CoCoA: Concurrent Continuous Group Key Agreement

Joël Alwen[1], Benedikt Auerbach[2], Miguel Cueto Noval[2], Karen Klein[3], Guillermo Pascual-Perez[2], Krzysztof Pietrzak[2], and Michael Walter[4]

[1]AWS Wickr
alwenjo@amazon.com
[2]ISTA, Klosterneuburg, Austria
{bauerbac, mcuetono, gpascual, pietrzak}@ist.ac.at
[3]ETH Zurich, Switzerland
karen.klein@inf.ethz.ch
[4]Zama, Paris
michael.walter@zama.ai

February 25, 2022

Abstract

Messaging platforms like Signal are widely deployed and provide strong security in an asynchronous setting. It is a challenging problem to construct a protocol with similar security guarantees that can *efficiently* scale to large groups. A major bottleneck are the frequent key rotations users need to perform to achieve post compromise forward security.

In current proposals – most notably in TreeKEM (which is part of the IETF's Messaging Layer Security (MLS) protocol draft) – for users in a group of size n to rotate their keys they must each craft a message of size $\log(n)$ to be broadcast to the group using an (untrusted) delivery server.

In larger groups, having users sequentially rotate their keys requires too much bandwidth (or takes too long), so variants allowing any $T \leq n$ users to simultaneously rotate their keys in just 2 communication rounds have been suggested (e.g. "Propose and Commit" by MLS). Unfortunately, 2-round concurrent updates are either damaging or expensive (or both); i.e they either result in future operations being more costly (e.g. via "blanking" or "tainting") or are costly themselves requiring $\Omega(T)$ communication for each user [Bienstock et al., TCC'20].

In this paper we propose CoCoA; a new scheme that allows for T concurrent updates that are neither damaging nor costly. That is, they add no cost to future operations yet they only require $\Omega(\log^2(n))$ communication per user. To circumvent the [Bienstock et al.] lower bound CoCoA increases the number of rounds needed to complete all updates from 2 up to (at most) $\log(n)$; though typically fewer rounds are needed.

The key insight of our protocol is the following: in the (non-concurrent version of) TreeKEM, a delivery server which gets T concurrent update requests will approve one and reject the remaining $T-1$. In contrast, our server attempts to apply all of them. If more than one user requests to rotate the same key during a round, the server arbitrarily picks a winner. Surprisingly, we prove that regardless of how the server chooses the winners, all previously compromised users will recover after at most $\log(n)$ such update rounds.

To keep the communication complexity low, CoCoA is a server-aided CGKA. That is the delivery server no longer blindly forwards packets, but instead actively computes individualized packets tailored to each user. As the server is untrusted, this change requires us to develop new mechanisms ensuring robustness of the protocol.

Contents

1 Introduction

End-to-end (E2E) secure cryptographic protocols are rapidly becoming ubiquitous tools in the daily life of billions of people. The most prominent examples are secure messaging protocols (such as those based on the Double Ratchet) and E2E encrypted VoIP conference calling protocols. The demands of practical E2E security are non-trivial; all the more so when the goal is to allow *groups* to communicate in a single session. Almost all current (aka. "1st generation") E2E protocols for groups are built on top of some underlying black-box 1-on-1 E2E secure protocol [26]. However, this approach seems to unavoidably result in the complexity of (at least some critical) operations scaling linearly in the group size n.[1] This has resulted in practical limits in the groups sizes for deployed E2E protocols (to date, often in 10s or low 100s and never more than 1000).

Motivated by this, Cohn-Gordon et al. [14] initiated the study of E2E protocols whose complexity scales *logarithmically* in n. Starting with this work, research in the area has focused on a fundamental class of primitives called *Continuous Group Key Agreement* (CGKA).[2] Intuitively, CGKA is to, say, E2E secure messaging what Key Agreement is to Public Key Encryption. That is, CGKA protocols capture many of the challenges involved in building practical higher-level E2E secure applications (like messaging) while still providing enough functionality to make building such applications comparatively easy using known techniques [2]. Thus they present a very useful subject for research in the area.

In a bit more detail, a CGKA protocol allows an evolving set of group members to continuously agree on a fresh symmetric key. Every time a new party joins, or an existing one leaves or refreshes (a.k.a. "updates") their cryptographic state, a new epoch begins in the session. Each epoch E is equipped with its own group key k_E which can be derived by all parties that are members of the group during E. CGKA protocol sessions are expected to last for very long periods of time (e.g. years). Thus, they must provide a property sometimes referred to as *post compromise forward security* (PCFS) [3]. That means that the group key of a target epoch should look random to an adversary despite having compromised any number of group members in both earlier and later epochs as long as the compromised parties either left the group or performed an update between their compromise and the target epoch.[3] In the spirit of distributed E2E security (and unlike, say, Broadcast Encryption or Dynamic Group Key Agreement) CGKA protocols must achieve this without the help of trusted group managers or other specially designated trusted parties.

With a few exceptions discussed below, most CGKA protocols today [14, 7, 1, 23, 3, 5] were designed with an asynchronous communication setting in mind (likely motivated by the application of secure asynchronous messaging). That is, parties may remain offline for extended periods of time, not ever actually being online at the same time as each other. Once they do come online, though, they should be able to immediately "catch up" and even initiate a new epoch (e.g. by unilaterally adding a new member to the group). To facilitate this, protocols are designed to communicate via an untrusted network which buffers protocol packets for parties until they come online again.

The Problem Of Coordination. One property the first generation of CGKA protocols [23, 1, 7, 14] share is that they require *all* protocol packets to be processed in exactly the same order by every group member. However, ensuring this level of coordination can present real challenges in a variety of settings; especially for large groups (e.g. with $50,000$ members as is targeted by the IETF's upcoming E2E secure messaging standard MLS [6]). In particular, it might lead to the problem sometimes called "starvation" where a client's packets are constantly rejected by the group (e.g. when the client is on a slow network connection and so can never distribute its own packets fast enough).

[1] In fact, this holds true even for the few 1st generation E2E protocols designed from the ground up with groups in mind [20].

[2] Also referred to as *Group Ratcheting* [9] or *Continuous Group Key Distribution* [11].

[3] We note that PCFS is strictly stronger than providing the two more commonly discussed properties of Forward Security (FS) and Post Compromise Security (PCS). Indeed, a successful attack on an epoch E may require compromises *both* before *and* after E. Such an attack is neither an FS attack nor a PCS attack. Moreover, literature usually speaks informally of FS and PCS as separate notions asking that they both hold. Yet the notions do not necessarily compose. For example, the MLS messaging standard has both strong FS and PCS properties but significantly worse PCFS [1]. Fortunately, all *formal* security definitions for CGKA we are aware of do in fact capture (some variation of) PCFS instead of treating FS and PCS separately.

There do not seem to be any practical solutions to convincingly provide this level of coordination without significant drawbacks. Implementing the buffering mechanism via a single server does not automatically address the issue of starvation of clients with a slow connection. Nor is a round-robin "speaking slot" approach a satisfactory solution (even assuming universal time), as it would severely impact responsiveness; especially for larger groups. It's also not just responsiveness that suffers from a reduction of the rate at which parties can send new packets to the group. The quality of the security of a session (e.g. the speed with which privacy is recovered after a group member's local state is leaked) is also tightly dependent on the rate at which participants can send out packets. After all, if a compromised party has not even been able to send anything new to the group since a compromise, they have no way to update the leaked cryptographic material to something the adversary cannot simply derive itself.

Concurrency At A High Price. To mitigate this problem the MLS messaging protocol introduced a new syntax referred to as the "propose-and-commit" (P&C) paradigm. Since then it was adopted by most second generation CGKA protocols [3, 5] as it allows for some degree of concurrency. In particular, group members (and even designated external parties) may concurrently propose changes to the group state e.g. "Alice proposes adding Bob", "Charlie proposes updating his keys", etc. At any point a group member can collect such proposal into a commit message which is broadcast to the group and actually affects all changes in the referenced proposals. Note, however, that the commit messages still must be processed in a globally unique order. Moreover, in each of these protocols there is a high price being paid for large amounts of concurrency. Namely, the greater the number of proposals in a single commit message, the less efficient (e.g. greater packet size) certain future commits will be. In fact, efficiency can degenerate to the point where (starting from an arbitrary group state) a commit to $\Theta(n)$ proposals can produce a state where the next commit packet is forced to have size $\Omega(n)$; a far cry from the desired $O(\log(n))$.

Lower-Bounds on Communication Complexity. Bienstock et al. [9] showed that there are limits to what we could hope for in terms of reducing communication complexity. Specifically, they show that T group members updating concurrently incurs a communication cost per user in the following round that is linear in T in any "reasonable" protocol.[4] In fact, if all n parties wish to update concurrently within 2 rounds then this has complexity at least $\Omega(n^2)$.

1.1 Our Contributions

In this paper we propose a new CGKA protocol called CoCoA (for *COncurrent COntinuous group key Agreement*) which is designed specifically to allow for efficient concurrent group operations. In contrast to past CGKA protocols, update operations may require more than 2 rounds (in the worst case $\log(n)$ rounds). However, even when all n users update their keys concurrently in $\log(n)$ rounds, the total communication complexity of any user is only roughly $(\log(n))^2$ (constant size) ciphertexts. This circumvents [9] as their lower-bound only holds for updates that complete in at most 2 rounds. So, for the price of more interaction CoCoA can *greatly* decrease the actual bandwidth consumed.

To emphasize this even more, consider the cost of transitioning from a fully blanked tree to a fully unblanked one. We believe this to be a particularly interesting case as it captures the transition from any freshly created group into a bandwidth-optimal one. The faster/cheaper this transition can be completed, the faster an execution can begin optimal complexity behaviour. TreeKEM [7], the CGKA scheme used in the MLS messaging protocol, needs $n/2$ rounds with receiver complexity, i.e. number of ciphertexts downloaded per user, $\Omega(n \log(n))$. The protocol in [9], in turn, would be able to unblank the whole tree in 2 rounds with linear sender and recipient communication per user. In contrast, in CoCoA the tree could be unblanked in 1 round with linear sender cost, but only logarithmic recipient cost. For big groups this difference is very significant.

[4] For the lower bound, [9] considered a symbolic model of execution, which only applies to protocols constructed by using "practical" primitives combined in a "standard" way. For definitions of what "practical" and "standard" mean in this context we refer to [9], but we remark, that our protocol and the TreeKEM variants considered in this work fall into this category.

With such low communication, a user cannot learn all the $2n-1$ fresh public-keys in the distributed group state (usually called "ratchet-tree" or "key-tree"). Fortunately, for CoCoA, users only need to know the $\log(n)$ secret keys and another $2\log(n)$ public keys. So in our protocol, users will not have a complete view of the public state as in previous protocols, but only know the partial state that is relevant to them. As a consequence, the server no longer acts as a relay but instead computes packets tailored to the individual receiving user. This comes with a new challenge that we address in this work: ensuring consistency across all users is not as straightforward anymore. This is crucial for security, since users disagreeing e.g. on the set of group members can lead to severe attacks.

Once we take into account operations like adding and removing group members, efficiency might degrade (though not to anything worse than past protocols). Nevertheless, in a typical execution we can expect to see far more updates than adds/removes. In particular, the more updates parties perform the faster the protocol heals from past compromises so it is generally in users' interest to perform updates as regularly as they can. By (greatly) reducing the cost of updates compared to past CGKA protocols, we allow groups to have quantitatively better security for the same amount of communication complexity spent.

In terms of security, we prove CoCoA secure in an "partially active setting". A bit more precisely, the adaptive adversary can (repeatedly) leak parties local states including any random coins they use and query users to generate protocol messages. As the sever is untrusted, the adversary is allowed to send arbitrary (potentially malformed) server messages and deviate from the server specification. However, as users sign their protocol messages and the adversary does not get access to signing keys, it is not able to generate such messages by itself. While the latter is a strong assumption, it is common for such protocols. We discuss this in more detail in Section 5. Note that [23] uses the term partially active to refer to security against weaker adversaries that have control of the delivery server, but are not allowed to send arbitrary messages.

Signature Keys. One caveat to the above discussion are signature keys. Apart from the aforementioned public and secret keys, each group member must know the signature verification key of each other group member (as these are used to authenticate packets, amongst other things). In principle, this means that just distributing n new signature key pairs (as part of n parties updating) already imposes $\Omega(n)$ communication complexity for each group member (regardless of how many rounds are used or even of concurrency).

However, in practice, there are several mitigating aspects to this problem. As was observed already during the design of MLS, in some real-world deployments of CGKAs fresh signature keys may be much harder to come by than simply locally generating new ephemeral key material. That is because each new signature key is typically bound to some external identity (like an account name) via some generic "authenticator" and this binding may be an expensive and slow process. E.g. a certificate that must be obtained manually from a CA. For this reason CoCoA (like MLS) explicitly permits *lite updates*; that is, updates which refresh all secret key material of the sender *except* for their signature keys. While lite updates are clearly not ideal from a security perspective, they do allow for frequently refreshing the remaining key material without being bogged down by the cost of certifying fresh signature keys. Moreover, CoCoA (like MLS) derives authenticity of packets not just from signatures but also by requiring senders to, effectively, prove knowledge of the previous epoch's group key. Thus, leaking a group members' signing keys does not automatically confer the ability to forge on their behalf. Indeed, if the victims all perform a lite update, a fresh epoch is initiated with a secure group key.

1.2 Related Work

The study of CGKA based protocols (in particular, with the explicit goal of PCS) was initiated by the ART protocol [14], based on which the first version of MLS [6] was built shortly before transitioning to TreeKEM [7]. TreeKEM has since been the subject of several security analysis including [1, 2, 11, 11, 8, 15]. More generally, the study of CGKAs has, roughly speaking, focused on several topics: stronger security definitions, more efficient constructions, better support for concurrency and new security properties.

Concurrent CGKA. Several works have studied CGKA's supporting varying degrees of concurrent operations. Weidner's Causal TreeKEM [24] explores the idea of updates *re-randomizing* key material instead of overwriting it (though it lacks forward secrecy and a complete security proof). Recently, [28] proposed a decentralized CGKA protocol; albeit with linear communication complexity. Finally, a paper by Bienstock *et al.* [9] studies the trade-off between PCS, concurrency and communication complexity, showing a lower bound for the latter and proposing a close to optimal protocol in their synchronous model for a fixed group in a weak security model (see Section 4).

CGKA Security Notions Protocols. Broadly speaking, there are at least three approaches to defining basic security properties for CGKA (e.g. confidentiality, authenticity and group agreement).

A similar security model to the one in our paper can be found in [23]. That work was the first to require security against an *adaptive* adversary (and also introduced the basic GSD proof technique used in most subsequent security proofs for adaptively secure CGKA.) They use their model to analyze Tainted TreeKEM which exhibits a different efficiency profile than TreeKEM that can prove advantageous in some common settings.

The security model of [1] is weak but incomparable to our model. Their adversaries must deliver all packets in the same order to all parties (though not at the same time or even at all) and they do not learn the coins of corrupt parties. On the other hand, the adversary may modify and even inject new packets; albeit only if honest parties would reject the resulting adversarial packet. They use the model to analyze the security of RTreeKEM which enjoys greatly improved forward secrecy (and PCFS) relative to other CGKAs (with the exeception of the ones in [3]).

A different approach to security notions was initiated in [2] where the *history graph* technique was introduced to describe the semantics of any given CGKA executions. They also provided the first black-box construction of secure group messaging from CGKA (and other primitives). Their game based notion still placed some restriction on when an active adversary could inject new packets. But it soon inspired the ideal/real CGKA security notion in [3] which presented the first ideal functionality for CGKA. Their notion captures security against powerful adaptive and fully active adversaries (i.e. they can deliver arbitrary packets without any restrictions) that can corrupt parties at will and even *set* their random coins. They used their notion to capture the security of a pair of CGKA protocols they described; in particular, showing them to satisfy maybe the strongest versions of the basic CGKA security properties of any existing protocols to date; albeit at the cost of quite impractical efficiency. Finally, building on that work, [5] extend their adversaries to also account for how corrupt insiders might interact with a (very weak) PKI. The resulting notion is called *insider security.* That work analyzes the insider security guaranteed by TreeKEM. The formal notion was later adapted in [19, 4] to capture essentially the same intuition but for the server-aided CGKA setting.

A third approach to defining adaptive and active security is taken in [8] who use an "event driven" language to define adaptive security a CGKA. E.g. authenticity is captured by roughly stating that if Alice believes a packet came from Bob then this must be preceded by an event where Bob sends such a packet. However, their adversaries are closer to those of [3] than the full insider adversaries of [5] as they can not interfere with the PKI. In [8], TreeKEM (Draft 7) along with 4 variants are analyzed.

Other Security Properties. The notion of server-aided CGKA was first formalized in [4]. They introduced the SAIK protocol which reduces receiver communication costs by as much as 1000 fold in groups of size $10,000$. However, the earlier work of [16] and the concurrent work of [19] both include (implicit) server-aided CGKAs as well.

The work of [22] initiated the study of post-quantum primitives for CGKA by building primitives designed for use in TreeKEM (and similar CGKAs). However, it turned out that their security notion was lacking (e.g. it seems to not allow for adaptive security of the resulting CGKA) so in a follow up paper [19] a new (more secure) PQ primitive is proposed along with a novel server-aided CGKA (proven secure in the classic model) designed to reduce receiver communication cost.

Cremers *et al.* compared the PCS properties in the multi-group setting of MLS to the Signal group protocol [15]. In [3] zero-knowledge proofs are used to improve the robustness of CGKA protocols. The approach was made a bit more practical in [16] by, amongst other things, introducing tailor-made ZK proofs. Very recently, [18] initiated the study of membership privacy for CGKAs.

Group Key Exchange. A closely related family of protocols to CGKA are the older Group Key Exchange (GKE) protocols which allow a fixed group of users to derive a common key. These can be traced to early publications like [21, 12]. In contrast to CGKA, GKE protocols do not target PCS and are designed for the synchronous setting. That is, they are highly interactive e.g. requiring all parties to contribute to any one operation via interactive rounds. Initial GKE results were followed by a long list of works exploring additional features; notably, supporting changes to group membership mid-session (aka. Dynamic GKE) [10, 17]. Another notion very related to CGKA are Logical Key Hierarchies [27, 29, 13]. Introduced as a solution to very related primitive of Multicast Encryption [25]. They allow a changing group of users to maintain a common key with the help of a trusted group manager.

2 Preliminaries

2.1 Continuous Group-key Agreement

To begin with, we define the notion of continuous group-key agreement (CGKA). Parties participating in the execution of a CGKA protocol will maintain a local state γ, allowing them to keep track of a common ratchet tree, to derive a shared secret. Parties will be able to add and remove users to the execution, and to rotate the keys along sections of the tree, thus achieving FS and PCS. Our definition is similar to that of [23], with the main difference that operations do not need to be confirmed individually by the server. Instead, the stateful server works in rounds, collects operations into batches and sends them out at the end of each round (note that setting the batch size equal to 1 would just return the definition from [23]). Accordingly, a party issuing an operation will no longer be able to pre-compute its new state should the operation be confirmed.

Definition 1 (Asynchronous Continuous Group-key Agreement)**.** *An* asynchronous continuous group-key agreement *(CGKA) scheme is an* 8*-tuple of algorithms* $\mathsf{CGKA} = (\mathsf{CGKA.Gen}, \mathsf{CGKA.Init}, \mathsf{CGKA.Add}, \mathsf{CGKA.Rem}, \mathsf{CGKA.Upd}, \mathsf{CGKA.Dlv}, \mathsf{CGKA.Proc}, \mathsf{CGKA.Key})$ *with the following syntax and semantics:*

Key Generation**:** *Fresh InitKey pairs* $((\mathsf{pk}, \mathsf{sk}), (\mathsf{ssk}, \mathsf{svk})) \leftarrow \mathsf{CGKA.Gen}(1^\lambda)$ *consist of a pair of public key encryption keys and a pair of digital signing keys. They are generated by users prior to joining a group, where* λ *denotes the security parameter. Public keys are used to invite parties to join a group.*

Initialize a Group**:** *Let* $\mathsf{G} = (\mathsf{ID}_1, \dots, \mathsf{ID}_n)$*. For* $i \in [2, n]$ *let* pk_i *be an InitKey PK of party* ID_i*. Party* ID_1 *creates a new group with membership* G *by running:*

$$(\gamma, [W_2, \dots, W_n]) \leftarrow \mathsf{CGKA.Init}\left(\mathsf{G}, [\mathsf{pk}_1, \dots, \mathsf{pk}_n], [\mathsf{svk}_1, \dots, \mathsf{svk}_n], \mathsf{sk}_1\right)$$

and sending welcome message W_i *for party* ID_i *to the server. Finally,* ID_1 *stores its* local state γ *for later use.*

Adding a Member**:** *A group member with local state* γ *can add party* ID *to the group by running* $(\gamma, W, T) \leftarrow \mathsf{CGKA.Add}(\gamma, \mathsf{ID}, \mathsf{pk}, \mathsf{svk})$ *and sending* welcome message W *for party* ID *and the* add message T *for all group members (including* ID*) to the server.*

Removing a Member**:** *A group member with local state* γ *can remove group member* ID *by running* $(\gamma, T) \leftarrow \mathsf{CGKA.Rem}(\gamma, \mathsf{ID})$ *and sending the* remove message T *for all group members (*ID*) to the server.*

Update**:** *A group member with local state* γ *can perform an update by running* $(\gamma, T) \leftarrow \mathsf{CGKA.Upd}(\gamma)$ *and sending the* update message T *to the server.*

Collect and Deliver: *The delivery server, upon receiving a set of CGKA protocol messages $T = (T_1, \ldots, T_k)$ (including welcome messages) generated by a set of parties, sends out a round message $(\gamma_{\text{ser}}, (\mathfrak{M}_1, \ldots, \mathfrak{M}_n)) = \mathsf{CGKA.Dlv}(\gamma_{\text{ser}}, T)$, where $\mathfrak{M}_i$ is the message for user i and γ_{ser} is the server's internal state. Each $\mathfrak{M}_i$ contains a counter c_i indicating whether $\mathfrak{M}_i$ includes an update message generated by user i, and which one of the potentially several they might have generated.*

Process: *Upon receiving an incoming CGKA message $\mathfrak{M}_i$, a party immediately processes it by running $\gamma \leftarrow \mathsf{CGKA.Proc}(\gamma, \mathfrak{M}_i)$.*

Get Group Key: *At any point a party can extract the current group key K from its local state γ by running $K \leftarrow \mathsf{CGKA.Key}(\gamma)$.*

2.2 Ratchet Trees

Our protocol builds on TreeKEM, and thus uses the same underlying structure of a *ratchet tree* for deriving shared secrets among the group members. A ratchet tree is a directed binary tree $\mathfrak{T} = (V_{\mathfrak{T}}, E_{\mathfrak{T}})$, with edges pointing towards the root node v_{root}[5] and each user in the group associated to a leaf. We will use the notation $\mathfrak{T}^i = (V^i_{\mathfrak{T}}, E^i_{\mathfrak{T}})$ to refer to the ratchet tree associated to round i.

Tree structure. Given a node v, we will denote its child by $\mathsf{child}(v)$, its left and right parents respectively as $\mathsf{lparent}(v), \mathsf{rparent}(v)$, and will write $\mathsf{parents}(v) = (\mathsf{lparent}(v), \mathsf{rparent}(v))$. Given a leaf node, we denote its path to the root as $\mathsf{path}(v) = (v_0 = v, v_1, \ldots, v_k = v_{root})$, where $v_i = \mathsf{child}(v_{i-1})$. Similarly, we denote its co-path as $\mathsf{co\text{-}path}(v) = (v'_1, \ldots, v'_k)$, where v'_i is the parent of v_i not in $\mathsf{path}(v)$. We will often just refer to such a (co-)path as *v's (co-)path*. For a user ID we will denote its associated leaf node by $\mathsf{leaf}(\mathsf{ID})$, and accordingly sometimes refer to $\mathsf{leaf}(\mathsf{ID})$'s (co-)path as just ID's (co-)path or ($\mathsf{co\text{-}path}(\mathsf{ID})$) $\mathsf{path}(\mathsf{ID})$. Given two leaves l, l', let $\mathsf{Int}(l, l')$ be their least common descendant, i.e. the first node where their paths intersect. For a node $v \in \mathfrak{T}$, we set $v.\mathsf{isLeaf} := \mathsf{true}$ if v is a leaf of $\mathfrak{T}$, and $v.\mathsf{isLeaf} := \mathsf{false}$ otherwise.

Node states. Each node v has an associated node state $\gamma(v)$. Sometimes during the protocol execution nodes can be marked as *blank*, meaning that their state is empty. Blank nodes become *unblanked* if their state is repopulated at a later point in time.

The non-blank node state contains: a PKE key-pair $(\gamma(v).\mathsf{sk}, \gamma(v).\mathsf{pk})$, sometimes written as $(\mathsf{sk}_v, \mathsf{pk}_v)$ for simplicity; a vector of public keys $\mathrm{PK}^{\mathrm{pr}}$ called the *predecessor keys* which correspond to the public keys of the nodes in the resolution of v (defined below) in the round right before the current key pk_v was first introduced, see Section 3.4; a pair of hash values h_v called the parent hash of v; an identifier corresponding to the party ID_v generating the node's key pair; a signature σ_v under the private signing key of ID_v; a transcript hash value, $\mathcal{H}_{trans}$, committing to the state of ID_v at the time of sampling that node's key pair (defined below in Section 3.3); a confirmation tag value $\mathsf{confTag}$ (defined below in Section 3.4); an optional pair of hash values $o_v = (o_{v,1}, o_{v,2})$ corresponding to partial openings of a Merkle commitment sent by the server and encoding the state of the parent nodes of v; and a set of of so called *unmerged leaves* $\gamma(v).\mathsf{Unmerged}$, or simply $\mathsf{Unmerged}(v)$, corresponding to the leaves (and their associated public keys) of the subtree rooted at v whose users have no knowledge of sk_v (this will be the case, temporarily, for newly added users). In a slight abuse of notation, given a set of nodes S, we define its set of unmerged nodes to be $\mathsf{Unmerged}(S) = \cup_{v \in S} \mathsf{Unmerged}(v)$. Finally, the state of leaves l additionally contains a signing and verification key-pair $(\mathsf{ssk}_l, \mathsf{svk}_l)$ corresponding to the user ID_l associated to that leaf. For an internal node v, we will write ssk_v to refer to the secret signing key of party ID_v. For a summary of node states see Table 1.

