DeCAF: Decentralizable CGKA with Fast Healing

Joël Alwen[1], Benedikt Auerbach[2], Miguel Cueto Noval[2], Karen Klein[3], Guillermo Pascual-Perez[2(✉)], and Krzyzstof Pietrzak[2]

[1] AWS Wickr, New York, USA
alwenjo@amazon.com
[2] ISTA, Klosterneuburg, Austria
{bauerbac,mcuetono,gpascual,pietrzak}@ist.ac.at
[3] ETH Zurich, Zürich, Switzerland
karen.klein@inf.ethz.ch

Abstract. Continuous group key agreement (CGKA) allows a group of users to maintain a continuously updated shared key in an asynchronous setting where parties only come online sporadically and their messages are relayed by an untrusted server. CGKA captures the basic primitive underlying group messaging schemes.

Current solutions including TreeKEM ("Messaging Layer Security" (MLS) IETF RFC 9420) cannot handle concurrent requests while retaining low communication complexity. The exception being CoCoA, which is concurrent while having extremely low communication complexity (in groups of size n and for m concurrent updates the communication per user is $\log(n)$, i.e., independent of m). The main downside of CoCoA is that in groups of size n, users might have to do up to $\log(n)$ update requests to the server to ensure their (potentially corrupted) key material has been refreshed.

In this work we present a "fast healing" concurrent CGKA protocol, named DeCAF, where users will heal after at most $\log(t)$ requests, with t being the number of corrupted users. While also suitable for the standard central-server setting, our protocol is particularly interesting for realizing decentralized group messaging, where protocol messages (add, remove, update) are being posted on some append-only data structure rather than sent to a server. In this setting, concurrency is crucial once the rate of requests exceeds, say, the rate at which new blocks are added to a blockchain.

In the central-server setting, CoCoA (the only alternative with concurrency, sub-linear communication and basic post-compromise security) enjoys much lower download communication. However, in the decentralized setting – where there is no server which can craft specific messages for different users to reduce their download communication – our protocol significantly outperforms CoCoA. DeCAF heals in fewer epochs ($\log(t)$ vs. $\log(n)$) while incurring a similar per epoch per user communication cost.

C. Galdi and D. H. Phan (Eds.): SCN 2024, LNCS 14974, pp. 294–313, 2024.
https://doi.org/10.1007/978-3-031-71073-5_14

1 Introduction

1.1 (Group) Messaging

Popular group messaging applications, like Signal [29], work in an asynchronous setting, where users need to be online only occasionally and their messages are relayed by an untrusted server. The underlying ratcheting protocol provides strong security; in particular, forward secrecy (FS), post-compromise security (PCS), and end-to-end encryption, which is important as conversations can last for years at a time. FS ensures that messages sent in the past remain secure if a user gets compromised, while PCS allows for the keys of a user to be refreshed after compromise ensuring future messages are secure again. It is a challenging problem, and the focus of recent IETF standard on "Messaging Layer Security" (MLS) [13], to efficiently scale messaging applications to larger groups without giving up on the strong security properties provided by two-party protocols like the Double Ratchet [29].

1.2 CGKA

Continuous group key agreement (CGKA) was identified as the key primitive underlying group messaging [4,5]. Accordingly, it has recently seen a lot of attention with works giving CGKA instantiations [3,4,7,9,14,19,21,23,24,27,28], analyzing the security of constructions [6,8,15,18,31], lower bounds [1,10,16,17], or targeting additional properties like CGKA for multiple groups [1,22], metadata-hiding [25], or tools for cryptographic administration of group membership [11]. See [30] for a SoK of security definitions for group key agreement.

CGKA allows a set of users to maintain a shared key in an asynchronous setting where protocol messages are relayed by an untrusted server. The operations CGKA must support are the users' addition and removal, and a key update functionality by which a user can rotate its secret key material so as to achieve forward secrecy and post-compromise security. Most of the so far proposed CGKA schemes with this motivation, beginning with ART [19] and TreeKEM [27], arrange users' keys in a binary tree structure. In this so-called ratchet tree, each node corresponds to a public/secret key pair. Leaves are identified with users who hold the secret keys of all nodes from their leaf to the root. The root secret key—known to all users—is used to define the group key which secures messages sent to the group. We think of the edges of the tree as being directed from the leaves to the root, and an edge $(pk, sk) \rightarrow (pk', sk')$ basically means that sk' is encrypted under pk in a ciphertext that can be retrieved from the delivery server. Thus, the user at a leaf with key-pair (pk, sk) will be able to retrieve all the secret keys on the path from its leaf to the root. The reason to use trees rather than, say, pairwise channels for maintaining the keys, is that in groups of size n, each user only has to send $\log(n)$ ciphertexts in order to update all secret keys they know (as opposed having to rekey $n-1$ independent channels). Concretely, as illustrated in Fig. 1 (tree on the top left, ignoring the blue nodes for now), if a user A wants to update, they resample the keys on their

path (the red path in the figure), encrypt the fresh keys to the nodes on their co-path (the red edges), and send these ciphertexts to the server. Other group members can fetch those ciphertexts and update their local states to reflect the new keys. An important "invariant" property of these tree-based schemes is that a user will always only learn the secret-key for nodes on their path to the root (which is why it is sufficient to replace just the keys on the user's path to root to achieve FS and PCS for that user).

Concurrent Updates. While updates in the initial versions of TreeKEM only need $\log(n)$ communication, they are inherently sequential: a user can only send an update request after processing the previous one. If two (or more) users A and B send an update request each rekeying their full paths to the server for the same previous ratchet tree state (as shown on the left in Fig. 1), the server will simply reject all but one of the requests. In fact, this is true for all CGKA variants with two exceptions discussed below.

Recent versions of TreeKEM do allow for a different type of concurrent updates through the "Propose and Commit" framework. Here, initial users concurrently announce their update operations in a first round, generating new key material *only for their own leaf*. Then, in a second round, one party "commits" the updates, along the way refreshing their own full path. But to ensure PCS, *all* nodes not on the paths of the initial users have their old keys replaced (or removed). TreeKEM and similar protocols address this by setting those nodes to be *blank*. That is, these node are effectively removed from the tree. Instead, each blank node's parent node now has edges to the blanked nodes children (if the child is blanked, then to its grandchild, etc.). Figure 1 (right side) shows the tree we get if A commits to an update proposal by B in this way. Note that the more concurrent updates, the more blanking ruins the tree structure, and as a consequence future operations become more expensive e.g., to commit A must send 4 ciphertexts before blanking B, but 6 after. In general the cost can grow from $\log(n)$ to n. If the group members want communication efficiency, they will have to commit to as few updates as possible at a time, relying instead on sequential commits to refresh keys. That means concurrency is not possible anymore, as commits need to be totally ordered, and the issue outlined above returns.

Causal TreeKEM. The first CGKA protocol supporting concurrent updates was Causal TreeKEM [28]. This protocol builds on a public key encryption primitive allowing for keys to be combined in a commutative way. This way, updates will no longer overwrite the previous key, but instead update it by combining the fresh key with the existing one. Since this combining process is commutative, several updates can be merged at the same time, without regard for the order in which users received them.

