Coffee: Fast Healing Concurrent Continuous Group Key Agreement for Decentralized Group Messaging

Joël Alwen[1], Benedikt Auerbach[*] [2], Miguel Cueto Noval[2], Karen Klein[†] [3], Guillermo Pascual-Perez[‡] [2], and Krzysztof Pietrzak[*] [2]

[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

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 ("Message Layer Security" (MLS) IETF draft) 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.

We present a new "fast healing" concurrent CGKA protocol, named Coffee, where users will heal after at most $\log(t)$ requests, with t being the number of corrupted users. Our new protocol is particularly interesting to realize decentralized group messaging,

[*]Funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT).

[†]Supported in part by European Research Council (ERC) CoG grant 724307. A part of this work was conducted at ISTA, funded by the ERC under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT).

[‡]Funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No.665385.

where protocol messages (add/remove/update) are being posted on a blockchain rather than sent to a server. In this setting, concurrency is crucial once requests are more frequent than blocks. Our new protocol significantly outperforms (the only alternative with sub-linear communication and PCS) CoCoA in this setting: it heals much faster ($\log(t)$ vs. $\log(n)$ rounds). The communication per round and user is $m \cdot \log(n)$, but in this setting – where there is no server who can craft specific messages to users depending on their position in the tree – CoCoA requires the same communication.

Contents

1 Introduction

1.1 (Group) Messaging

Popular group messaging applications, like Signal [PM16], work in an asynchronous setting, where users only must occasionally be online and their messages are relayed by an untrusted server. The underlying ratcheting protocol provides strong security; apart from end-to-end encryption, also forward secrecy (FS) and post compromise security (PCS), which is important as conversations can go on for long times. 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.

It is a challenging problem, and the focus of the "Message Layer Security" (MLS) IETF working group, to efficiently scale messaging applications to larger groups without giving up on the strong security properties provided in the two party case by protocols like the Double Ratchet [PM16]. Most of the so far proposed group messaging schemes with this motivation, starting with ART [CCG⁺18] and TreeKEM [KPPW⁺21] and variants (discussed in the related work below), use a binary tree structure to maintain the keys of the users. In this so called ratchet tree, each node corresponds to a public/secret key pair. Leaves are identified with users who hold the corresponding secret keys, while the key at the root is the group key used to exchange messages for the group. We think of the edges of the tree as being directed from the leaves to the root, and an edge $(pk, sk) \to (pk', sk')$ basically means that sk' is encrypted under pk in a ciphertext that can be retrieved from the delivery server. This way, the user of a leaf with key-pair (pk, sk) will be able to retrieve all the secret keys on the path from its leaf to the group key at the root.

1.2 CGKA

Continuous group key agreement (CGKA) was defined in [ACDT20] as the key primitive underlying group messaging. CGKA allows a set of users to maintain a shared key in an asynchronous setting where 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.

The reason to use trees rather than, say, pairwise channels for maintaining the keys, lies in the fact that in groups of size n, each user only has to send $\log(n)$ ciphertexts in order to perform a key update, as opposed to n. Concretely, as illustrated in Figure 1 (the two trees on the left, ignoring the blue nodes for now), if a user A wants to update, they will 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. The other group members can fetch those ciphertexts and update to 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 (so it is sufficient to replace keys on a path to achieve FS and PCS). This invariant becomes difficult to ensure once we