The *secret part* of $\gamma(v)$ consists of sk_v, and ssk_v in case v is a leaf. In turn, the *public part*, ${}^{p}\gamma(v)$, of $\gamma(v)$ consists of $\gamma(v)$ minus the secret part. While the public part of nodes' states can be accessed by all users, users should only have partial knowledge of the secret parts. Indeed, the protocol ensures that the

[5] The non standard direction of the edges here captures that knowledge of (the secret key associated to) the source node implies knowledge of (the secret key associated to) the sink node. Note that nodes therefore have one child and two parents.

$(\gamma(v).\mathsf{sk} = \mathsf{sk}_v, \gamma(v).\mathsf{pk} = \mathsf{pk}_v)$	The node's key-pair
$(\mathsf{ssk}_l, \mathsf{svk}_l)$	Signing key-pair (Only present if v is a leaf)
$\mathrm{PK}^{\mathrm{pr}}$	vector of predecessor keys
$h_v = (h_{v,1}, h_{v,2})$	Parent hash value
ID_v	Identifier of party setting v's key
σ_v	Signature under ID_v's key
$\mathcal{H}_{trans,v}$	Transcript hash
$\mathsf{confTag}_v$	Confirmation tag
$o_v = (o_{v,1}, o_{v,2})$	Pair of partial Merkle commitment openings
$\gamma(v).\mathsf{Unmerged} = \mathsf{Unmerged}(v)$	Set of unmerged leaves keys

Table 1: State $\gamma(v)$ of non-blank node v.

secret part of $\gamma(v)$ is known only by users whose leaf is in the sub-tree rooted at v; this is known as the *tree invariant.*

Looking ahead, parties might end up (through a misbehaving delivery server) having different views on the state of a given node, and so we will refer to the view of party ID_i of v at round n as $\gamma_i^n(v)$.

Resolution and effective parents. The set of blank and non-blank nodes in a ratchet tree gives rise to the *resolution* of a node v. Intuitively, it is the minimal set $\mathcal{S}$ of non-blank nodes such that for each ancestor v' of v the set $\mathcal{S}$ contains at least one node on the path from v' to v. Formally, it is defined as follows.

Definition 2. *Let $\mathfrak{T}$ be a tree with vertex set $V_{\mathfrak{T}}$. The resolution $\mathsf{Res}(v) \subset V_{\mathfrak{T}}$ of $v \in V_{\mathfrak{T}}$ is defined as follows:*

- *If v is not blank, then $\mathsf{Res}(v) = \{v\}$.*
- *If v is a blank leaf, then $\mathsf{Res}(v) = \emptyset$.*
- *Otherwise, $\mathsf{Res}(v) = \cup_{v' \in \mathsf{parents}(v)} \mathsf{Res}(v')$*

In a slight abuse of notation, given a set of nodes V, we define the resolution of V to be $\mathsf{Res}(V) = \bigcup_{v \in V} \mathsf{Res}(v)$.

Re-keying. Users will often need to sample new keys along their leaves' paths. This is done following the MLS specification, through the hierarchical derivation captured in the algorithm $\mathsf{Re\text{-}key}(v)$ (Algorithm 1). Given a leaf v, it outputs a list of seeds and key-pairs for nodes along v's path. We use Δ_{root} or the expression *root seed* to refer to the seed associated to v_{root}. The algorithm will use two independent hash functions H_1 and H_2. These can be easily defined by taking a hash function H, fixing two different tags x_1 and x_2 and defining $\mathsf{H}_i(\cdot) = \mathsf{H}(\cdot, x_i)$.

3 The CoCoA Protocol

We start with a high level description of the CoCoA protocol in Section 3.1. Section 3.2 covers users' states and the key schedule, Section 3.3 robustness and the round hash, Section 3.4 the parent hash mechanism, and Section 3.5 formally defines the protocol procedures.

3.1 Overview

Concurrent updates in CGKA. To recover from compromise, CGKA protocols allow users to refresh the secret key material known to them. Broadly, a user does this by re-sampling all keys they know (those on the user's path in the case of a ratchet tree), encoding them in an update message, and sending this to

Algorithm 1: Re-key computes new seeds and keys along a path.

```
Input: A leaf node $v$ in a tree $\mathfrak{T}$ of depth $d$.
Output: A vector of hierarchically derived seeds, and another vector of corresponding keys for all
        nodes in $v$'s path to the root.
$\Delta_1 \xleftarrow{\$} \mathfrak{S}(\lambda)$        // $\lambda$ security parameter; $\mathfrak{S}(\lambda)$ seed space
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{PKE.Gen}(\mathsf{H}_2(\Delta_1))$
for $i \leftarrow 2$ to $d$ do
    $\Delta_i = \mathsf{H}_1(\Delta_{i-1})$
    $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{PKE.Gen}(\mathsf{H}_2(\Delta_i))$
$\Delta \leftarrow (\Delta_1, \ldots, \Delta_d)$
$\mathsf{K} \leftarrow ((\mathsf{sk}_1, \mathsf{pk}_1), \ldots, (\mathsf{sk}_d, \mathsf{pk}_d))$ return $(\Delta, \mathsf{K})$
```

the server, which broadcasts it to the other group members. However, it is unclear how to handle concurrent update attempts by several users.

As a first approach, it seems natural to simply reject all but one update. Using a fixed rule to determine whose update to implement, however, might lead to starvation, with users blocked from updating and thus not recovering from compromise (compare Fig. 1, column (a)). Even if parties that did not update for the longest time are prioritized, it may take a linear number of update attempts to fully recover security of the ratchet tree (compare Fig. 1, column (b)).

To amend this issue the MLS protocol introduced the "propose and commit" paradigm. Roughly, update proposals refresh a user's leaf key and signal the intent to perform an update. A commit then allows a user to implement several concurrent update proposals. While this allows the ratchet tree to fully recover within two rounds, this comes at the cost of destroying the binary structure of the tree, as, in order to preserve the tree invariant, nodes not on the path of the committing party are blanked. In the worst case, this can lead to future updates having a size linear in the number of parties (compare Fig. 1, column (c)).

The approach we take with the CoCoA protocol is to implement all updates simultaneously, albeit some of them only partially. Intuitively, while the ratchet tree might not fully recover immediately, every updating party still makes progress towards recovery; and after logarithmically many updates of every compromised user, security is restored (compare Fig. 1, column (d)).

Updates in the CoCoA protocol. The main idea in the CoCoA protocol is, given several concurrent update messages, to apply all of them simultaneously, while resolving conflicts by means of an ordering of the operations. As a consequence, some updates might only be applied partially. More precisely, the protocol parameters contain an ordering $\prec$. This could be, e.g. the lexicographic ordering, however, the particular choice does not affect our security results. Then, given a set of update messages $\{U_1, \ldots, U_k\}$, if a node in the ratchet tree would be affected by several U_i, the one that is minimal with respect to $\prec$ takes precedence and replaces its key pair. Consider the example of Fig. 2, in which the users A, C, G in a group of size 8 concurrently update, with C's update taking precedence over the other two. Note that since the updates are concurrent, new keys get encrypted to keys of the previous round. Assume, e.g., that C and G were compromised. Then, after the updates, all compromised keys are replaced. However, only the first three keys in C's and G's update paths are secure, while the new Δ_{root} was encrypted to an old, compromised key and hence is known to the adversary. So, while the ratchet tree did not fully recover, it made progress towards it. In Section 5 we discuss the security of CoCoA in more detail.

Adds and Removes. To be able to add users to and remove users from the group, CoCoA combines features from different versions of TreeKEM, together with a new rule for the server to handle concurrent operations. In general, it follows the approach of TreeKEM v7 in that it does not distinguish between Proposals and Commits. However, add operations are handled in a way reminiscent of the latter technique, as they are executed in two rounds: a first round where they get announced to the rest of group members,

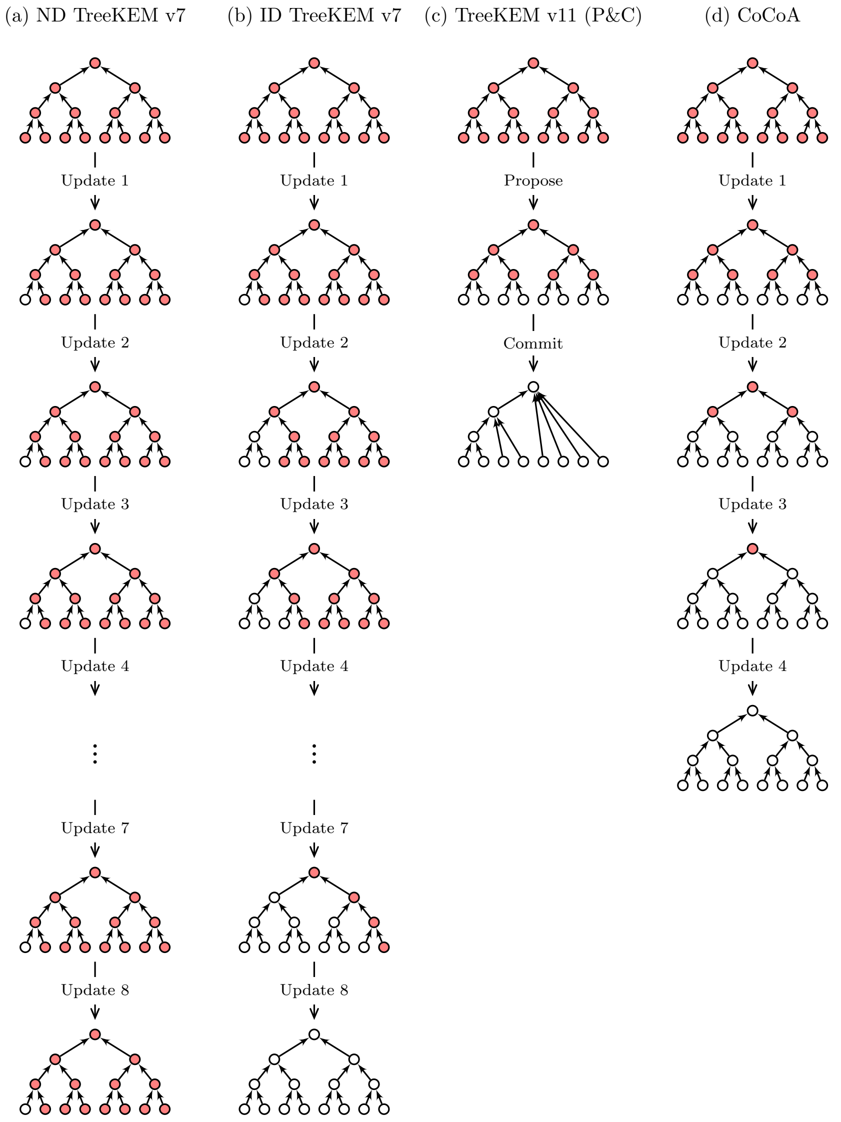

Figure 1: Comparison of number of rounds required to recover from corruption for different TreeKEM variants, ND stands for "Naïve Delivery", ID for "Ideal Delivery". Red nodes indicate key material known to the adversary. In each round all parties (try to) update. In columns (a) and (d) update requests are prioritized from left to right. In column (b) update requests are prioritized from left to right among all parties that did not update yet. In column (c) all parties propose an update, then the leftmost party commits.

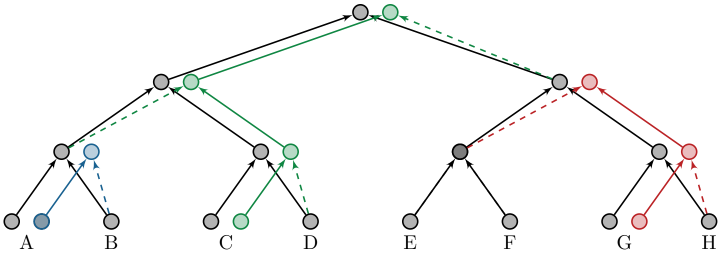

Figure 2: Example; concurrent updates in the CoCoA protocol. The former state of the ratchet tree (black) is changed by concurrent updates of A (blue), C (green), and G (red). The ordering is $U_C \prec U_A \prec U_G$. In the updates solid edges correspond to seeds obtained by hashing, dashed edges to encryptions.

and a second where the parties actually join the group, after receiving a *welcome message*. We stress that add operations taking two rounds seems to be an inherent consequence of allowing concurrency: an updating user cannot compute encryptions for a user added to the group in the same round by a different party. Moreover, adds are executed following the *unmerged leaves* technique, introduced in TreeKEM v9, which can be thought of as initially connecting a new user's leaf directly to v_{root} and progressively connecting it to lower nodes as the keys for these get rotated. As in MLS, it is assumed that parties have a public (list of) key(s) available to all other users, termed *init keys*, which can be used to add them to groups.

Removes can also be seen as a two-round process, as the removal of a party will cause v_{root} to get blanked, and an extra round will be needed to create a new group key - this is the same dynamic already existing in TreeKEM versions up to v7. Note that while this does not apply to later versions, these do need two rounds: one for the removal to be proposed, and another for it to be committed.

Note that concurrent operations could be conflicting: consider the case of two parties removing each other, a removed party adding a new one, or a removed party updating. Thus, special care needs to be taken in how to handle these conflicts. We refer the reader to Section 3.5 for more details.

Saving on communication complexity. A consequence of allowing concurrent operations is that arbitrarily many keys can change in a given round (even all the keys in the tree if all parties decide to update). In previous protocols, like MLS, every user stores a complete copy of the current ratchet tree. Sticking to this principle while allowing concurrent updates would imply that the communication cost of a round could be linear, as a party would have to download all new public keys in the tree. To avoid this, in the CoCoA protocol users only keep track of the state of nodes that are relevant to them when issuing an operation: those in their path and in the resolution of nodes in their co-path, since the latter will be the ones they need to encrypt to when sending an Update. As an important consequence, the server no longer only acts as a relay server broadcasting the same message to all users. Instead, given a set of Update, Add, and Remove operations, it prepares an individual packet for every user. We discuss the efficiency benefits of this approach in more detail in Section 4.[6]

Authenticity. Naturally, protocol messages have to be authenticated in any real-world deployment. In the MLS protocol, all users keep track of signature verification keys of the other users, and sign the Update, Add, and Remove messages that they generate. As these messages are no longer simply forwarded by the server, in the CoCoA protocol users sign every component of the message separately. Removes and Adds (and therefore Initialization messages) consist of a single block of information that everyone needs to receive, so a single signature suffices. In case of an Update, on the other hand, every ciphertext, and public key needs

[6] One could also imagine a variant of CoCoA that sticks closer to the principles of TreeKEM v11, by having the server forward complete update messages. While in this case t concurrently updating users would lead to a recipient complexity of $t \cdot \log(n)$, the protocol would still allow for concurrent updates that preserve the binary structure of the ratchet tree and thus outperform TreeKEM v11 for certain sequences of operations at the cost of slower healing.

to be signed individually. However, this increase in computational cost and size of sender packets allows us to greatly decrease recipient packets in the way discussed above.

Robustness. An important property CGKA protocols aim for is robustness: ensuring that parties have consistent views of the tree. TreeKEM achieves what [5] refers to as *weak robustness*: all honest parties accepting some message $\mathfrak{M}$ (potentially generated adversarially) will transition to compatible states.[7]

To capture the CoCoA protocol, the definition of weak robustness needs to be slightly adapted: parties now receive different personalized packages as opposed to a unique one that gets broadcast to everyone, and do not have access to the complete ratchet tree. Accordingly, we require that if two parties receive and accept messages $\mathfrak{M}_i$ and $\mathfrak{M}_j$ satisfying a certain relation, they will transition into consistent states (where malformed messages not satisfying this relationship will immediately force users into inconsistent states). TreeKEM achieves weak robustness through a value called *confirmation tag* [5]. This consists of a MAC of the entire CGKA transcript (encoded in a running hash, called the *transcript hash*) up to and *including* that epoch, which is sent together with every Commit message. The MAC key, a.k.a. the *confirmation key*, is derived from the new epoch key schedule, which ensures correct processing of the commit message and also that the sender had knowledge of the previous epoch's key schedule. To ensure consistency, users compute the transcript hash locally and verify the MAC. There are two issues when attempting to apply this to our scheme: 1) a user issuing an operation at a given round n will not have knowledge of the operations taking place concurrently, and thus will not be able to pre-compute the resulting transcript hash at the moment of crafting their message. And 2), since users only have a partial view of the ratchet tree, they are not able to compute the transcript hash. Note that users *need* to ensure they received consistent sets of operations, as e.g. in Figure 2, if C is not sent A's partial update, they will disagree on the key for node $\mathsf{Int}(A, B)$ after processing.

We solve 1) by effectively only authenticating the transcript up to the last round, i.e. not including the current operations. This ensures that if ID accepts a packet, it comes from a user whom they agreed with up until the beginning of that round. We solve 2) by what we call a *round hash*: a hash value computed over the public part of the new state of the ratchet tree (and any add and remove operations applied concurrently in that round). Clearly, none of the users can compute this hash value from their local state, so we shift this computation to the delivery server, who sends the round hash value to every party. However, the delivery server might act maliciously, and hence we need to ensure that the users can verify this computation. We do this by letting the round hash be a Merkle commitment to the current state. Users then expect the server to provide the openings of the commitment necessary to verify that it matches their partial view of the tree, which ensures consistency. See Section 3.3 for details.

Parent hash. TreeKEM aims for further security through two different mechanisms, mainly targeted at allowing new group members to verify the legitimacy of the received information. Informally, the first one, *parent hashing*, provides newly added users with the guarantee that the ratchet tree was well formed, i.e., that it resulted from a legitimate execution of the protocol. The second, *tree hashing*, consists of a locally-computed hash commitment to the whole ratchet tree, which the new users can verify upon joining. Our round hash can be seen as subsuming it, just with the difference that it can no longer be computed locally.

We adapt the parent hash construction from TreeKEM into our protocol, with some modifications. While TreeKEM relies on parent hash to ensure new users can verify the tree received when joining a group, we also use it to verify any new keys sent by the server to current users. This is needed since, as a result of blanks, the set of nodes whose states users need to keep track of varies.

3.2 Users' states and the Key Schedule

Each user keeps track solely of the state of nodes on either their path or the resolution of their co-path; we define $\mathcal{P}(\mathsf{ID}) = \mathsf{path}(\mathsf{ID}) \cup \mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$ to be the set comprising exactly those nodes. More in detail,

[7] We mention in passing that [5] also defines a notion of *strong robustness*, seemingly hard to achieve using practically efficient protocols, so we do not discuss it further.

$\gamma.\mathsf{ID}$	An identifier for the party.
$\gamma.G$	The set of current members of the group.
$\gamma.\mathsf{ssk}$	The party's signing key
$\gamma(v)$	Node state for every $v \in \mathcal{P}(\gamma.\mathsf{ID})$, only public part for $v \in \mathsf{Res}(\mathsf{co\text{-}path}(\gamma.\mathsf{ID}))$.
$\gamma.\mathcal{H}_{trans}$	Current value of the transcript hash.
$\gamma.\mathsf{appSecret}$	Current round's application secret.
$\gamma.\mathsf{confKey}$	Current round's confirmation key.
$\gamma.\mathsf{initSec}$	Current round's initialization secret.
γ'	Pending state encoding operations not yet confirmed.

Table 2: User's local state γ.

each user stores a local state γ, described in Table 2, which gets updated after every round message. We will write γ^n to refer to a state corresponding to round n.

Key schedule. CoCoA's *key schedule* for round n is defined via hash function H_5 as follows:

$$\begin{aligned}
\gamma.\mathsf{epochSecret}(n) &= \mathsf{H}_5(\gamma.\mathsf{initSec}(n-1) \mid\mid \Delta_{root}(n) \mid\mid \mathcal{H}_{trans}(n)) \\
\gamma.\mathsf{appSecret}(n) &= \mathsf{H}_5(\gamma.\mathsf{epochSecret}(n) \mid\mid \text{'appsecret'}\ ) \\
\gamma.\mathsf{confKey}(n) &= \mathsf{H}_5(\gamma.\mathsf{epochSecret}(n) \mid\mid \text{'confirm'}\ ) \\
\gamma.\mathsf{initSec}(n) &= \mathsf{H}_5(\gamma.\mathsf{epochSecret}(n) \mid\mid \text{'init'}\ )
\end{aligned}$$

The epoch secret $\gamma.\mathsf{epochSecret}(n)$ is used to derive all other keys from it; the application secret $\gamma.\mathsf{appSecret}(n)$ serves as the group key in epoch n and is to be used in higher level protocols, e.g. secure group messaging; the confirmation key $\gamma.\mathsf{confKey}(n)$ will be used to authenticate next epoch's protocol messages through a MAC termed the confirmation tag;[8] and the initialization secret $\gamma.\mathsf{initSec}(n)$ seeds next round's key schedule, tying it to the current one. Finally, the transcript hash $\mathcal{H}_{trans}(n)$ encodes the transcript of the execution up until round n - it is defined in the following section.

3.3 Robustness, Round Hash, and Transcript Hash

In this section we discuss CoCoA's robustness. We show that two parties accepting messages containing the same round hash value, will transition into consistent states. We start by defining the concept of a round hash and consistent states.

In the following we assume a fixed rule, that can be locally computed by the users on input a ratchet tree $\mathfrak{T}$ and a set of operations that determines a total ordering of said operations. This ordering ensures all users will compute the same round hash and also, when applied to adds A_i, determines the free leaf that the user added by A_i is assigned to.

Definition 3. *Let H_3 be a hash function, and n a round with associated protocol messages $T = (U, R, A) = ((U_1, \ldots, U_k), (R_1, \ldots, R_l), (A_1, \ldots, A_m))$, where the U_i correspond to Update messages; and the R_i and A_i correspond to the packets, as sent by their issuers, of any remove and add operation, respectively; and let each vector U, R, A be ordered with respect to the ordering $\prec$. Let $\mathfrak{T}^n$ be the ratchet tree resulting from applying the operations in T with respect to $\prec$ to $\mathfrak{T}^{n-1}$, and ${}^p\gamma(v)$ the public state of v in $\mathfrak{T}^n$ (note that ${}^p\gamma(v) = \mathtt{\textbf{blank}}$ if the node is to be blanked as a result of some removal in R). We define the map ℓ taking nodes in $\mathfrak{T}^n$ to labels as follows:*

$$\ell(v) = \begin{cases} \mathsf{H}_3({}^p\gamma(v)), & \textit{if } v \textit{ is a leaf.} \\ \mathsf{H}_3(\ell(\mathsf{lparent}(v)), \ell(\mathsf{rparent}(v)), {}^p\gamma(v)), & \textit{if } v \textit{ is an internal node.} \end{cases}$$

[8] This MAC, also present in TreeKEM, is there to mitigate active attacks. The latter are not reflected in our security model, but we chose to keep it, as it is the main security mechanism in response to a leaking of signature keys.

The round hash $\mathcal{H}_{round}(n)$ of n is defined to be

$$\mathcal{H}_{round}(n) = \mathsf{H}_3\left(\ell(v_{root}), R, A\right) \quad .$$

In short, the round hash is essentially a Merkle commitment to the ratchet tree's public keys and the round's dynamic operations. The benefit of this approach is that every user can verify that the round hash sent by the server faithfully encodes the operations affecting their local state, by just receiving at most a logarithmic number of values irrespective of the number of updates (note that a user will necessarily need to hear about all dynamic operations). In particular, a user ID receiving the appropriate group operations should have access to the inputs corresponding to dynamic operations, and to the new keys of nodes in $\mathcal{P}(\mathsf{ID})$. The server does this by sending the user $\mathcal{H}_{round}(n)$, as well as the output of openRH(Algorithm 2), which, on input a user ID, returns a vector of hash values, corresponding to the labels of nodes not in $\mathcal{P}(\mathsf{ID})$, but that are parents of a node in $\mathcal{P}(\mathsf{ID})$. Given these values, the user is able to verify the received round message by running verifyRH(Algorithm 3), which recomputes $\mathcal{H}_{round}(n)$ with respect to their updated ratchet tree and compares it to the round hash provided by the server. In order to formally define the above algorithms, we make use of the following helper functions:

- extract on input a round message $\mathfrak{M}$ and a state γ, outputs a list of updates $(U_1, \dots, U_p)$, removes $(R_1, \dots, R_q)$, and adds $(A_1, \dots, A_s)$, a list of indices $\mathtt{i}$ corresponding to the leaves users added by the A_i are assigned to, and a round hash value h.

- update-info on input vectors U, R, A of update, remove and add operations, ordering $\prec$, a state γ corresponding to ID and a node $v \in \mathcal{P}(\mathsf{ID})$, outputs a node public state ${}^{p}\gamma_v$ corresponding to the public state of v resulting from applying the input operations with respect to $\prec$ and state γ (${}^{p}\gamma_v = \mathtt{blank}$ if v is blanked as a result of some R_i); and, if v is a leaf node, additionally, outputs the identifier ID^* corresponding to v after applying the operations as above.