CoCoA. The CGKA scheme CoCoA [3] processes concurrent update proposals in a "greedy" manner and simply accepts as many keys in a concurrent proposal as possible. As illustrated in Fig. 1, fresh keys from concurrent updates

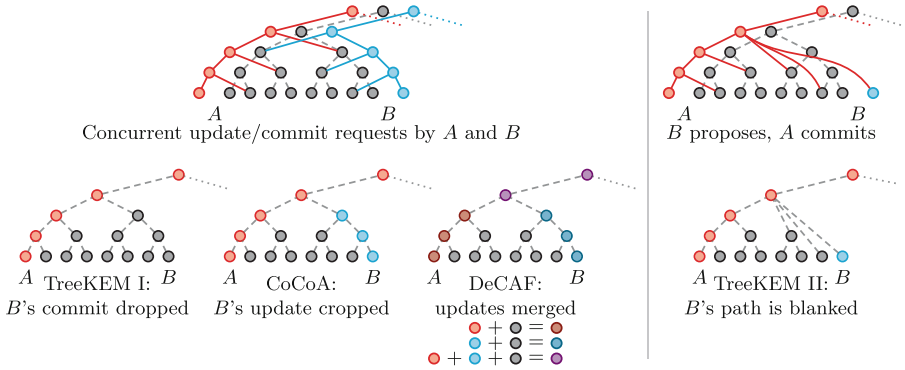

Fig. 1. (left): Illustration of how TreeKEM, CoCoA, and DeCAF handle a concurrent update by parties A and B who want to replace their (potentially compromised) keys. TreeKEM I refers to the conservative approach where users commit one at a time. In DeCAF instead of replacing old keys, the new key-material is merged with the existing one. (right): An illustration of blanking used to commit an update proposal (removing B would be similar, with their leaf node blanked instead.) (Color figure online)

are accepted, and if there is a conflict as two updates want to replace the same node, one of the two updates is rejected *from this point upwards*. While this process does not guarantee that the key is safe after every compromised party updated,[1] somewhat surprisingly [3] proves that the tree does heal after every party updated $\log(n)$ times in the worst case.

Moreover, CoCoA enjoys very low communication complexity, as each party must only download at most $\log(n)$ ciphertexts to process each set of concurrent updates. Note that, this is independent of both the number m of parties that update in this epoch, which can be as large as $m = n$, as well as the number t of corruptions, which can be as small as $t = 1$. For this to be theoretically possible, the untrusted server must be more sophisticated than just relaying every protocol message it gets to all users in the group. Instead, it only sends a subset of the ciphertexts to each user based on their position in the tree and some commitment to its actions, allowing users to check if they received consistent messages.

Server- and Blockchain-Aided CGKA. In order to distribute protocol messages among the members of the group, CGKA protocols typically rely on an untrusted server. Most CGKA protocols like TreeKEM [13], rTreeKEM [4], and Tainted TreeKEM [27] require a simple relay server. CoCoA, however, is a server-aided CGKA protocol, a primitive formally defined in [7], and where the server is expected to do non-trivial computation and provide users with personalized packages. To achieve end-to-end security the server is untrusted. Despite this, reliance on the server can still be problematic. For example, it allows it to reject

[1] In the example from Fig. 1, if B was compromised, after the update, the two topmost red nodes would still be compromised, as their keys were encrypted to compromised keys.

protocol messages by a particular user, thus preventing them from healing. Or to selectively forward messages to only part of the users, leading to a group split.

Note that these issues could be amended by replacing the server with a decentralized solution, an example of which would be a blockchain. Throughout the paper we will use the term blockchain for convenience to refer to any append-only data structure with the property that when the data is distributed among multiple nodes there is a consensus mechanism that guarantees that the data is arranged into blocks with a total ordering on these that all nodes agree on. New data can be added by making use of a peer-to-peer network or any other suitable type of channels. The use of such an append-only structure (permissioned or permissionless) allows us to realize group messaging which enjoys the same robustness and security guarantees as the underlying structure. More concretely, instead of sending their CGKA protocol messages (update/add/remove) to the server, the users would post them on the append-only ledger. Only the key-management must be on-chain, text messages (encrypted under the current group key) can be gossiped or shared on a public bulletin board.

Note that any CGKA in the classical setting can be "compiled" to the blockchain setting: in the latter, the block producer simply emulates the server to compile the protocol messages that would be broadcast in the classical setting, and adds this message to the block. In the case of server-aided CGKA the users, after downloading all protocol messages stored on chain, can simply locally emulate the computation that would be done by the smart server. Note that this potentially increases the download communication-complexity, as the users no longer receive personalized packages. The opposite holds as well, any server being able to emulate the outputs of the decentralized consensus protocol.

There are at least three separate properties which are achieved in the decentralized setting, but not in the "classical" server setting. Namely (1) security against splitting attacks, (2) censorship resistance, and (3) robustness. Regarding (1), an attack which is unavoidable in the classical setting is a splitting attack, where the (corrupted) server splits the users into two or more groups, and then only relays messages within those groups, forcing parties in different groups into different and inconsistent states. With such an attack one can, for example, enforce that only a particular subset of users sees some set of messages. If the protocol messages are on a blockchain, all parties will agree on the same view, and thus this attack is prevented. With regards to (2), another attack that is unavoidable in the single server setting is the censoring of a particular party. An untrusted server can ignore messages from a party, this way e.g. preventing them from ever updating. This is severe as, should this party be corrupted, the corrupted key can be indefinitely prevented from healing. In the blockchain setting, the "liveness property" of the blockchain, in combination with the fact that our protocol allows for concurrent updates (so there are no DOS-type attacks where some parties prevent another one from updating by flooding the mempool) prevents this attack: if a user wants to update, their request will be added with high probability within a few blocks. Finally, and regarding (3), in the single server setting the group can be shut down by taking out a single server. Better

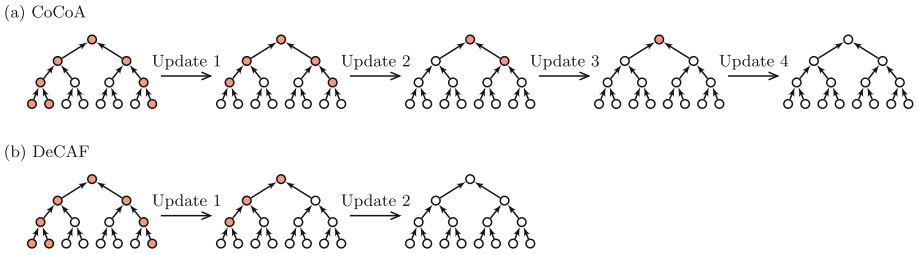

Fig. 2. Comparison of the number of epochs required to recover in CoCoA (a) and DeCAF (b) for n users, of which t are corrupted. Red nodes correspond to compromised keys. In each epoch all parties update concurrently, in CoCoA update requests are prioritized from left to right. CoCoA requires $\lceil \log(n) \rceil + 1 = 4$ epochs to recover, DeCAF only $\lfloor \log(t) \rfloor + 1 = 2$. (Color figure online)

resilience can be achieved with several servers, but then one needs to solve the state machine replication problem. This is what our protocol does if using a permissioned blockchain. With a permissionless blockchain, resilience would be even stronger.

Let us mention that in order to avoid all three issues mentioned above we need to record all the protocol messages on chain, which is probably no problem in the permissioned setting, but could be expensive in a permissionless blockchain. Permissionless blockchains like Bitcoin or Ethereum have slow block arrival rates (and even slower confirmation times), there also is a non-trivial cost to record transactions on chain. A permissioned blockchain, on the other hand, just requires a fixed small number of servers and provides the required security as long as a majority of the servers behave honestly (e.g., 3 out of 5). The cost of running such a protocol is only a small constant factor larger than just having a single server, but greatly reduces the trust required. If we are only interested in (1) and (2), but not (3), one can just post a single hash of all the messages which each block contains on chain, while the actual messages are stored off chain. This loses property (3) unless we solve the data availability problem separately[2].