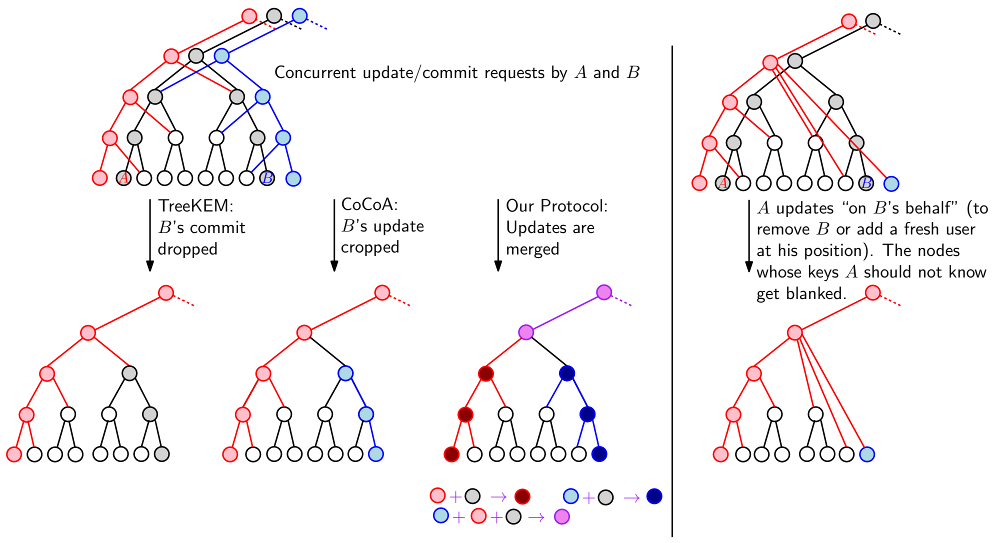
Concurrent update/commit requests by A and B
A
B
TreeKEM: B's commit dropped
CoCoA: B's update cropped
Our Protocol: Updates are merged
A updates "on B's behalf" (to remove B or add a fresh user at his position). The nodes whose keys A should not know get blanked.

Figure 1: (left) Illustration of how TreeKEM, CoCoA and our protocol handle a concurrent update by parties A and B who want to replace their (potentially compromised) keys. (right) An illustration of blanking used to commit an update proposal (removing B would be similar, with their leaf node blanked instead.)

consider additions or removals of users from the group. Assume a user A wants to remove a user B. For this, A could propose an update where it replaces B's path. But unless A is the sibling of B in the tree, now the invariant no longer holds (adding a new user has the same issue). To address this, in TreeKEM all the nodes whose keys A should not know are blanked, which simply means the node is removed and its ingoing edges point directly to its child (if the child is blanked, to its grandchild etc.). Figure 1 (right side) shows the tree we get if A removes B this way. Adds are instead handled through the "unmerged leaves" technique, where the leaves of the new users get attached directly to the root, increasing its indegree. Note that this means an add or remove operation destroys the nice tree structure, and as a consequence future operations become more expensive e.g., to update A must send 4 ciphertexts before blanking B, but 6 after, in general the cost can grow from $\log(n)$ to n. Fortunately, adds and removes are typically rare operations, and the tree structure heals as parties update their keys (e.g., a single update of B or its sibling will fix the tree completely).

Concurrent updates. While updates in the initial versions of TreeKEM just need $\log(n)$ communication and leave the tree structure intact, they are inherently sequential: a user can only send an update request after having processed the previous one. If two (or more) users A and B send an update request to the server referring to the same previous state (as illustrated on the left in Figure 1), the server will simply reject all but one of the requests. This is also the case for variants such rTreeKEM [ACDT20] and Tainted TreeKEM [KPPW$^+$21].

Recent versions of TreeKEM do allow for concurrent updates through the "Propose and Commit" framework. Here, the users concurrently announce the add/remove/update operations in a first round. One party then "commits" them in a second round by sampling new keys on its path and blanking all nodes not on its path that need to be refreshed, thus severely ruining the tree structure (similar to removals). If the group members want to preserve the tree structure, and its associated optimal structure, one would expect this to rarely occur, and for users to refresh their keys by issuing a commit. Here, concurrency is not possible anymore, as commits need to be totally ordered, and the issue outlined above comes back.

