′}
        else if Prop[$p$].act = rem-id′ $\wedge$ id′ $\in G$ then
            $P \leftarrow P \parallel$ (Prop[$p$].orig, Prop[$p$].act)
            $G \leftarrow G \setminus$ {id′}
    return $(G, P)$

helper valid-proposal($c$, $p$)
    return Prop[$p$].par = $c$ $\wedge$ Prop[$p$].orig $\in$ Node[$c$].mem
        $\wedge$ $\neg$(Prop[$p$].act = add-id′ $\wedge$ id′ $\in$ Node[$c$].mem)
        $\wedge$ $\neg$(Prop[$p$].act = rem-id′ $\wedge$ id′ $\notin$ Node[$c$].mem)

helper set-key($c$)
    if $\neg$safe($c$) then
        Send (Key, id) to the adversary and receive $I$.
        Node[$c$].key ← $I$
        Node[$c$].chall ← false
    else
        Node[$c$].key ←$ $\mathcal{I}$
        Node[$c$].chall ← true

helper rand-stat(id)
    if RndPool[id] = good $\vee$ RndCor[id] = good then
        RndPool[id] ← good
        return good
    else
        return bad

helper update-status-after-expose(id)
    for each $p$ s.t. Prop[$p$] $\neq \perp$ and
        (a) Prop[$p$].par = Ptr[id] and
        (b) Prop[$p$].orig = id and
        (c) Prop[$p$].act = up
    do Prop[$p$].stat ← bad
    for each $c$ s.t. Node[$c$] $\neq \perp$ and
        (a) Node[$c$].par = Ptr[id] and
        (b) Node[$c$].orig = id
    do Node[$c$].stat ← bad
```

Fig. 3: The helper functions for the CGKA functionality, defined in Fig. 2. The behavior related to corruptions is marked in boxes.

Predicate safe

```
helper safe($c$)
    return $\neg\exists$id s.t. key-and-state-leaked(id, $c$)

helper key-and-state-leaked(id, $c$)
    return true iff id $\in$ Node[$c$].mem and either
        (a) (id, true) $\in$ Node[$c$].exp or
        (b) state-leaked(id, Node[$c$].par) $\wedge$ $\neg$heals(id, $c$)
    or
        (c) Node[$c$].par = $\perp$ or
        (d) last $p \in$ Node[$c$].pro s.t. Prop[$p$].act = add-id
    and
            (if there exists one) has Prop[$p$].stat = adv

helper state-leaked(id, $c$)
    return true iff id $\in$ Node[$c$].mem and either
        (a) (id, false) $\in$ Node[$c$].exp or
        (b) key-and-state-leaked(id, $c$)