1.3 Our Contribution

DeCAF. In this work we consider a new CGKA protocol, DeCAF (for DEcentralizable Continuous group key Agreement with Fast healing), that allows for concurrent updates. In DeCAF we use a key-updatable PKE scheme, and updates no longer *replace* keys, but *update* them. We show that the protocol provides forward security in the same vein as most other CGKAs (albeit slightly weaker than TreeKEM due to a potential delay until update messages are received and processed by other users), and only needs $\log(t)$ epochs to heal, with t being the number of corrupted parties. The latter point contrasts to CoCoA, where it is only guaranteed that the tree healed once each compromised party updated

[2] https://blog.polygon.technology/the-data-availability-problem-6b74b619ffcc/.

$\log(n)$ times. This difference is illustrated in Fig. 2. The root of this difference is the fact that, while in CoCoA we must drop one of two concurrent updates for the same node, in DeCAF we can perform them both, which turns out to have a significant impact on security. As we can expect t to be small compared to n (in fact, for most of the lifetime one should hope that $t = 0$), DeCAF will provide comparable security to CoCoA with fewer updates. On the downside, as in DeCAF every user must process all updates by other users (while in CoCoA at most $\log(n)$ other updates matter), the download communication (from server to users) will be larger.

The above discussion suggests a trade-off between DeCAF and CoCoA, and which one is better will depend on the context. If run using a server, CoCoA and DeCAF are incomparable; DeCAF heals faster ($\log(t)$ vs $\log(n)$ epochs) and therefore has lower sender communication, but CoCoA has lower recipient communication (since the server crafts individual messages for each party). However, in the decentralized setting (where we do not want to rely on a(n intelligent) server to relay messages), CoCoA loses its advantage in recipient communication and DeCAF is strictly better in all aspects. This is discussed in greater detail below, where we give a comparison of DeCAF to CoCoA and other concurrent CGKA protocols.

Our protocol is also similar to Causal TreeKEM [28] in some aspects, but differs largely in others. In particular, the main element in common is the above-mentioned use of updatable PKE, which is exclusive to these two protocols. While the primitive is also part of other constructions, such as rTreeKEM [4], it is employed in a very different way, as the focus is another (improved FS, in that case). However, while Causal TreeKEM requires the key-update functionality to be commutative, we do not. Furthermore, mechanisms for adding and removing parties are different, with those used by DeCAF being both simpler and in line with what is currently used by MLS, making a potential adoption by the standard much easier. Another big difference is the security guarantees provided by both protocols. Indeed, Causal TreeKEM does not consider FS and PCS is only claimed after each corrupted user issues an update in a separate epoch, thus needing n epochs to heal (in the model where corrupted users are not aware of their corruption). The latter claim lacks a formal security proof. We believe that, for static groups, Causal TreeKEM might enjoy a similar PCS guarantee to DeCAF, but this is unclear for dynamic groups.

Maintaining a Group on Chain. Given the particular suitability of DeCAF in a decentralized network, we cast it as making use of a blockchain, access to which is shared by all group members. The use of blockchain for CGKA protocols is novel as far as we know, but note that there exist previous messaging protocols making use of it, like Elixxir [20]. We stress that this is not a requirement for the protocol to run, which could instead simply rely on a central server, as discussed above. Now we explain how to make use of such a structure to maintain a group. In its simplest instantiation, a group would be initialized once some ith block B_i in the blockchain contains the welcome messages which defines a ratchet tree T_i for some group. Users in the group can post add/remove/update messages on

Table 1. Overview of the cost incurred to heal t corruptions in a group of size n (it is not known which t of the n users are corrupted). Column 'Conc.' indicates, whether the protocol allows for concurrent updates, column 'Epochs' the number of epochs required to recover from corruption, column 'Sender comm.' the cumulative uploaded communication, column 'Recipient comm.' the per-user download communication cost, and column 'Cost after rec.' the sender communication incurred by an update of a single user after the recovery process has concluded. TreeKEM I corresponds to the conservative approach of only healing by sending commits, TreeKEM II to using update proposals to heal at the expense of extra blanking. *: [17] only achieves weak PCS, obtaining PCS guarantees similar to the rest would need $\mathcal{O}(n)$ cost after healing, due to extensive tainting.

Protocol	Conc.	Epochs	Sender comm.	Recipient comm.	Cost after rec.
TreeKEM I [14]	No	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
TreeKEM II [14]	Yes	2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Causal TreeKEM [28]	Yes	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
Bienstock *et al.* [17]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log(n))^*$
Weidner *et al.* [32]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
CoCoA [3]	Yes	$\log(n)$	$\mathcal{O}(n\log^2(n))$	$\mathcal{O}(\log^2(n))$	$\mathcal{O}(\log(n))$
DeCAF (this work)	Yes	$\log(t)$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(\log(n))$

the blockchain, and the ratchet tree T_j is defined to be the ratchet tree T_{j-1} after processing the protocol messages contained in block B_j. One issue with this basic protocol is the fact that a message created referring to T_i can only be created after learning block B_i and must be added to the next block B_{i+1}. Depending on the block-arrival time of the chain, we might want to give messages more time to get included in the blockchain. We use a simple way to achieve this by introducing a parameter k, and only update the ratchet tree every k blocks, so messages referring to this tree can be included in any of the k blocks following the block specifying the tree. The parameter k should not be chosen larger than necessary, as only one update per k-block epoch will contribute towards healing (except if a corruption occurs in between two updates from the same epoch). If a message is not included in time this just means it can no longer be included, so the user can simply create a new message referring to the new ratchet tree.

To achieve FS, users should delete secret keys of outdated ratchet trees as soon as possible. For blockchains with immediate finality (i.e., no forks) this means old keys can be deleted immediately once a new ratchet tree is computed, while in longest-chain protocols one should wait to delete keys until the corresponding blocks are considered confirmed. Otherwise they might lose access to the group should a fork occur.

Efficiency. We now discuss the efficiency of DeCAF in healing a group with t compromises, and how it compares to related protocols. Throughout we refer to Table 1. There, we distinguish between two modes of TreeKEM (Propose and Commit). TreeKEM I corresponds to the conservative approach of only healing by sending commits (which would be expected behaviour, as argued below),

hence is not concurrent. TreeKEM II, in turn corresponds to using update proposals to heal at the expense of extra blanking. Note that an execution where, as a rule, users achieve PCS by sending update proposals instead of commit is not compatible with retaining logarithmic communication in the long term, due to the large amount of blanks, as illustrated on the last column of Table 1. Thus, the data shown for the communication complexity of the latter mode of TreeKEM during healing is only short term. In order to have the fairest comparison, we consider the complexity of DeCAF in the decentralized setting and that of CoCoA in the centralized one, in which it was proposed.

We consider the process by which the group heals from t compromises. We first stress that since a party does not know if they are corrupted, they cannot decide whether to update based on this. The main novelty of our protocol is that the number of epochs that it takes to heal depends on the number of corrupted parties, but *not on relative update behaviour of users*. Indeed, while several previous protocols could heal faster than what is shown on the table in an optimal execution, this execution needs for the users and/or the server to coordinate and/or make "optimal" choices obliviously (since, again, there is no reason the identities of corrupted parties are known); for instance, give preference to the corrupted parties in the case of concurrency, or coordinate to not concurrently commit or update. In the table we consider thus all users updating. This is the case for TreeKEM I and Causal TreeKEM, who could heal optimally in t epochs, and thus reduce the communication complexity accordingly; but also for TreeKEM II, [17] and [32], for which the number of epochs is not affected, but whose communication complexity could be reduced in an optimal execution.

