CoCoA: Concurrent Continuous Group Key Agreement

Joël Alwen[1], Benedikt Auerbach[2][0000−0002−7553−6606], Miguel Cueto Noval[2], Karen Klein[3], Guillermo Pascual-Perez[2][0000−0001−8630−415X], Krzyzstof 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

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].

* Benedikt Auerbach and Krzysztof Pietrzak have received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT); Karen Klein was supported in part by ERC CoG grant 724307 and conducted part of this work at IST Austria, funded by the ERC under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT); Guillermo Pascual-Perez was funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No.665385; Michael Walter conducted part of this work at IST Austria, funded by the ERC under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT).

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.

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 [28]. However, this approach seems to unavoidably result in the complexity of (at least some critical) operations scaling linearly in the group size n.[5] 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. [16] 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).[6] 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 [4]. Thus they present a very useful subject for research in the area.

[5] In fact, this holds true even for the few 1st generation E2E protocols designed from the ground up with groups in mind [22].

[6] Also referred to as *Group Ratcheting* [11] or *Continuous Group Key Distribution* [13].

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) [5]. 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. [7] 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 [16, 9, 3, 25, 5, 7] 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 [25, 3, 9, 16] 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 [8]). 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).

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 star-

[7] 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 [3]. 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.

vation 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 [5, 7] 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. [11] 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.[8] 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 (full version in [2]) 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

[8] For the lower bound, [11] 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 [11], but we remark, that our protocol and the TreeKEM variants considered in this work fall into this category.

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 [11] 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 [9], 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 [11], 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.

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 [25] 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 was initiated by the ART protocol [16], based on which the first version of MLS [8] was built shortly before transitioning to TreeKEM [9]. TreeKEM has since been the subject of several security analysis including [3, 4, 13, 10, 17]. 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.

Several works have studied CGKA's supporting varying degrees of concurrent operations. Weidner's Causal TreeKEM [26] explores the idea of updates *re-randomizing* key material instead of overwriting it (though it lacks forward secrecy and a complete security proof). Recently, [30] proposed a decentralized CGKA protocol; albeit with linear communication complexity. Finally, a paper by Bienstock *et al.* [11] 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.

A similar security model to the one in our paper can be found in [25]. That work was the first to require security against an *adaptive* adversary. The security model of [3] 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. A different approach to security notions was initiated in [4] 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). [5] presented the first ideal/real CGKA security notion. Their notion captures security against powerful adaptive and fully active adversaries that can corrupt parties at will and even *set* their random coins. Finally, building on that work, [7] extend their adversaries to also account for how corrupt insiders might interact with a (very weak) PKI. The formal notion was later adapted in [21, 6] 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 [10] who use an "event driven" language to define adaptive security a CGKA. The notion of server-aided CGKA was first formalized in [6]. However, the earlier work of [18] and the concurrent work of [21] both include (implicit) server-aided CGKAs as well.

The work of [24] 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 [21] 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 [17]. In [1] more efficient CGKA constructions in the multi-group setting are given. In [5] zero-knowledge proofs are used to improve the robustness of CGKA protocols. The approach was made a bit more practical in [18] by, amongst other things, introducing tailor-made ZK proofs. Very recently, [20] initiated the study of membership privacy for CGKAs.

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 [23, 14]. In contrast to CGKA, GKE protocols do not target PCS and are designed for the synchronous setting. 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) [12, 19]. Another notion very related to CGKA are Logical Key Hierarchies [29, 31, 15], introduced as a solution to very related primitive of Multicast Encryption [27]. 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 [25], 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 [25]). 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* $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$*. 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:*

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

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}[9] 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

[9] 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.

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.

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 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, but refer the reader to the full version [2] for a more in detail introductory discussion. 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 (again, more complete in [2]), and Section 3.5 formally defines the protocol procedures.