helper heals(id, $c$)
    if Node[$c$].orig = id $\wedge$ Node[$c$].stat = good then
        return true
    else Let $p$ be the last element in Node[$c$].pro s.t.
        Prop[$p$].orig = id $\wedge$ Prop[$p$].act = up
        (return false if no such $p$ exists)
        return Prop[$p$].stat = good
```

Fig. 4: The predicate **safe**, that determines if the key in a node c is secure. The part in the box is only relevant in the active setting (Section 5).

- Any key material id stored locally in order to process future control messages.
- The current group key, if id has not retrieved it yet via $\mathtt{Key}$. The flag $\mathsf{HasKey}[\mathsf{id}]$ indicates if id currently holds the key.
- The key material for update proposals and commits that id has created from its current epoch (but not processed yet).

The functionality records this symbolically as follows: the pair $(\mathsf{id}, \mathsf{HasKey}[\mathsf{id}])$ is added to the "corrupted set" exp of id's current node. To address the third point, the helper function **update-status-after-expose**(id) sets the status of all child nodes (update proposals and commits) created by id to $\mathsf{stat} = \mathsf{bad}$, i.e., they are marked as no longer healing the party.

The second avenue for the attacker to obtain information about group keys is when the parties use bad randomness. Note that this work assumes that CGKA schemes use their own randomness pool, which is refreshed with randomness from the underlying operating system (OS) before every use. This guarantees that a party uses good randomness whenever (1) the OS supplies random values or (2) the pool is currently random (from the attacker's perspective).

In $\mathcal{F}_{\text{CGKA-AUTH}}$, the flag RndCor[$\mathsf{id}$] records for each party id whether id's OS currently supplies good randomness; the flag can be changed by the adversary at will via $\mathtt{CorrRand}$. Moreover, for each party id, the functionality stores the status of its randomness pool in $\mathsf{RndPool}[\mathsf{id}]$. Whenever id executes a randomized action, the functionality checks whether id uses good randomness by calling **rand-stat**(id) and stores the result as the stat flag of the created history graph node. As a side effect, **rand-stat**(id) updates the pool status to good if good fresh OS randomness is used.

The safety predicate. The predicate **safe**(c) is defined as follows: The key corresponding to c is secure if and only if it has not been exposed via one of the parties. This can happen in two situations: either if the party's state has been exposed in this particular state c while the party still stored the key ($(\mathsf{id}, \mathtt{true}) \in \mathsf{Node}[c].\mathsf{exp}$), or its previous state (not necessary with the key) is known to the adversary and c did not heal the party. This can also be interpreted as a reachability condition: the key is exposed if the party has been corrupted in any ancestor of c and there is no "healing" commit in between.

The commit c is said to be healing, iff it contains an update by id with good randomness or id is the committer and used good randomness. Observe that this is optimal as those are the only operations, that by definition of a CGKA scheme, are supposed to affect the party's own key material.

Adaptive corruptions. Exposing a party's state may trigger some keys that were already output as secure (i.e., random) now to become insecure. Unfortunately, this is a stereotypical situation of the so-called commitment problem[14] of simulation-based security. Hence, we define two variants of $\mathcal{F}_{\text{CGKA-AUTH}}$, which

[14] Roughly, the simulator, having already outputted a commit message that "binds" him to the group key, now has to produce a secret state, such that processing this message results in the (random) key from the functionality.

differ in the behavior upon exposure (see the part in the dashed box) — in the weaker notion (**restrict-corruptions** $=$ `true`), the environment is restricted not to corrupt a party if it would cause a challenged key to become insecure, while in the stronger notion the adversary is simply given all now insecure keys.

5 Security of CGKA in the Active Setting

This section introduces the functionality $\mathcal{F}_{\text{CGKA}}$, which deals with active network adversaries, i.e., it allows the environment to input arbitrary messages. It is defined in Fig. 5, and the differences from $\mathcal{F}_{\text{CGKA-AUTH}}$ are marked in $\boxed{\text{boxes.}}$

On a high level, the main difficulty compared to the passive setting is that $\mathcal{F}_{\text{CGKA}}$ has to account for inherent injections of valid control messages, where the adversary uses leaked states of parties. To this end, $\mathcal{F}_{\text{CGKA}}$ marks history graph nodes created by the adversary via injections by a special status flag $\mathsf{stat} = \mathsf{adv}$. It maintains the following *history graph invariant*, formally defined in Fig. 7:

1. Adversarially created nodes only occur if inherent, that is, their (claimed) creator's state must have leaked in the parent node. (We explain the special case of creating orphan nodes later.)
2. The history graph is consistent.

The invariant is checked at the end of every action potentially violating it (cf. lines [g]). We now describe the changes and additional checks in more detail.

Injected proposals and commits. First, consider the case where a party calls commit with an injected proposal p (i.e., $\mathsf{Prop}[p] = \bot$). In such case, the simulator is allowed to reject the input (if it is invalid) by sending $\mathit{ack} = \texttt{false}$. Otherwise, $\mathcal{F}_{\text{CGKA}}$ asks the simulator to interpret the missing proposals by providing properties (action etc.) for new proposal nodes (line [d]) and marks them as adversarial by setting their status to adv. (Those interpretations must be valid with respect to the corresponding actions, cf. line [e], as otherwise the simulator must reject the input.) The behavior of $\mathcal{F}_{\text{CGKA}}$ in case a party calls process with an injected commit or proposal message is analogous, except that the simulator also interprets the commit message, creating a new commit node (line [h]).

While in the authentic-network setting we could enforce that each honest propose and commit call results in a unique proposal or commit message, this is no longer the case when taking injections into account. For example, add proposals are deterministic, so if the adversary uses a leaked state to deliver an add proposal p, then the next add proposal computed by the party is p as well. The same can happen with randomized actions where the adversary controls the randomness. Accordingly, we modify the behavior of $\mathcal{F}_{\text{CGKA}}$ on propose and commit inputs and allow outputting messages corresponding to existing nodes, as long as this is consistent. That is, in addition to the invariant, $\mathcal{F}_{\text{CGKA}}$ at this point also needs to enforce that the values stored as part of the preexisting node correspond to the intended action, and that this does not happen for randomized actions with fresh randomness (see lines [a]). If all those checks succeed, the node is treated as non-adversarial and we adjust its status accordingly (see lines [b]).

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

The functionality expects as part of the instance's session identifier sid the group creator's identity $\text{id}_{\text{creator}}$. It is parameterized in:

- the predicate **safe**, specifying which keys are leaked via corruptions
- the flag **restrict-corruptions**, indicating if it restricts the environment (avoiding the commitment problem), or if it provides full adaptive security
- the flag **robust**, indicating that parties must be able to process "honest" messages.

Initialization and input Create from $\text{id}_{\text{creator}}$

This is the same as in $\mathcal{F}_{\text{CGKA-AUTH}}$ in Fig. 2.

Inputs from a party id

```
Input (Propose, act), $act \in \{up, add\text{-}id', rem\text{-}id'\}$
      if $Ptr[id] \in \{fresh, removed\}$ then return $\bot$
      Send (Propose, id, act) to the adversary; receive $p$.
      stat $\leftarrow$ good
      if act = up then
          stat $\leftarrow$ rand-stat(id)
      if $Prop[p] = \bot$ then
          $Prop[p] \leftarrow$ create-prop(Ptr[id], id, act, stat)
      else
a:        check-prop-consistency($p$, id, act, stat)
b:        $Prop[p]$.stat $\leftarrow$ stat
      return $p$

Input (Commit, $\vec{p}$)
      if $Ptr[id] \in \{fresh, removed\}$ then return $\bot$
      Send (Commit, id, $\vec{p}$) to the adversary
                and receive ($ack$, $c$, $w$).
c:    if valid-comm-by-correctness(id, $\vec{p}$) $\vee$ $ack$ then
d:        fill-proposals(id, $\vec{p}$)
e:        $\forall p \in \vec{p}$: assert valid-proposal(Ptr[id], $p$)
          stat $\leftarrow$ rand-stat(id)
          mem $\leftarrow$ members(Ptr[id], $\vec{p}$)
          assert id $\in$ mem
          if $Node[c] = \bot$ then
              $Node[c] \leftarrow$ create-child(Ptr[id], id, $\vec{p}$, mem, stat)
          else
a:            check-comm-consistency($c$, id, $\vec{p}$, stat, mem)
              $Node[c]$.stat $\leftarrow$ stat
f:            if $Node[c].par = \bot$ then attach($c$, id, $\vec{p}$)
          assert $w \neq \bot$ iff $(mem \setminus Node[Ptr[id]].mem) \neq \varnothing$
          if $w \neq \bot$ then
              assert $Wel[w] \in \{\bot, c\}$
              $Wel[w] \leftarrow c$
g:        assert invariant
          return $(c, w)$
      else return $\bot$