Causal TreeKEM. The first CGKA protocol supporting concurrent updates was Causal TreeKEM [Mat19]. 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 me merged at the same time, without regard for the order in which users received them. Our protocol is similar to Causal TreeKEM in several aspects, and can generally be seen as both an improvement and formalization of it. On the one hand, Causal TreeKEM does not give any FS guarantees, and PCS requires a number of updates equal to the amount of corrupted group members, each of which needs to take place in a different round. In contrast, our protocol does provide FS (albeit slightly weaker than TreeKEM due to the potential delays in getting messages into the blockchain), and only needs $\log(t)$ epochs to heal, with t being the number of corrupted parties (this faster healing might also be true for Causal TreeKEM, due to the protocol similarities, but is not alluded to, or proved, in their paper). On the other hand, Causal TreeKEM does not formalize the security of the "key merging" functionality, and does not give full security proofs.

CoCoA. The recent proposal CoCoA [AAC$^+$22] processes concurrent update proposals in a "greedy" manner and simply accept as many keys in a concurrent proposal as possible. As illustrated in Figure 1, fresh keys from concurrent updates are accepted, and if there is a conflict as two updates want to replace the same node, one of the two updates is rejected *from this point upwards*. While this process does not guarantee that the key is safe after every compromised party updated,[1] somewhat surprisingly [AAC$^+$22] proves that the tree does heal after every party updated $\log(n)$ times in the worst case (where everyone is initially compromised, the adversary can schedule all requests and also decide on the rule which of two concurrent updates "wins" in every case).

Moreover, CoCoA enjoys extremely 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 the number m of parties that update in this round, which can be as large as $m = n$. 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

[1] In the example from Figure 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.

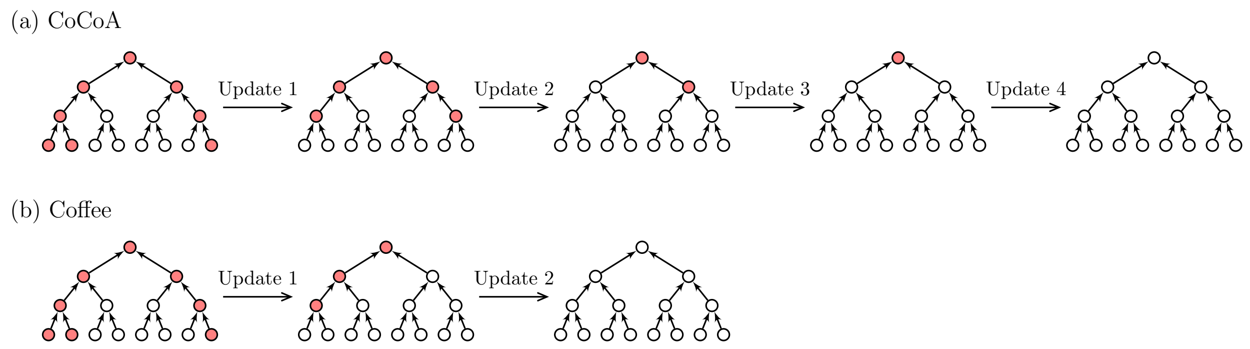

Figure 2: Comparison of the number of rounds required to recover in CoCoA (a) and Coffee (b) for n users, of which t are corrupted. Red nodes correspond to compromised keys. In each round all parties update concurrently, in CoCoA update requests are prioritized from left to right. CoCoA requires $\lceil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

tree, together with some commitment to its actions, allowing users to check if they received consistent messages.

1.3 Our Contribution

Coffee. In this work we consider a new CGKA protocol, Coffee (for COncurrent Fairly-Fast healing continuous group key agrEEment), that allows for concurrent updates. In Coffee we use a key-updatable PKE scheme, and updates no longer *replace* keys, but *update* them. While in CoCoA we must drop one of two concurrent updates for the same node, in Coffee we can perform them both. We prove that this has a significant implication on security. While in CoCoA we can only guarantee that the tree healed once each compromised party updated $\log(n)$ times, in Coffee that number drops to $\log(t)$ where t is the number of compromised users. This corresponds to the fairly-fast healing property reflected in the name (recall one could heal even faster following the P&C protocol, at the discussed cost). This is illustrated in Figure 2.