One can see that, among the protocols that provide sub-linear communication costs for sending updates over the long term, our protocol manages to heal in the least amount of epochs. On the recipient side, our protocol performs within a logarithmic factor of all others, except for CoCoA, which naturally outperforms all other in this regard, due to users only storing a partial view of the tree. We stress that, if run in the decentralized setting, CoCoA loses its advantage in terms of recipient communication, leading to a cost of $\mathcal{O}(n \log(n)^2)$. Thus, in this setting it is outperformed by DeCAF in every aspect.

2 Preliminaries

In this section we provide syntax for secretly key-updatable PKE, define the notion of a blockchain-aided continuous group-key agreement and the concept of ratchet trees.

2.1 Secretly Key-Updatable Public-Key Encryption

We now recall the definition of secretly key-updatable public-key encryption (skuPKE) schemes [26]. A skuPKE scheme is essentially a public-key encryption scheme, that additionally allows the sampling of pairs (Δ, δ) of public and secret update information, which can be used to update secret and public keys, in a consistent way.

Definition 1. *A* secretly key-updatable public-key encryption scheme skuPKE *consists of the tuple of algorithms* (skuPKE.Gen, skuPKE.Enc, skuPKE.Dec, skuPKE.Sam, skuPKE.UpdP, skuPKE.UpdS).
Key-generation algorithm skuPKE.Gen *on input of the security parameter* 1^λ *returns a key pair* (pk, sk). *Encryption algorithm* skuPKE.Enc *on input of public key* pk *and message* m *returns a ciphertext* c. *The deterministic decryption algorithm* skuPKE.Dec *receives as input a secret key* sk *and a ciphertext* c *and returns either a message* m *or the symbol* $\perp$ *indicating a decryption failure. Sampling algorithm* skuPKE.Sam(1^λ) *is used to sample pairs* (Δ, δ) *consisting of public and secret update information. The key-update algorithms* skuPKE.UpdP *and* skuPKE.UpdS *get as input* (pk, Δ) *and* (sk, δ)*, respectively, and output a rerandomized key* pk' *or* sk'.

Correctness requires that updating the public and secret key of a key-pair with the same sequence of rerandomization factors preserves compatibility of the updated keys with each other. For *security* we essentially require that, on one hand, messages encrypted to a secret key that was generated by updating a potentially compromised secret key are secure as long as the secret update information to do so was not leaked, and, on the other hand, that leaking an updated key does not compromise ciphertexts encrypted to its predecessor as long as the secret update information was not leaked. We defer the formal definition of correctness and security, as well as, an instantiation based on the ElGamal scheme to the full version [2] of this paper.

2.2 Blockchain-Aided Continuous Group-Key Agreement

We now introduce the syntax of blockchain-aided continuous group-key agreement (baCGKA), which allows the set up of a group $G = (id_1, \ldots id_n)$ of users sharing an evolving group key. We assume all users id have an initialization key packet $((pk_{id}, sk_{id}), (svk_{id}, ssk_{id}))$, known to all other users. Here, (pk_{id}, sk_{id}) will be used to encrypt group invitation messages to id and (svk_{id}, ssk_{id}) to authenticate messages from id. In practice, this would be implemented by a PKI that allows users to deposit their and recover other users' key packets.

A baCGKA scheme baCGKA specifies algorithms baCGKA.Init, baCGKA.Upd, baCGKA.Add, baCGKA.Rem, baCGKA.Proc, baCGKA.Key, baCGKA.Send, and baCGKA.Fetch. The first 6 algorithms are local, in the sense that they only affect the executing user's state, and generate protocol messages to be sent to the rest of the group. The last two algorithms, on the other hand, interact with the distributed protocol by sending transactions and fetching blocks, respectively.

We consider a setting in which an append-only data structure is used to store the protocol messages and the data is distributed among several nodes. Users send their protocol messages to these nodes and then these nodes run a consensus algorithm that guarantees that they agree on their view of the data and on a total ordering of the blocks formed by the protocol messages. A blockchain is an example of this and that is why we use the term "blockchain-aided" CGKA.

Initialization. User id_1 runs $(id_1.st, W) \leftarrow \mathsf{baCGKA.Init}(G, (pk_{id_1}, \ldots, pk_{id_n}), ssk_{id_1})$ to initialize a session . Here $G = (id_1, \ldots, id_n)$ specifies the group, pk_{id_i} is the initialization encryption public-key of user id_i, and ssk_{id_1} the initialization authentication secret key of the party setting up the group. The output consists of user id_1's initial state and a welcome message W.

Updates. To update their state, id runs $(id.st, U) \leftarrow \mathsf{baCGKA.Upd}(id.st)$, updating their state and generating an update message.

Adding a Group Member. To add user id' to the group member id can run $(id.st, A) \leftarrow \mathsf{baCGKA.Add}(id.st, id', pk_{id'})$. Here $pk_{id'}$ is the initialization public key of id' and A an add message.

Removing a Group Member. User id can remove a (not necessarily different) user id' from the group by running $(id.st, R) \leftarrow \mathsf{baCGKA.Rem}(id.st, id')$. The output consists of an updated state and a removal message R.

Processing a Block. To process a block B consisting of update, welcome, add, and remove messages, and move to an updated state, user id runs $id.st \leftarrow \mathsf{baCGKA.Proc}(id.st, B)$.

Retrieving the Group Key. At any point a party id in the group can extract the current group key K from its local state st by running $K \leftarrow \mathsf{baCGKA.Key}(id.st)$.

Sending a Transaction. To send a protocol message M generated by one of the previous algorithms, user id runs $\mathsf{baCGKA.Send}(id.st, M)$.

Fetch New Blocks. Algorithm $(B_1, \ldots, B_\ell) \leftarrow \mathsf{baCGKA.Fetch}(id.st)$ returns all blocks added to the chain since the user last fetched them.

2.3 Ratchet Trees

Similarly to other efficient CGKA protocols, our protocol relies on a *ratchet tree*. This is a directed binary tree $T = (V, E)$, edges pointing towards the root v_{root}. Intuitively, the root corresponds to the group secret and every user id has an associated leaf v_{id}. For node v we denote its child by $v.child$, its parents by $v.par$, and its left and right parent by $v.lpar$ and $v.rpar$. If v is a leaf we denote its path to the root by $v.path$ and by $v.copath$ its copath, i.e. the set of parents of $w \in v.path$ that are not themselves in $v.path$.

Further, v has an associated state $v.st$ consisting of a skuPKE key pair ($v.pk$, $v.sk$), sets $v.unm_0$ and $v.unm_1$, and, if $v = v_{id}$ is a leaf, user id's signature key pair (svk_{id}, ssk_{id}). $v.unm_0$ and $v.unm_1$ are sets of *unmerged leaves*, capturing the leaves below v, whose users do not know the secret key $v.sk$. More precisely, $v.unm_0$ corresponds to unmerged users such that there has not yet been an epoch with an update affecting v since they joined the group, $v.unm_1$ to unmerged users, for whom a single such epoch exists. We denote by $v.stpub$ the public part of the state, i.e. $(v.pk, v.unm_0, v.unm_1)$ and, if $v = v_{id}$ is a leaf, the signature verification key svk_{id}. The secret part $v.stsec$ of v's state consists of $v.sk$ and, if

$v = v_{id}$ is a leaf, the signature signing key ssk_{id}. Similarly, we denote by $T.stpub$ the public part of the ratchet tree, i.e., (V, E) together with $v.stpub$ for all $v \in V$. A node's state can also be *blank*, meaning its state is empty. For the purpose of later populating this node with a new state, a blank node is considered to have a dummy key-pair (pk_c, sk_c), sampled when the group is created, and whose secret key is public knowledge. Updates unblanking a node will then update this dummy key-pair. Finally, we define the *resolution* $v.res$ of v as $v.res = \{v\}$ if v not blank, $v.res = \emptyset$ if v is a blank leaf, and $v.res = \bigcup_{v' \in v.par} v'.res$ else.