Input Key
      if $Ptr[id] \in \{fresh, removed\} \vee \neg HasKey[id]$ then
          return $\bot$
      if $Node[Ptr[id]].key = \bot$ then
          set-key(Ptr[id])
      HasKey[id] $\leftarrow$ false
      return Node[Ptr[id]].key

Input (Process, $c$, $\vec{p}$)
      if $Ptr[id] \in \{fresh, removed\}$ then return $\bot$
      Send (Process, id, $c$, $\vec{p}$) to the adversary
                and receive ($ack$, orig$'$).
c:    if valid-proc-by-correctness(id, $c$, $\vec{p}$) $\vee$ $ack$ then

d:        fill-proposals(id, $\vec{p}$)
e:        $\forall p \in \vec{p}$: assert valid-proposal(Ptr[id], $p$)
          mem $\leftarrow$ members(Ptr[id], $\vec{p}$)
          if $Node[c] = \bot$ then
h:            $Node[c] \leftarrow$ create-child(
                        Ptr[id], orig$'$, $\vec{p}$, mem, adv)
          else
i:            check-valid-successor($c$, id, $\vec{p}$, mem)
f:            if $Node[c].par = \bot$ then attach($c$, id, $\vec{p}$)
          if $\exists p \in \vec{p}$ : $Prop[p]$.act = rem-id then
              Ptr[id] $\leftarrow$ removed
          else
              $Ptr[id] \leftarrow c$
              rand-stat(id)
              HasKey[id] $\leftarrow$ true
g:        assert invariant
          return get-output-process($c$)
      else return $\bot$

Input (Join, $w$)
      if $Ptr[id] \neq \{fresh, removed\}$ then return $\bot$
      Send (Join, id, $w$) to the adversary
                and receive ($ack$, $c'$, orig$'$, mem$'$).
c:    if valid-join-by-correctness(id, $w$) $\vee$ $ack$ then
          $c \leftarrow Wel[w]$
          if $c = \bot$ then
              $c \leftarrow c'$
j:            $Wel[w] \leftarrow c$
          if $Node[c] = \bot$ then
k:            $Node[c] \leftarrow$ create-child($\bot$, orig$'$, $\bot$, mem$'$, adv)
          $Ptr[id] \leftarrow c$
          rand-stat(id)
          HasKey[id] $\leftarrow$ true
g:        assert invariant
          return get-output-join($c$)
      else return $\bot$
```

Corruptions

This is the same as in $\mathcal{F}_{\text{CGKA-AUTH}}$ in Fig. 2.

Fig. 5: The ideal CGKA functionality for the active setting. The behavior related to injections is marked in boxes. The corresponding helper functions are defined in Figs. 3 and 6, the invariant in Fig. 7, and the optimal predicate **safe** in Fig. 4.

Helper Functions

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

helper check-prop-consistency($p, \mathsf{id}, \mathsf{act}, \mathsf{stat}$)
  // Preexisting $p$ valid for id proposing act?
  assert $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Prop}[p].\mathsf{act} = \mathsf{act}$
          $\wedge\ \mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[\mathsf{id}] \wedge \mathsf{Prop}[p].\mathsf{stat} = \mathsf{adv}$
  if $\mathsf{act} = \mathsf{up}$ then
      assert $\mathsf{stat} \neq \mathsf{good}$

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

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

helper attach($c, \mathsf{id}, \vec{p}$)
  // Attach (detached) node $c$ as successor of id's current
     node with proposals $\vec{p}$
  $\mathsf{Node}[c].\mathsf{par} \leftarrow \mathsf{Ptr}[\mathsf{id}]$; $\mathsf{Node}[c].\mathsf{pro} \leftarrow \vec{p}$

helper valid-comm-by-correctness($\mathsf{id}, \vec{p}$)
  // Does correctness enforce the commit-call to succeed?
  return $\forall p \in \vec{p} : \mathsf{Prop}[p] \neq \bot\ \wedge$ valid-proposal($\mathsf{Ptr}[\mathsf{id}], p$)
           $\wedge\ \mathsf{id} \in$ members($\mathsf{Ptr}[\mathsf{id}], \vec{p}$)

helper valid-proc-by-correctness($\mathsf{id}, c, \vec{p}$)
  // Does correctness enforce the process-call to succeed?
  if $\mathsf{Node}[c] = \bot \vee \mathsf{Node}[c].\mathsf{par} \neq \mathsf{Ptr}[\mathsf{id}] \vee \mathsf{Node}[c].\mathsf{pro} \neq \vec{p}$
  then return false
  else if robust then
      return $\neg(\mathsf{Node}[c].\mathsf{orig} = \mathsf{id} \wedge \mathsf{Node}[c].\mathsf{stat} = \mathsf{adv})$
            $\wedge\ \neg\big($last $p \in \vec{p}$ s.t. $\mathsf{Prop}[p].\mathsf{orig} = \mathsf{id} \wedge$
              $\mathsf{Prop}[p].\mathsf{act} = \mathsf{up}$ exists and has $\mathsf{Prop}[p].\mathsf{stat} = \mathsf{adv}\big)$
  else
      return $\mathsf{Node}[c].\mathsf{stat} \neq \mathsf{adv} \wedge\ \forall p \in \vec{p} : \mathsf{Prop}[p].\mathsf{stat} \neq \mathsf{adv}$

helper valid-join-by-correctness($\mathsf{id}, w$)
  // Does correctness enforce the join-call to succeed?
  $c \leftarrow \mathsf{Wel}[w]$
  return robust $\wedge\ \mathsf{Ptr}[\mathsf{id}] = \mathsf{fresh} \wedge c \neq \bot \wedge \mathsf{Node}[c] \neq \bot$
           $\wedge\ \mathsf{Node}[c].\mathsf{stat} \neq \mathsf{adv}$
           $\wedge\ \mathsf{id} \in (\mathsf{Node}[c].\mathsf{mem} \setminus \mathsf{Node}[c].\mathsf{par}.\mathsf{mem})$
```