- retrieve-labels on input a local state γ, a node v and a vector O of tuples of the form (v_i, h, h'), outputs vector (v, h_l, h_r) if there is a unique vector in O with $v_i = v$, where $h_l = h$ and $h_r = h'$, and $\perp$ otherwise.

- tree on input a set of nodes V outputs the smallest subtree of $\mathfrak{T}$ which contains V.

- depth on input a local state γ and a node v, outputs the depth of v, as defined by the length of the path from v to v_{root} in $\mathsf{tree}(\mathcal{P}(\gamma.\mathsf{ID}))$.

Algorithm 2: openRH

```
Input: User $\mathsf{ID}$ and labeled ratchet tree $\mathfrak{T}_\ell$
Output: A vector $O$ of nodes and hash values.
$O := ()$
for $v \in \mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$ do
    if $v.\mathsf{isLeaf} = \mathsf{false}$ then
        $O \leftarrow O \cup (v, \ell(\mathsf{lparent}(v)), \ell(\mathsf{rparent}(v)))$
return $O$
```

Correctness of Algorithm 3 follows by inspection. The following lemma shows that we get the desired robustness. Note that if two users process a round message containing different round hash values, they will immediately be forced into inconsistent states, so we can just concern ourselves with the case where the round hash values are the same.

The transcript hash is defined as $\mathcal{H}_{trans}(0) = 0$, and, for subsequent rounds, given a verified round hash:

$$\mathcal{H}_{trans}(n) = \mathsf{H}_3(\mathcal{H}_{trans}(n-1) || \mathcal{H}_{round}(n)) \quad .$$

```
Algorithm 3: verifyRH
Input: Local state $\gamma$, round message $\mathfrak{M}$, and vector $O$ of nodes and hash values
Output: $b \in \{0,1\}$
$(U, R, A, \mathsf{i}, h) \leftarrow \mathsf{extract}(\mathfrak{M}, \gamma)$
for $d \in \{\mathsf{depth}(\mathsf{leaf}(\gamma.\mathsf{ID})), \ldots, 0\}$ do
    for $v \in \mathsf{tree}(\mathcal{P}(\gamma.\mathsf{ID}))$ s.t. $\mathsf{depth}(v) = d$ do
        if $v.\mathsf{isLeaf} = \mathsf{true}$ then
            ${}^{p}\gamma_v \leftarrow \mathsf{update\text{-}info}(U, R, A, \prec, \gamma, v)$
            $\ell(v) = \mathsf{H}_3({}^{p}\gamma_v)$
        else if $v \in \mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$ then
            ${}^{p}\gamma_v \leftarrow \mathsf{update\text{-}info}(U, R, A, \prec, \gamma, v)$
            $(v, h_l, h_r) \leftarrow \mathsf{retrieve\text{-}labels}(O, \gamma, v)$
            $\ell(v) = \mathsf{H}_3(h_l, h_r, {}^{p}\gamma_v)$
        else
            ${}^{p}\gamma_v \leftarrow \mathsf{update\text{-}info}(U, R, A, \prec, \gamma, v)$
            $\ell(v) = \mathsf{H}_3(\ell(\mathsf{lparent}(v)), \ell(\mathsf{rparent}(v)), {}^{p}\gamma_v)$
if $h = H(\ell(v_{root}), R, A, \lambda)$ then
    $b \leftarrow 1$
else
    $b \leftarrow 0$
return $b$
```

With this we can define what it means for parties to have consistent states, which informally requires them to have consistent views of the tree (i.e. agree on the states of nodes on the intersection of their states), and agree on the group key, group members, and group history, i.e. on the transcript hash.

Definition 4. *Let* ID *and* ID^* *be two group members with states* γ *and* γ^**. They have consistent states if* ${}^{p}\gamma(v) = {}^{p}\gamma^*(v)$ *for all* $v \in \mathcal{P}(\mathsf{ID}) \cap \mathcal{P}(\mathsf{ID}^*)$*),* $\gamma.\mathsf{appSecret} = \gamma^*.\mathsf{appSecret}$*, and* $(\gamma.G, \gamma.\mathcal{H}_{trans}) = (\gamma^*.G, \gamma^*.\mathcal{H}_{trans})$*.*

Note that we only define consistency of states for users who have joined the group. More in detail, we say that a user ID has (in their view) joined the group if there exists a query $\mathsf{CGKA.Proc}(\mathsf{ID}, \cdot)$ in the execution, where ID accepts the corresponding round message, i.e. where the state for ID changes (is initialized) as a result of said query.

The following proposition shows that we get the desired robustness. Note that if two users process a round message containing different round hash values, they will immediately be forced into inconsistent states, so we can just concern ourselves with the case where the round hash values are the same.

Proposition 1. *Let* ID_i *and* ID_j *be two group members with consistent local state after round* n *and let them receive round messages* $\mathfrak{M}_i$ *and* $\mathfrak{M}_j$ *respectively, both containing the same round hash value* h*, and opening vectors* O_i *and* O_j*. If* $\mathsf{verifyRH}(\gamma_i^n, \mathfrak{M}_i, O_i) = \mathsf{verifyRH}(\gamma_j^n, \mathfrak{M}_j, O_j) = 1$*, then they will have consistent states after processing their respective round messages.*

Proof. Assume for contradiction that the parties arrive to inconsistent states after processing their respective messages, we will show that this implies a collision for H. Since we assumed them to start the round in a consistent state, and the received round hash is the same, they will agree on the transcript hash. Similarly, agreement on the group ID and round follows trivially from the agreement in the previous round. Thus, disagreement over the group state must come from either membership or key material in the tree. First, let $(U_k, R_k, A_k, \prec_k, \lambda_k, h) \leftarrow \mathsf{extract}(\mathfrak{M}_k, \gamma_k)$ for $k \in \{i, j\}$ and denote by ℓ_k the map from nodes to labels as derived by party ID_k when running algorithm $\mathsf{verifyRH}$. Define $\alpha_k = (\ell_k(v_{root}), R_k, A_k, \lambda_k)$. We know from $\mathsf{verifyRH}(\gamma_i^n, \mathfrak{M}_i, O_i) = \mathsf{verifyRH}(\gamma_j^n, \mathfrak{M}_j, O_j) = 1$ that $H(\alpha_i) = H(\alpha_j)$. Now, if $\gamma_i^{n+1}.G \neq \gamma_j^{n+1}.G$, it follows that $(R_i, A_i) \neq (R_j, A_j)$, since $\gamma_i^n.G = \gamma_j^n.G$. In particular, this implies that $\alpha_i \neq \alpha_j$, i.e. a collision for H. Assume, therefore, that $\gamma_i^{n+1}.G = \gamma_j^{n+1}.G$ and that there is $v \in \mathcal{P}(\mathsf{ID}_i) \cap \mathcal{P}(\mathsf{ID}_j)$ such that ${}^p\gamma_i^{n+1}(v) \neq {}^p\gamma_j^{n+1}(v)$. If $h_i \neq h_j$, a collision for H follows as before, so assume that $h_i = h_j$. Let u, u' be the left and right parents, respectively, of v_{root}. By definition, we have that $h_k = H(\ell_k(u), \ell_k(u'), {}^p\gamma_k^{n+1}(v_{root}))$. If any of $\ell_i(u) \neq \ell_j(u)$, $\ell_i(u') \neq \ell_j(u')$ and ${}^p\gamma_i^{n+1}(v_{root}) \neq {}^p\gamma_j^{n+1}(v_{root})$ hold, a collision for H follows. Assume thus equality of the labels and public states, and assume w.l.o.g that v is on the left subtree, i.e. is an ancestor of u. We know that $\ell_i(u) = \ell_j(u)$, and that $\ell_k(u) = H(\ell_k(w), \ell_k(w'), {}^p\gamma_k^{n+1}(u))$, by definition, where w_k, w_k' are the left and right parents of u. Thus, as before, either we have some inequality between the corresponding values for i and j, from which a collision can be extracted, or the labels for the parents are the same for both parties. We can repeat this process until arriving to v, for which we know ${}^p\gamma_i^{n+1}(v) \neq {}^p\gamma_j^{n+1}(v)$. We will therefore eventually find a collision for H.

Therefore, an adversarial delivery server making ID_i and ID_j accept messages with a consistent round hash value, which prompt them into inconsistent states would equivalently be able to derive a collision for H. Since we assume H to be collision resistant, it follows that no such adversary exists. □

The next proposition shows that, in fact, it is to consider the group keys or transcript hash values in users states to determine if these are consistent.

Proposition 2. *If H_3 and H_5 are modeled as random oracles then, with all-but-negligible probability, the states γ and γ^* of users ID and ID^* are consistent if and only if $\mathsf{CGKA.Key}(\gamma) = \mathsf{CGKA.Key}(\gamma^*)$. Equivalently, if they have the same transcript hash value $\gamma.\mathcal{H}_{trans} = \gamma^*.\mathcal{H}_{trans}$.*

Proof. The only if implication is clear by definition. Assume thus that $\mathsf{CGKA.Key}(\gamma) = \mathsf{CGKA.Key}(\gamma^*)$. Since the application secret in their states is the same, so must be $\mathcal{H}_{trans}$, by collision resistance of H_5. If the states of the two parties differ in the group membership, then one of them must have processed some Add or Remove operation the other has not. These operations are directly hashed into the appropriate round hash, which itself gets hashed into $\mathcal{H}_{trans}$. Thus, there must have been a collision in either H_3 or H_5. Finally, the public states for all nodes in a user's state get hashed by the user in order to compute the round hash. As before, if any of the values in the states of nodes in the intersection of user's states $\mathcal{P}(\mathsf{ID}) \cap \mathcal{P}(\mathsf{ID}^*)$ differs, we can extract a collision for either H_3 or H_5. Hence, the states of the two users must be consistent.

Finally, to see that two users having the same transcript hash value implies them having the same epoch secret $\mathsf{CGKA.Key}(\gamma)$, recall that $\gamma.\mathsf{epochSecret}(n) = \mathsf{H}_5(\gamma.\mathsf{initSec}(n-1) \,||\, \Delta_{root}(n) \,||\, \mathcal{H}_{trans}(n))$, where $\gamma.\mathsf{initSec}(n) = \mathsf{H}_5(\gamma.\mathsf{epochSecret}(n) \,||\, \text{'init'})$. Moreover, a user always checks that the secret key at any node v, and in particular at v_{root}, is consistent with the public key set at that node. The former is derived from Δ_v, and the latter is input into the computation of the round hash, and therefore of $\mathcal{H}_{trans}$. Thus, were the seeds $\Delta_{root}(n)$ used by ID and ID^* to compute their respective epochs secrets different, so would be their transcript hash values, up to the negligible probability of collisions of $\mathsf{PKE.Gen}$ and the random oracle H_2 (since, recall, $(\mathsf{sk}_i, \mathsf{pk}_i \leftarrow \mathsf{PKE.Gen}(\mathsf{H}_2(\Delta_i)))$. Last, suppose for contradiction that the initialization secrets are different while their transcript hash values match. Since the $\gamma.\mathsf{initSec}$ are derived deterministically from $\gamma.\mathsf{epochSecret}$, if $\gamma.\mathsf{initSec}(n-1) \neq \gamma.\mathsf{initSec}^*(n-1)$, then we would have $\gamma.\mathsf{epochSecret}(n-2) \neq \gamma.\mathsf{epochSecret}^*(n-2)$. By the collision resistance of the random oracle H_3, $\mathcal{H}_{trans}(n) = \mathcal{H}_{trans}{}^*(n)$ implies $\mathcal{H}_{trans}(n-1) = \mathcal{H}_{trans}{}^*(n-1)$, so we could apply the same argument, eventually reaching the first epoch where the last one of them joined, i.e. to the first point in the execution where $\mathcal{H}_{trans} = \mathcal{H}_{trans}{}^*$. If at this point one of the users, say w.l.o.g. ID, was already a part of the group (i.e. this was not the value with which one of them initialized their state), then ID^* got both their $\gamma.\mathsf{initSec}^*$ and $\mathcal{H}_{trans}{}^*$ from another party ID', so we can repeat the argument with respect to ID and ID'. If, instead, these values corresponded to those with which both of them initialized their state, then by collision resistance, they must have been sent by different parties ID_1 and ID_2, for which the same must be true: their transcript hash values match, but not their initialization secrets. Eventually, since the number of users in the group is finite and there is a single initial user, the group creator, it must be that these two users are the same or were added by the same party with respect to the same state (by collision resistance of the random oracle H_3), which is a contradiction.

□

3.4 Parent Hash

Ratchet trees in TreeKEM contain so-called *parent hashes*, which were introduced to the standard in TreeKEM v9, and analyzed and improved by Alwen *et al.* [5]. These ensure, on the one hand, that for every node $v \in \mathfrak{T}$, whoever sampled sk_v had knowledge of the secret signing key for some leaf l of the subtree rooted at v; and on the other, that at the moment this secret was generated it was not communicated to any user whose leaf is not in this subtree. This protects against active attacks where a user is added to a malformed group where the tree invariant is violated, potentially causing him to communicate to a set of users different to the one he believes to be communicating to.

To adapt parent hash to CoCoA we have to overcome the two issues that (a), since parties update concurrently, parent hash values can be defined with respect to keys on the copath that were overwritten by a concurrent update, and (b), since the resolution of a user's copath and in turn the corresponding public keys that are known to the user may change from round to round, the user needs to be able to verify the authenticity of such keys without having access to the state of leaves below it. We address the first issue by having users store the public keys of one previous round: each node state $\gamma(v)$ now contains an associated list of predecessor keys, $\mathrm{PK}^{\mathrm{pr}}$, containing the public keys corresponding to nodes in the resolution of v in the epoch when the current key was sampled, and excluding those that where unmerged at $\mathsf{child}(v)$;[9] i.e. if the Update sampling pk_v unblanked v, the predecessor keys will be a list, else it will just contain the previous public key. The second issue we solve by not only signing the parent hash value of users' leaves but by introducing a signature at every node in their update path (that which is sent with the packet containing the new public key when it is first announced). Last, to ensure consistency between users' views, we add two further values to the parent hash and node state: a commitment to the subtree under the node's sibling and a commitment to the whole ratchet tree. We now define more formally the slightly modified parent-hash algorithm, compatible with our construction, with respect to signature scheme Sig.

As in TreeKEM, parent hash values of a node are updated whenever the key corresponding to the node is updated. More in detail, let ID compute an Update U containing new keys for nodes along their path (see full definition in Section 3.5), which get stored in pending state γ'. Parent hashing algorithm $\mathsf{PHash.Sig}$ on input (ID, γ') first fetches ID's update path $\mathsf{path}(v_{\mathsf{ID}}) = (v_0 = v_{\mathsf{ID}}, v_1, \ldots, v_k = v_{root})$. For $i \in \{0, \ldots, k-1\}$ let v'_i denote the parent of v_{i+1} that is not part of $\mathsf{path}(v_{\mathsf{ID}})$, and let $\mathcal{R} = \mathsf{Res}(v'_i) \setminus \mathsf{Unmerged}(v_{i+1})$. Then, we define $h_{1,k} = h_{2,k} = 0$, and using hash function H_4, compute:

$$\begin{aligned}
h_{1,i} &\leftarrow \ell(v'_i) && \text{for } i \in (k-1, \ldots, 0)\\
h_{2,i} &\leftarrow \mathsf{H}_4(\mathsf{pk}_{v_{i+1}}, \mathrm{PK}^{\mathrm{pr}}_{v_{i+1}}, h_{2,i+1}, \{\mathsf{pk}_v\}_{v \in \mathcal{R}}) && \text{for } i \in (k-1, \ldots, 0)\\
\sigma_i &\leftarrow \mathsf{Sig.Sig}\left(\gamma(\mathsf{ID}).\mathsf{ssk}, (\mathsf{pk}_{v_i}, \mathrm{PK}^{\mathrm{pr}}_{v_i}, (h_{1,i}, h_{2,i}), \mathcal{H}_{trans}, \mathsf{confTag})\right) && \text{for } i \in (0, \ldots, k)
\end{aligned}$$

where $\ell(v)$ is the label of v as in Def. 3 above, $\mathrm{PK}^{\mathrm{pr}}_v \leftarrow 0$ if v did not have a key before U, $h_i = (h_{1,i}, h_{2,i})$.

Algorithm $\mathsf{PHash.Sig}$ then adds the values $(H, \Sigma) = (h_0, \ldots, h_k, \sigma_0, \ldots, \sigma_k)$ to U, substitutes the parent hash values h_i and signatures σ_i in γ' by the newly computed ones, and returns U.

Verification. A user receiving a tree T from the server can verify its authenticity by running the algorithm $\mathsf{PHash.Ver}(T)$. This will be run by users in two different scenarios: on the one hand, when joining the group, they will verify the whole ratchet tree (in this case $T = \mathfrak{T}$); on the other, when processing a round message containing one or more Removes, they will verify the received keys for nodes in the new resolution of their co-path (in this case T is the union of $\mathcal{P}(\mathsf{ID}_i)$ for all removed ID_i). The algorithm runs as follows:

The algorithm first checks that all non-blank nodes in the tree have a complete public state, and that for any internal node v, the associated identifier ID_v is associated to one of the leaves of the sub-tree rooted at v.[10] If any of these checks does not pass, the algorithm aborts. Next, it checks that $h_{2,v_{root}}$, and then, verifies the following conditions hold:

[9] The exclusion of these unmerged leaves responds to the fact that these could correspond to parties added *after* the state for $\mathsf{child}(v)$ was last updated.

[10] A user who is already part of the group will have knowledge of the leaf index of each group member, and can check this without necessarily having a full view of the tree.

1. For any non-blank non-leaf node v in T the following equalities hold with either p and p' being the left and right parents of v or, if not, with p' being the left parent of v and p the right parent, setting $p \leftarrow \mathsf{lparent}(p)$ if p is blank, until p is either non-blank or an empty leaf, in which case $0 \leftarrow \mathsf{PHash.Ver}(T)$.[11]

   (a) $h_{2,p} = \mathsf{H}_4(\mathsf{pk}_v, \mathrm{PK}_v^{\mathrm{pr}}, h_{2,v}, \{\mathsf{pk}_w\}_{w \in R})$ and $h_{1,p} = \ell(p')$ or

   (b) $h_{2,p} = \mathsf{H}_4(\mathsf{pk}_v, \mathrm{PK}_v^{\mathrm{pr}}, h_{2,v}, \mathrm{PK}_{p'}^{\mathrm{pr}})$ and $\mathcal{H}_{trans,p} = \mathcal{H}_{trans,p'}$.

   where $R = \mathsf{Res}(p') \setminus \mathsf{Unmerged}(v)$.

2. For every node $w \in T$, $\mathsf{Sig.Ver}_{\mathsf{svk}_w}((\mathsf{pk}_w, \mathrm{PK}_w^{\mathrm{pr}}, h_w, \mathcal{H}_{trans,w}, \mathsf{confTag}_w), \sigma_w) = 1$.

We say that p and p' as defined above are the *effective parents* of v. If p is the effective parent satisfying condition 1, we say that v *verifies through* p, and note that this corresponds to the situation where p's key was sampled in the same Update as v's. Observe that if we have two concurrent updates, the parent hash value of the node at the intersection cannot possibly have been computed with respect to the new key in its co-path, as these were generated at the same time, by different parties. Thus, the parent hash value will be computed w.r.t. the predecessor key - condition (b) corresponds to this case. Moreover, we check that in this case the users computing the concurrent updates where actually in consistent states, by checking that transcript hash values they included in their update packets are the same. In case v is not the intersection between two concurrent updates, the public state of the nodes in the resolution of the co-parent will not have changed, so we only need to check with respect to the newest key in the resolution nodes. However, we also check that the user who sampled v indeed had in their view the same (commitment to) subtree under the parent of v not in their path, by checking that the parent hash value h_1 is consistent with the label of p'. Finally, unmerged leaves are excluded from R since these might differ between the time the parent hashes were computed and the new user joins the group.

3.5 The Protocol: CoCoA and Partial Updates

In the description below, we use γ for the state of the party issuing the appropriate operation. The ordering used to resolve conflicts caused by concurrent updates is denoted by $\prec$. An overview of the content of user-generated messages can be found in Table 3, and one for the contents of round messages in Table 4.

Initialization. To initialize a group with parties $G = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$, ID_1 creates a ratchet tree as follows. First, ID_1 retrieves the public initialization keys $(\mathbf{pk}, \mathbf{svk}) = (\{\mathsf{pk}_{\mathsf{ID}_1}, \ldots, \mathsf{pk}_{\mathsf{ID}_n}\}, \{\mathsf{svk}_{\mathsf{ID}_1}, \ldots, \mathsf{svk}_{\mathsf{ID}_n}\})$ of all group members (including themselves), redefines $G \leftarrow (G, \mathsf{pk}, \mathsf{svk})$ to include these, and initializes a left-balanced binary tree with n leaves, assigning each pair of keys in $(\mathbf{pk}, \mathbf{svk})$ to a leaf. Let v be ID_1's leaf. They then sample new secrets for v's path $(\Delta, K) \leftarrow \mathsf{Re\text{-}key}(v)$, store the new keypairs $(\mathsf{sk}_j, \mathsf{pk}_j)$ in the corresponding nodes on the created tree and compute and store in γ' the parent hashes and signatures for the nodes in $\mathsf{path}(v)$: $(H, \Sigma) \leftarrow \mathsf{PHash.Sig}(\mathsf{ID}_1, \gamma')$, where recall that each $\sigma_j \in \Sigma$ is a signature of $(\mathsf{pk}_j, 0, h_j, \mathcal{H}_{trans}, 0)$ for some $v_j \in \mathsf{path}(v)$ with $h_j \in H$ its corresponding new parent hash pair (here $\mathrm{PK}_v^{\mathrm{pr}}$ and $\mathsf{confTag}$ are set to 0 initially). For every $v_j \in \mathsf{path}(v) \setminus v$, let w_j be the parent of v_j not in $\mathsf{path}(v)$. Then, for each $y_{j,l} \in \mathsf{Res}(w_j)$, ID_1 computes $e_{j,l} = \mathsf{PKE.Enc}(\mathsf{pk}_{y_{j,l}}, \Delta_j)$, together with the signature $\sigma_{j,l}^s = \mathsf{Sig.Sig}_{\mathsf{ssk}_i}(e_{j,l})$. Next, they send out the initialization message $I = ('init', \mathsf{ID}_1, G, \mathbf{pk}, P, S)$; where P is the vector with entries $p_j = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \sigma_j)$, one per node in $\mathsf{path}(\mathsf{ID}_1)$; and S is the vector with entries $s_{j,l} = (e_{j,l}, \sigma_{j,l}^s)$, containing all the necessary encryptions and values to be authenticated, for each $v_j \in \mathsf{path}(v)$. Finally, ID_1 erases the seeds Δ_i, and sets all internal nodes in their local copy of the tree which are not in their path to be blank.

[11] The recursion in the second case is needed to account for the possible blank nodes introduced between p and v as a result of adding to new leaves to accomodate new parties, so that p and p' correspond to the parents of v at the time the state of v was created.

Init	
ID	The identifier of the group creator.
G	List of group members.
$P = (p_j = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \sigma_j) \ : \ j \in [\lvert\mathsf{path}(\mathsf{ID})\rvert])$	Public states for every $v_j \in \mathsf{path}(\mathsf{ID})$.
$S = (s_{j,l} = (e_{j,l}, \sigma^s_{j,l}) \ : \ j \in [\lvert\mathsf{path}(\mathsf{ID})\rvert],\ l \in L_j)$	Encryptions of the seeds along ID's path.
Update	
ID	The identifier of the updating user.
c_i	Update counter
$P = (p_j = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \mathsf{confTag}, \sigma_j) \ : \ j \in [\lvert\mathsf{path}(\mathsf{ID})\rvert])$	New public states for every $v_j \in \mathsf{path}(\mathsf{ID})$.
$S = (s_{j,l} = (e_{j,l}, \mathsf{confTag}, \sigma^s_{j,l}) \ : \ j \in [\lvert\mathsf{path}(\mathsf{ID})\rvert],\ l \in L_j)$	Encryptions of the new seeds.
Remove/Add	
ID	Identifier of removed/added party.
$\mathsf{confTag}$	Confirmation tag.
σ	Signature of message contents under sender's signing key.
$\mathsf{pk}_{\mathsf{ID}}$	Public key of added party (Only adds).
$\mathcal{W} = (\mathcal{H}_{round}, \gamma.\mathcal{H}_{trans}, \gamma.G, \gamma.\mathsf{confKey}, \gamma.\mathsf{initSec})$	Welcome message (Sent next round, signed, only adds).

Table 3: Contents of user generated messages.