As we can expect t to be small compared to n (in fact, for most of the lifetime one should hope that $t = 0$), Coffee will provide comparable security to CoCoA with much fewer updates. On the downside, as in Coffee every user must process the update of every other user (while in CoCoA at most $\log(n)$ other updates will matter), the communication complexity (from server to users) will be larger in Coffee. A discussion of the efficiency of our protocol and its relationship to others can be found on Section 4.

Decentralized Group Messaging. The above discussion suggests a trade-off between Coffee and CoCoA, and which one is better will depend on the context. But there is one interesting setting where Coffee shines, namely in a decentralized setting where we do not want to rely on a(n intelligent) server to relay protocol messages between the users. Even though the server does not need to be trusted, it can still be problematic for various reasons.

We suggest to use a blockchain (permissioned or permissionless) to realize group messaging which enjoys the same robustness and security guarantees as the underlying blockchain. More concretely, instead of sending their protocol messages (update/add/remove) to the server, the users would post them on the blockchain. 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.

In particular, there are at least three separate properties which are achieved in the blockchain 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 denial-of-service-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, in the single server setting, the group can be shut down by taking out a single server. One can achieve better robustness with several servers, but then needs to solve the machine state replication problem. This is what our protocol does if using a permissioned blockchain. With a permissionless blockchain, robustness guarantees are 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. 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]

As a further observation, note that any CGKA in the classical setting can be "compiled" to the blockchain setting: in the blockchain setting, the block producer simply emulates the server to compile the message that would be broadcast in the classical setting, and adds this message (or hash) to the block. One further advantage over the classical server setting is that block producers could in this case check the validity of the received messages before including them in a block, which is something that a server is not generally able to do.

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

Maintaining a Group on Chain. 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 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.

1.4 Related Work

The primitive of Group Key Exchange (GKE) has been around for a long time, but it was not until recently, following the inception of the double ratchet protocol [PM16] and related work, where participants could achieve PCS, that the study of CGKA (with explicit PCS) started. The first construction of a CGKA is (implicitly) ART [CCG+18], adopted by the first version of MLS [BBM+20], which later switched to TreeKEM [BBR18]. In the last few years, different modifications of TreeKEM have been published, aiming to improve the original design along different angles. Notably, rTreeKEM [ACDT20] improves the FS guarantees from TreeKEM, achieving it after a single update by any party. Tainted TreeKEM [KPPW+21] introduces an alternative to blanking, called *tainting*, which reduces the efficiency drawback caused by dynamic operations in certain scenarios. Further, a recent paper by Hashimoto *et al.* [HKP+21] proposes the use of multi-recipient PKE in order to improve the download cost of users, at the cost of linear size commits. This primitive, in combination with reducible signatures, is also used by Alwen *et al.* in [AHKM21]. This work introduces server-aided CGKA as well as notable efficiency improvements by greatly reducing constant factors (though communication stays similar to TreeKEM's asymptotically). Further, Alwen *et al.* [ACDT21] formalize secure group messaging and casts it modularly in terms of the primitives CGKA, forward-secure group AEAD and PRF-PRNG.

In terms of security, the first security proof for any CGKA was for ART in [CCG+18]. Their proof has an exponential loss against *adaptive* adversaries. The first proof of adaptive security with sub-exponential loss (in fact, polynomial) for a variant of TreeKEM was given in [KPPW$^+$21]. While their security proof captured adversaries who can make adaptive choices, it did not capture fully active adversaries who can arbitrary deviate from the protocol specification and send malformed messages. Subsequent works [ACJM20] and [AJM20] propose stronger security models, allowing the adversary to set the random coins of parties, and to corrupt and impersonate them, respectively. Our model is closer to the one of [KPPW$^+$21], in that the adversary is not allowed to corrupt the parties signing keys, though we do not assume the existence of a server, which is replaced by a blockchain. We should also note two recent papers, the one by Devigne *et al.* [DDF21], achieving stronger robustness against malicious insiders by means of zero knowledge and verifiable encryption, and the SoK by Poettering *et al.* [PRSS21], surveying different GKE security models. Finally, formal analyses TreeKEM's security were carried out in [BCK21] and [BBN19].