3 Protocol Description

We now describe DeCAF in detail. Section 3.1 describes how the protocol proceeds in epochs determined by the blockchain's blocks, Sect. 3.2 how the structure of the ratchet tree is modified when handling changes to the group membership, and Sect. 3.3 how update information for a path in the ratchet tree is sampled and applied. Finally, in Sect. 3.4 we give the description of the protocol's algorithms. For a more formal description of DeCAF in pseudocode see the full version [2] of this paper.

3.1 Blocks and Epochs

DeCAF proceeds in epochs consisting of k blocks. More precisely the ith epoch corresponds to blocks $i \cdot k+1, \ldots, i \cdot k+k$ of the blockchain. Updates are generated with respect to the ratchet tree of the *first* block of the current epoch. This is to handle potential delays of up to k blocks from the moment a user sends a message containing group operations information to the moment it makes it into the blockchain. At the beginning of a new epoch, the group switches to a new ratchet tree that incorporates all updates of the last epochs, as well as the dynamic changes made to the group. One consequence of having to accommodate for such delays is that users need to store at least the keys at the beginning of an epoch for the entire duration of it, and if the underlying blockchain does not have immediate finality potentially keys from further back. This translates into weaker FS guarantees than in the server setting as a user cannot immediately delete keys after updating to the next state. But this difference will be marginal as the length of an epoch (or confirmation time of the blockchain, whichever is larger) will still be tiny compared to the duration for which users are typically offline. A second consequence is that these delays introduce a further delay in the execution of dynamic operations. Indeed, updating information generated during an epoch is computed without taking into account users that were being removed or added during that epoch. Thus, in the case of epochs with adds, the key at the end of that epoch will not be known to the new parties, who will need to wait one more epoch to learn it. In the case of epochs with removes, the key at the end of that epoch will be blank, so a new key will be necessary to establish a new group key that the removed users do not have knowledge of. We remark that this seems to be somewhat inherent. In fact, if we set $k = 1$,

the situation is not that different than that in other protocols like CoCoA or TreeKEM, where a first round of dynamic operations needs to be followed by a subsequent one where the commit effecting the operations takes place. In summary, using a blockchain for decentralization gives improved consistency and security guarantees, but the delay between protocol epochs is now dictated by the block arrival and typical inclusion times of the underlying blockchain. Therefore, FS is (marginally) affected by the confirmation time of blocks.

User id's state $id.st$ contains the user's identifier id, two ratchet trees $T = (V, E)$ and $T_{\text{next}} = (V_{\text{next}}, E_{\text{next}})$, lists O_{next}, and U_{pending}, epoch counter e_{ctr}, a key pair (pk_c, sk_c), the (potentially empty) group key K, and a working copy of the group key K_{next} for the next epoch.

T contains the state of the ratchet tree at the beginning of the current epoch. More precisely, this encompasses the public states $v.stpub$ of all nodes $v \in V$ and, if we denote id's leaf in T by v_{id}, additionally the secret node states $v.stsec$ for all nodes v in id's update path $v_{id}.path$. Ratchet tree T_{next} serves as a working copy for the next epoch, i.e., it contains keys updated according to the blocks already processed in the current epoch—excluding dynamic operations. Note that the two trees differ only in the node states, but not the general tree structure. To clarify whether we consider nodes in T or T_{next}, we will denote nodes in the latter by v^{n}. O_{next} is a list of the dynamic operations included in the blocks of the current epoch that were already processed. These changes will be applied to T_{next} at the end of the epoch. List U_{pending} stores pending update information. The epoch counter e_{ctr} is used to generate and confirm protocol messages for the current epoch. Finally (pk_c, sk_c) is the dummy key-pair used for blank nodes.

3.2 Implementing Dynamic Operations

As a result of dynamic operations, the tree structure will change. Here, we describe this change, ahead of the protocol description.

To add parties we use the *unmerged leaves* technique, introduced in TreeKEM v9 [12]. Note that a new user might not be able to receive the keys for all nodes in their path to the root the moment they are added, since all other parties under any of these nodes might be offline at the time. Thus, new parties are joined directly to the root, and sent the keys in their path in subsequent epochs. More in detail, whenever id, whose path shares a node with that of a new party id', generates an update in a follow-up epoch, they need to encrypt the current key for that node, together with the seed used to sample the update information to id'. However, this key might already have been present in an epoch which preceded that in which id' was added. Hence, sending it to id could cause problems with forward secrecy—id must ensure that the key sent to id' was updated *after* they joined the group. Thus, this process is done in two steps. First, upon being added to the group, id' is included into the set $v.unm_0$ for all v in their path, except for the root. Updates that apply to v, issued while id' is in this set $v.unm_0$, do not encrypt any secret information about v to id. Whenever an epoch first contains such an update for v, however, id' is removed from the set $v.unm_0$ and added to $v.unm_1$, at the end of the epoch. This signals that

the key at v is now safe to be communicated to id'. Any following update that applies to v once $id' \in v.unm_1$, will then encrypt the current key plus the update information to id'. Once such an update occurs, id' learns the key at v, and is then removed from $v.unm_1$. The one exception to this is the root node v_{root}, where id' is directly added to $v_{root}.unm_1$. The reason is that all add operations are coupled with an update from the issuing party, thus ensuring that the root key at the end of that epoch is updated, and thus safe to communicate to id'.

Removes are handled via *blanking*, where the keys that removed users had knowledge of get set to the dummy key-pair (pk_c, sk_c) and get ignored by users encrypting new secret update information δ_i until they get updated again.

All these changes are executed once at the end of each epoch. While all group operations in the following epoch will take the new tree into account, added and removed users will not be properly added and removed until the end of that following epoch, though. This seems inherent if we want to allow concurrency: the author of an operation concurrent with a dynamic one will be oblivious to the latter, thus unable to prepare their operation taking it into account.

More in detail, at the end of an epoch where adds $A = (A_1, \ldots, A_{\ell_a})$, removes $R = (R_1, \ldots, R_{\ell_r})$, and modifications $M = (M_1, \ldots, M_{\ell_m})$ to the sets of unmerged users took place, users will call algorithm $\mathsf{upd\text{-}tree}(T_{\text{next}}, A, R, M)$, which will output the tree resulting from applying these operations. First, the algorithm in order processes the M_i, which are lists of nodes that were affected by updates in the current epoch (their exact definition is given in Sect. 3.3 below). For every $v \in M$ the sets of unmerged leaves are updated to $v.unm_1 \leftarrow v.unm_0$ and $v.unm_0 \leftarrow \emptyset$. Then, the algorithm will set the state of all in the paths of any of the removed users to *blank*, and associate with them the dummy key-pair (pk_c, sk_c). Added parties will get assigned a leaf in the tree in a canonical way, determined by the ordering of operations in the corresponding block. The first leaves to be assigned will be blank ones, and new leaves to the right of the existing ones will be added, if there are not enough blanked ones, adding any internal nodes necessary to maintain the binary structure of the tree. If a new root node must be added to accommodate for the new parties, this will be given the dummy key-pair until updated at the end of the next epoch. Then, for each newly-added party id_i with init key pk_{id}, it sets the state of their new leaf v_{id} to (pk_{id}, svk_{id}), and for any $v \in l_i.path$ except the root v_{root}, it adds id_i to $v.unm_0$. The root id_i is added to $v_{root}.unm_1$. Finally, it outputs the resulting tree.