Update. To issue an update, user ID_i with state γ and at leaf v, first computes new secrets along their path $(\Delta, K) \leftarrow \mathsf{Re\text{-}key}(v)$, stores the new keys in γ' and computes and stores the parent hashes and signatures for the nodes in $\mathsf{path}(v)$: $(H, \Sigma) \leftarrow \mathsf{PHash.Sig}(\mathsf{ID}_i, \gamma')$. Second, they set $\mathsf{confTag} = \mathsf{MAC.Tag}(\gamma.\mathsf{confKey}, \gamma.\mathcal{H}_{trans})$. For every $v_j \in \mathsf{path}(v) \setminus v$, let w_j be the parent of v_j not in $\mathsf{path}(v)$ and let $L_j = \mathsf{Res}(w_j) \cup \mathsf{Unmerged}(\mathsf{Res}(w_j))$ be the set of nodes that are either in the resolution of w_j or are leaves that are unmerged at some node in said resolution. Then, for each $y_{j,l} \in L_j$, ID_i computes $e_{j,l} = \mathsf{PKE.Enc}(\mathsf{pk}_{y_{j,l}}, \Delta_j)$, together with the signature $\sigma^s_{j,l} = \mathsf{Sig.Sig}_{\mathsf{ssk}_i}(e_{j,l}, \mathsf{confTag})$. Next, they send out the update message $U = (\mathsf{ID}_i, P, S, c_i)$; where P is a vector of entries $p_j = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \mathsf{confTag}, \sigma_j,)$ containing the new public states and necessary authentication values for each $v_j \in \mathsf{path}(v)$;[12] S is the vector with entries $s_{j,l} = (e_{j,l}, \mathsf{confTag}, \sigma^s_{j,l})$, containing all the necessary encryptions and values to be authenticated, for each $v_j \in \mathsf{path}(v)$; and a counter c_i, the number of updates (including this one) sent by ID_i since they last processed a round message. Last, they erase the seeds Δ.

Remove. To remove party ID_j, ID_i sends out a $\mathit{remove}(\mathsf{ID}_j)$ plaintext request together with $\mathsf{confTag} = \mathsf{MAC.Tag}(\gamma.\mathsf{confKey}, \gamma.\mathcal{H}_{trans})$, and a signature σ under their signing key of the remove message and the confirmation tag. This will have the effect of blanking the nodes in ID_j's path. Following a removal, an Update operation must be issued immediately so that a new group key is created.

Add. Additions of parties work in two rounds. To add party ID_j, ID_i first sends a plaintext add request $\mathit{add}(\mathsf{ID}_j, \mathsf{pk}, \mathsf{svk})$ containing ID_j's public init key pair $(\mathsf{pk}, \mathsf{svk})$, $\mathsf{confTag} = \mathsf{MAC.Tag}(\gamma.\mathsf{confKey}, \gamma.\mathcal{H}_{trans})$ and a signature under ID_i's signing key of the add request and the confirmation tag. This will allow all group members to learn the identity of the new party and therefore to encrypt future protocol messages to them. In the following round, ID_i[13] must send ID_j a signed welcome message $\mathcal{W} = (\mathcal{H}_{round}, \gamma.\mathcal{H}_{trans}, \gamma.G, \gamma.\mathsf{confKey}, \gamma.\mathsf{initSec})$, encrypted under pk, containing the necessary information to initialize their local state and seed that round's key schedule, as well as check the received tree from the server is correct. Moreover, some user must send an Update during that round, ensuring that way that a new application secret will be created.

[12] note that, as in an initialization message, the signature included in each of the p_j does not exactly cover the rest of the elements of p_j, but also includes the predecesor key $\mathrm{PK}^{\mathrm{pr}}$ at that node. This is not a problem for verification, as this is set to 0 for new groups, and in any other cases, parties will have access to the key at that node before they processed said update.

[13] an alternative specification could allow any group member online to do this instead

R	Vector of Remove operations
A	Vector of Add operations
c_i	Update counter
$\mathcal{H}_{round}$	Round hash value
O_i	Openings to verify $\mathcal{H}_{round}$
$\gamma(v)$ for $v \in N_i$	public states of nodes needed to update $\mathcal{P}(\mathsf{ID}_i)$
$u_v = (\mathsf{ID}, p, s)$	public state and encryption (if appropriate) of each relevant updated v
$\mathcal{W}$	Welcome message (only for new joiners)
$\mathfrak{T}$	Entire ratchet tree (only for new joiners)

Table 4: Contents of Round message $\mathfrak{M}_i$ to party ID_i.

Collect and Deliver. Whenever the server receives an initialization message I, it just forwards it to all the new group members, initializing its local state γ_{ser} with the members of the new group and the public information of the ratchet tree included in I. For all other messages, it does as follows: given concurrent group messages $T = (U, R, A, W) = (U_a, R_b, A_c, W_d : a \in [p], b \in [q], c \in [r], d \in [s])$ sent during a round, corresponding to Updates, Removes, Adds, and Welcome messages, respectively, the delivery server will first check if any two or more updates come from the same user, deleting all of them except for the one received last. The server first updates its local copy of the public state of $\mathfrak{T}$, stored in γ_{ser}, by updating the public keys of nodes refreshed by any U_a, blanking any nodes affected by any R_b, and adding a public key and identifier to any leaf newly populated as a result of an A_c; here if two or more operations affect a given node, the operation that is minimal with respect to $\prec$ will be the one determining the state of the node. A few considerations must be observed here, which we discuss further below: first, all Removes must precede any Updates, so that a node is blanked whenever a leaf under it is removed, irrespective of which Updates take place; second, conflicting Removes take effect simultaneously, blanking nodes in both paths; third, new users are added on the left-most free leaves in the tree according to some fixed rule that the receiving parties can reproduce locally. Once the server's view $\mathfrak{T}$ is updated, it computes the labels for it, defining $\mathfrak{T}_\ell$, and the round hash $\mathcal{H}_{round}$, as prescribed in Definition 3; and computes opening vectors $O_i \leftarrow \mathsf{openRH}(\mathsf{ID}_i, \mathfrak{T}_\ell)$ for all group members $\mathsf{ID}_i \in G$ (note that these will be computed with respect to the set $\mathcal{P}(\mathsf{ID}_i)$ resulting from (un)blanking nodes as implied by T). Then, it crafts round messages $\mathfrak{M}_i$ for each user, containing the following information: first, the vectors R and A or Removes and Adds; second the vector O_i and the round hash $\mathcal{H}_{round}$; third, the public states $\gamma(v) = (\mathsf{pk}_v, \mathrm{PK}_v^{\mathrm{pr}}, h_v, \mathsf{ID}_v, \sigma_v, \mathcal{H}_{trans,v}, \mathsf{confTag}_v, o_v, \mathsf{Unmerged}(v))$ at the beginning of the round of the nodes $v \in \mathcal{N}_i = (\cup_{j \in R_{id}} \mathcal{P}(\mathsf{ID}_j)) \setminus \mathcal{P}(\mathsf{ID}_i)$ where R_{id} is the set of indices of parties removed by R, i.e., the new nodes on the resolution of ID_i's and the extra states needed to verify the validity of the received keys[14]; and fourth, for each node $v \in \mathcal{P}(\mathsf{ID}_i)$ (after the (un)blanking implied by T) whose keys get rotated as a result of some (winning w.r.t. $\prec$) update $U_a = (\mathsf{ID}, P, S)$, the server adds $u_v = (\mathsf{ID}, p_j)$ to $\mathfrak{M}_i$, where $p_j \in P$ is the public state of corresponding to v; if, besides, $v \in \mathsf{path}(\mathsf{ID}_i)$ and is the lowest node in $\mathsf{path}(\mathsf{ID}_i)$ updated by U_a, the server also includes the tuple $s_{j,l} \in S$ into u_v, corresponding to the encryption of v's seed to the node in $\mathsf{path}(\mathsf{ID}_i)$ which is also in the resolution of the parent of v not on U_a's path. Last, the server will also include a counter c_i, equal to that of the update by ID_i included in $\mathfrak{M}_i$ if there is one, and equal to 0 otherwise. Finally, for each ID_i, recipient of a welcome message $\mathcal{W}$, the round message $\mathfrak{M}_i$ will additionally contain $\mathcal{W}$, as well as a copy of the public state of $\mathfrak{T}$.

Process. Upon receipt of a round message $\mathfrak{M}$ containing associated Updates $U = (U_1, \ldots, U_p)$, Removes $R = (R_1, \ldots, R_q)$, Adds $A = (A_1, \ldots, A_r)$, openings vector O, public states for nodes in $\mathcal{N}$, round hash $\mathcal{H}_{round}$, and counter c, user ID processes it as follows. First, if $c \neq 0$, they check if, from the time they last processed a round message, they issued an update with counter c, aborting if not. Next, for every update,

[14] note that the leaves of the sub-tree of $\mathfrak{T}$ with vertex set $\mathcal{N}_i$ correspond to the new nodes in the resolution of ID that were not part of their state

remove and add, they check that $\mathsf{MAC.Ver}(\gamma.\mathsf{confKey}, \mathsf{confTag}) = 1$; that for the all update packets U_a the transcript hash value included with the new public values for a node is the same as $\gamma.\mathcal{H}_{trans}$; and that the associated signature verifies under the public key of the sender (using the current node key in place of $\mathrm{PK}^{\mathrm{pr}}$ to verify signatures of updates); and similarly abort if any of these verifications does not pass. [15] If these checks pass, they copy their local state γ corresponding to the current round to γ', incorporating into it any node states previously stored there as part of the generation of said update with counter c (this update is empty if $c = 0$). Then, they update the public state of nodes needed to verify the round hash, as prescribed by the received operations: first, for every $v \in \mathcal{P}(\mathsf{ID})$, they blank v if it is in the path affected by some R_i and update $\mathcal{P}(\mathsf{ID})$ to include its new resolution as follows: they check that the set of nodes $\mathcal{N}$ consists of the nodes outside $\mathcal{P}(\mathsf{ID})$ that are in the paths and resolutions of co-paths of removed users. If more than one user is removed, it could be that $\mathcal{N}$ consists of several disconnected subtrees of $\mathfrak{T}$. For each such subtree T, ID checks that its leaves are all non-blank; that all the leaves (w.r.t to $\mathfrak{T}$) of removed parties as described in R are included in it; and, finally, that $\mathsf{PHash.Ver}(\gamma, T) = 1$. Moreover, for each blanked node w (as a result of $\mathfrak{M}$), they will use the received openings for the leaves of T, together with the received states, to reconstruct the Merkle hash openings o_v associated to v and check that the stored values match these. If all the checks pass, ID incorporates in γ' the public states of the nodes in $\mathcal{N}$ that belong to the new nodes in $\mathcal{P}(\mathsf{ID})$, together with the received openings for each such node, and aborts otherwise. Next, if any v in the new $\mathcal{P}(\mathsf{ID})$ set is affected by an update U_a, they overwrite its public key, parent hash value, signature, identifier, transcript hash value, and confirmation tag to the one set by U_a, and update the unmerged leaves and predecessor keys appropriate; and else, if corresponding to a newly populated leaf, determine the corresponding added party from (U, R, A) and add the new public key and identifier ID^* to the leaf. If several nodes in their state are affected by updates, they also check that for every such node in their path, the update setting a new state for it is the same setting a state for one of its parents. Once the updating of the public state of $\mathfrak{T}$ is done, they run $\mathsf{verifyRH}(\gamma', \mathfrak{M})$, aborting if the output is 0. Once those verifications are passed, for all nodes affected by some U_i, they decrypt the appropriate seed, derive the new key-pairs from it as in algorithm $\mathsf{Re\text{-}key}$, check that the received public key matches the derived one, aborting if not, and otherwise, overwrite the public and secret keys with them; set $\mathsf{Unmerged}(v) \leftarrow \emptyset$, and then $\mathsf{Unmerged}(v) \leftarrow \mathsf{Unmerged}(v) \cup l_i$ for each leaf l_i that is an ancestor of v corresponding to an added party. After that, they update $\gamma.G$ to account for membership changes as per R and A. Finally, they compute the key schedule for the current round, set $\gamma \leftarrow \gamma'$, deleting both the old key schedule and the old key material from node states, and delete $\gamma' \leftarrow \emptyset$.

If the user is not yet part of the group, $\mathfrak{M}$ will also contain a welcome message $\mathcal{W} = (\mathcal{H}_{round}, \mathcal{H}_{trans}, G, \mathtt{confKey}, \mathtt{initSec})$ together with a copy of the public state of the ratchet tree $\mathfrak{T}$, allowing the user to initialize their state prior to executing the instructions above. The newly added user ID_i will first check that G matches the leaf identifiers in $\mathfrak{T}$, compute the round hash from $\mathfrak{T}$, R and A as in Def. 3, and check that it matches the received value $\mathcal{H}_{round}$ (and skip this step when later processing the rest of the round message). If any of these checks fails, the user immediately aborts. Next, they will initialize their state γ by setting $\gamma.\mathsf{ID} \leftarrow \mathsf{ID}_i$, $\gamma.\mathcal{H}_{trans} \leftarrow \mathcal{H}_{trans}$, $\gamma.G \leftarrow G$, $\gamma.\mathsf{confKey} \leftarrow \mathtt{confKey}$, and $\gamma.\mathsf{initSec} \leftarrow \mathtt{initSec}$. Finally, they set the state $\gamma(l)$ of the leaf l to contain the init key with which they were added - note that they will not have at this point knowledge of the secret keys of any other node, but they will obtain some as soon as they process any U_a. When doing so, note that for the verification of the signature they will need to make use of the keys in $\mathfrak{T}$. Last, to process an initialization message $I = ('init', \tilde{\mathsf{ID}}, G, P, S)$, ID verifies the parent hash for the node public states in P, using $\mathrm{PK}_v^{\mathrm{pr}} = 0$ for all nodes $v \in \mathsf{path}_{\tilde{\mathsf{ID}}}$, derives the keys for $\tilde{\mathsf{ID}}$'s path from S, and creates a ratchet tree with users in G as leaves and the obtained keys. Last, they initialize the key schedule, with initial value 0 for $\gamma.\mathsf{initSec}$ and $\mathcal{H}_{trans}$, storing all in the newly created state γ.

Get group key. To extract the current group key a user ID with local state γ fetches $K = \gamma.\mathsf{appSecret}$.

[15] Observe that this could allow an active adversary to continuously send inconsistent messages, preventing users from updating. Since this falls outside of our model, we do not consider it here for simplicity, but note that it could be prevented by having users process all operations that do verify and compute an updated round hash, hashing together the received value and the operations that failed verification, inputting this into the transcript hash instead. This would ensure that parties agree on the transcript hash if and only if they processed exactly the same operations.

Protocol type	Rounds to heal t corruptions	Cumulative sender communication		Per-user recipient communication	Subsequent per-user update cost	
		no coordination	coordination		worst	average
(a) corrupted parties unknown						
Original TreeKEM & variants [1, 6, 23, 24]	n	$n^2 \log(n)$	$n \log(n)$	$n \log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [6]	2	n^2	n	n	n	n
Bienstock et al. [9]	2	n^2	n	$n^\dagger$	$\log(n)^\dagger$	$\log(n)$
Bidirectional channels [28]	2	n^2	n^2	n	n	n
This work	$\lceil \log(n) \rceil + 1$	$n \log^2(n)$	$n \log^2(n)$	$\log^2(n)$	$\log(n)$	$\log(n)$
(b) corrupted parties known						
Original TreeKEM & variants [1, 6, 23, 24]	t	$t^2 \log(n)$	$t \log(n)$	$t \log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [6]	2	$t^2(1 + \log(n/t))$	$t(1 + \log(n/t))$	$t(1 + \log(n/t))$	$t(1 + \log(n/t))$	$\frac{t^2+(n-t)\log(n)}{n}$
Bienstock et al. [9]	2	$t^2(1 + \log(n/t))$	$t(1 + \log(n/t))$	$t(1 + \log(n/t))^\dagger$	$\log(n)^\dagger$	$\log(n)$
Bidirectional channels [28]	2	tn	tn	t	n	n
This work	$\lceil \log(n) \rceil + 1$	$t \log^2(n)$	$t \log^2(n)$	$\log(n) \cdot \min(t, \log(n))$	$\log(n)$	$\log(n)$

Table 5: Comparison of the communication complexity of different CGKA protocols. For a detailed discussion of the table see Section 4. The values x depicted in the last 5 columns are to be understood as $\mathcal{O}(x)$. We assume that the ratchet-tree based protocols start with a fully unblanked tree. †: In the uncoordinated case, the protocol's recipient communication is n^2 (case (a)) and $t^2(1 + \log(n/t))$ (case (b)), respectively. Regarding the subsequent update cost, while the protocol formally has a worst case subsequent update cost of $\log(n)$, it is only secure in a weak security model. Modifying it to obtain PCS guarantees similar to the other protocols, e.g. by tainting [23], would lead to future worst-case update cost of n (case (a)) and $t(1 + \log(n/t))$ (case (b)), respectively.

Handling concurrent changes to the group membership. Regarding the considerations on handling concurrent dynamic operations in the collect and deliver operation, note that it is mandatory that Removes precede Updates, as the new keys sampled by the latter might be encrypted to keys under the knowledge of some of the removed parties; thus, if the node is not blanked, there is not guarantee the removed party will no longer have knowledge of any node's state - our restriction on the ordering prevents this, enforcing that a node is blanked whenever a leaf under it is removed, irrespective of which Updates take place. With regards to conflicting removes, i.e., those where the removed party in one is the remover in the other, so that processing the one would render the other syntactically incorrect, this could be left up to the group policy. For example, if two parties ID_i and ID_j concurrently remove each other, different policies could be a) both take effect, b) none take effect and c) only one takes effect. We consider the first option to be the most desirable, e.g. so users could not avoid being removed by issuing removals of other parties, and so apply this one in the protocol.

4 Efficiency

In this section we discuss the communication complexity of our protocol and compare it with other CGKA schemes. We focus on the cost incurred by several users updating concurrently to recover from compromise, as this is the main setting we aim to tackle with this work. An overview is given in Table 5.

Considered setting. Not only does the sequence of operations preceding concurrent update operations (in the case of ratchet-tree based CGKA schemes) have a crucial impact on the resulting communication cost, but also, whether the participating parties know which of the other parties have been compromised and when they are planning to update. Thus, there are plenty of different settings one could compare. We restrict our view to the following, that in our opinion are quite natural.

We consider a group of n users, t of which have been compromised. For ratchet-tree-based protocols we assume that the tree is fully unblanked/untainted, as this should typically be the case, with Updates being the most common operation. Our analysis differentiates between the settings (a) where it is only known that the group has been compromised, but not who the particular t corrupted users are, and (b) where the set of compromised users is known to everyone. Note that the former essentially forces every member of the group to update, while in the latter scenario only the t compromised users have to act.

The first value we are interested in is the number of rounds of (potentially) concurrent updates, after which the group key is guaranteed to be secure again. The second is the cumulative sender complexity (measured over all rounds), which essentially corresponds to the number of public keys and ciphertexts sent to the server. Here, we again distinguish between two settings. Namely, whether the parties act coordinated or not. In the latter case the participating parties are not aware of whether other parties are concurrently preparing updates/commits, which, depending on the scheme, potentially leads to the server having to reject packages. In the former case, on the other hand, they have this knowledge. In practice, this could be implemented by introducing an additional mechanism, that requires parties to wait for a confirmation by the server before preparing and sending update packages. We further track the per-user recipient communication complexity, again measured as a total over all rounds required to recover from compromise. The final considered value is the sender communication cost of a single, non-concurrent, update/commit in a subsequent round. Here, we state both the cost of the worst-case party as well as the average cost.

In Table 5 we mark schemes that perform substantially better or worse in one of the categories in green and red, respectively.

The communication complexity of CoCoA. We first discuss the number of rounds required to recover from compromise of t users. As we will show in Section 5, it is sufficient for the group to recover that all corrupted users concurrently update in $\lceil \log(n) \rceil + 1$ rounds.

Regarding the sender communication complexity, the size of update packages sent by a user ID to update in the CoCoA protocol is proportional to the size of the resolution of ID's co-path, which will be of order $\log(n)$ for a fully unblanked tree[16]. However, this value could be up to linear in a tree with many blanks, as is the case in TreeKEM and its variants, where blanks (or taints in the case of TTKEM) degrade communication efficiency. In CoCoA concurrent updates are merged and thus none are ever rejected by the server. Hence, in the considered scenario CoCoA in both the coordinated and uncoordinated setting has the same sender communication complexity of order $n \log(n)^2$ (corresponding to n users sending an update of size $\log(n)$ in $\lceil \log(n) \rceil + 1$ many rounds) and $t \log(n)^2$ (corresponding to t users sending an update of size $\log(n)$ in $\lceil \log(n) \rceil + 1$ many rounds), for cases (a) and (b) respectively.

With regards to the recipient communication complexity, user ID in our protocol needs to only receive at most a single ciphertext per update (zero if said update does not rotate the keys of any node in their state), and never more than $\mathsf{path}(\mathsf{ID}) = \lceil \log(n) \rceil$ in total. They will also receive at most $|\mathcal{P}(\mathsf{ID})|$ public keys per round.[17] Thus in case (a) ID would incur a download cost of order $\log(n)$ per round, and $\mathcal{O}(\log(n)^2)$ across the $\lceil \log(n) \rceil + 1$ rounds. In case (b) only t parties are updating per round, implying that the per round recipient cost is of order $\min(t, \log(n))$ and the cost over all $\lceil \log(n) \rceil + 1$ rounds is of order $\log(n) \cdot \min(t, \log(n))$. Finally, as in CoCoA concurrent updates do not affect the ratchet tree structure and in particular do not require blanks, the cost of subsequent updates remains of order $\log(n)$.

The communication complexity of other CGKA schemes. We now give a brief overview on the communication cost of other CGKA schemes in the considered scenarios as presented in Table 5. The first considered class are ratchet-tree based schemes that do not rely on the propose-commit framework, as TreeKEM v7 and earlier versions [6], rTreeKEM [1], TTKEM [23], and Causal TreeKEM [24]. These schemes require n (in case (a)) or t (in case (b)) rounds to recover, as only one update per round can be implemented[18]. Regarding the sender communication complexity, in the uncoordinated case every (in case

[16] an additional ciphertext would need to be sent for each unmerged leaf across ID's path, but this will not account for much in typical protocol executions.

[17] Note that the size of $\mathcal{P}(\mathsf{ID})$ grows at most by 1 per every blank node.

[18] Causal TreeKEM proposes an interesting idea of re-randomizing node secrets through a concrete homomorphic operation, instead of re-sampling them. Thus it actually allows for concurrent updates. However, the presented security statement still requires updates of every compromised party in *different* rounds, thus leading to communication complexity as presented in the table.

(b) corrupted) user would try to update in every round, thus leading cost of order $n^2 \log(n)$ (case (a)) and $t^2 \log(n)$ (case (b)), respectively. In the coordinated case, only one user per round would send an update attempt, leading to costs of $n \log(n)$ and $t \log(n)$. The per-user recipient cost is of order $n \log(n)$ as n packets of size $\log(n)$ have to be downloaded, and, finally, the cost of subsequent updates is of order $\log(n)$, as updating does not result in blanks.

The second class of protocols are ratchet-tree based protocols following the propose-commit paradigm as TreeKEM v8[6] and later versions, and the protocol by Bienstock et al. [9]. In these protocols the group can recover very quickly, by all compromised users proposing an Update in the first round, and having someone send a commit to all updates of the previous round in a second round. But this comes at the elevated cost of blanking (or tainting in the case of a PCS version of [9]) the paths of all those T users. This leads to the commit having a cost roughly proportional to the number of updates of the previous round. In case (a) this leads to a cumulative sender communication of n^2 in the uncoordinated case (all n user try to commit at cost n) and, with coordination, of n (only one user commits). In case (b) the authors of [9] show that the cost of a commit to t updates in their protocol is of order $t(1 = \log(n/t))$ that also applies to propose-commit TreeKEM. Thus, in this case the sender complexity is given by $t^2(1 = \log(n/t))$ (no coordination) and $t(1 = \log(n/t))$ (coordination), respectively. The per-user recipient communication is of order n (case (a)) and $t(1 = \log(n/t))$ (case (b)). The ability to recover in only two rounds in propose-commit TreeKEM comes at the cost of introducing blank nodes in the ratchet tree. Concretely, in case (a) the commit to n updates would lead to a fully blanked tree, meaning that the average (and worst case) cost of subsequent updates is going to be linear in n. Note that fully recovering from this state requires linearly many commits to single updates. In case (b) there always exists a user with a subsequent update cost of order $t(1+\log(n/t))$ and the average update cost is at least of order $(t^2 + (n-t)\log(n))/n$. While the protocol of [9] formally has a worst case subsequent update cost of $\log(n)$, it is only secure in a weak security model. Modifying it to obtain PCS guarantees similar to the other protocols, e.g. by tainting [23], would lead to future worst-case update costs matching the ones of propose-commit TreeKEM.

Finally, the protocol by Weidner et al. [28] based on bidirectional channels also supports concurrent operations and allows the group to recover in 2 rounds. We point out that this work targets a different network model and has thus a different focus than ours. In particular, the communication complexity for each update is linear in the number of parties, which is considered impractical in the contexts we are interested in. The protocol has a cumulative sender complexity of order n^2 (case (a)) and tn (case (b)) both in the coordinated and uncoordinated case, and a per-user recipient complexity of order n and t, respectively. The cost of subsequent updates is of order n.

Summary and comparison. CoCoA diverges across two different axes from what could be considered a common paradigm until now. On the one hand, users are no longer required to keep track of the full state of the ratchet tree, reducing the recipient communication cost and the storage costs for users, and making this cost differ substantially from the total amount of upload communication. Indeed, this is a big change, as this distinction is not really present in previous works, where the majority of uploaded packets are downloaded by everyone. On the other hand, we consider a more flexible PCS guarantee that only requires users to heal after $\lceil \log(n) \rceil + 1$ rounds. This is in contrast to previous works requiring PCS to hold after a constant number of rounds or only after n rounds. The effect of allowing concurrent updates to be merged is that, on one hand, the protocol is agnostic to coordination, i.e., no additional mechanism is needed that ensures that users do not send update/commit packages that will be rejected by the server, and, on the other hand, it allows the protocol to handle concurrent update operations without introducing blanks in the ratchet tree.