Fig. 6: The additional helper functions for $\mathcal{F}_{\text{CGKA}}$, defined in Fig. 5.

Injected welcome messages. If a party calls join with an injected welcome message, we again ask the simulator to interpret the injected welcome message by providing the corresponding commit message c (line [j]), which can either refer to an existing node or a new one the simulator is allowed to set the corresponding values for (line [k]). The main difficulty compared to injected proposals and commits, however, is that sometimes this node's position in the history graph cannot be determined. For example, consider an adversary who welcomes a party id to a node at the end of a path that he created in his head, by advancing the protocol a number of steps from a leaked state. Unless welcome messages contain the whole conversation history (and not just e.g. a constant size transcript hash thereof), it is impossible for any efficient simulator to determine the path.

As a result, $\mathcal{F}_{\text{CGKA}}$ deals with an injected welcome message w as follows: if the commit node to which w leads does not exist (c is provided by the simulator), then a new detached node is created, with all values (except parent and proposals) determined by the simulator. The new member can call propose, commit and process from this detached node as from any other node, creating an alternative history graph rooted at the detached node. Moreover, new members can join the alternative graph. The node, together with its alternative subtree, can be attached to the main tree when the commit message c is generated or successfully

Predicate invariant

Return `true` if all of the following are true:

- *adversarial nodes created only by corrupted parties:*
  - $\forall c$, if $\mathsf{Node}[c].\mathsf{stat} = \mathsf{adv}$ then **state-leaked**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{Node}[c].\mathsf{orig})$ or $\mathsf{Node}[c].\mathsf{par} = \bot$
  - $\forall p$, if $\mathsf{Prop}[p].\mathsf{stat} = \mathsf{adv}$ then **state-leaked**$(\mathsf{Prop}[p].\mathsf{par}, \mathsf{Prop}[p].\mathsf{orig})$
- *the history graph's state is consistent:* $\forall c$ s.t. $\mathsf{Node}[c].\mathsf{par} \neq \bot$:
  - $\mathsf{Node}[c].\mathsf{pro} \neq \bot$ and $\forall p \in \mathsf{Node}[c].\mathsf{pro}\ \mathsf{Prop}[p].\mathsf{par} = \mathsf{Node}[c].\mathsf{par}$
  - $\mathsf{Node}[c].\mathsf{mem} =$ **members**$(\mathsf{Node}[c].\mathsf{par}, \mathsf{Node}[c].\mathsf{pro})$
- *pointers consistent:* $\forall \mathsf{id}$ s.t. $\mathsf{Ptr}[\mathsf{id}] \notin \{\mathsf{fresh}, \mathsf{removed}\} : \mathsf{id} \in \mathsf{Node}[\mathsf{Ptr}[\mathsf{id}]].\mathsf{mem}$
- *the graph contains no cycles*

Fig. 7: The graph invariant. The predicate **state-leaked** is defined as part of the predicate **safe** in Fig. 4.

processed by a party from the main tree. The function **check-valid-successor**, invoked during commit and process (lines [i,a]) verifies if attaching is allowed.

Security. So far we have explained how the CGKA functionality maintains a consistent history graph, even when allowing (inherent) injections. It remains to consider how such adversarially generated nodes affect the secrecy of group keys. First, obviously an adversarial commit or update does not heal the corresponding party. Note that **heals** from the safe predicate (cf. Fig. 4) already handles this by checking for $\mathsf{stat} = \mathsf{good}$. Second, for adversarial add proposals we have to assume that the contained public-key was chosen such that the adversary knows the corresponding secret key, implying that the adversary can read its welcome message. Hence, both secret state of the added party and the new group key are considered exposed (see the part (d) in Fig. 4 marked with a $\boxed{box}$).

Finally, consider a detached node created by an injected welcome message. Recall that new members join using a welcome message, containing all the relevant information about the group. Since our model does not include any long-term PKI, this welcome message is the only information about the group available to them and we cannot hope for a protocol that detects injected welcome messages. Moreover, we don't know where in the history graph a detached node belongs, and in particular whether it is a descendant of a node where another party is exposed or not. This means that we cannot guarantee secrecy for keys in detached nodes or their children (the part (c) of in Fig. 4 marked with a $\boxed{box}$). Still, we can at least express that this does not affect the guarantees of existing group members, and can start considering the subtree's security once it is attached to the main tree (e.g. by a party from the main tree moving there).