Both blanks and unmerged leaves sets can disappear over the protocol execution, bringing the tree back to its optimal binary structure. Whenever an Update including new update information for a node v takes place, v will become unblanked if it was not so already. Moreover, unmerged leaves in unm_1 will become merged, and those in unm_0 will then pass to unm_1.

3.3 Updating the States of an Update Path

During group creation and updating, users will update the keys along some path. Before describing our protocol's algorithms, we detail this operation.

Consider user id with associated leaf v_{id}. Update information for the keys of $v_{id}.path$ is sampled using $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathsf{gen\text{-}path\text{-}upd}(id.st)$. The algorithm, on input of the user's state, first fetches $(v_1 = v_{id}, \ldots, v_r = v_{root}) = v_{id}.path$ with respect to ratchet tree T corresponding to the beginning of the epoch. Let m be maximal such that $id \in v_{m-1}.unm_0 \cup v_{m-1}.unm_1$. If no such m exists, we set $m = 2$. The algorithm samples a seed s_1 uniformly at random and computes $s_m = \mathsf{H}_1(s_1)$ as well as $s_i = \mathsf{H}_1(s_{i-1})$ for $i = m+1, \ldots, r$. For $i \in \{1, m, \ldots, r\}$ it samples update information $(\Delta_i, \delta_i) \leftarrow \mathsf{skuPKE.Sam}(\mathsf{H}_2(s_i))$ using randomness $\mathsf{H}_2(s_i)$. It then for $i \in \{m, \ldots, r\}$ computes vectors of ciphertexts $C_i = (c_{i,j})_{z_j}$ with $c_{i,j} \leftarrow \mathsf{skuPKE.Enc}(z_j.pk, s_i)$, where the nodes z_j are chosen as $z_j \in v_{i-1}.res \cup v_i.unm_1 \backslash v_{i-1}.unm_1$ for $i = m+1, \ldots, r$ and $z_j \in (v_i.lpar).res \cup (v_i.rpar).res \cup v_i.unm_1 \backslash \{id\}$ for $i = m$. Finally, $\kappa = \mathsf{H}_1(s_r)$ will be used to update the group key. The algorithm's output is $((\Delta_i, \delta_i, C_i)_i, \kappa)$. Looking ahead, $(\Delta_i, C_i)_i$ will be sent out as the update message and $((\Delta_i, \delta_i)_i, \kappa)$ saved in the user's pending state.

When user id' wants to apply a path update $(\Delta_i, C_i)_i$ with $i \in \{1, m, \ldots, r\}$ generated by user id, they call algorithm $id'.st \leftarrow \mathsf{proc\text{-}path\text{-}upd}(id'.st, (\Delta_i, C_i)_i)$. It first fetches user id's update path $(v_1^{\mathrm{n}} = v_{id}^{\mathrm{n}}, \ldots, v_r^{\mathrm{n}} = v_{root}^{\mathrm{n}}) = v_{id}^{\mathrm{n}}.path$ from the working copy T_{next} of the ratchet tree. Then, for all i it updates the public keys along the path, i.e., $v_i^{\mathrm{n}}.pk \leftarrow \mathsf{skuPKE.UpdP}(v_i^{\mathrm{n}}.pk, \Delta_i)$. Here, if v_i^{n} was blank the public key of a constant dummy key-pair (pk_c, sk_c) is used as $v_i^{\mathrm{n}}.pk$. Note that this implies that v_i^{n}'s resolution is now $\{v_i^{\mathrm{n}}\}$.

Let v_i denote the first node that is shared between $v_{id}.path$ and $v_{id'}.path$ and for which $id' \notin v_i.unm_0$. Then, if the update was generated during the current epoch, C_i contains an encryption $c_{i,j}$ of seed s_i under the public key of some node $w_{i,j}$ for which the secret key is contained in id's copy of tree T that is part of $v_{id'}.st$. The algorithm recovers $s_i \leftarrow \mathsf{skuPKE.Dec}(w_{i,j}.sk, c_{i,j})$ and for $j \in \{i+1, \ldots, r\}$ computes $s_j = \mathsf{H}_1(s_{j-1})$ and update information $(\Delta_j, \delta_j) \leftarrow \mathsf{skuPKE.Gen}(\mathsf{H}_2(s_j))$. It then updates the corresponding secret keys in T_{next} as $v_j^{\mathrm{n}}.sk \leftarrow \mathsf{skuPKE.UpdS}(v_j^{\mathrm{n}}.sk, \delta_j)$, where, analogous to the above, if v_j is blank, sk_c takes the role of $v_j.sk$. Finally, the algorithm computes group key update information $\kappa = \mathsf{H}_1(s_r)$, incorporates it in the working copy of the group key $K_{\mathrm{next}} \leftarrow K_{\mathrm{next}} \oplus \kappa$, and adds the list $M = (v_m, \ldots, v_r)$ to O_{next}. The latter will be used to update the sets of unmerged users at the end of the epoch.

3.4 Protocol Algorithms

To **initialize a group** for users $(id_1, \ldots, id_n)$, user id_1 first generates the dummy key-pair $(pk_c, sk_c) \leftarrow \mathsf{skuPKE.Gen}(1^\lambda)$. They then set up a left-balanced binary ratchet tree $T = (V, E)$. Every node in T is blank, except for the leaves. The public state of the i^{th} leaf contains the corresponding user's initialization public key and their signature verification key. Further, the secrets state of id_1, $v_{id_1}.stsec$, contains id_1's secret decryption and signing keys. Group creator id_1 incorporates (pk_c, sk_c), T, a copy T_{next} of T, and an empty list O_{next} in their state and computes $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathsf{gen\text{-}path\text{-}upd}(id_1.st)$. The tuple

$((\Delta_i, \delta_i)_i, \kappa)$ is added to id_1's state together with epoch counter $e_{\text{ctr}} = (0,0)$ (where the first coordinate denotes the epoch and the second one the block inside the epoch) and $K_{\text{next}} \leftarrow 0$. The algorithm outputs the resulting state and welcome message $W = (T.stpub, (\Delta_i, C_i)_i, (pk_c, sk_c), \sigma, id_1)$, where σ is a signature of $(T.stpub, (\Delta_i, C_i)_i, (pk_c, sk_c))$ under ssk_{id_1}.

To issue an **update**, id computes $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathsf{gen\text{-}path\text{-}upd}(id.st)$. The secret update information $(\delta_i)_i$ and κ are stored in id's pending state U_{pending}. Let $(v_1, \ldots, v_r) = v_{id}.path$ be id's update path. Update messages also communicate the current secret key of nodes to unmerged users that have already processed an update on this node. More precisely, the updating user for all $i \in [2, \ldots, r]$ such that $id \notin v_i.unm_0 \cup v_i.unm_1$ computes a vector of ciphertexts $\tilde{C}_i = (\tilde{c}_{i,j})_{z_j}$, where $\tilde{c}_{i,j} = \mathsf{skuPKE.Enc}(z_j.pk, v_i.sk)$ and z_j are the nodes satisfying $z_j \in v_i.unm_1$. For users who just joined the group, and are thus unmerged at the root, this ciphertext contains the key K_{next}. The algorithm outputs message $U = ((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\text{ctr}}, \sigma, id)$, where σ is a signature of $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\text{ctr}})$ under ssk_{id}.