The trade-off with TreeKEM versions preceding the propose-commit paradigm is clear: we are paying a $\log(n)$ factor in sender communication in exchange for faster PCS that is independent from the number of compromised users. The comparison with propose-commit TreeKEM is not as straightforward, as the t compromised users can heal in only 2 rounds. The main advantage CoCoA over has this scheme is that it does not introduce blanks in the ratchet tree when handling concurrent operations, which leads to an improved update cost in subsequent rounds. However, this comes at the cost of slower healing and a factor of $\log^2(n)$ (or roughly $\log(n)$ in case (b)) in terms of sender communication cost. We point out that the propose-commit framework of TreeKEM allows for more flexibility, e.g. by performing the required updates in several batches over multiple rounds. The exact trade-off achieved by such an intermediate approach is hard to quantify, but, again, due to blanking the cost of future updates will suffer. Finally, CoCoA, compared to all versions of TreeKEM, has the advantage of reduced recipient communication complexity and that users can prepare updates without the need of extra communication with the server to prevent rejection of said updates.

As a final remark, CoCoA seems to have a slightly worse efficiency than TreeKEM based protocols predating the propose-commit paradigm, since it requires slightly larger sender communication overall. However, as we show in Section5, this is only the case if fast PCS is required for many users. In fact, a round with a single update will immediately grant PCS to its sender, just as in TreeKEM. Thus, CoCoA can be seen as an extension of pre propose-commit TreeKEM, which incorporates the possibility of trading bandwidth for faster collective healing.

5 Security

Given a set of parties whose state has leaked, TreeKEM and related variants achieve PCS exactly after all of them perform an update. This is still true in our protocol *as long as the updates are applied sequentially.* Furthermore, we also allow for concurrent updates, which results in some updates only being applied partially. Not surprisingly, it is not sufficient for every party to perform a partial update in order to achieve PCS. Consider the following scenario: parties ID_i and ID_j, both of which were corrupted since their last update, update concurrently by generating and processing simultaneously messages $\mathfrak{M}_i$ and $\mathfrak{M}_j$, respectively, resulting in a round message that refreshes both their paths; and let's say that $\mathfrak{M}_i \prec \mathfrak{M}_j$. Then, the seed of the node at the intersection of both paths gets encrypted under a node in the path of ID_j; in particular, it gets encrypted under a key the adversary knows. Thus, PCS is not achieved. However, as a group the two users have made progress towards achieving PCS: all nodes up to that intersection have healed. And hence, if at least one of the parties (potentially concurrently with other users) updates again, the ratchet tree will recover. In our security proof we capture this in more generality. In particular, we show that parties may heal after an individual, non-concurrent update, or by at most logarithmically many updates.

We consider our security proof a significant contribution of this work, not only because it increases our confidence in the security of our protocol, but also because there are significant differences to prior work. There are two main sources of obstacles for us to overcome: 1) Parties only have partial views of the key material in the tree and are potentially not even aware of changes to the key material outside of their view. This introduces difficulties in defining when a party should consider another party as *safe.* 2) The delivery server is not purely a relay server that one can force to behave honestly using signatures. In our case, the server is expected to actively compute parts of the messages that are exchanged, but still we do not want to put any trust in the server. Accordingly, the adversary gains additional active capabilities compared to prior works of similar flavor. Another aspect new to our proof is a combinatorial result over the challenge graph that establishes the ratchet tree's healing after $\log(n)$ updates.

5.1 Security Model and Safe Predicate

To analyze the security of CoCoA, we essentially use the security model from [23], which allows the adversary to act partially actively and fully adaptively: in this model, the adversary can adaptively decide which users perform which operations, and can actively control the delivery server; however it can not issue messages on behalf of the users. In [23] this is enforced by assuming authenticated channels. Since in CoCoA the signing of protocol messages is more involved, parent hash plays an important role also for security against partially active adversaries, and the server no longer just relays messages, we make the use of signatures explicit in this work. As we restrict our analysis to partially active adversaries, the adversary does not get access to signing keys via corruptions. While this might look artificial, it has importance in practice as discussed in the introduction, and we still obtain meaningful results in the vein of [23]. Nevertheless, we consider the analysis of CoCoA's security against fully active adversaries an important question for future work.

Except for explicit signatures, the differences in the setting of *concurrent* CGKA to the one of [23] are that 1) users process concurrent messages, 2) no messages are ever rejected by the server, and 3) the server is allowed to send arbitrary (potentially malformed) messages. Regarding 2), it is however possible that messages get lost and even that a user does not process an update they generated. Whether a user ID_i's update message (and which one) is contained in a round message $\mathfrak{M}_i$, is represented by a counter c_i. Finally, regarding 3), while our security notion is strictly stronger than the one from [23] (where the server could only forward existing messages), the security of protocols such as TreeKEM and TTKEM can trivially be upgraded to our notion: This is true since round messages in these protocols only consist of *signed* messages and the adversary does not learn any party's signing key. In our protocols, in contrast, the server is assumed to perform some computation on users' messages, hence it makes sense to consider a stronger model where this computation is not trusted.

Definition 5 (Asynchronous CGKA Security)**.** *The security for CGKA is modeled using a game between a challenger* C *and an adversary* A*. At the beginning of the game, the adversary queries* **create-group**(G) *and the challenger initializes the group* G *with identities* $(\mathsf{ID}_1, \ldots, \mathsf{ID}_\ell)$*. The adversary* A *can then make a sequence of queries, enumerated below, in any arbitrary order. On a high level,* **add-user** *and* **remove-user** *allow the adversary to control the structure of the group, whereas* **process** *allows it to control the scheduling of the messages. The query* **update** *simulates the refreshing of a local state. Finally,* **start-corrupt** *and* **end-corrupt** *enable the adversary to corrupt the users for a time period. The entire state and random coins of a corrupted user are leaked to the adversary during this period, except for the user's signing key.*

1. **add-user**$(\mathsf{ID}, \mathsf{ID}')$*: a user* ID *requests to add another user* ID' *to the group.*

2. **remove-user**$(\mathsf{ID}, \mathsf{ID}')$*: a user* ID *requests to remove another user* ID' *from the group.*

3. **update**(ID)*: the user* ID *requests to refresh its current local state* γ*.*

4. **process**$(\mathfrak{M}, \mathsf{ID})$*: for some message* $\mathfrak{M}$ *and party* ID*, this action sends* $\mathfrak{M}$ *to* ID *which immediately processes it.*

5. **start-corrupt**(ID)*: from now on the entire internal state and randomness of* ID *except for the signing key* $\mathsf{ssk}_{\mathsf{ID}}$ *is leaked to the adversary.*

6. **end-corrupt**(ID)*: ends the leakage of user* ID*'s internal state and randomness to the adversary.*

7. **challenge**(q^*)*:* A *picks a query* q^* *corresponding to an action* $\mathsf{a}^* =$ **update**(ID) *or the initialization (if* $q^* = 0$*). Let* K_0 *denote the group key that is sampled during this operation and* K_1 *be a fresh random key. The challenger tosses a coin* b *and – if the safe predicate below is satisfied – the key* K_b *is given to the adversary (if the predicate is not satisfied the adversary gets nothing).*

At the end of the game, the adversary outputs a bit b' *and wins if* $b' = b$*. We call a CGKA scheme CGKA-secure if for any PPT adversary* A *it holds that*

$$\mathsf{Adv}_{\mathsf{CGKA}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 1]|$$

is negligible.

In contrast to the security definition of [23], process queries do not point to specific queries here. Thus, in order to define our safe predicate, we first need to define what we mean by saying that a party processed another party's update.

Definition 6. *Let* ID *and* ID^* *be two (not necessarily different) users and* $(\gamma_q, T) \leftarrow \mathsf{CGKA.Upd}(\gamma_{q-1})$ *an update with associated counter* c*, generated by* ID *in query* q*. Let* $\mathcal{R}(\mathsf{ID}, \gamma_q)$ *be the set of round messages* $\mathfrak{M}$ *that*

(a) are efficiently computable from the public transcript and private states of all parties,

(b) have counter c for party ID*, and*

(c) will be accepted by ID *in state* γ_q*, i.e.,* $\mathsf{CGKA.Proc}(\gamma_q, \mathfrak{M})$ *outputs a new state* γ_{q+1} *such that* $\mathsf{CGKA.Key}(\gamma_{q+1}) \neq \mathsf{CGKA.Key}(\gamma_q)$.

Then we say that ID^* processes the update T (or equivalently q) at time $q^* > q$ *if* ID^* *processes some round message* $\mathfrak{M}^*$ *at time* q^* *resulting in state* γ_{q^*}*, and* $\mathsf{CGKA.Key}(\gamma_{q^*}) \in \{\mathsf{CGKA.Key}(\mathsf{CGKA.Proc}(\gamma_q, \mathfrak{M})) \mid \mathfrak{M} \in \mathcal{R}(\mathsf{ID}, \gamma_q)\}$.

As a special case we say that ID^* *processes the* single *update T (or equivalently q), if in item (c) additionally the only changes to* $\mathcal{P}(\mathsf{ID})$ *resulting from updates are due to T.*

With this notion in place, we will now define the safe predicate similar to the one in [23]. In particular, it rules out all trivial winning strategies, while preserving simplicity by ignoring protocol-specific details such as the relative position of users within the tree.

Definition 7 (Critical window, safe user)**.** *Let* ID *and* ID^* *be two (not necessarily different) users and* $q^* \in [Q]_0$ *be some* **update**$(\cdot)$ *or* **create-group**$(\cdot)$ *query. Let* $q^- < q^*$ *be maximal such that one of the following holds:*

- *There exist* $L := \lceil \log(n) \rceil + 1$ *update queries* $\mathsf{a}^i_{\mathsf{ID}} := \textbf{update}(\mathsf{ID})$ *($i \in [L]$) that were generated for* ID *and processed by* ID^* *within the time interval* $[q^-, q^*]$*. If* ID^* *does not process L such queries then we set* $q^- = 1$*, the first query. We denote the last such update query as* q^L*.*
- *There exists an update query* $\mathsf{a}^-_{\mathsf{ID}} := \textbf{update}(\mathsf{ID})$ *that was generated by* ID *and processed by* ID^* *as a* single *update within the time interval* $[q^-, q^*]$*. In this case, we set* $q^L := q^-$*.*

Furthermore, let $q^+ > q^L$ *be the first query that invalidates* ID*'s current key (in the view of* ID^**), i.e., in query* q^+*,* ID *processes a (partial) update* $\mathsf{a}^+_{\mathsf{ID}} := \textbf{update}(\mathsf{ID}) \notin \{\mathsf{a}^i_{\mathsf{ID}}\}_{i \in [L]}$*. If* ID *does not process any such query then we set* $q^+ = Q$*, the last query.*
We say that the window $[q^-, q^+]$ *is* critical *for* ID *at time* q^* *in the view of* ID^**. Moreover, if the user* ID *is* not corrupted *at any time point in the critical window, we say that* ID *is* safe *at time* q^* *in the view of* ID^**.*

Similar to [23], we define a group key as *safe* if all the users that ID^* considers to be in the group are individually safe, i.e., not corrupted in their critical windows, in the view of ID^*.

Definition 8 (Safe predicate)**.** *Let* K^* *be a group key generated in an action*

$$\mathsf{a}^* \in \{\textbf{update}(\mathsf{ID}^*), \textbf{create-group}(\mathsf{ID}^*, \cdot)\}$$

at time point $q^* \in [Q]_0$ *and let* G^* *be the set of users which would end up in the group if query* q^* *was processed, as viewed by the generating user* ID^**. Then the key* K^* *is considered* safe *if for all users* $\mathsf{ID} \in G^*$ *(including* ID^**) we have that* ID *is safe at time* q^* *in the view of* ID^* *(as per Definition 7).*

Note that the second case in Definition 7 exactly captures the case where only *single* updates are accepted in each round. Thus, the security of CoCoA is strictly stronger than sequential variants of TreeKEM. Further, the bound of $\lceil \log(n) \rceil + 1$ updates as required in Definition 7 is indeed tight, an example is shown in Section 5.2, Figure 3.

Remark 1. We make the following observations about Definition 8.

- If we were to consider a weaker security model where the delivery server is honest, Definitions 7 and 8 could be considerably simplified as follows: Let G^* be the group of users at time q^*. Then q^* is safe, if all users $\mathsf{ID} \in G^*$ either did $\lceil \log(n) \rceil + 1$ partial or one full update *before* q^*, and one update (partial or full) *after* q^*, and are not compromised between/during these updates.

- A more permissible safe predicate could be defined when considering the relative position of updating users within the tree. To this aim, one could define a function that evaluates the *progress* a user makes during a concurrent update. This progress corresponds to the number of keys the party updates along its path to the root and depends on the ordering imposed on concurrent updates. e.g. a single update refreshes the entire path, hence makes progress $\lceil\log(n)\rceil + 1$, a concurrent update makes progress at least 1. An even stronger notion of safe group keys could be defined by also considering the effect updates of other users have on a member ID's path: E.g. if ID does a single concurrent update, all keys along her path can heal if her sibling continues to update. In this work, however, we prefer to define a simple and protocol-independent security notion that allows to compare our scheme to other constructions.

5.2 Security of CoCoA

Regarding the security of CoCoA we obtain the following.

Theorem 1. *If the encryption scheme used in CoCoA is IND-CPA-secure, the signature scheme is UF-CMA-secure, and the used hash functions are modeled as random oracles, then CoCoA is CGKA-secure.*

To prove security of CoCoA, we follow the approach of [23] and consider the graph structure that is generated throughout the security experiment. A node i in the so-called *CGKA graph* is associated with seeds Δ_i and $s_i := \mathsf{H}_2(\Delta_i)$, and a key-pair $(pk_i, sk_i) := \mathsf{Gen}(s_i)$. The edges of the graph, on the other hand, are induced by dependencies via the hash function H_1 or (public-key) encryptions. To be more precise, an edge (i, j) corresponds to either:

(a) a ciphertext of the form $\mathsf{Enc}_{\mathsf{pk}_i}(\Delta_j)$; or

(b) an application of H_1 of the form $\Delta_j = \mathsf{H}_1(\Delta_i)$ used in hierarchical derivation.

Naturally, the structure of the CGKA graph depends on the **update**, **add-user** or **remove-user** queries made by the adversary, and is therefore generated adaptively. To argue security of a challenge group key, we consider the subgraph of the CGKA graph that consists of all ancestors of the node associated to the challenge group key – the so-called *challenge graph.* By functionality of the CGKA protocol, the challenge group key can be derived from any secret key/seed associated to a node in the challenge graph. For an example of CGKA graph and challenge graph see Figure 3. To argue security, none of the secret keys in the challenge graph must be leaked to the adversary by corruption. We prove (in Lemma 2 below) that this is indeed the case for CoCoA if the safe predicate is satisfied. Our proof follows the ideas from [23], but involves a new combinatorial argument to establish the upper bound of $\lceil\log(n)\rceil + 1$ updates for healing the state of every user. Further, the fact that in CoCoA users only keep track of a part of the ratchet tree substantially complicates the proof of this statement.

In more detail, the proof for the protocol in [23] relies on the property that every key in the challenge graph must stem from an update that the party ID^*, who generated the challenge key, processed. This can easily be ensured for protocols keeping track of the full ratchet tree, by forcing parties, who do not agree for every point in time in the protocol execution on every key associated to a node in the ratchet tree, into inconsistent states, thus making future communication between them impossible. Note that this implies the desired property. In this case, if a user ID, while generating an update, encrypts the seed of a key pk to some pk', and later ID^* encrypts to pk, then ID^* must have had pk' in their state at some point in time, and thus in particular processed the update establishing it.

Unfortunately, while in an execution where the server behaves honestly, this property would also be true with respect to the relatively simple definition of processing an update of Definition 6, it is no longer true if we allow an untrusted server. Since in CoCoA the server might send malformed round messages, this property turns out to not hold anymore. We overcome this issue by giving a more involved definition (which is equivalent to Definition 6 in the honest server setting) of *weakly processing* an update and then essentially show, in the ROM, that every key in a user's state must stem from a weakly processed update (Lemma 5). Further, we show that users that do not agree on the same history of weakly processed updates transition to

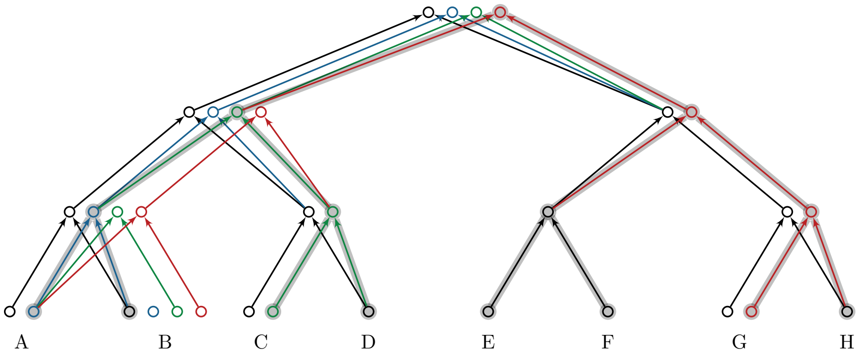

Figure 3: Example; CGKA graph and challenge graph. Sequence of operations; we write update($X \prec Y$), to indicate that parties X and Y updated concurrently and X's update took precedence over Y's. A group with 8 parties is set up (black), update($A \prec B$) (blue), update($C \prec B$) (green), update($G \prec B$) (red), G's update is challenged. Vertices and edges that are part of the challenge graph are shaded in gray. Note that even though B updated three times her leaf key in the challenge graph lags behind by $3 = \log(8)$ steps.

inconsistent states (Lemma 4). For this we have to show, for example, that all keys introduced into a user's state after a change to the resolution of their copath must have been weakly processed in an earlier round (even in the case that at this point in time this update did not affect the user's limited view of the ratchet tree). To prove these properties we rely on the consistency mechanisms of transcript hash and parent hash.

With these statements in place we are finally able to show that no key in the challenge graph is leaked to the adversary, where we use the observation that this property holding with respect to processing an update as defined in Definition 6 is implied by it holding with respect to the relaxed definition.

5.3 Overview of Proof Structure

In this section we give an overview of the proof structure, introducing the definition of *weakly processing* and a few small preliminary results, along with the statements of the main lemmas we build the proof on. For all security statements in the remainder of the paper we assume all hash function H_i, $i \in [5]$, to be random oracles. Our security proof of CoCoA is based on the proof of the following Lemma.

Lemma 2. *Assume that* Sig *is a secure signature scheme and* H_3 *a hash function. Then, for any* safe *challenge group key in the asynchronous CGKA security game instantiated with CoCoA, it holds that none of the seeds and secret keys in the challenge graph are leaked to the adversary via corruption.*

With this Lemma in place, Theorem 1 now follows from the corresponding results in [23, Section 3.5]. There, security of a variant of TreeKEM called TTKEM is proven by reducing to a game that is known as generalized selective decryption (GSD), where the adversary can query for encryptions of secret keys under other keys, corrupt keys, and finally has to distinguish a key from random that it did not trivially learn by its previous queries. While GSD had only been defined in the secret-key setting, in [23] the authors introduce and analyze a public key variant of this game in the random oracle model, which allows to consider the security experiment of ratchet-tree based CGKA protocols as a special case of GSD. This is true thanks to Lemma 2, which guarantees that any safe challenge in the CGKA security game implies a valid challenge in the GSD game. The security bound for CoCoA now follows analogously to the corresponding result for TTKEM [23, Theorem 4] from the more general result on public-key GSD in the ROM [23, Theorem 3].

In order to prove Lemma 2 we will rely on the notion of weakly processing an update. Since an adversarial server can send wrong openings to a user, when ID processes a round message, they might still be introducing into their state some keys or some hash values that depend on updates they have not processed as per the definition above. For example, consider the case of a round message sent to ID, containing a single Update from a user ID^* who is not their sibling: a malicious server could include the correct keys and encryptions in the round message, but substitute the new Merkle commitment by an arbitrary value; this will cause ID to still accept the message and update the corresponding keys in their state, while not having processed ID^*'s update (since the server cannot fool the latter into accepting a round message with a round hash dependent on the wrong opening). Below we define a notion of *weakly processing* an update, which captures this scenario. Before we do that, however, we will define the further notion of *explicitly processing*, capturing the process of users introducing keys into their state by means of (weakly) processing an update. Note, however that explicitly processing an update will not imply processing it, for the above mentioned reason.

In what follows the adversary queries as defined in the security game (Definition 5) are also denoted using the protocol notation (Definition 1) in order to explicit refer to the CGKA protocol in use.

Definition 9 (Explicitly Process)**.** *We say that a user ID^* has explicitly processed the update U generated by user ID in query $q = \mathsf{CGKA.Upd}(\gamma_{\mathsf{ID}})$ if ID^* processed and accepted a round message $\mathfrak{M}$ in query $q^* > q$ and either*

- *$\mathfrak{M}$ contained a counter c corresponding to update U generated by ID, or*
- *$\mathfrak{M}$ contained a tuple $p_v = (\mathsf{pk}_v, h_v, \mathcal{H}_{trans}, \mathsf{confTag}, \sigma_v)$ corresponding to the public state of some node v, where $p_v \in U$ and its transcript hash value $\mathcal{H}_{trans}$ is the same as the one in the state of ID.*

Definition 10. *We say that a user ID^* has weakly processed the update U generated by user ID in query $q = \mathsf{CGKA.Upd}(\gamma_{\mathsf{ID}})$ if ID^* processed a round message $\mathfrak{M}^*$ in query $q^* > q$ and either:*

- *q^* was the first query ID^* processed, and the full tree which ID^* received in the welcome message contained a node state sampled in U,*
- *there exists ID' (not necessarily different from ID^*), in the group in the view of ID^*, such that the following conditions below hold. Let first q' be either $q' = \mathsf{CGKA.Upd}(\gamma')$ the last Update from ID' weakly processed by ID^* or, if such update does not exist, the query in which ID' processed their welcome message into the group, initializing state γ'. Then:*
  - *there exists a series of efficiently computable round messages (given access to the RO queries, the public transcript, and private states of all parties) $\mathfrak{M}'_1, \ldots, \mathfrak{M}'_t$ such that ID' with respect to state γ' would process (and accept) all $\mathfrak{M}'_i$ in order, where γ'_i is the state resulting from processing and accepting $\mathfrak{M}'_i$ with respect to state γ'_{i-1} (and $\gamma'_0 = \gamma'$).*
  - *ID' explicitly processed U by processing $\mathfrak{M}'_t$.*
  - *$\mathsf{CGKA.Key}(\mathsf{CGKA.Proc}(\gamma^{q^*}_{\mathsf{ID}^*}, \mathfrak{M}^*)) = \mathsf{CGKA.Key}(\gamma'_t)$, i.e. they derive the same group key after processing the respective messages,*

A few remarks regarding this definition: first, note that the update query q' in the second case of the definition is required to have been weakly processed by ID^*. In fact, it must also hold that if this case is satisfied, such query was also *processed* by ID^*, since weakly processing but not processing an update implies the user processing will never have consistent states with the update issuer. Second, note that considering the case where $\mathsf{ID}' = \mathsf{ID}^*$ is equivalent to saying ID^* explicitly processed U. In particular, explicitly processing an update means weakly processing it. Similarly, by considering the case where $\mathsf{ID}' = \mathsf{ID}$, we can see that processing also implies weakly processing. We recall again that explicitly processing does not imply processing. From this definition, we can define the predicate $\mathsf{wProcess}(\mathsf{ID}, U, q)$, which evaluates to 1 if ID weakly processed U at time q; and else evaluates to 0.

Below we define formally what it means for a user to join the group at the same time or after another user. Note that this is defined with respect to the views that users have of the group. With regards to the situation where a user leaves the group and joins again, we assume that it does so with a new identifier. That is, party identifiers are assumed unique and cannot rejoin once removed. Further, for security we assume also that init keys are only used once.

Definition 11. *We say that user* ID^* *is added at the same time or after* ID *(in the view of* ID*) if there exist* $q_A = \mathsf{CGKA.Add}(\cdot, \mathsf{ID})$ *and* $q_A^* = \mathsf{CGKA.Add}(\cdot, \mathsf{ID}^*)$ *and queries* q *and* q^* *such that* $\gamma_{\mathsf{ID}}^{q} \equiv \gamma_{\mathsf{ID}^*}^{q^*}$ *and, moreover, either* ID *processed* q_A^* *in some query* $\leq q$*, or the equivalence holds for* q *and* q^* *corresponding to the first process queries for each party.*

Throughout the following, for simplicity of notation, we assume that the state of a party is the same before and after issuing an add, remove and update operation, i.e. if q is a query of the form $\mathsf{CGKA.Add}(\mathsf{ID}, \cdot)$, $\mathsf{CGKA.Rem}(\mathsf{ID}, \cdot)$ or $\mathsf{CGKA.Upd}(\gamma_{\mathsf{ID}}^{q-1})$, we consider the states of party ID at $q-1$ and q the same. While in practice this is not the case, since issuing any of these operations will change the pending state γ'_{ID}, which is a part of the local state, it will not change any value in the rest of the state, and thus will not influence any statement regarding states consistency.

When a user ID is added to the group, we count them as effectively processing two round messages, the first one corresponding to the welcome message, the second corresponding to the round message that comes with it. Thus, we will say ID^* has a state consistent with that of ID at the time the latter joined the group if the state of ID^* is consistent with the contents of the welcome message or, equivalently, with the user that added ID. Note that although the welcome message contains no group key $\gamma.\mathsf{appSecret}$, it does contain a transcript hash value $\mathcal{H}_{trans}$, and users having the same transcript hash value in their state is equivalent (up to collision resistance of the random oracle) to users having the same group key, by Proposition 2.