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.
$\varDelta_1 \xleftarrow{\$} \mathfrak{S}(\lambda)$            // $\lambda$ security parameter; $\mathfrak{S}(\lambda)$ seed space
$(\mathsf{sk}_1, \mathsf{pk}_1) \leftarrow \mathsf{PKE.Gen}(\mathsf{H}_2(\varDelta_1))$
for $i \leftarrow 2$ to $d$ do
    $\varDelta_i = \mathsf{H}_1(\varDelta_{i-1})$
    $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{PKE.Gen}(\mathsf{H}_2(\varDelta_i))$
$\varDelta \leftarrow (\varDelta_1, \ldots, \varDelta_d)$
$\mathsf{K} \leftarrow ((\mathsf{sk}_1, \mathsf{pk}_1), \ldots, (\mathsf{sk}_d, \mathsf{pk}_d))$
return $(\varDelta, \mathsf{K})$
```

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 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,

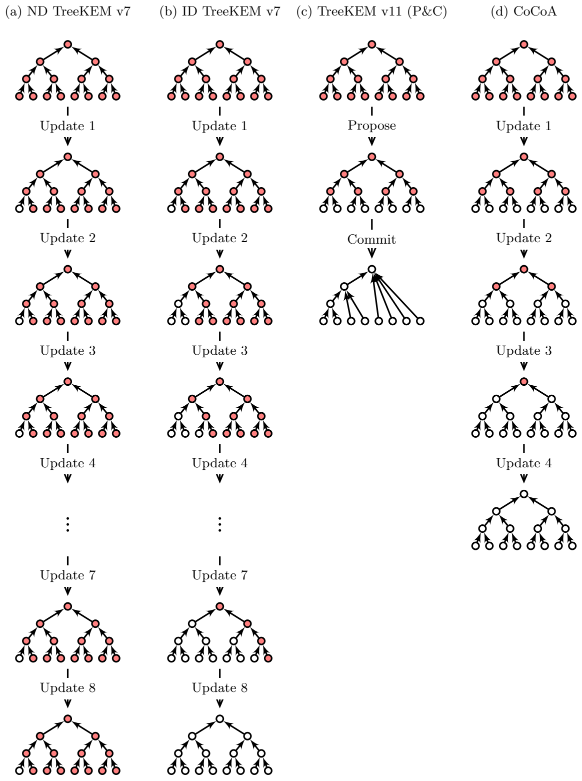

Fig. 1. Comparison of number of rounds required to recover from corruption for different TreeKEM variants, ND stands for "Naïve Delivery", ID for "Ideal Delivery". Red nodes indicate key material known to the adversary. In each round all parties (try to) update. In columns (a) and (d) update requests 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.

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

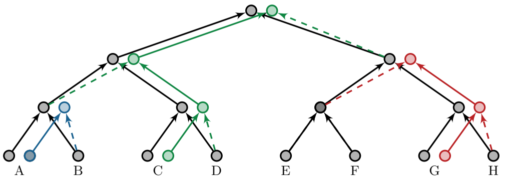

Fig. 2. Example; concurrent updates in the CoCoA protocol. The former state of the ratchet tree (black) is changed by concurrent updates of A (blue), C (green), and G (red). The ordering is $U_C \prec U_A \prec U_G$. In the updates solid edges correspond to seeds obtained by hashing, dashed edges to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

$\gamma.\mathsf{ID}$	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 1. User's local state γ.

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.

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, each user stores a local state γ, described in Table 1, 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:

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

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;[10] 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 ${}^{p}\gamma(v) = \mathtt{blank}$ 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:*

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

The round hash $\mathcal{H}_{round}(n)$ of n is defined to be $\mathcal{H}_{round}(n) = \mathsf{H}_3\left(\ell(v_{root}), R, A\right)$.

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, 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

[10] 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.

$\mathcal{P}(\mathsf{ID})$. Given these values, the user is able to verify the received round message by running $\mathsf{verifyRH}$, which recomputes $\mathcal{H}_{round}(n)$ with respect to their updated ratchet tree and compares it to the round hash provided by the server. For a formal description of both algorithms we refer to the full version of this work [2].

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

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

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.

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.* [7]. 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 $\mathsf{child}(v)$;[11] i.e. if the Update sampling

[11] 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.

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:

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

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, $h_i = (h_{1,i}, h_{2,i})$, and $m = (\mathsf{pk}_{v_i}, \mathrm{PK}^{\mathrm{pr}}_{v_i}, (h_{1,i}, h_{2,i}), \mathcal{H}_{trans}, \mathsf{confTag})$

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.[12] 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:

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

[12] 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.

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 $0 \leftarrow \mathsf{PHash.Ver}(T)$.[13]

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

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 $\prec$.

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 $\sigma^s_{j,l} = \mathsf{Sig.Sig}_{\mathsf{ssk}_i}(e_{j,l})$. 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 $s_{j,l} = (e_{j,l}, \sigma^s_{j,l})$, 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 outside their path in their local tree copy to be blank.

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

[13] 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.

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 $v_j \in \mathsf{path}(v)$;[14] 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 $\varDelta$.