To **add** a user, when called by id, the addition algorithm outputs $\tilde{A} = (A, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}}, \sigma, id)$, containing an add request $A =$ "$\texttt{add.user}(id')$", where id' is the new user. Further, it contains a copy of the public ratchet tree state, the dummy key pair, an update message U generated as described in the previous paragraph, the epoch counter, a signature σ of the message $(A, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}})$ under ssk_{id}, and the identity id.

To **remove** a user, when called by user id, the removal algorithm outputs $\tilde{R} = (R, e_{\text{ctr}}, \sigma, id)$, with $R =$ "$\texttt{remove.user}(id')$" for id' the removed user, and where σ is a signature of (R, e_{ctr}) under ssk_{id}.

To **process a block**, user id processes a block $B = (W, U, \tilde{A}, \tilde{R})$ consisting of (a potential) welcome message W, update messages $U = (U_1, \ldots, U_{\ell_u})$, add messages $\tilde{A} = (\tilde{A}_1, \ldots, \tilde{A}_{\ell_a})$, and removal messages $\tilde{R} = (\tilde{R}_1, \ldots, \tilde{R}_{\ell_r})$ as follows. We first describe the case of users already in the group. User id starts by processing the update messages given by the block as follows. Update message U_ℓ for $\ell \in [\ell_u]$ has the form $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\text{ctr}}, \sigma, id)$. First, the user checks whether the signature σ verifies under svk'_{id} and that e_{ctr} matches the value stored in $id.st$. If one of the checks fails the update is discarded.

If $id = id'$, i.e., U_ℓ is an update generated by the processing user, id retrieves from U_{pending} the corresponding update information $((\Delta_i, \delta_i)_i, \kappa)$ with $i = \{1, m, \ldots, r\}$ for some m, deletes it from U_{pending}, and applies it to their update path $v^{\text{n}}_{id}.path = (v^{\text{n}}_1, \ldots, v^{\text{n}}_r)$ with respect to T_{next} as $v^{\text{n}}_i.pk \leftarrow \mathsf{skuPKE.UpdP}(v^{\text{n}}_i.pk, \Delta_i)$ and $v^{\text{n}}_i.sk \leftarrow \mathsf{skuPKE.UpdS}(v^{\text{n}}_i.sk, \delta_i)$ (note that this updates all key pairs on id's update path for which the user has access to the secret key). Then they set $K_{\text{next}} \leftarrow K_{\text{next}} \oplus \kappa$. Else, let $v_{u_1}, \ldots, v_{u_t}$ be the nodes in $v_{id}.path \cap v_{id'}.path$ such that $id \in v_{u_i}.unm_1$ and $u_i \geq m$. Then, $\tilde{C}_{u_i}$ contains an encryption of $v_{u_i}.sk$ under id's leaf key $v_{id}.pk$. For $i \in [u_1, \ldots, u_t]$, id uses the corresponding secret key to recover $v_{u_i}.sk$ and adds it to $v_{u_i}.st$ in T and T_{next} unless the state already contains a secret key. Then id calls $id.st \leftarrow \mathsf{proc\text{-}path\text{-}upd}(id.st, (\Delta_i, C_i)_i)$, which updates the keys affected by the

update in the working copy T_{next} of the ratchet tree (note that the secret keys added in the previous step ensure id is able to decrypt the ciphertext relevant to them), the working copy of the group key, and the list of merges to be implemented at the end of the epoch.

After processing all update operations, id processes adds $\tilde{A}$ and subsequently removes $\tilde{R}$. First, they check that the signature included in a message verifies and that the message was generated for the current epoch, discarding it if not. In the case of an add message $\tilde{A}_\ell = (A_\ell, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}}, \sigma, id)$ the user processes the update message U as described above and appends A_ℓ to O_{next}. For valid remove message $\tilde{R}_\ell = (R_\ell, e_{\text{ctr}}, \sigma, id)$ the request R_ℓ is added to O_{next}. Finally, if B was the last block of an epoch, i.e., B is the ith block with $i = 0 \bmod k$, then id prepares the transition to the next epoch. To this end, id recovers from O_{next} the ordered lists of merges $M = (M_1, \ldots, M_{\ell_m})$, adds $A = (A_1, \ldots, A_{L_a})$, and removes $R = (R_1, \ldots, R_{L_r})$ that were included in the blocks of the current epoch. Then they apply these changes to the working copy of the ratchet tree $T_{\text{next}} \leftarrow \mathsf{upd\text{-}tree}(T_{\text{next}}, A, R, M)$ to be used in the next epoch, update $T \leftarrow T_{\text{next}}$, increase the epoch counter to $e_{\text{ctr}} \leftarrow ((e_{\text{ctr}})_1 + 1, 0)$, set O_{next} to the empty list, and update the group key to $K \leftarrow \mathsf{H}_1(\text{“}\mathtt{key}\text{”}, K_{\text{next}})$, and afterwards $K_{\text{next}} \leftarrow \mathsf{H}_1(\text{“}\mathtt{next}\text{”}, K_{\text{next}})$.

Let us now describe the second case, that is, that of users not in the group. We distinguish two further cases according to whether id (a) was added in an add operation or (b) in the group initialization (i.e., $W \neq \perp$). In case (a) let $B_1^{\text{p}}, \ldots, B_k^{\text{p}}$ be the blocks of the previous epoch. Then one of these blocks contains an add message $\tilde{A} = (A, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}}, \sigma, id)$ with $A = \text{“}\mathtt{add.user}(id', id)\text{”}$ being the add request for user id. The user, after validating the signature and epoch, incorporates $T.stpub, (pk_c, sk_c)$ in $id.st$. As $T.stpub$ is the ratchet tree of the previous epoch, id brings it up to date by processing, in order, the blocks $B_1^{\text{p}}, \ldots, B_k^{\text{p}}$. Here, as they do not have access to any secret keys of the tree, they only update the public keys. After this operation T and its copy T_{next} match the current epoch and the user adds to $v_{id}.stsec$ their init decryption key and ssk_{id}, and then processes the current block $B = (U, A, R)$ as described above.

Finally, assume that id was added as part of the group initialization, i.e., case (b) above, with $W = (T.stpub, (\Delta_i, C_i)_i, (pk_c, sk_c), \sigma, id_1)$. In this case id checks that the signature σ verifies under svk_{id_1}, rejecting it if this is not the case. If id is the user who issued the initialization message, they recover $((\Delta_i, \delta_i)_i, \kappa)$ from their state, apply the update information to their update path, set $K_{\text{next}} \leftarrow \kappa$, and $K \leftarrow \mathsf{H}_1(\text{“}\mathtt{key}\text{”}, K_{\text{next}})$. If id did not issue the initialization message, they incorporate $(T.stpub, (pk_c, sk_c))$ in their state, add to $v_{id}.stsec$ their init decryption key and ssk_{id}, set K_{next} to the zero string, and run $id'.st \leftarrow \mathsf{proc\text{-}path\text{-}upd}(id'.st, (\Delta_i, C_i)_i)$ to update T_{next}. K is set to $\mathsf{H}_1(\text{“}\mathtt{key}\text{”}, K_{\text{next}})$, O_{next} is initialized as empty list, as there are no merge, add, or remove operations yet, and $e_{\text{ctr}} \leftarrow (0, 0)$.

We conclude by describing the remaining operations of the CGKA scheme. To **extract** the current group key, a user id fetches K from its state, and deletes

this value afterwards. To **send** a protocol message, *id* simply uses the underlying blockchain protocol to send it as a transaction to the blockchain. To **fetch** the last blocks of operations, *id* uses the underlying blockchain protocol to retrieve the blocks added to it since it last did.