Before beginning with the proofs, we present here an observation that will be used throughout most of them. Informally, this is the following: due to updates being signed and the adversary not having access to signing keys, every key in a user's state and, in turn, in the challenge tree, must stem from an update or add query. More precisely, any key stemming from an add query, that is introduced in the state, gets delivered in a packet that comes with a signature by the party who authored the add, together with a signature of the party whom that leaf belongs to; and any key introduced in the state by either explicitly processing an update, or by receiving a node state from the server after a removal, is received as part of a tuple $(\mathsf{pk}_v, h_v, \mathcal{H}_{trans}, \mathsf{confTag}, \sigma_v)$, together with a party identifier ID_v. Before accepting, a user will check that σ_v is a signature for $(\mathsf{pk}_v, h_v, \mathrm{PK}_v^{\mathrm{pr}}, \mathcal{H}_{trans}, \mathsf{confTag})$ under ID_v's verification key. Thus, any key, and moreover, any of those signed values in the state of any $v \in \mathcal{P}(\mathsf{ID})$, must have come from an update by ID_v. In other words, if some state in $\mathcal{P}(\mathsf{ID})$ contained a key, or value from the ones above, not corresponding to a query, we could build a forger for the signature scheme with a loss of n in the advantage by guessing the user that supposedly generated the key in question. Note that every update, add, or remove query requires at most $n + 2\log(n)$ signatures (this happens, e.g., in the case of a fully blanked tree).

The following small proposition shows that, as we would expect of the more general definition of weak processing, users can only weakly process updates issued by parties with a consistent state.

Proposition 3. *Let* U *be an output from* $q^* = \mathsf{CGKA.Upd}(\gamma_{\mathsf{ID}^*}^{q^*-1})$*, for some user* ID^* *with state* $\gamma_{\mathsf{ID}^*}^{q^*-1}$ *at time* q^*-1*. Let* ID *be a party with state* γ_{ID} *and such that* $\mathsf{wProcess}(\mathsf{ID}, U, q) = 1$ *for some query* q*, and such that* ID *was already part of the group before* q*. Then the states of* ID *at time* $q-1$ *and* ID^* *at time* q^* *are consistent.*

Proof. If ID weakly processed U after joining the group, then, by definition there must exist a user ID' and some message $\mathfrak{M}'$ such that ID' would explicitly process U by processing $\mathfrak{M}'$ in query q' and such that its state right after would be consistent with the state of ID at q. Now, for ID' to explicitly process U, their state just before processing $\mathfrak{M}'$, at $q'-1$, must have been consistent with the state of ID^* at the time they issued U, i.e. at q^*. This is the case since for ID' to accept $\mathfrak{M}'$, the signed packet contained in said round message and which corresponded to U must have contained the same transcript hash value as ID' had at $q'-1$ that moment in their state. However, by collision resistance of the random oracle H_3, since the states of ID at q and ID' at q' were consistent, so were their respective states at $q-1$ and $q'-1$. But this implies that ID's state at $q-1$ was consistent with that of ID^* at q^*. □

It follows from the proposition above that no user can weakly process the same update twice:

Corollary 3. *Let* $\mathsf{wProcess}(\mathsf{ID}, U, q) = 1$ *for some* ID*,* U *and* q*. Then, except with negligible probability, for any* $q' \neq q$*,* $\mathsf{wProcess}(\mathsf{ID}, U, q') = 0$*.*

Proof. Assume for contradiction that there is an update U and distinct queries q, q' with $\mathsf{wProcess}(\mathsf{ID}, U, q) = 1$ and $\mathsf{wProcess}(\mathsf{ID}, U, q') = 1$. Let $q < q'$, and assume first that ID did not join the group in q. By Proposition 3, the state of ID at both $q-1$ and $q'-1$ must be consistent with the state of U's author at the time U was generated. But this is not possible, since ID has processed and accepted some round message (and therefore updated its state) in between these times.

Suppose now that ID did join the group in q. Since the welcome packet that they received was (except with negligible probability) part of the output of a query, the transcript hash $\mathcal{H}_{trans}$ in it was computed with respect to the round hash value $\mathcal{H}_{round}$ in it. Further, we know $\mathcal{H}_{round}$ is consistent with the tree that ID received upon joining, which contained some node state from U, by the assumption that ID weakly processed U at q. However, this node state from U also contained a transcript hash value. By Proposition 3, ID at $q'-1$ was consistent with U's author at the time they issued U. In particular, the transcript hash value in U must be the same as the transcript hash value in ID's state at $q'-1 \geq q$, by Proposition 2. But this is a contradiction to the properties of the random oracle H_3, since the transcript hash value of ID at this time depended on that of ID at q, which itself depended on the transcript hash value in U. □

With this in place, we will be ready to prove Lemma 2 above, with the help of two further Lemmas, capturing the consistency guarantees given by our round hash and parent hash mechanisms.

The first one states that users with consistent states have weakly processed equivalent sequences of operations.

Lemma 4. *Let ID and ID^* be two users with consistent states at time q, and such that ID^* joined the group at the same time or after ID. Then:*

$$\{U \;:\; \exists q' \leq q \text{ s.t. } \mathsf{wProcess}(\mathsf{ID}^*, U, q') = 1\} \subseteq \{U \;:\; \exists q' \leq q \text{ s.t. } \mathsf{wProcess}(\mathsf{ID}, U, q') = 1\}$$

and, for any q_1^*, q_2^*, with $q_1^* < q_2^*$ and any Updates U, U' such that $\mathsf{wProcess}(\mathsf{ID}^*, U, q_1^*) = \mathsf{wProcess}(\mathsf{ID}^*, U', q_2^*) = 1$, there exist q_1 and q_2, with $q_1 < q_2$ such that $\mathsf{wProcess}(\mathsf{ID}, U, q_1) = \mathsf{wProcess}(\mathsf{ID}, U', q_2) = 1$.

The second one states that all the keys that a user incorporates into their state as part of the protocol execution stem from adds or updates that they have weakly processed, and moreover, that these do not correspond to operations that have already been superseded by other weakly processed Updates.

Lemma 5. *For any user ID and any key pk_v in their local state γ associated to a node $v \in \mathcal{P}(\mathsf{ID})$ at time q^*, the following hold:*

1. *there is a query $q \in \{1, \ldots, q^*\}$, such that pk_v is either the InitKey of a party whose Add was processed by ID in query q, or was sampled in some update U that the user had weakly processed in q; i.e., $\mathsf{wProcess}(\mathsf{ID}, U, q) = 1$;*
2. *for any Update U' generated before q^*, which affects v and such that $\mathsf{wProcess}(\mathsf{ID}, U', q') = 1$ for some $q' \leq q^*$, it holds that $q' \leq q$.*

To conclude the security proof, in Section 5.4 we give the proof of Lemma 2 assuming Lemmas 4 and 5, which are proven afterwards, in Section 5.5.

5.4 Proof of Lemma 2

We now prove Lemma 2. To do so, we first argue that all parties associated to leaves in the challenge graph belong to the group as viewed by the party generating the challenge key.

Lemma 6. *Let ID^* be the party who generated the challenge key K^* in query q^*. Then, all parties associated to leaves in the challenge graph belong to G^*, the set of group members at time q^* in ID^*'s view.*

Proof of Lemma 6. Let $\mathsf{ID} \notin G^*$ and assume for contradiction that ID is associated to a leaf in the challenge graph. Let $v_1, \ldots, v_k$ be the path in the challenge path from ID's leaf v_1 to the root v_k. Further, let

$i_1 < \cdots < i_\ell$ be such that the edges from (v_{i_j-1}, v_{i_j}) correspond to encryptions, while the other edges on the path stem from hierarchical derivation. Note that either ID^* never considered ID to be in the group, or ID^* processed a **remove-user**$(\cdot, \mathsf{ID})$ message (and no subsequent **add-user**$(\cdot, \mathsf{ID})$ message). Further, it is not possible that $\ell = 1$ since in this case ID^* would have encrypted a seed to the key of v_{i_1-1} that in this case was generated by ID. However, in the course of processing any round message, before incorporating any key into their local state ID^* checks that it comes with a signature which verifies under the key of some user from the corresponding sub-tree, in particular from a user in G^*.

Thus, we may assume $\ell \geq 2$. We denote the user that generated the key at v_{i_j} by ID_j and the corresponding update query by q_j. So, when ID_1 generated the key at v_{i_1} they encrypted its seed to the a key that was generated by ID, implying that ID_1 considered ID to be in the group at this point in time. Further, user ID_2 encrypted the seed of node v_{i_2} to a key generated in update q_1 implying in particular that at time q_2 they had weakly processed the same operations (and in particular the same add/remove operations) as ID_1 at q_1 by Proposition 3 and Lemma 4, plus potentially some others, but none affecting nodes on the the subtree under v_{i_2-1} (since any such operation would overwrite the keys created in q_1). In particular, they cannot have processed any **remove-user**$(\cdot, \mathsf{ID})$ between q_1 and q_2, so they must consider ID to be part of the group.

By repeatedly applying this argument we obtain that $\mathsf{ID}_{\ell-1}$, when generating update $q_{\ell-1}$, considered ID to be part of the group. Similarly, as $\mathsf{ID}^* = \mathsf{ID}_\ell$ processed this update, they also must have considered ID to be in the group at this point in time and, further, cannot have processed a **remove-user**$(\cdot, \mathsf{ID})$ message before generating the root of the challenge graph at q_ℓ, as, again, otherwise the key at $v_{i_\ell-1}$ would have been blanked or overwritten; a contradiction.

□

Proof of Lemma 2. Let ID^* be the party who generated the challenge key K^* in query q^*. Since K^* is safe, for all parties ID in G^*, the set of group members at time q^* in ID^*'s view, ID must be safe in the view of ID^*. We will show that any key in the challenge graph can only be in the state of ID during the critical window $[q^-_{\mathsf{ID}}, q^+_{\mathsf{ID}}]$ of ID in the view of ID^*. Together with Lemma 6 this will show the result. We note that all the secret keys in the state of ID during its critical window are overwritten in the state of ID *before* the next corruption of ID. This is the case because, by definition of the safe predicate, after ID generated the last update $\mathsf{a}^{\mathsf{L}}_{\mathsf{ID}}$ she (partially) processes some further update $\mathsf{a}^{+}_{\mathsf{ID}}$ before the next corruption. Even though her own update $\mathsf{a}^{+}_{\mathsf{ID}}$ might only be partially processed, the way updates are merged in $\mathsf{CGKA.DIv}$ guarantees that *all* keys along ID's path will be refreshed. Moreover, by Corollary 3, ID will not add to their state any keys that were already part of it. It is sufficient to prove that any key in the challenge graph was generated by a party that had already entered its critical window in the view of ID^* when generating the update resulting in this key. This is because seeds can only be encrypted to keys that have already been generated – since every key in the challenge graph must stem from a query, as argued above –, which means that all receiving parties are already in their respective critical window. Accordingly, we now prove that each key in the challenge graph was generated by some user $\mathsf{ID} \in G^*$ within the first period $[q^-_{\mathsf{ID}}, q^L_{\mathsf{ID}}]$ of ID's critical window for q^* in the view of ID^*.

To this aim, first note that the challenge node was generated by user ID^* in query q^*, and q^* by definition lies in the first period of the critical window of ID^*. We will proceed by induction: consider an arbitrary internal node v in the challenge graph, and assume it was generated at time q_v (which is in the first period of the critical window of the generator). Then, all except one parent of v must have been generated *before* query q_v. More precisely, for all parents w of v in the challenge graph it holds: Either (1) w was generated in the same query q_v (this happens for exactly one parent), or (2) w was generated by a party $\mathsf{ID} \in G^*$ in the subtree rooted at w, by Lemma 6, at some time $q_w < q_v$. In case (1), if v was generated during the first period of the critical window of its generator, then clearly the same holds for w. In case (2), since ID^* only processes messages from parties with a consistent state, and q_w was weakly processed by the party who generated q_v *before* it generated q_v (by Lemma 5), having weakly processed q_v, user ID^* must have also weakly processed q_w before. This follows from Lemma 4, as both users must have then weakly processed the same updates. In other words, we have $q^+_w < q^+_v$, where q^+ refers to the query when ID^* weakly processes q. Similarly, we can deduce that q_w must have been the last update generated by ID before q_v, which was weakly processed by ID^*. This means that, for all update queries $q'_{\mathsf{ID}} > q_w$ generated by ID that are weakly processed by ID^*, we have $(q'_{\mathsf{ID}})^+ \geq q_v^+$.

We will now look at the critical window of ID. Consider first the case where ID's critical window is defined through a single update q^-_{ID} that was processed individually by ID^*. We will show that $q^-_{\mathsf{ID}} \leq q_w$ (so that, in fact, equality holds here), which means that ID generated q_w in the first period of its critical window. Assume for contradiction that $q^-_{\mathsf{ID}} > q_w$, so that $q^* > (q^-_{\mathsf{ID}})^+ > q_v^+$. Consider the partition of the path in the challenge graph from v to K^* given by encryption edges, as in the proof above of Lemma 6: let $v_1, \ldots, v_k$ be the nodes in such path, and let $i_1 < \cdots < i_\ell$ be such that the edges (v_{i_j-1}, v_{i_j}) correspond to encryptions, while the other edges correspond to hierarchical derivations. If $\ell = 1$ then, by Lemma 5, in q^* ID^* encrypted to a key generated in q^-_{ID}, but q^-_{ID} overwrote all keys generated by q_v, so v cannot be in the challenge graph, which contradicts our assumption on v. Assume, thus $\ell \geq 2$ and denote the user that generated the key at v_{i_j} by ID_j and the corresponding update query by q_j. In that case, by the same argument as in the previous case, ID_1 must have not weakly processed q^-_{ID}, before q_v, since otherwise they would encrypt v_{i_1} under a key sampled by the former, and not the latter, and v would not be in the challenge graph. But we can recursively apply this argument to deduce that $\mathsf{ID}_\ell = \mathsf{ID}^*$ must also not have weakly processed q^-_{ID} at q^*, a contradiction.

Finally, consider the case where ID's critical window is defined through a sequence of $L = \lceil \log(n) \rceil + 1$ updates $q^-_{\mathsf{ID}} = q_1, \ldots, q_L$. We will argue with respect to the potentially larger sequence $q^-_{\mathsf{ID}} = q_1, \ldots, q_{L'}$ of weakly processed updates in the first period of the critical window; here $L' \geq L = \lceil \log(n) \rceil + 1$, since processing implies weakly processing. Applying the same argument as in the previous paragraph, we can deduce that at most one of these updates q_i could have been weakly processed by ID^* in the window $[q_w^+ + 1, q_v^+]$, and if this is the case, then this update must have been weakly processed *concurrently* with q_v. Note that if $q_i^+ = q_v^+$ for any $i > 1$, we have $q_1^+ \leq q_w^+$, which by Lemma 4 implies $q_1 \leq q_w$ and thus ID is in its critical window when it generates q_w. If this is not the case, no q_i can belong to the mentioned interval, meaning it suffices to show that $q_1^+ < q_v^+$, as this implies $q_1 \leq q_w$. Assume for contradiction $q_1^+ \geq q_v^+$. Consider v's child u_1; following the same argument, at most one of ID's queries can be weakly processed in the time interval $[q_v^+ + 1, q_{u_1}^+]$. But there are at most $\lceil \log(n) \rceil - 1$ descendants of (non-leaf node) v, let these be denoted by $u_1, \ldots, u_l$ with $l \in [\lceil \log(n) \rceil - 1]$ and $u_0 := v$. Now, by the above, for each time interval $[q_{u_{i-1}}^+ + 1, q_{u_i}^+]$ $(i \in [\lceil \log(n) \rceil - 1])$ there is at most one (concurrent) update from ID weakly processed by ID^*. Since u_l is the root of the challenge graph, which was created in query q^*, and by assumption $q_1^+ \geq q_v^+$, a simple counting argument implies that $(q_{L'})^+ > q^*$, a contradiction.

□

5.5 Proofs of Lemmas 4 and 5

At the core of the proofs of both of the lemmas in the previous section (Lemmas 2 and 6) lies the fact that an adversary cannot commit to subtrees outside the users' states containing wrong information in a way that will not push users out of consistent states. This is captured by the following game and lemma:

Definition 12. *The ability of the adversary to open wrong commitments to the parts of the ratchet tree the users do not see is modeled through the following* OPEN *game, played by an PPT adversary* A. *The game is syntactically equivalent to the CGKA security game from Definition 5, except for the challenge queries, which now take a different shape, and the winning condition. In this game,* A *is able to issue a query* $\mathsf{OPEN.Chall}(q)$ *at any point, with* q *being a query of the form* $q = \mathsf{CGKA.Proc}(\mathsf{ID}, \cdot)$ *already made in the CGKA game, and output a ratchet tree* T_q.

The game outputs $1 \leftarrow \mathsf{OPEN}$ *if, after such a* $\mathsf{OPEN.Chall}(q)$ *query, the following is true for* T_q*:*

- T_q *is a well-formed ratchet tree, every node except for its leaves has indegree two, it has no blank leaves and it contains at least one ratchet tree leaf (i.e., one containing a public verification key);*

- $\mathsf{PHash.Ver}(T_q) = 1$, *i.e. parent hash verifies;*

- *the state of every node* v *in* T_q *corresponds to that output by some add or update query* $q_v < q$*;*

- *the label $\ell(v_{root}^{T_q})$ (as in Def. 3) of the node at the root of T_q is the same as the label of some node in $\mathcal{P}^q(\mathsf{ID})$;*
- *there exist a query q_i to which some node state in T_q correspond to, that has not been weakly processed by* ID *at time q or before.*

Else, $0 \leftarrow \mathsf{OPEN}$*. Finally, we say that* A *wins the OPEN game if* $1 \leftarrow \mathsf{OPEN}$*.*

We now show that no adversary can win game OPEN with more than negligible probability.

Lemma 7. *The probability that any adversary* A *making at most Q queries in the CGKA security game wins the OPEN game with respect to any $q \in [Q]$ is negligible.*

In order to prove this, we first prove the existence of an extractor algorithm that can return the queries (weakly) processed by any user at any point in the game. This is formalized in the following lemma.

Lemma 8. *There exists an efficient algorithm* E *that, given access to the random oracle queries, the public transcript, and private states of all parties, can compute the set of operations Q that any user* ID *has weakly processed in any query $q = \mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}}, \mathfrak{M})$.*

Proof. Note first that, having access to the private states of the party, E can learn if ID accepts $\mathfrak{M}$, just by running CGKA.Proc. Also, it is clear from $\mathfrak{M}$ which adds and removes are processed, as well as any update that is explicitly processed. We will prove the statement of the lemma by induction, showing that E can compute the set of operations weakly but not explicitly processed by ID, as well as the set of users in the group with whom ID could still have consistent states with.

To begin, note that if $\mathfrak{M}$ is the first message ID processes, either because they set up the group, or are added to it, E can easily extract all the updates weakly processed by ID. In the first case, these are none, and in the second they are given by the ratchet tree ID receives as part of the welcome message, as by the first bullet point of Definition 10. Before continuing, we argue that in the latter case these are all updates ID weakly processes, i.e. any update satisfying the second bullet point of said definition must be one of those given by the ratchet tree ID receives. Suppose that was not the case, and that there was some user ID′ whom, after processing some round message and, through it, explicitly processed some update U, was in a consistent state with ID. From the fact the states were consistent, we know the respective round hash values for both parties must be the same. However, that of ID′ was computed using as input some key sampled in U which, by assumption, was not used in the computation of ID's round hash value. This is however, a contradiction to the collision resistance of the RO H_3.

Now, let U^i_{ID} be the set of users with whom ID could still have consistent states with respect to their state γ^i_{ID}; i.e. those for which a sequence of round messages as described in the second case of Definition 10 exists. More in detail, if $\mathsf{ID}' \in U^i_{\mathsf{ID}}$, there is a sequence of round messages $\mathfrak{M}'_1, \ldots, \mathfrak{M}'_{r_i}$ that ID′ would process with respect to their state at query q' (recall this corresponds to either the last update query issued by id' and weakly processed by ID or the time ID′ joined the group), and which would bring them to a state consistent with γ^i_{ID}. We will show how E can compute the sets U^i_{ID} along with the queries weakly processed by ID. The initial set U^i_{ID} can be determined by E as follows. If ID sets up the group, this set starts empty; whereas if ID is added to the group by ID′ through an add query CGKA.Add sampled in query q', this set is $U^1_{\mathsf{ID}} = U^{q'}_{\mathsf{ID}'}$. Further, E can track the impact of adds and removes in these sets as follows. Whenever ID processes and accepts a round message in query $\tilde{q}$ containing a remove or an add, E respectively removes the corresponding party from $U^{\tilde{q}}_{\mathsf{ID}}$, and adds it if it was added by a party that is in $U^{\tilde{q}-1}_{\mathsf{ID}}$. To justify the latter, observe that a newly added party will be added with a state consistent with that of the party who added them.

It remains to show that E, given knowledge of the updates weakly processed by ID up until that point, and the set U^{q-1}_{ID} of parties with whom ID has not been pushed to inconsistent states with, can extract the updates that are weakly processed by ID when processing $\mathfrak{M}$ and compute U^q_{ID}. Let γ^q be the state of ID after processing $\mathfrak{M}$, and let γ^{q-1} be its state before it. In particular, we want to identify the set Q' of updates that were weakly processed but not explicitly processed by ID when transitioning from γ^{q-1} to γ^q.

We start with some terminology and a couple of observations. Given a pair of openings $o_v = (o_{v,1}, o_{v,2})$ associated to node v, received in $\mathfrak{M}$ during query process q (and supposed to be commitments to the keys on the subtree under v's parents), we say that o_v is a *correct opening pair* if there is are queries to the RO with output $o_{v,1}$ and $o_{v,2}$ and input a tuple $(o_{v_i,1}, o_{v_i,2}, \mathsf{pk}_{v_i})$, for $i \in \{1,2\}$ where $o_{v_i,j}$ are hash values in the image of the RO, and pk_i is either equal to $\mathtt{blank}$, or is the public key associated to the corresponding parent of v in the output of some update query q' issued by user ID' and taking place before q. In the former case, we require that $o_{v_i,1}$ and $o_{v_i,2}$ are also correct openings, or their parents, should the corresponding key be $\mathtt{blank}$, and so on. Moreover, we require that, either the update pk_i to which belongs to was (one of, if several were concurrent) the last affecting that node which ID weakly processed, or that the state of ID' at time q' is consistent with that of ID at $q-1$. This will ensure that if it corresponds to an update not yet weakly processed by ID, this it will be possible for ID to weakly process it at q. Last, we further require that one of the pk_{v_i} belongs to the same update as the pk_v. This, is also needed to ensure ID would process any such update, as any user for which those two nodes are in their state will not accept a round message not satisfying that.

Now, we observe that, for ID to weakly process in q the query to which pk corresponds to or, in fact, any update query coming from a user in v's subtree other than that setting the new key for v, it must be that the opening pair o_v is a correct opening pair. Indeed, for ID to weakly process such update query, there must exist a user ID^* in the subtree under v that would, with respect to some of its states and a certain chain of round messages, ultimately explicitly process said update, and end up in a state consistent with γ^q. However, said user's state will necessarily include a key for (the resolution of) the parents of v, as well as two openings corresponding to the node's parents (of the nodes in the resolution). Thus, were these openings not correct, either because they were never output by the RO, or because their inputs were not of the right format, or sampled by a user with an inconsistent state, we would have a contradiction to either the preimage or collision resistance of the RO, or the fact that said user would explicitly process the update, respectively. Also, note that that user, before processing that last message in the chain, would have a state consistent with that of ID at $q-1$, and would thus accept said update.

Further, observe that it must be the case that, for any update coming from a user in v's subtree to be weakly (and only weakly) processed by ID, there must be a path of correct openings all the way from v to a leaf. Indeed, ID^*'s state will contain public keys (and implicitly openings) for each of the nodes in its path. Were the above not true, the fact that ID and ID^* end up sharing the same round hash would again imply a contradiction with the properties of the RO.

In order to identify the exclusively weakly processed updates and update U^q_{ID}, E performs the following steps for each of the non-leaf nodes $v \in \mathsf{Res}^q(\mathsf{co\text{-}path}(\mathsf{ID}))$, i.e. in the resolution of ID's co-path after processing $\mathfrak{M}$, for which its opening pair is correct, following the observations above.

We distinguish several cases. Case (1), $v \in \mathsf{Res}^{q-1}(\mathsf{co\text{-}path}(\mathsf{ID}))$, i.e. v is not a node introduced in ID's state in q. E now looks at each of the two openings $o_{v,1}$ and $o_{v,2}$ and distinguishes two further situations in each case, depending on whether the opening values are new. Case (1.1) $o^q_{v,i} = o^{q-1}_{v,i}$, i.e. the opening $o_{v,i}$ does not change by processing $\mathfrak{M}$, and (1.2) the opposite, $o^q_{v,i} \neq o^{q-1}_{v,i}$. In case (1.1), E does not add any update to Q' and stops further examining any values or nodes in the subtree under the corresponding parent of v. The reason being that in this case it is not possible for ID to have weakly processed any update coming from a user in said subtree. To see why this is the case, assume for contradiction that it is not. Then there exists some user under said subtree that could explicitly process some update and end up in a state consistent with ID, making ID to have weakly processed it. This user must have been in consistent states with ID before processing the update, and in particular have the same transcript hash value as ID at that time. However, when processing the message through which they explicitly processed this update, the labels for v must necessarily change for them, i.e. the labels for v they input into their computation of the round hash must be different for this process query and the previous one. But then it is impossible for the two of them to have the same round hash value in both rounds and therefore impossible for them to have consistent states, by the collision resistance of random oracle H_3.