Remove. To remove party ID_j, ID_i sends out a $remove(\mathsf{ID}_j)$ 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 $add(\mathsf{ID}_j, \mathsf{pk}, \mathsf{svk})$ 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[15] 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, allowing them to initialize their state and key schedule, as well as checking the correctness of the tree sent by the server. Moreover, some user must send an Update during that round, thus creating a new application secret.

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,

[14] 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.

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

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[16]; 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 in the resolution of the co-path of U_a's author. Last, the server also includes a counter c_i, equal to that of ID_i's update included in $\mathfrak{M}_i$ if there is one, and 0 otherwise. Finally, for each newly-added ID_i, the round message $\mathfrak{M}_i$ additionally contains the corresponding $\mathcal{W}$, 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, they check that $\mathsf{MAC.Ver}(\gamma.\mathsf{confKey}, \mathsf{confTag}) = 1$ for every update, remove and add; 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. [17] If these checks pass, they copy

[16] 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

[17] 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.

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

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 [3, 8, 25, 26]	n	$n^2 \log(n)$	$n \log(n)$	$n \log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [8]	2	n^2	n	n	n	n
Bienstock et al. [11]	2	n^2	n	$n^\dagger$	$\log(n)^\dagger$	$\log(n)$
Bidirectional channels [30]	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 [3, 8, 25, 26]	t	$t^2 \log(n)$	$t \log(n)$	$t \log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [8]	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. [11]	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 [30]	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 2. 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 [25], would lead to future worst-case update cost of n (case (a)) and $t(1 + \log(n/t))$ (case (b)), respectively.

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. A user with local state γ fetches $K = \gamma.\mathsf{appSecret}$.

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 2.

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. Among the different settings one could compare, we restrict our view to the following, quite natural in our opinion.

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 2 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 sufficientthat 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[18]. 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 $t \log(n)^2$

[18] 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.

(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.[19] 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. In Table 2 we contrast CoCoA to other CGKA schemes. For a more detailed breakdown of theses values we refer to the full version of this work [2]. The first class of considered schemes are ratchet-tree based schemes that do not rely on the P&C framework, as TreeKEM v7 and earlier versions [8], rTreeKEM [3], TTKEM [25], and Causal TreeKEM [26][20]. Further, with TreeKEM v8 [8] and later versions and the protocol by Bienstock et al. [11] we consider ratchet-tree based protocols following the P&C paradigm. Finally, we give values for the protocol by Weidner et al. [30] based on bidirectional channels. We point out that this work targets a different network model and has thus a different focus than ours.

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.

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

[20] 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.

The trade-off with TreeKEM versions that precede the P&C 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 P&C 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 sender communication cost. We point out that the P&C 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 has the advantage, over all versions of TreeKEM, 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 P&C paradigm, since it requires slightly larger sender communication overall. However, as we show in Section 5, 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-P&C 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.* In the case of concurrent updates, on the other hand, we show that every corrupted party sending logarithmically many updates is sufficient.

5.1 Security Model and Safe Predicate

To analyze the security of CoCoA, we essentially use the security model from [25], 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 [25] 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 [25]. 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 [25] 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 [25] (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 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* (Q, ϵ, t)*-CGKA-secure if for any adversary* A *making at most* Q