Robustness. Finally, we consider robustness, i.e., correctness guarantees with respect to honestly generated ciphertext when parties might have processed adversarially generated ones beforehand. We define two variants of $\mathcal{F}_{\text{CGKA}}$, differing in the level of robustness. Intuitively, the stronger variant (**robust** $=$ `true`) requires that honestly generated ciphertexts can always be processed by the intended recipients, while in the weaker variant (**robust** = `false`) the adversary can inject ciphertexts resulting parties to reject subsequent honest ones.

6 Construction for the Passive Setting

We now introduce the protocol P-Pas for the authenticated setting. A formal description can be found in the full version [5]. The protocol P-Pas is a modification of the TTKEM protocol [2] when executed in a propose-and-commit manner. We thus first briefly describe TTKEM and its vulnerability to cross-group attacks.

6.1 TTKEM

The (distributed) group state consists primarily of a *labeled left-balanced binary tree* (LT) τ, where each group member is assigned one of the leafs. All nodes (except the root that stores the group key) have labels epk and esk, denoting the public and secret key of a PKE scheme, respectively. The public key and the overall structure of the tree is considered public information, whereas the secret keys are only known by the members whose leaf is in the corresponding sub-tree.

Proposals represent suggestions for modifying the current LT. More concretely, a *Remove* proposal deletes all the labels of the specified member's leaf, an *Add* proposal adds a new leaf assigned to the new member id_t where epk is fetched from the PKI, and an *Update* proposal suggests replacing esk and epk in a member's leaf by a fresh pair, with the public key specified in the proposal.

The proposals are applied upon creating or processing a commit message, to derive the new state of the tree. When applying the proposals, some of the keys stored at intermediate leafs can no longer be used: to achieve PCS in case of updatesa party all nodes (potentially) known to that party, and analogously for removals. Moreover, for freshly added parties it cannot be assumed that they know the keys on their direct path (to the root). TreeKEM deals with those issues by *blanking* all those nodes. TTKEM, on the other hand, simply lets the committer choose fresh keys for all those nodes and marks them as *tainted* (i.e., sampled) by the committer via an additional public taintedby label. In particular, those keys now in turn must be assumed to be known to that party[15], and thus when committing one of his updates, also needs to be replaced in turn.

The commit message must allows all other group members to compute their respective views of the new LT, i.e., to learn all new public keys but also all replaced secret keys on their direct path. To simplify this, the commit algorithm generates the keys by first partitioning the to be re-keyed nodes into a number of path segments from a start node u to one of its ancestors v. Each of this segment is then re-keyed by "hashing up the path". Namely, it chooses a random secret s and iterates over the nodes in $(u \rightsquigarrow v)$. At each step it derives a new key pair

[15] While the party does not need to store any secret keys off his direct path, they might still have leaked to the adversary e.g. when using adversarially chosen randomness.

for the node using random coins $\mathsf{H}_2(s)$ and updates the secret for use with the next node on the path by setting $s \leftarrow \mathsf{H}_1(s)$.

Hence, each user only needs to learn one ciphertext for each segment: the seed for where the path first meets the user's direct path. Thus, the protocol simply encrypts each fresh key it to all its children's secrets that are not replaced as well. Care has to be taken where two different segments meet, i.e., where a one's segment's node is the parent of another segment's node. There, we still have to encrypt the node to the child, yet ensure that the child node's new key is used (for PCS). This can be done by processing the segments from "lower" to "higher", according to the depth of their end point.

Most users then process the above commit message in the obvious way. The only exception is the committer himself, which for PCS cannot decrypt the commit using any of his state. He can, however, simply stores the new ratchet tree at the time of creating it.

A welcome message prepared by the committer contains the public part of the (new) LT τ'. Additionally, for each freshly added member, the welcome message contains the secret labels of all nodes on the new member's direct path (except the leaf) encrypted under a second public key epk' stored as part of the add proposal. (An add proposal in TTKEM contains two public keys epk and epk'.)

Cross-Group Attacks. The TTKEM protocol is vulnerable against so-called cross-group attacks. Intuitively, those attacks are possible against protocols where commits affect only part of the group state. Consider the following example:

1. Create a group with A, B and C. Move all parties to the same node $\mathsf{Node}[c_0]$.
2. Make A send a commit c_1 and B send a commit c_2, neither containing an update for C.
3. Move C to $\mathsf{Node}[c_1]$, and A and B to $\mathsf{Node}[c_2]$.
4. Expose C's secret state.

In an optimally secure protocol, the two sub-groups should evolve independently, without the exposure of C in the one branch affecting the security of the other branch. In case of TTKEM, however, the group states in epochs c_0, c_1 and c_2 all share the same key pair for C's leaf. Moreover, if C is added last, then his node in the tree will be the direct right child of the root. Thus, when generating c_1 and c_2, both A and B encrypt the new root secret under C's leaf public key. [16] Hence, the adversary can derive the group key of $\mathsf{Node}[c_2]$ by using C's leaked secret key to decrypt the corresponding ciphertext in c_2.

We note that this cannot be easily fixed by just mixing the old group key into the new one. For this, we modify the above attack and corrupt B after Step 1. This leaks the old group key. Still, by PCS the key in c_2 should be secure.