In case (1.2), E recovers the inputs $(o_{w_i,1}, o_{w_i,2}, \mathsf{pk}_i)$ of the RO query with output $o_{v,i}$ (these exist and have this form by the assumption on $o_{v,i}$ belonging to a correct opening pair), where w_i is the corresponding effective parent of v. Then it checks that $(o_{w_i,1}, o_{w_i,2})$ is a correct opening pair, ignoring the subtree under w_i if not, and makes the same distinction as before, now applied on the $o_{w_i,j}$, iteratively excluding subtrees under any node for which its opening pair is not correct, or for which its key corresponds to updates already weakly processed by ID. At the end of this iterative process, E compiles a list of all the updates given by the keys in the correct openings, which had not yet been weakly processed by ID. It remains to determine for which of these updates there are users in U_{ID}^{q-1} which could explicitly process them through processing some round message and arrive at a state consistent with γ^q. To this end, first note that, following the observation above, only users for which there is a path of correct opening pairs from their path to v will be able to process some message and end up in such a consistent state. Thus, E considers the users in U_{ID}^{q-1} for which such a path exists, and looks at the nodes in this path and the resolution of its copath which contained keys (those the openings were committing to) belonging to updates in the previously mentioned list. The updates from this list for which this is the case are then added to Q'. To see why, note that if ID^* is such a user, the message $\mathfrak{M}^*$ output by collecting all the operations and updates processed or explicitly processed by ID in $\mathfrak{M}$, together with those in Q', with the same ordering applied, will be accepted by ID^* (with respect to their state after processing the corresponding messages $\mathfrak{M}^*_1, \dots, \mathfrak{M}^*_{r_{q-1}}$ given by the fact $\mathsf{ID}^* \in U_{\mathsf{ID}}^{q-1}$). Moreover, by setting the openings in $\mathfrak{M}^*$ corresponding to the parents of nodes in the resolution of ID^*'s copath, to be the same as those given by the RO queries, we ensure that ID^* will derive a state consistent with γ^q. Last, E removes from U_{ID}^{q} those user for which such path of correct opening pairs from their leaf to v does not exist.

We now examine case 2), where v has been added to ID's state as a result of processing $\mathfrak{M}$. In particular, this means that v is in the resolution of a node blanked by some removal in $\mathfrak{M}$. Recall that ID receives from the server the states of nodes in a subtree spanning the paths and (resolutions of) co-paths of removed members. A subtree that, in particular, contains a state from v. E first checks whether the key at v after processing $\mathfrak{M}$ matches that in the subtree (this will not be the case if there was some update concurrent with the removal included in the round message), and adds no query to Q' if this is the case. If it is not, then whichever update the new key at v corresponds to was explicitly processed by ID at q, and E then performs the same steps as in the previous case, looking at the keys commited in the openings if these are correct, and working its way down the subtree under v through examining the RO queries, all the way to the ratchet tree leaves. Note that the removals taking place in $\mathfrak{M}$ would also be processed by whichever users under v's subtree could still remain in consistent states with ID, since these will receive node states for the same nodes in the resolutions of all blanked descendants of v (and any openings sent to ID can also be mirrored in the last round message $\mathfrak{M}'_t$ sent to said user). □

Proof of Lemma 7. Now, in order to prove the statement of the Lemma, we will argue for contradiction, assuming an adversary A that wins the OPEN game exists. We will show that if A had a non-negligible probability of winning the game at some given query, then it must have had it also with respect to an earlier query. Since there is no chance for A to win the game with respect to the first process query (as this must correspond to either the group creator processing their own update or a user joining the group right after initialization), this will show that all queries corresponding to node states in T_q must have been processed by ID at time or before q, giving us the desired contradiction. Accordingly, consider an execution of the OPEN game, and let q be the first query of the form $q = \mathsf{CGKA.Proc}(\mathsf{ID}, \cdot)$ where A could win, i.e. the first query for which a tree T_q, computable in polynomial time by A and satisfying all conditions in Def. 12 exists.

By assumption, there is a node $v \in \mathcal{P}^q(\mathsf{ID})$ in the state of ID at time q, with label ℓ_v, which is the same as the label of the root node of T_q. Recall that if the state of v contains openings $o_v = (o_{v,1}, o_{v,2})$ and a public key pk_v, then $\ell_v = \mathsf{H}_3(o_v^1, o_v^2, {}^p\gamma(v))$. Note that we could assume w.l.o.g. that $v \in \mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$. Indeed, note that for any node in $T_q \bigcap \mathcal{P}^q(\mathsf{ID})$, by collision resistance of H_3, the states of that node in T_q and $\mathcal{P}^q(\mathsf{ID})$ must match. Moreover, ID must have weakly processed (in fact, explicitly processed) any query setting any keys for nodes in $\mathsf{path}(\mathsf{ID})$. Thus if T_q satisfies the conditions of Def. 12, then so must a subtree of it rooted at a node in $\mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$.

Moreover, by assumption, we know that $\mathsf{PHash.Ver}(T_q) = 1$. Recall this means that every non-blank node in the T_q has a complete public state; and that for any internal node w, its associated ID_v is that of a user whose leaf is in the sub-tree rooted at w, and if w_1 and w_2 are w's effective parents, we have that either:

(a) $h^2_{w_1} = \mathsf{H}_4(\mathsf{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h^2_w, \{\mathsf{pk}_y\}_{y \in R})$ and $h^1_{w_1} = \ell(w_2)$ or

(b) $h^2_{w_1} = \mathsf{H}_4(\mathsf{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h^2_w, \mathrm{PK}^{\mathrm{pr}}_{w_2})$ and $\mathcal{H}_{trans,w_1} = \mathcal{H}_{trans,w_2}$.

for $R = \mathsf{Res}(w_2) \setminus \mathsf{Unmerged}(w)$. Moreover, $\mathsf{Sig.Ver}_{\mathsf{svk}_w}((\mathsf{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h_w, \mathcal{H}_{trans,w}, \mathsf{confTag}_w), \sigma_w) = 1$.

Let ID_1 be the user located at the end of the path from a ratchet tree leaf in T_q to v which, again, we know exists by assumption. For ease of exposition, we will assume w.l.o.g. that ID_1 is the left-most leaf in the subtree of $\mathfrak{T}^q$ under v .(where $\mathfrak{T}^q$ is the ratchet tree of the whole group in the view of ID; even though ID does not have a full view of $\mathfrak{T}_q$, they do know which users are associated to which leaves.) ID_1's path in T_q will contain a number of updates from some of the users under v's subtree, which allow us to partition said path as follows. Let v_1 be the leaf of ID_1, and $v_1, \ldots, v_k = v$ the nodes in said path. As noted above, every key associated to the v_i must stem from either an add 9in the case of v_1 or an update query, since all the signature at each v's state verifies. We can partition this path into sub-paths given by sets of nodes that were sampled in the same update: v_i and v_{i+1} are in the same sub-path exactly when v_{i+1} verifies through v_i, according to $\mathsf{PHash.Ver}$. We denote the different updates that make up said path, and such that each corresponds to one element in the partition, as $q_1, \ldots, q_m$, in order, with q_1 being the update setting the key for v_1, and q_m the one setting the key for v. Accordingly, we denote by ID_i the user who authored q_i. Let q_i be an update corresponding to a partition element with two or more nodes, i.e. to which two or more nodes in the path correspond to. Then, we denote by $q_{i,j}$, $j \in (1, \ldots, m_i)$ the updates that correspond to the nodes that are in $\mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}_1))$, i.e. in the resolution of ID_1's co-path, which for which their child corresponds to q_i. If there any blank nodes in between those sampled by q_i and q_{i+1}, we denote by $q^b_{i,j}$ those updates corresponding to nodes in the resolution of said blank nodes, and which are not q_1. Here $j < j'$ if $q_{i,j}$ (resp. $q^b_{i,j}$) was sampled by a party whose index is to the left of the party who sampled $q_{i,j'}$ (resp. $q^b_{i,j'}$). As before, we denote the user who sampled $q_{i,j}$ (resp. $q^b_{i,j}$) by $\mathsf{ID}_{i,j}$ (resp. $\mathsf{ID}^b_{i,j}$).

Along the way we will distinguish based on whether the parent hash of the lowest node in $\mathsf{path}(\mathsf{ID}_1)$ sampled by any given q_i verifies through conditions (a) or (b) above. Recall that condition (a) corresponds to the case where ID_i was aware of q_{i-1} at the time q_i, and had already weakly processed it by that time (as we will now show). In turn, condition (b) corresponds to the case where q_i and q_{i-1} were concurrent, in the sense that both users were in consistent states when sampling them and were not aware of the existence of the opposite update when sampling theirs. Thus, for ease of notation, we will write $q_{i-1} \lhd q_i$ to represent the first case, and $q_{i-1} \sim q_i$, to represent the second one. For any query $q_{i,j}$, i.e. those affecting only the nodes on v's copath, we say, similarly, that $q_{i,j} \lhd q_j$ and $q_{i,j} \sim q_j$ if the node $\mathsf{Int}(\mathsf{ID}_i, \mathsf{ID}_{i,j})$ verifies through condidions (a) and (b), respectively.

We will now prove the following claim, from which the Lemma will follow easily. Informally, it states that from its state at q_1 onwards, there is a chain of round messages that ID_1 would accept and that would bring them to a state consistent with ID's current state, along the way explicitly processing all updates corresponding to nodes in the path and copath of ID_1.

Claim. *For any $j \in (1, \ldots, m-1)$, such that $q_j \lhd q_{j+1}$, there is a sequence of efficiently computable round messages $\mathfrak{M}_1, \ldots, \mathfrak{M}_{r_j}$ that ID_1 would process and accept with respect to their state at q_1, and such that the state of ID_1 after processing that last message $\mathfrak{M}_{r_j}$, $\gamma^{q_{r_k}}_{\mathsf{ID}_1}$, is consistent with that of ID_{j+1} at time q_{j+1}, i.e. with $\gamma^{q_{j+1}}_{\mathsf{ID}_{j+1}}$.*

Proof of Claim. First, we make the following observation: if $\tilde{q} \lhd \hat{q} < q$, then it must be that the user $\hat{\mathsf{ID}}$ that issued the update in $\hat{q}$ had already weakly processed $\tilde{q}$ at the time $\hat{q}$. Indeed, by the fact that parent hash verifies at the node where the both update paths meet, we know that $\hat{\mathsf{ID}}$ had some key from $\tilde{q}$ in their state. If there are not blank nodes in between the nodes set by $\hat{q}$ and the node set by $\tilde{q}$, then the update in $\tilde{q}$ must have been explicitly processed by $\hat{\mathsf{ID}}$, since a node they sampled belongs to the copath of $\hat{\mathsf{ID}}$. If, instead, there is at least one blank node in between $\hat{q}$ and $\tilde{q}$ and $\hat{(\mathsf{ID})}$ did not explicitly process $\tilde{(q)}$ at any point, then, by the minimality of q, it must be that $\hat{\mathsf{ID}}$ had already weakly processed $\tilde{q}$ at $\hat{q}$. To see why, note that $\hat{\mathsf{ID}}$ introduced the node corresponding to $\tilde{q}$ into their state at some point before $\hat{\mathsf{ID}}$, by processing a round message containing at least one remove operation. However, since $\hat{\mathsf{ID}}$ accepted that round message, it must be that the parent hash verification of said tree passed and, moreover, that it was a well-formed tree containing at least one user leave (that of the removed party). Further, all node states in it must have corresponded to past queries, except with negligible probability, since the adversary is not allowed use of signing keys and all node states come with a signature. Thus, that tree would have satisfied all constraints in Def. 12 and, moreover, have contained a node state corresponding to an operation not yet weakly processed by $\hat{\mathsf{ID}}$. In particular, this means A could have won the OPEN game with respect to some query earlier than q, which is a contradiction to the minimality of q.

We will prove the claim by induction on j, but will first start with a couple of simplified case, capturing most of the arguments used in the inductive argument, in order to build intuition. Suppose first $j = 1$, i.e. $q_1 \lhd q_2$, and we assume that there are no blank nodes between the nodes sampled by q_1 and those sampled by q_2. We will show the claim is true for this particular case.

We first show that ID_2 was at some point in a state consistent with that of ID_1 at q_1. To see this, first note that at least one node state set by q_1 is part of ID_2's state at q_2, from parent hash verification. Now, if q_1 corresponds to the query where ID_1 processed their welcome message, then the state of ID_1 after processing the corresponding round message must have been consistent with that of ID_A, the party who added them, at the time they issued the add operation. However, since ID_2 also must have processed the corresponding add query from ID_A, their state must have also been consistent with ID_A's at that time and, by transitivity of $=$, also with ID_1's at q_1. If, in turn, q_1 was sampled by ID_1 after they joined the group, then we know it must have been explicitly processed by ID_2 (since by assumption there are no blank nodes between nodes from q_1 and nodes from q_2, i.e. the highest node sampled by q_1 is the parent of a node sampled in q_2 and therefore a node in the copath of ID_2), which, by Proposition 3, means their state before processing it was consistent with that of ID_1 at q_1.

Now, from the time the state of ID_2 was consistent with ID_1's at q_1 all the way to q_2, ID_2 must have processed a series of round messages (we know they processed at least one: the one containing q_1). For any $q_{1,j}$ such that $q_{1,j} \lhd q_1$, we know that ID had already weakly processed $q_{1,j}$ at q_1, by the observation at the beginning of the claim's proof. Now, note that all packets from the $q_{1,j}$ such that $q_{1,j} \sim q_1$ will be accepted by ID_1 with respect to its state at q_1, since by parent hash verification of nodes sampled by q_1 (in particular the check involving $\mathcal{H}_{trans}$), ID_1 was in consistent states at that time with the corresponding user $\mathsf{ID}_{1,j}$, and moreover, by the equality check involving the predecessor keys $\mathrm{PK}^{\mathrm{pr}}$, ID considered those parties as belonging to the corresponding subtree. This is necessary, as ID_1 will only accept an packet corresponding to some node u if it is signed by a user belonging to the subtree under u. Note that if $\mathrm{PK}^{\mathrm{pr}}$ was not part of the state or not included in the signatures, a tree resulting from rearranging update paths from users that update concurrently and from consistent states, would pass parent hash verification. For example, consider the case where all users update concurrently, with the left update always winning; the tree where the update paths of any even-index users are rearranged would still verify (this issue is similar as the one that earlier versions of parent hash for TreeKEM suffered of [5]).

Consider now all other packets included in the round message $\mathfrak{M}_{\mathsf{ID}_2,1}$ that ID_2 processed and through which they processed q_1, which were either adds or removes, or corresponded to node states for nodes that are either $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)$, nodes above it, or parent of these. All this packets would also be accepted by ID_1 with respect to $\gamma_{id_1}^{q_1}$, since they were accepted by ID_2 at a time where their state was consistent with it; and the information used to determine aceptance of this packets is based on values that lie in the intersection of the states of both parties. Indeed, the acceptance of update or add packets is based uniquely on values corresponding to the key schedule and membership set, and the acceptance of removes is based also on values stored on node states. Since the nodes mentioned are in the states of both parties, and the node states must match by the fact that the states are consistent, the statement follows. Thus, the message $\mathfrak{M}_1$ containing the packets corresponding to the $q_{1,j}$ such that $q_{1,j} \sim q_1$, and all other packets from $\mathfrak{M}_{\mathsf{ID}_2,1}$ that are either adds, removes or correspond to the node states above will be accepted by ID_1. Moreover, if this message contains, for packets corresponding to the $q_{1,j}$, the same openings as the corresponding nodes in T_q, and, for all other node states, the same openings included in the message $\mathfrak{M}_{\mathsf{ID}_2,1}$ to ID_2, then ID_1 after processing $\mathfrak{M}_1$ will be in a state consistent with ID_2 after they weakly processed q_1. To see this, note that the states of nodes in ID_1's state below $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)$ will match exactly those in T_q, and that those for the remaining nodes must match those in ID_2's state. Since by parent hash verification, we know $h^1_{w_2} = \ell(w_1)$, where w_1 and w_2 are the left and right parents of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)$, we know the label both parties compute for w_1 must be the same as well.

As to the remaining round messages processed by ID_2 from this time until q_2, it must be that none of this included any changes to the subtree under the highest node sampled by q_1. Indeed, since we are for now assuming that there are no blank nodes in between those sampled by q_1 and q_2, for ID_2 to process something affecting that node after processing q_1, they would have to explicitly process q_1 again in order to have a node state from that query in their state at q_2. However, by Corollary 3, we know this cannot be the case, since explicitly process implies weakly process. Thus, all other messages processed by ID_2 from that point onwards until q_2 must have only affected nodes that are, are above, or are parents of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)$. By the same argument as before, ID_1 would process and accept the corresponding round messages $\mathfrak{M}_2, \ldots, \mathfrak{M}_{r_1}$, containing those same packets, in the same order, bringing them to a state consistent with ID_2's at q_2. This proves the claim above for simpler case where $j = 1$, there are no blank nodes between the nodes sampled by q_1 and q_2.

Before we move on, we will now discuss the effect of blank nodes between the nodes sampled by q_1 and q_2 in the above argument. First note that now we do not have the guarantee that ID_2 explicitly processed q_1, however this is taken care of by the observation from the beginning of the claim, which implies ID_2 weakly processed q_1 before q_2. By Proposition 3, this implies ID_2 was at some point in a state consistent with that of ID_1 when issuing q_1.

Further, we now need to show that ID_2 did not process any other operations affecting nodes on the subtree under the highest node sampled in T_q by q_1. Above we relied on the fact that ID_1 would have to explicitly process q_1 twice but, again, we do not have this guarantee. We argue as follows. In the case wjere blanks between q_1 and q_2, if ID_2 weakly processed but did not explicitly process q_1, say by processing round message $\mathfrak{M}_{\mathsf{ID}_2,1}$, it must be that they processed and accepted a round message containing some removal *after* they processed $\mathfrak{M}_{\mathsf{ID}_2,1}$. Let $\mathfrak{M}_{\mathsf{ID}_2,2}$ be some round message, processed and accepted by ID_2 between weakly processing q_1 (through processing $\mathfrak{M}_{\mathsf{ID}_2,1}$) and issuing q_2, where ID_2 weakly processed such an operation affecting the nodes on the subtree under the highest node sampled in T_q by q_1. First, note that if ID_2 processed no round messages including removals between processing $\mathfrak{M}_{\mathsf{ID}_2,2}$ and time q_2, then they must have processed some round message in that interval setting the key of some node in $\mathsf{Res}(\mathsf{Iparent}(\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)))$ to one set by q_1. But this node must have already been part of ID_2's state at that time, meaning they would have explicitly processed q_1 after already weakly processing it, a contradiction to Corollary 3. Thus, ID_2 must have processed some round message containing a remove after processing $\mathfrak{M}_{\mathsf{ID}_2,2}$ and, in particular, it must have been such a message that put the key from q_1 into their state. However, for ID_2 to accept such a message the received tree corresponding to the removed user(s) path and co-path resolution must have been consistent with said key from q_1. Since we know the openings left by $\mathfrak{M}_{\mathsf{ID}_2,2}$ were not consistent with it, ID_2 must have processed yet another $\mathfrak{M}_{\mathsf{ID}_2,3}$ in between these two messages, in particular including openings now again consistent with the key from q_1. Assume $\mathfrak{M}_{\mathsf{ID}_2,3}$ is the last such round message processed by ID_2 before they the aforementioned round message that put the key from q_1 into their state. We know that that round message containing a remove must have contained a tree T that was consistent with the openings sent in $\mathfrak{M}_{\mathsf{ID}_2,3}$ (and thus such that all queries corresponding to node states in T had already been weakly processed by ID_2 at the time, by the argument at the beginning of the claim's proof). Now T must contain some query q' that ID_2 weakly processed between (and including) $\mathfrak{M}_{\mathsf{ID}_2,2}$ and $\mathfrak{M}_{\mathsf{ID}_2,3}$, such that $q_1 \lhd q'$ in T. If there are no blank nodes between q' and q_1, then whoever issued q' must have explicitly processed q_1, implying ID_2 weakly processed q_1 a second time in that interval, a contradiction. Else, we can recurse and repeat the same argument with respect to ID', which must eventually lead us to some user weakly processing q_1 twice, since we are considering progressively lower nodes in the tree.

The remaining part of the argument that is affected by these blank nodes is that now there are queries $q^b_{1,j}$ that we need to show ID_1 would weakly process by the time they process $\mathfrak{M}_{r_1}$. To see this is the case, note that all such blank nodes between q_1 and q_2 must have been created (in the view of ID_2) between (possibly including) them processing q_1 and the query q_2. Moreover, during this interval, ID_2 cannot have processed any removes coming from the subtree under the highest node sampled by q_1 in T_q, following the argument above. However, note that if we were to construct the sequence of round messages for ID_1 as before, we would now need to include any updates affecting nodes in the subtree under $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_2)$ but outside the subtree under the highest node sampled by q_1. Further, note that these might no longer be reflected in T_q, since there could have been several of them, and earlier ones could have been overwritten. Nevertheless, we can use the algorithm E from Lemma 8 to determine which operations affecting these nodes were processed by ID_2 in this time interval, and add those to the crafted messages to ID_1, with the corresponding openings. Further, tree that the server would need to send to ID_1 for each message containing a remove would contain exactly the same node states as the tree received by ID_2 in their corresponding messages. Thus, they would be accepted by ID_1. In particular, any of the nodes corresponding to the $q^b_{1,i}$ either corresponds to an update ID_2 explicitly processed (either after or in one of the messages where the blanks were created), or would have been included in at least one of such trees. In the first case, ID_1 would also explicitly process such query in the process; in the second, again, by minimality of q, we know that ID_1 must have weakly processed the corresponding query. This proves the claim for $j = 1$ in the general case, where blank nodes can exist between q_1 and q_2.

Now, we will start the induction proof of the claim. Let $j \in (2, \ldots, m-1)$ and suppose that $q_j \lhd q_{j+1}$ and, further, that for any $j' < j$ for which $q_{j'} \lhd q_{j'+1}$, a sequence $\mathfrak{M}_1, \ldots, \mathfrak{M}_{r'_j}$ as in the claim exists. For the base case, we consider the case where j is the minimal such index for which $q_j \lhd q_{j+1}$. In that case all, for all $j' < j$, $q_{j'} \sim q_{j'+1}$, and thus, by parent hash verification (in particular by the fact that the transcript hash values in those updates match), we know that all $\mathsf{ID}_{j'}$ were consistent with each other with respect to their respective states $\gamma^{q_{j'}}_{\mathsf{ID}'_j}$. In particular, it means that ID_1 would accept any packet from those updates that was included in a round message. Moreover, as above, we will show that ID_{j+1} must have weakly processed q_1 at some point, which in turn implies that ID_{j+1} must have at some point had a state consistent with that of ID_1 at q_1. To show this, note that by parent hash or verification of the node $\mathsf{Int}(\mathsf{ID}_{j+1}, \mathsf{ID}_1)$, we know that, at q_{j+1}, the label of the left parent of said node, in the view of ID_{j+1} corresponded exactly to the label of that subtree of T_q, to which the node(s) sampled by q_1 belong to. By minimality of q, we know then that ID_{j+1} must have weakly processed q_1.

Similar to above, $\mathfrak{M}_1$ can now be constructed as containing all $q_{j'}$ for $j' \in \{1, \ldots, j\}$, together with all the $q_{j',\hat{j}'}$ such that $q_{j',\hat{j}'} \sim q_{j'}$, as well as all queries contained in the message that ID_{j+1} processed when they weakly processed q_1, and which affected nodes outside of the subtree under the highest node sampled by q_j. A similar argument to the one above shows that ID_1 would accept all these individual packets and thus the round message, and, if this round message contained the correct openings (those given by T for nodes the subtree under the highest node sampled by q_j, and those given by the round message for ID_{j+1} for the remaining nodes), then ID_1 will transition to a state consistent with that of ID_{j+1} at the time the latter processed q_1. The exact same argument above shows that there is a sequence $\mathfrak{M}_2, \ldots, \mathfrak{M}_{r_j}$ of round messages ID_1 would process and accept after processing $\mathfrak{M}_1$ that bring them to a state consistent with that of ID_{j+1} at q_{j+1}. Moreover, along the way, ID_1 processes all $q^b_{j,\hat{j}}$ that they had not already weakly processed, as before, those would correspond to nodes whose states must have been included by the server in the corresponding round messages including the removals that caused any blank nodes between the nodes sampled by q_j and those sampled by q_{j+1} (note that no such $q^b_{j,\hat{j}}$) exist if no such blank nodes exist). Finally, just as before, ID_1 must also before q_1 have processed any $q_{j',\hat{j}}$ for $j' \leq j$ for which $q_{j',\hat{j}} \lhd q_{j'}$. Indeed, we know the label of the highest node sampled by q_j in T matches the label of the same node in ID_1's state after processing q_1, so by minimality of q, ID_1 must have processed all queries corresponding to this subtree. This concludes this case.

Now, we consider the inductive step, where j is not minimal, i.e. there exist queries q_κ satisfying $q_\kappa \lhd q_{\kappa+1}$ with $\kappa < j$, and we assume that the claim is true for any such $\kappa < j$. Let $q_{j'}$ be the query where $j' < j$, $q_{j'-1} \lhd q_{j'}$, and such that j' is maximal satisfying those two properties. In particular, j' is such that for all $\hat{j} \in \{j'+1, \ldots, j-1\}$ (if such interval exists) we have $q_{j'} \sim q_{\hat{j}}$. By the induction hypothesis there is a sequence of messages $\mathfrak{M}_1, \ldots, \mathfrak{M}_{r_{j'-1}}$ which ID_1 would process with respect to their state at q_1 and which ushers them into a state consistent with that of $\mathsf{ID}_{j'}$ at time $q_{j'}$.