queries of the form **add-user**$(\cdot,\cdot)$*,* **remove-user**$(\cdot,\cdot)$*, or* **update**$(\cdot)$ *and running in time t it holds*

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

In contrast to the security definition of [25], 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 [25]. 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* $\mathsf{a}^i_{\mathsf{ID}} :=$ **update**(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* $\mathsf{a}^-_{\mathsf{ID}} :=$ **update**(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* $\mathsf{a}^+_{\mathsf{ID}} :=$ **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 [25], 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 $\mathsf{a}^* \in \{\mathbf{update}(\mathsf{ID}^*), \mathbf{create\text{-}group}(\mathsf{ID}^*, \cdot)\}$ 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, as we show with an example given in the full version of this work [2].

5.2 Security of CoCoA

Regarding the security of CoCoA we obtain the following.

Theorem 1. *If the encryption scheme used in CoCoA is $(\epsilon_{\mathrm{Enc}}, t)$-IND-CPA-secure, the signature scheme is $(\epsilon_{\mathrm{Sig}}, t, (n+2\log(n))Q)$-UF-CMA-secure, and the used hash functions are modeled as random oracles, then CoCoA is $(Q, \mathcal{O}(\epsilon_{\mathrm{Enc}} \cdot (nQ)^2 + \epsilon_{\mathrm{Sig}} \cdot n), t)$-CGKA-secure.*

Due to space limitations we only give a high level overview on the proof, and refer to the full version of this work [2] for the formal proof. To prove security of CoCoA, we follow the approach of [25] 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. To argue security, none of the secret keys in the challenge graph must be leaked to the adversary by corruption. We prove that this is indeed the case for CoCoA if the safe predicate is satisfied. Our proof follows the ideas from [25], 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 [25] 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, 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. Further, we show that users that do not agree on the same history of weakly processed updates transition to inconsistent states. 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.

Acknowledgements

We thank Marta Mularczyk and Yiannis Tselekounis for their very helpful feedback on an earlier draft of this paper.

References

1. J. Alwen, B. Auerbach, M. A. Baig, M. Cueto Noval, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter. Grafting key trees: Efficient key management for overlapping groups. In K. Nissim and B. Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 222–253. Springer, Heidelberg, Nov. 2021.

2. J. Alwen, B. Auerbach, M. Cueto, K. Klein, G. Pascual-Perez, K. Pietrzak, and M. Walter. Cocoa: Concurrent continuous group key agreement. Cryptology ePrint Archive, Report 2022/xxxx, 2022. `https://ia.cr/2022/xxxx`.
3. 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.
4. J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, Nov. 2021.
5. 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.
6. 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`.
7. 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`.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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`.
14. 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.
15. 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.
16. 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.
17. 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, *USENIX Security 2021*, pages 1847–1864. USENIX Association, Aug. 2021.

18. 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.
19. 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.
20. 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.
21. 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 G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, Nov. 2021.
22. 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.
23. I. Ingemarsson, D. Tang, and C. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, 1982.
24. 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.
25. 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*, pages 268–284. IEEE Computer Society Press, May 2021.
26. Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.
27. 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.
28. T. Perrin and M. Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.
29. 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.
30. M. Weidner, M. Kleppmann, D. Hugenroth, and A. R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In G. Vigna and E. Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, Nov. 2021.
31. 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.