6.2 The Protocol P-Pas

To avoid cross-group attacks, we modify TTKEM so that a commit evolves all key pairs in the LT. For this, we first replace the standard encryption scheme

[16] This attack can be easily extended to C's leaf not being a direct child of the root.

by HIBE. That is, each node, instead of labels epk and esk, has two public mpk and $\vec{\mathrm{id}}$, as well as one private label hsk. In the order listed, these labels contain a (master) HIBE public key, and a HIBE identity vector and the corresponding HIBE secret key for identity $\vec{\mathrm{id}}$. Encryption for a node is done with mpk and $\vec{\mathrm{id}}$. Whenever a new key pair is created for an internal node (e.g. during rekeying), the node's $\vec{\mathrm{id}}$ is initialized to the empty string. For leaf nodes, the first ID in the vector $\vec{\mathrm{id}}$ is set to the ID of the user assigned to that leaf.

Second, we can now evolve all keys with every commit: For nodes whose keys does not get replaced with the commit, we simply append the the hash of the commit message $\mathsf{H}_3(c)$ to the HIBE ID vectors, and update all secret keys on the processor's direct path accordingly.

Intuitively, this provides forward secrecy for individual HIBE keys in the LT. First, HIBE schemes ensure that secret keys for an ID vector can not be used to derive secrets for prefixes of that ID vector. So, the HIBE key of a node can not be used to derive its keys from previous epochs. Second, this guarantees in the event the group is split into parallel epochs (by delivering different commit messages to different group members) that the keys of a node in one epoch can not be used to derive the keys for that node in any parallel epochs. That is because, more generally, HIBE schemes ensure that secret keys for an ID vector $\vec{\mathrm{id}}$ can not be used to derive keys for any other ID vector $\vec{\mathrm{id}}'$ unless $\vec{\mathrm{id}}$ is a prefix of $\vec{\mathrm{id}}'$. But as soon as parallel epochs are created, the resulting ID vectors of any given node in both LTs have different commit messages in them at the same coordinate ensuring that no such vector is a prefix of another.

We prove two statements about $\mathsf{P\text{-}Pas}$. First, if the hash functions are modeled as non-programmable random oracles, then the protocol realizes the relaxed functionality that restricts the environment not to perform certain corruptions. Second, for programmable random oracles it achieves full UC security. Formally, we obtain the following theorems, proven in the full version [5].

Theorem 1. *Assuming that* HIBE *and* PKE *are* IND-CPA *secure, the protocol* $\mathsf{P\text{-}Pas}$ *realizes* $\mathcal{F}_{\text{CGKA-AUTH}}$ *with* **restrict-corruptions** $=$ `true` *in the* $(\mathcal{F}_{\text{PKI}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where* $\mathcal{G}_{\text{RO}}$ *denotes the global random oracle, calls to hash functions* H_i *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* H_i *and calls to* $\mathsf{PRG.eval}$ *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix PRG.*

Theorem 2. *Assuming that* HIBE *and* PKE *are* IND-CPA *secure, the protocol*$\mathsf{P\text{-}Pas}$ *realizes* $\mathcal{F}_{\text{CGKA-AUTH}}$ *with* **restrict-corruptions** $=$ `false` *in the* $(\mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{RO}})$*-hybrid model, where* $\mathcal{F}_{\text{RO}}$ *denotes the (local) random oracle, calls to hash functions* H_i *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* H_i *and calls to* $\mathsf{PRG.eval}$ *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix PRG.*

7 Constructions for the Active Setting

This section explains how to gradually enhance our protocol with passive security from Section 6 to deal with active network adversaries.

7.1 Basic Modifications of the Passive Protocol

Authentication. The goal of the first modification is to prevent injections whenever they are not inherent given correctness, i.e., the adversary should be able to make some party accept a message as coming from id in epoch c only if id's state in c is exposed (in the sense of the safe predicate). We achieve this using key-updatable signatures (KUS) [21], a signature analogon of HIBE, where verification additionally takes an identity vector and signing keys can be updated to lower-level ones, for longer identity vectors.

To this end, we modify the group state with each leaf in the LT getting two additional labels: a KUS verification key spk and a corresponding signing key ssk for the leafs identity vector $\vec{\mathsf{id}}$ (the same one as for HIBE). The leaf's KUS keys live alongside its HIBE keys: each update and commit of the user id assigned to the leaf contains a fresh spk, and whenever id processes a commit message c, he updates ssk using the identity c. All messages sent by id are signed with his current signing key and verified by receiving parties respectively. Accordingly, the PKI key generation outputs an additional KUS key pair for the new member.

Binding control messages to epochs. The actively secure protocols have to ensure that control messages are not used of context, e.g., trying to process a commit message that does not originate from the current state, or using a proposal belonging to a different epoch in a commit message. This is achieved by each control message (commit or proposal) contains an epoch id epid, which is simply a hash of the last commit message, and additionally each commit message containing a hash of the list of committed proposals.

Proposal validation. For active security, the commit and proc algorithms have to check that all proposals being committed to were created by a current member of the group, that add- and remove-proposals only add and remove parties that are currently not yet in the group, respectively already in the group, and that the proposals don't remove the party executing commit from the group (as this party chooses the next group key).

Validating the public state in welcome messages. Recall that $\mathcal{F}_{\text{CGKA}}$ allows the environment to inject a welcome message, making a party join a detached node. If afterwards the environment makes a different party process the corresponding commit message, the node is attached to its parent. $\mathcal{F}_{\text{CGKA}}$ requires that in such case the joining and the processing party end up in a consistent state (e.g. they agree on the member set). Our protocol guarantees this by 1) including in a commit message a hash $\mathsf{H}_5(\tau_{pub})$, where τ_{pub} is the public part of the LT in the new epoch, and 2) including the whole commit c in the welcome message.