4 Security

To analyze the modified protocol, we essentially use the security model from [27], which allows the adversary to act partially active and fully adaptive. The only differences in the setting of baCGKA are that 1) users are processing concurrent messages, and 2) no messages will ever be rejected. It is however possible that messages get lost and hence sent but not processed.

Asynchronous baCGKA security is defined through a game between an adversary and a challenger, where the adversary can request to see arbitrary execution patterns of the protocol, i.e. decide on how many parties to initiate a group key agreement, then dictating parties to update their state (by posting a respective message on the blockchain), remove/add other parties, download and process updates, and also to start/end corruption of users (which leaks the users entire state during the corrupted period). The adversary can decide on this sequence of actions fully adaptively and can request arbitrary actions to be performed concurrently. For security (see the full version of this paper [2] for the formal definition), intuitively, we aim to guarantee that all group keys which were not leaked to the adversary via (processing of updates using) corrupted keys remain indistinguishable from random.

To precisely define the set of group keys for which we can guarantee security, similar to previous work, we define a *safe predicate*. Intuitively, in our protocol a group key will be considered *safe* if all users to which this key was communicated (i.e., the current group members in the view of the party generating the group key) have either performed a single update (with no-one else performing a concurrent update) or participated in at least $\lfloor \log(C) \rfloor + 1$ concurrent updates (C denoting the total number of corrupted users since the last time the group key was secure) since their last corruption, and furthermore have processed a further own (potentially concurrent) update before the next corruption. This is in contrast to the predecessor CoCoA [3], which also allows for concurrent updates, but requires each party to perform $\lfloor \log(n) \rfloor + 1$ concurrent updates (n being the group size, which can be assumed significantly larger than the number C of corrupted parties). In the full version of this paper [2] we prove the following theorem.

Theorem 1. *If the secretly key-updatable public key encryption scheme used in DeCAF is* $(\varepsilon_{\text{Enc}}, t)$*-IND-CPA-secure (*$t$ *denoting the runtime,* ε_{Enc} *the advantage of adversaries) and the used hash functions are modeled as random oracles, then DeCAF is* $(O(\varepsilon_{\text{Enc}} \cdot 2(nQ^2)^2), t, Q)$*-baCGKA-secure, where* Q *denotes the number of oracle queries made in the security game.*

References

1. Alwen, J., et al.: Grafting key trees: efficient key management for overlapping groups. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part III. LNCS, vol. 13044, pp. 222–253. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90456-2_8
2. Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K.: DeCAF: decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559 (2022). https://eprint.iacr.org/2022/559
3. Alwen, J., et al.: CoCoA: concurrent continuous group key agreement. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part II. LNCS, vol. 13276, pp. 815–844. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-07085-3_28
4. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56784-2_9
5. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021, pp. 1463–1483. ACM Press (2021)
6. Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64378-2_10
7. Alwen, J., Hartmann, D., Kiltz, E., Mularczyk, M.: Server-aided continuous group key agreement. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022, pp. 69–82. ACM Press (2022)
8. Alwen, J., Jost, D., Mularczyk, M.: On the insider security of MLS. Cryptology ePrint Archive, Report 2020/1327 (2020). https://eprint.iacr.org/2020/1327
9. Alwen, J., Mularczyk, M., Tselekounis, Y.: Fork-resilient continuous group key agreement. Cryptology ePrint Archive, Paper 2023/394 (2023). https://eprint.iacr.org/2023/394
10. Auerbach, B., Cueto Noval, M., Pascual-Perez, G., Pietrzak, K.: On the cost of post-compromise security in concurrent continuous group-key agreement. In: Rothblum, G., Wee, H. (eds.) TCC 2023. LNCS, vol. 14371, pp. 271–300. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-48621-0_10
11. Balbás, D., Collins, D., Vaudenay, S.: Cryptographic administration for secure group messaging. In: 32nd USENIX Security Symposium (USENIX Security 2023), Anaheim, CA, pp. 1253–1270 . USENIX Association (2023)
12. Barnes, R., Beurdouche, B., Millican, J., Omara, E., Cohn-Gordon, K., Robert, R.: The messaging layer security (MLS) protocol. Internet-Draft draft-ietf-mls-protocol-09, Internet Engineering Task Force, Work in Progress (2020)
13. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The messaging layer security (MLS) protocol. RFC 9420 (2023)
14. Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: asynchronous decentralized key management for large dynamic groups (2018)
15. Bhargavan, K., Beurdouche, B., Naldurg, P.: Formal models and verified protocols for group messaging: attacks and proofs for IETF MLS. Research report, Inria, Paris (2019)
16. Bienstock, A., Dodis, Y., Garg, S., Grogan, G., Hajiabadi, M., Rösler, P.: On the worst-case inefficiency of CGKA. In: Kiltz, E., Vaikuntanathan, V. (eds.) TCC 2022, Part II. LNCS, vol. 13748, pp. 213–243. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-22365-5_8

17. Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64378-2_8
18. Brzuska, C., Cornelissen, E., Kohbrok, K.: Cryptographic security of the MLS RFC, Draft 11. Cryptology ePrint Archive, Report 2021/137 (2021). https://eprint.iacr.org/2021/137
19. Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, pp. 1802–1819. ACM Press (2018)
20. Coin, X.: Elixxir architecture brief v2.0. https://xx.network/elixxir-architecture-brief-v1.0.pdf
21. Cong, K., Eldefrawy, K., Smart, N.P., Terner, B.: The key lattice framework for concurrent group messaging. Cryptology ePrint Archive, Paper 2022/1531 (2022). https://eprint.iacr.org/2022/1531
22. Cremers, C., Hale, B., Kohbrok, K.: The complexities of healing in secure group messaging: why cross-group effects matter. In: 30th USENIX Security Symposium (USENIX Security 2021), pp. 1847–1864. USENIX Association (2021)
23. Devigne, J., Duguey, C., Fouque, P.-A.: MLS group messaging: how zero-knowledge can secure updates. In: Bertino, E., Shulman, H., Waidner, M. (eds.) ESORICS 2021. LNCS, vol. 12973, pp. 587–607. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88428-4_29
24. Hashimoto, K., Katsumata, S., Postlethwaite, E., Prest, T., Westerbaan, B.: A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021, pp. 1441–1462. ACM Press (2021)
25. Hashimoto, K., Katsumata, S., Prest, T.: How to hide MetaData in MLS-like secure group messaging: simple, modular, and post-quantum. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022, pp. 1399–1412. ACM Press (2022)
26. Jost, D., Maurer, U., Mularczyk, M.: Efficient ratcheting: almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 159–188. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_6
27. Klein, K., et al.: Keep the dirt: tainted TreeKEM, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA, pp. 268–284. IEEE Computer Society (2021)
28. Weidner, M.A.: Group messaging for secure asynchronous collaboration. Master's thesis, University of Cambridge (2019)
29. Perrin, T., Marlinspike, M.: The double ratchet algorithm (2016). https://signal.org/docs/specifications/doubleratchet/
30. Poettering, B., Rösler, P., Schwenk, J., Stebila, D.: SoK: game-based security models for group key exchange. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 148–176. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75539-3_7
31. Wallez, T., Protzenko, J., Beurdouche, B., Bhargavan, K.: TreeSync: authenticated group management for messaging layer security. In: 32nd USENIX Security Symposium (USENIX Security 2023), Anaheim, CA, pp. 1217–1233. USENIX Association (2023)
32. Weidner, M., Kleppmann, M., Hugenroth, D., Beresford, A.R.: Key agreement for decentralized secure group messaging with strong security guarantees. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021, pp. 2024–2045. ACM Press (2021)