In this case, using the same argument as above, ID_{j+1} processed all $q_{j'}, \ldots, q_j$, by minimality of q, before q_{j+1}. Thus, ID_{j+1} was at some point in the past in a state consistent with $\mathsf{ID}_{j'}$ at $q_{j'}$, and thus with ID_1

after they processed $\mathfrak{M}_{r_{j'-1}}$. Moreover, it must be that since that point and all the way to q_{j+1}, ID_{j+1} has not weakly processed any updates, adds or removes affecting nodes in the subtree under the left parent of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_{j'})$. To see why, note first that all the $q_{j'}, \dots, q_j$, must have processed at the same time since they all had the same transcript hash values. At the time just before processing those, $q_{j'}$ had to have already weakly processed all updates below $q_{j'}$. Indeed, we know that there is a tree that would agree with $\mathsf{ID}_{j'}$'s state at that time and which would contain the nodes for all those queries (following the fact that the parent hash verification of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_{j'})$ is through condition (a)). But since the state $\mathsf{ID}_{j'}$ had at time $q_{j'}$, is consistent with ID_j's state at the time they processed $q_{j'}$, such tree must have also been consistent with ID_{j+1}'s state, showing, again, by minimality of q, that ID_{j+1} had weakly processed those queries by then. Moreover, we know that at q_{j+1} their state was also consistent with the subtree under the highest node affected by q_j in T_q. Thus, if ID_{j+1} processed anything else in between weakly processing $q_{j'}$ and time q_{j+1} affecting those nodes on the subtree under the highest node affected by $q_{j'}$, then must have processed some other round message afterwards that brought their state consistent with the subtree under q_{j+1} again. In particular, they must have processed something that affected the left parent of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_{j+1})$, and later processed something that set the key at the highest node sampled by q_j in T to be the same one as they already had in their state at the time of weakly processing $q_{j'}$ (and therefore also q_j). This means that they must have weakly processed q_j twice, which is a contradiction to Corollary 3. Thus, it must be that since the time ID_{j+1} weakly processed $q_{j'}$ and all the way to q_{j+1}, ID_{j+1} has not weakly processed any updates, adds or removes affecting nodes in the subtree under the left parent of $\mathsf{Int}(\mathsf{ID}_1, \mathsf{ID}_{j'})$.

In particular, this means that we can treat this as the case where j is minimal, and argue that there is a sequence of round messages $\mathfrak{M}'_1, \dots, \mathfrak{M}'_{r'_j}$ that $\mathsf{ID}_{j'}$ would process from time $q_{j'}$ that would put them into a state consistent with ID_{j+1} at q_{j+1}, where $\mathsf{ID}_{j'}$ would along the way weakly process all queries corresponding to nodes between $q_{j'}$ and q_j; and, moreover, such that those messages would contain no changes from the subtree under $q_{j'}$. But then, the corresponding sequence of round messages $\mathfrak{M}^1_1, \dots, \mathfrak{M}^1_{r'_j}$, sent this time to ID_1 after they processed $\mathfrak{M}_{r_{j'-1}}$, and containing the same packets as the $\mathfrak{M}'_1, \dots, \mathfrak{M}'_{r'_j}$, would have the same effect and would also be accepted, since they only concern nodes that in the intersection of the states of the two parties. Thus, if we let $\mathfrak{M}_{r_{j'-1}+k} \leftarrow \mathfrak{M}^1_k$ for $k \in \{1, \dots, r'_j\}$ then the sequence of messages $\mathfrak{M}_1, \dots, \mathfrak{M}_{r_{j'-1}+r'_j} =: \mathfrak{M}_{r_j}$ brings ID_1 from their state at q_1 to a state consistent to ID_{j+1}'s at q_{j+1}, along the way weakly processing all the necessary queries. This completes the proof of the claim.

□*(End of proof of Claim)*

If q_j is the maximal query in ID_1's path in T_q for which $q_{j-1} \lhd q_j$, then we know from the proved claim above that there is a sequence of round messages that ID_1 would process from its state in q_1 that lead them to a state consistent with that of ID_j at q_j. Moreover, for any $j' > j$, $q_{j'-1} \sim q_{j'}$, just by definition of q_j. Thus, we will argue now that there exists one last message $\mathfrak{M}_m$ that ID_1 would process with respect to their state after processing $\mathfrak{M}_{r_{j-1}}$, and which will take them to a state consistent with that of ID at the time they weakly processed q_m. First, observe that ID's state before weakly processing q_m, in particular the corresponding openings, must have been consistent with the subtree of T_q below q_j, since they must have been in consistent states with ID_m at q_m and therefore with ID_j at q_j (since $q_j \sim q_m$). Moreover, after processing the round message through which they weakly processed q_m, their state must still be consistent with said subtree of T_q, just because of the assumption that $\ell(T)$ is equal to the label $\ell(v)$ is ID's state. This implies that this round message through which they processed q_m contained no changes for that subtree of T_q in question, following the same argument used above. Thus, following the arguments in the proof of the claim, the round message, now sent to ID_1 containing the same packets as the one sent to ID would be accepted and would prompt ID_1 into a state consistent with that of ID.

However, this implies that all queries $\tilde{q}$ in T_q such that $\tilde{q} \sim q_j$ would be explicitly processed by ID_1 by processing this message, and thus weakly processed by ID at this time. Moreover, all other queries corresponding to all other nodes in T_q were already weakly processed by ID_1 at the time before they processed $\mathfrak{M}_m$. In particular, since the states of ID and ID_1 before they each weakly processed q_j were consistent, so must have been their round hash value at that round. Thus, by minimality of q, again, ID must have already weakly processed all those queries by that time. This concludes the proof. □

Finally, we turn to the proofs of Lemmas 4 and 5.

Proof of Lemma 4. Before starting the proof we make the following observation. Let ID and ID^* be in consistent states after each processing queries $q = \mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}}, \mathfrak{M})$ and $q^* = \mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}^*}, \mathfrak{M})$, respectively, and assume that ID^* joined the group at the same time or after than ID. Then, by collision resistance of H_5, the sequence of transcript hash values (and round hash values) ID^* has computed throughout their execution, up to time q^*, must be a suffix of those ID has computed up to time q. In particular, for any query of the form $\mathsf{CGKA.Proc}(\gamma^*_{\mathsf{ID}}, \cdot)$ in the protocol execution where ID^* accepts the received round message, there must be a query $\mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}}, \cdot)$ where ID accepts the received round message; and moreover, their states between any two pairs of such corresponding queries must be consistent, by Proposition 2. Indeed, this must be the case since accepting and processing a round message always changes the users' state, as e.g. the transcript hash is updated by hashing in a new round hash value. We can thus draw a 1-to-1 correspondence between processed messages by one and the other. More formally, if $(q_1^*, \ldots, q_k^*)$ are the process queries $q_i^* = \mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}^*}^{q^*_{i-1}}, \cdot)$ that ID^* processed since joining the group up until q, then there is a bijection $q_i^* \mapsto q_i$ between those and the k last process queries $(q_1, \ldots, q_k)$, $q_i = \mathsf{CGKA.Proc}(\gamma_{\mathsf{ID}}^{q_{i-1}}, \cdot)$, that ID processed before q, such that the states of ID and ID^* between q_i and q_{i+1} and between q_i^* and q_{i+1}^*, respectively, are consistent.

We will start by arguing that every update U that ID^* weakly processed at or before time q was also weakly processed at or before time q by ID. Consider first the case where ID^* weakly processed U at the moment of joining the group, i.e. ID the tree T that ID received as part of the welcome message contained a node with a public key which was part of U. Because ID^* accepted the welcome message, we know that $\mathsf{PHash.Ver}(T) = 1$, and that it has the format required in Def. 12, that is, indegree two everywhere except at the leaves, no blank leaves, the state at each node corresponds to some update sampled during the protocol execution (since the signatures at each node verify), and at least one of its leaves contains a public signing key. Moreover, since the state of ID^* after processing q_1^* is consistent with that of ID after processing q_1, it holds that the label of T must match the label of v_{root} in ID's state, $\gamma_{\mathsf{ID}}^{q_1}$, at that time. However, if T contained a node state corresponding to some query not yet processed by ID, we could then build an adversary against the OPEN game that would win using T with respect to query where ID processed q_1. It follows that ID had at that time already processed U.

Next, consider the alternative case, where ID^* weakly processed an update U after joining the group, through query q_i^*, $i \geq 2$. By definition, there exists a user ID' and a series of efficiently computable round messages $\mathfrak{M}'_1, \ldots, \mathfrak{M}'_t$ such that ID', with respect to their state γ' at time q'_{ID^*}, would process (and accept) all $\mathfrak{M}'_i$ in order; and such that ID' would explicitly process U by processing $\mathfrak{M}'_t$ and arrive at a state consistent with $\gamma_{\mathsf{ID}^*}^{q_i^*}$. Here, recall q'_{ID^*} is either $q'_{\mathsf{ID}^*} = \mathsf{CGKA.Upd}(\gamma')$ the last update from ID' weakly processed by ID^* or, if such update does not exist, the query in which ID' processed their welcome message into the group, initializing state γ'. Now, note that since the state of ID at q_i is consistent with that of ID^* at q_i^*, then so it is with the state of ID' after processing $\mathfrak{M}'_t$. We will show that there must exists a similar sequence of round messages from q'_{ID}, the query where ID' last issued an update weakly processed by ID or, if such update does not exist, the query in which ID' processed their welcome message into the group. Assume first that q'_{ID^*} is the query where ID' processed their welcome message. In that case, $q'_{\mathsf{ID}^*} \leq q'_{\mathsf{ID}}$, so (a subsequence of) the $\mathfrak{M}'_i$ satisfy the definition with respect to ID, meaning ID weakly processed U in q_i. If that is not the case, ID^* must have weakly processed query q'_{ID^*} at or after joining. If ID^* did so at the moment of joining, then we know that ID must have also weakly processed said query, so $q'_{\mathsf{ID}^*} \leq q'_{\mathsf{ID}}$ (in fact, equality holds here) so, again, the $\mathfrak{M}'_i$ satisfy the definition with respect to ID. Last, assume ID^* weakly processed q'_{ID^*} in q_i^*, with $i \geq 2$. But then, we know that the state of ID' at q'_{ID^*} is consistent with ID^*'s state, $\gamma_{\mathsf{ID}^*}^{q^*_{i-1}}$, and therefore also with ID's, $\gamma_{\mathsf{ID}}^{q_{i-1}}$. Thus the $\mathfrak{M}'_i$, preceded with whatever round messages ID' processed between q'_{ID} and q'_{ID^*} satisfy the definition with respect to ID^*.

Finally, this last argument shows, in fact, that if ID^* weakly processes U in q_i^*, for $i \geq 2$, then so does ID in q_i. This completes the last part of the lemma statement, regarding the ordering of weakly processed queries (and, in particular, implies that $q'_{\mathsf{ID}^*} = q'_{\mathsf{ID}}$ in the last case above). □

Proof of Lemma 5. There are three actions that trigger ID to add a key to their local state: processing an add operation, and adding the new InitKey of the added party; explicitly processing an update operation; and processing a remove operation, where they add extra keys to the resolution of their copath. Statement 1 is trivially true for keys added through the first or second action. We will first show that statement 2 is also true for these keys.

Consider now the case where ID added pk_v to their state by processing an add operation, adding party ID_A, and assume for contradiction that they weakly processed an update U' affecting v in query $q' \in \{q+1, \ldots, q^*\}$. This update must be authored by ID_A, but since their leaf v is in the state of ID, they must have then explicitly processed it, which would mean v would no longer be in ID's state, a contradiction.

Now, consider the case where pk_v was introduced in ID's state by explicitly processing Update U_v in q and, again, assume for contradiction they weakly processed some U' in $q' \in [q+1, \ldots, q^*]$ affecting v. For pk_v to still be part of ID's state at q^*, they must have not explicitly processed any Update coming from the subtree under v. However, by the definition of weakly processing, there must be a user $\hat{\mathsf{ID}}$ under v's subtree whose Update was explicitly processed by ID in q', which is a contradiction.

Thus, it remains to show the lemma for keys pk_v that were added by ID to their state by processing a remove operation. This is, however, a straight forward application of Lemma 7. Observe that pk_v must have then been part of a tree sent by the server which satisfied all conditions in Def. 12, because ID accepted them round message containing it. Indeed, the checks a party does on this tree before accepting the round message that contains it ensure that it satisfies the requirements estipulated in the lemma: parent hash verifies and, by virtue of consisting of the path(s) and co-path resolution(s) of one (or more) user(s), it will have no blank leaves, indegree two except at the leaves, and contain a leaf with a public verification key. The fact that all node states correspond to previous queries follows, in turn, by the fact the adversary is not allowed to corrupt signatures. Moreover, its label must match that of the intersection of the removed user(s)'s path(s) and $\mathcal{P}(\mathsf{ID})$. Thus, it must be that all keys in it correspond to operations ID had already weakly processed. If that was not the case, we could build an adversary for the OPEN game which would win using the mentioned tree. with respect to ID and the tree(s) received from the server as part of the round message.

It remains to show that statement 2 is true for v, i.e. that ID cannot have weakly processed an update U^+ coming from a user in the subtree under v, after weakly processing U. However, note that if this is the case, ID must have explicitly processed an update between q and q^*, and later process another round message again making them weakly process U twice, following the same argument as in the proof of Lemma 7, which is a contradiction to Corollary 3. □

Acknowledgements

We thank Marta Mularczyk and Yiannis Tselekounis for their very helpful feedback on an earlier draft of this paper. Benedikt and Krzysztof were funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT). Karen is supported in part by ERC CoG grant 724307. Karen and Michael conducted parts of this work at IST Austria, where they were funded by the ERC under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT). Guillermo is funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No.665385.

References

[1] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, Aug. 2020.

[2] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1463–1483. ACM, 2021.

[3] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk. Continuous group key agreement with active security. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, Nov. 2020.

[4] J. Alwen, D. Hartmann, E. Kiltz, and M. Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456, 2021. `https://eprint.iacr.org/2021/1456`.

[5] J. Alwen, D. Jost, and M. Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Report 2020/1327, 2020. `https://eprint.iacr.org/2020/1327`.

[6] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-11, Internet Engineering Task Force, Dec. 2020. Work in Progress.

[7] K. Bhargavan, R. Barnes, and E. Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. `https://mailarchive.ietf.org/arch/attach/mls/pdf1XUH6o.pdf`, May 2018.

[8] K. Bhargavan, B. Beurdouche, and P. Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, Dec. 2019.

[9] A. Bienstock, Y. Dodis, and P. Rösler. On the price of concurrency in group ratcheting protocols. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, Nov. 2020.

[10] E. Bresson, O. Chevassut, and D. Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 321–336. Springer, Heidelberg, Apr. / May 2002.

[11] C. Brzuska, E. Cornelissen, and K. Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. `https://eprint.iacr.org/2021/137`.

[12] M. Burmester and Y. Desmedt. A secure and efficient conference key distribution system (extended abstract). In A. D. Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 275–286. Springer, Heidelberg, May 1995.

[13] R. Canetti, J. A. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, Mar. 21–25, 1999.

[14] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, Oct. 2018.

[15] C. Cremers, B. Hale, and K. Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1847–1864. USENIX Association, 2021.

[16] J. Devigne, C. Duguey, and P. Fouque. MLS group messaging: How zero-knowledge can secure updates. In E. Bertino, H. Shulman, and M. Waidner, editors, *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 587–607. Springer, 2021.

[17] R. Dutta and R. Barua. Dynamic group key agreement in tree-based setting. In C. Boyd and J. M. G. Nieto, editors, *ACISP 05*, volume 3574 of *LNCS*, pages 101–112. Springer, Heidelberg, July 2005.

[18] K. Emura, K. Kajita, R. Nojima, K. Ogawa, and G. Ohtake. Membership privacy for asynchronous group messaging. Cryptology ePrint Archive, Report 2022/046, 2022. `https://eprint.iacr.org/2022/046`.

[19] K. Hashimoto, S. Katsumata, E. Postlethwaite, T. Prest, and B. Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient pkes. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1441–1462, New York, NY, USA, 2021. Association for Computing Machinery.

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

[21] I. Ingemarsson, D. Tang, and C. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, 1982.

[22] S. Katsumata, K. Kwiatkowski, F. Pintore, and T. Prest. Scalable ciphertext compression techniques for post-quantum KEMs and their applications. In S. Moriai and H. Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 289–320. Springer, Heidelberg, Dec. 2020.

[23] K. Klein, G. Pascual-Perez, M. Walter, C. Kamath, M. Capretto, M. Cueto, I. Markov, M. Yeo, J. Alwen, and K. Pietrzak. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[24] Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.

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

[26] T. Perrin and M. Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.

[27] D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet Draft, Sept. 1998. `http://www.ietf.org/ID.html`.

[28] M. Weidner, M. Kleppmann, D. Hugenroth, and A. R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2024–2045, New York, NY, USA, 2021. Association for Computing Machinery.

[29] C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM*, pages 68–79, Vancouver, BC, Canada, Aug. 31 – Sept. 4, 1998.

A Glossary

Continuous Group-key Agreement		
CGKA	Asynchronous Continuous Group Key Agreement	Def. 1
CGKA.Gen	Key generation.	Def. 1
CGKA.Init	Initialization of a group.	Def. 1
CGKA.Add	Adding a member.	Def. 1
CGKA.Rem	Removing a member.	Def. 1
CGKA.Upd	Updating.	Def. 1
CGKA.Dlv	Server collects protocol messages and delivers a round message.	Def. 1
CGKA.Proc	Processing a round message.	Def. 1
CGKA.Key	Extraction of group key.	Def. 1
Ratchet Trees		
$\mathfrak{T}^n = (V_{\mathfrak{T}}^n, E_{\mathfrak{T}}^n)$	Ratchet tree associated to round n	p.8
$\gamma(v)$	State associated to a node v which contains:	p.8
$(\mathsf{sk}_v, \mathsf{pk}_v)$	PKE associated to v,	
$\mathrm{PK}^{\mathrm{pr}}$	predecessor keys associated to v,	
h_v	parent hash associated to v,	
ID_v	identifier corresponding to a party,	
σ_v	signature under the private signing key of ID_v,	
$\mathcal{H}_{trans,v}$	transcript hash at the time of sampling,	
$\mathsf{confTag}_v$	confirmation tag at the time of sampling,	
$o_v = (o_1, o_2)$	hashes that are part of a Merkle tree,	
$\mathsf{Unmerged}(v)$	set of keys of unmerged leaves belonging to the subtree rooted at v,	
$(\mathsf{ssk}_v, \mathsf{svk}_v)$	signing and verification key-pair if v is a leaf,	
$\mathsf{Res}(v)$	Resolution of a node.	Def. 2
Re-key	Hierarchical derivation of seeds and keys along a path.	Alg. 1
CoCoA: Local state		
$\mathcal{P}(\mathsf{ID})$	$\mathcal{P}(\mathsf{ID}) = \mathsf{path}(\mathsf{ID}) \cup \mathsf{Res}(\mathsf{co\text{-}path}(\mathsf{ID}))$.	p.13
γ	Local state stored by a user which contains:	Tab. 2
$\gamma.\mathsf{ID}$	identifier for the party,	
$\gamma.G$	set of current group members,	
$\gamma.\mathsf{ssk}$	the party's signing key,	
$\gamma(v)$	for each $v \in \mathsf{path}(\gamma.\mathsf{ID})$,	
${}^{p}\gamma(v)$	public part for each $v \in \mathsf{Res}(\mathsf{co\text{-}path}(\gamma.\mathsf{ID}))$,	
$\gamma.\mathcal{H}_{trans}$	current value of the transcript hash,	
$\gamma.\mathsf{epochSecret}(n)$	epoch secret,	
$\gamma.\mathsf{appSecret}(n)$	application secret,	
$\gamma.\mathsf{confKey}(n)$	confirmation key,	
$\gamma.\mathsf{initSec}(n)$	initialization secret,	
γ'	pending state.	
ℓ	Map that takes nodes in $\mathfrak{T}^n$ to labels.	Def. 3
$\mathcal{H}_{round}(n)$	Round hash.	Def. 3
CoCoA: Parent hash		
$(\mathsf{Sig.Sig}, \mathsf{Sig.Ver})$	Signature scheme.	p.18
PHash.Sig	Computation of new parent hash values and signatures.	p.18
PHash.Ver	Verification of the authenticity of a tree.	p.18
CoCoA: User generated messages		
$(\mathsf{MAC.Tag}, \mathsf{MAC.Ver})$	MAC.	Tab. 3
Init messages		Tab. 3
ID	The identifier of the group creator.	
G	List of group members.	
$P = (p_j = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \sigma_j))$	Public states for every $v \in \mathsf{path}(\mathsf{ID})$.	
$S = (s_{j,l} = (e_{j,l}, \sigma^s_{j,l}))_{j,l}$	Encryptions of the seeds.	
$\sigma^s_{j,l}$	Signature under $\mathsf{ssk}_{\mathsf{ID}}$ of $e_{j,l}$.	
Update messages		Tab. 3
ID	The identifier of the updating user.	
c	Number of updates by ID since last processed round message.	
$P = (\mathsf{pk}_j, h_j, \mathcal{H}_{trans}, \mathsf{confTag}, \sigma_j)_j$	New public states for every $v \in \mathsf{path}(\mathsf{ID})$.	
confTag	Confirmation tag under $\gamma.\mathsf{confKey}$ of $\gamma.\mathcal{H}_{trans}$	
σ_j	Signature under $\mathsf{ssk}_{\mathsf{ID}}$ of $(\mathsf{pk}_j, h_j, \mathsf{confTag},)$.	

$S = (e_{j,l}, \mathsf{confTag}, \sigma^s_{j,k})_{j,l}$	Encryptions of the new seeds	
$\sigma^s_{j,k}$	Signature under $\mathsf{ssk}_{\mathsf{ID}}$ of $(e_{j,l}, \mathsf{confTag})$.	
Remove messages		Tab. 3
ID	The identifier of the removed/added user.	
$\mathsf{confTag}$	Confirmation tag under $\gamma.\mathsf{confKey}$ of $\gamma.\mathcal{H}_{trans}$	
σ	Signature of $\mathsf{confTag}$ under sender's signing key.	
Add messages		Tab. 3
ID	The identifier of the removed/added user.	
$\mathsf{confTag}$	Confirmation tag under $\gamma.\mathsf{confKey}$ of $\gamma.\mathcal{H}_{trans}$	
σ	Signature of $\mathsf{confTag}$ under sender's signing key.	
$\mathsf{pk}_{\mathsf{ID}}$	Public key of ID.	
$\mathcal{W}$	Welcome message sent next round.	
CoCoA: Round messages		
$\mathfrak{M}_i$	Round message sent to party ID_i which contains:	Tab. 4
R	vector of Remove messages,	
A	vector of Add messages,	
c_i	update counter, if ID_i updated, 0 otherwise,	
$\mathcal{H}_{round}$	round hash value,	
O_i	openings to verify $\mathcal{H}_{round}$,	
$\gamma(v)$	public states of nodes needed to update $\mathcal{P}(\mathsf{ID}_i)$,	
$u_v = (\mathsf{ID}, p, s)$	public state of each relevant updated v,	
W	welcome message (for new parties),	
$\mathfrak{T}$	entire ratchet tree (for new parties).	
CoCoA: CGKA security		
C	Challenger in the security game.	Def. 5
A	Adversary in the security game.	Def. 5
create-group(G)	C initializes the group.	Def. 5
add-user$(\mathsf{ID}, \mathsf{ID}')$	ID adds ID' to the group.	Def. 5
remove-user$(\mathsf{ID}, \mathsf{ID}')$	ID removes ID' from the group.	Def. 5
update(ID)	ID refreshes its current local state.	Def. 5
process$(\mathfrak{M}, \mathsf{ID})$	$\mathfrak{M}$ is sent to ID which immediately processes it.	Def. 5
start-corrupt(ID)	γ and the randomness of ID except for $\mathsf{ssk}_{\mathsf{ID}}$ leak to A.	Def. 5
end-corrupt(ID)	End of the leakage of ID's γ and randomness to A.	Def. 5
challenge(q^*)	A chooses a query q^* for the challenge.	Def. 5
CoCoA: Round hash		
$\mathsf{extract}(\mathfrak{M}, \gamma)$	Extracts a list of updates, removes, adds and indices.	p.15
$\mathsf{update\text{-}info}(U, R, A, \prec, \gamma, v)$	Outputs the new public key pk_v at node v.	p.15
$\mathsf{retrieve\text{-}labels}(\gamma, v, O)$	O is a vector of tuples of the form (v_i, h, h'). It outputs (v, h_l, h_r) if $\exists!(v_i, h_l, h_r) \in O$ with $v_i = v$, $h_l = h$ and $h_r = h'$, and $\perp$ otherwise.	p.15
$\mathsf{openRH}(\mathsf{ID}, \mathfrak{T}_\ell)$	Outputs a vector of hash values, corresponding to the labels of nodes that are parents of a node in $\mathcal{P}(\mathsf{ID})$ but are not in $\mathcal{P}(\mathsf{ID})$.	Alg. 2
$\mathsf{verifyRH}(\gamma, \mathfrak{M}, O)$	Outputs 0 or 1 depending on whether the round hash included in the round message is equal to the one computed using the updated ratchet tree.	Alg. 3
CoCoA: Hash functions		
H_1	Hash function used to obtain new seeds.	Alg. 1
H_2	Hash function used to obtain PKE keys.	Alg. 1
H_3	Hash function used to define the map ℓ and the round hash.	Def. 3
H_4	Hash function used to define the parent hash.	p.18
H_5	Hash function used to define the key schedule.	p.14