If after processing a commit c the resulting LT doesn't match the hash, the protocol returns $\perp$. The joining party verifies that τ_{pub} in the welcome message matches the hash in the commit c and computes epid as hash of c.[17]

[17] Recall that the functionality identifies epochs by c, so in order for the simulator to determine the epoch for injected welcome messages, it has to contain the whole c.

7.2 The Non-Robust Protocol

Well-formedness via hashing. The next goal is to prevent the adversary from successfully injecting (using leaked states) malformed control messages. This is not a problem for our proposal messages, since they only contain public information, which can be easily verified.[18] However, commit and welcome messages both contain a number of ciphertexts of (supposedly) related data, only part of which can be decrypted by a given party. The following simple solution to this problem provides security, but not robustness.

Consider first commit messages, which contain a number of public keys and a number of ciphertexts, used by a party to derive his slice of corresponding secret keys and the new group key. While validity of derived secret keys can be verified against the public keys, this is not the case for the group key. Hence, we add to the message an analogue of a public-key for the group key — we use hash functions H_6 and H_7 and whenever a party is ready to send a commit c creating an LT τ, it attaches to c a *confirmation key* $\mathsf{H}_6(c, \tau.\mathsf{grpkey})$ (recall that grpkey is the label in the root of τ) with which grpkey can be validated. The actual group key for the new epoch is then defined to be $\mathsf{H}_7(\tau.\mathsf{grpkey})$.

Second, a welcome message contains the public part of the LT τ_{pub}, encryption of the new member's slice of the secret part and the commit message c. The join algorithm performs the same checks as proc: it verifies the decrypted secret keys against the public keys in τ_{pub} and the decrypted $\tau.\mathsf{grpkey}$ against the confirmation key in c.

Putting it all together. Combining the above techniques results in our first protocol, $\mathsf{P\text{-}Act}$. In particular, a commit in $\mathsf{P\text{-}Act}$ is computed as follows: (1) Generate the message c as in the protocol with passive security (taking into account the additional KUS labels). (2) Add the hash of the public state and epoch id: $c \leftarrow (c, \mathsf{epid}, \mathsf{H}_5(\tau_{pub}))$. (3) Compute the confirmation key as $\mathsf{conf\text{-}key} = \mathsf{H}_6(c, \tau.\mathsf{grpkey})$. (4) Output $(c, \mathsf{conf\text{-}key})$, signed with the current KUS secret key. (We note that we use KUS with unique signatures).

Security. We note that the confirmation key has in fact two functions. Apart from guaranteeing that parties end up with the same group keys, in the random oracle model, it also constitutes a proof of knowledge of the group key with respect to the commit message. This prevents the adversary from copying parts of commits sent by honest parties, where he does not know the secrets, into his injected commits (he cannot copy the honest committer's confirmation key, because the control message c no longer matches).

As in the case of protocols with passive security, we prove two statements: if the hash functions are modeled as non-programmable random oracles, then we achieve security with respect to a restricted class of environments, while if the random oracles are programmable, we achieve full UC security. Both theorems are proven in the full version [5].

[18] Note that the validity of the public-key contained in add-proposals cannot be verified as our model does not consider an identity PKI.

Theorem 3. *Assuming that* HIBE *and* PKE *are* IND-CCA *secure, and* KUS *is* EUF-CMA *secure the non-robust protocol* $\mathsf{P\text{-}Act}$ *realizes* $\mathcal{F}_{\text{CGKA}}$ *with* **robust** $=$ `false` *and* **restrict-corruptions** $=$ `true` *in the* $(\mathcal{F}_{\text{PKI}}, \mathcal{G}_{\text{RO}})$*-hybrid model, where* $\mathcal{G}_{\text{RO}}$ *denotes the global random oracle, calls to hash functions* H_i *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* H_i *and calls to* $\mathsf{PRG.eval}$ *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* PRG*.*

Theorem 4. *Assuming that* HIBE *and* PKE *are* IND-CCA *secure, and* KUS *is* EUF-CMA *secure the non-robust protocol* $\mathsf{P\text{-}Act}$ *realizes* $\mathcal{F}_{\text{CGKA}}$ *with* **robust** $=$ `false` *and* **restrict-corruptions** $=$ `true` *in the* $(\mathcal{F}_{\text{PKI}}, \mathcal{F}_{\text{RO}})$*-hybrid model, where* $\mathcal{F}_{\text{RO}}$ *denotes the (local) random oracle, calls to hash functions* H_i *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* H_i *and calls to* $\mathsf{PRG.eval}$ *are replaced by calls to* $\mathcal{G}_{\text{RO}}$ *with prefix* PRG*.*

7.3 The Robust Protocol using NIZKs

Unfortunately, the solution with the confirmation key not provide robustness, since a party cannot verify all ciphertexts, and so it may accept a commit message that will be rejected by another party. In order to provide robustness, we need a mechanism that allows parties to verify well-formedness of all ciphertexts in a commit message. For this, we replace the simple method of well-formedness verification via hashing by a non-interactive zero-knowledge argument (NIZK). In particular, in our robust protocol $\mathsf{P\text{-}Act\text{-}Rob}$ a commit message contains a NIZK of knowledge of randomness r and secret state γ such that (1) running th commit of $\mathsf{P\text{-}Pas}$ with r and γ results in the given message and (2) secret keys in γ match the public keys in the receiver's ratchet tree. Intuitively, this is secure since $\mathsf{P\text{-}Pas}$ is already secure against adversarial randomness, and since r and γ can be extracted from the NIZK of knowledge.

For lack of space, we leave details and the security proof to the full version [5]. The statement we prove is that $\mathsf{P\text{-}Act\text{-}Rob}$ realizes in the *standard model* a static version of $\mathcal{F}_{\text{CGKA}}$, where, whenever an id commits, the environment must specify if the key in the new node should be secure, in which case certain corruptions are disabled, or not. The reason for using a different type of statement than for other protocols is that $\mathsf{P\text{-}Act\text{-}Rob}$ uses a NIZK for a statement involving hash evaluations. On the other hand, our proofs of adaptive security require modeling the hash as a random oracle.

References

[1] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019, Part I. Lecture Notes in Computer Science, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2˙5

[2] Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. Cryptology ePrint Archive, Report 2019/1489 (2019), `https://eprint.iacr.org/2019/1489`

[3] Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging. Private communication (2020)

[4] Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the ietf mls standard for group messaging. In: Advances in Cryptology – CRYPTO 2020 – 40th Annual International Cryptology Conference, Proceedings (2020), full version: `https://eprint.iacr.org/2019/1189`

[5] Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. Cryptology ePrint Archive, Report 2020/752 (2020), `https://eprint.iacr.org/2020/752`

[6] Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., Wichs, D.: Multiparty computation with low communication, computation and interaction via threshold FHE. In: Pointcheval, D., Johansson, T. (eds.) Advances in Cryptology – EUROCRYPT 2012. Lecture Notes in Computer Science, vol. 7237, pp. 483–501. Springer, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4˙29

[7] Backes, M., Dürmuth, M., Hofheinz, D., Küsters, R.: Conditional reactive simulatability. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006: 11th European Symposium on Research in Computer Security. Lecture Notes in Computer Science, vol. 4189, pp. 424–443. Springer, Heidelberg (Sep 2006). https://doi.org/10.1007/11863908˙26

[8] Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Advances in Cryptology—ASIACRYPT 2020 (2020), to appear

[9] Barnes, R., Beurdouche, B., , Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: The messaging layer security (mls) protocol (draft-ietf-mls-protocol-09). Tech. rep., IETF (Mar 2020), `https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/`

[10] Barnes, R., Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: Message layer security (mls) wg. https://datatracker.ietf.org/wg/mls/about/

[11] Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017, Part III. Lecture Notes in Computer Science, vol. 10403, pp. 619–650. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63697-9˙21

[12] Bhargavan, K., Barnes, R., Rescorla, E.: Treekem: Asynchronous decentralized keymanagement for large dynamic groups. (May 2018)

[13] Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science. pp. 136–145. IEEE Computer Society Press (Oct 2001). https://doi.org/10.1109/SFCS.2001.959888

[14] Canetti, R., Krawczyk, H., Nielsen, J.B.: Relaxing chosen-ciphertext security. In: Boneh, D. (ed.) Advances in Cryptology – CRYPTO 2003. Lecture Notes in Computer Science, vol. 2729, pp. 565–582. Springer, Heidelberg (Aug 2003). https://doi.org/10.1007/978-3-540-45146-4˙33

[15] 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: 25th Conference on Computer and Communications Security. pp. 1802–1819. ACM Press (Oct 2018). https://doi.org/10.1145/3243734.3243747

[16] Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. In: Attrapadung, N., Yagi, T. (eds.) IWSEC 19: 14th International Workshop on Security, Advances in Information and Computer Security. Lecture Notes in Computer Science, vol. 11689, pp. 343–362. Springer, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26834-3˙20

[17] Facebook: Messenger secret conversations (technical whitepaper version 2.0) (2016), retrieved 05/2020 from `https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf`

[18] Freire, E.S.V., Hesse, J., Hofheinz, D.: Universally composable non-interactive key exchange. In: Abdalla, M., Prisco, R.D. (eds.) SCN 14: 9th International Conference on Security in Communication Networks. Lecture Notes in Computer Science, vol. 8642, pp. 1–20. Springer, Heidelberg (Sep 2014). https://doi.org/10.1007/978-3-319-10879-7˙1

[19] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th Annual ACM Symposium on Theory of Computing. pp. 218–229. ACM Press (May 1987). https://doi.org/10.1145/28395.28420

[20] Howell, C., Leavy, T., Alwen, J.: Wickr messaging protocol : Technical paper (2019), `https://1c9n2u3hx1x732fbvk1ype2x-wpengine.netdna-ssl.com/wp-content/uploads/2019/12/WhitePaper_WickrMessagingProtocol.pdf`

[21] Jaeger, J., Stepanovs, I.: Optimal channel security against fine-grained state compromise: The safety of messaging. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018, Part I. Lecture Notes in Computer Science, vol. 10991, pp. 33–62. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1˙2

[22] Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019, Part I. Lecture Notes in Computer Science, vol. 11476, pp. 159–188. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17653-2˙6

[23] Jost, D., Maurer, U., Mularczyk, M.: A unified and composable take on ratcheting. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019: 17th Theory of Cryptography Conference, Part II. Lecture Notes in Computer Science, vol. 11892, pp. 180–210. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-36033-7˙7

[24] Marlinspike, M., Perrin, T.: The double ratchet algorithm (11 2016), `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`

[25] Poettering, B., Rösler, P.: Towards bidirectional ratcheted key exchange. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018, Part I. Lecture Notes in Computer Science, vol. 10991, pp. 3–32. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1˙1

[26] WhatsApp: Whatsapp encryption overview (2017), retrieved 05/2020 from `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`