Concurrency was initially approached in Causal TreeKEM [Mat19]. Later, it was more formally analyzed by Bienstock *et al.* [BDR20], who study the trade-off between PCS, concurrency and communication complexity, showing a lower bound for the latter and proposing a close to optimal protocol efficiency-wise, though in a synchronous, static-group model, and with much weaker PCS. A further paper by Weidner *et al.* [WKHB21] proposes a decentralized and concurrent protocol, at the cost of linear communication cost for updates.

In the multi-group setting, Cremers *et al.* [CHK21] study the PCS guarantees and Alwen *et al.* [AAB$^+$21] study more efficient solutions for groups with overlapping user sets.

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 [Coi].

Finally, updatable public key encryption is also an ingredient of rTreeKEM, which uses a somewhat different version, where decryption keys are updated upon being used. The paper by Jost *et al.* [JMM19] first introduced the primitive of secret key updatable public key encryption (skuPKE) in the context of two-party messaging. It is this primitive and syntax that we use here, albeit with different security requirements.

2 Preliminaries

2.1 Decentralized 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 $((pk_{id}, sk_{id}), (svk_{id}, ssk_{id}))$ that is 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 uploaded by id. In practice, this would be implemented by a key-server that allows users to deposit their and recover other users' key packets.

A baCGKA scheme baCGKA specifies algorithms $\mathsf{baCGKA.Init}$, $\mathsf{baCGKA.Upd}$, $\mathsf{baCGKA.Add}$, $\mathsf{baCGKA.Rem}$, $\mathsf{baCGKA.Proc}$, $\mathsf{baCGKA.Key}$, $\mathsf{baCGKA.Send}$, and $\mathsf{baCGKA.Fetch}$. 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.

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

Adding a group member. Member id can run $(id.st, A) \leftarrow \mathsf{baCGKA.Add}(id.st, id', pk_{id'})$ to add user id' to the group. 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 transaction, i.e. a protocol message M generated by one of the previous algorithms, user id runs algorithm $\mathsf{baCGKA.Send}(id.st, M)$.

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.

2.2 Secretly Key-Updatable Public-Key Encryption

We now recall the definition of secretly key-updatable public-key encryption (skuPKE) schemes [JMM19]. 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 *specifies algorithms* $(\mathsf{skuPKE.Gen}, \mathsf{skuPKE.Enc}, \mathsf{skuPKE.Dec}, \mathsf{skuPKE.Sam}, \mathsf{skuPKE.UpdP}, \mathsf{skuPKE.UpdS})$. *Key-generation algorithm* $\mathsf{skuPKE.Gen}$ *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 essentially 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. More precisely let $\lambda, k \in \mathbb{N}$, $(pk_0, sk_0) \in [\mathsf{skuPKE.Gen}(1^\lambda)]$, and $(\Delta_0, \dots, \Delta_k)$, $(\delta_0, \dots, \delta_k)$ with $(\Delta_i, \delta_i) \in [\mathsf{skuPKE.Sam}(1^\lambda)]$ for all i. Further, for $i \in \{0, \dots, k\}$ let $pk_{i+1} = \mathsf{skuPKE.UpdP}(pk_i, \Delta_i)$ and $sk_{i+1} = \mathsf{skuPKE.UpdS}(sk_i, \delta_i)$. We require that for all messages m and all i, $\mathsf{PKE.Dec}(sk_i, \mathsf{PKE.Enc}(pk_i, m)) = m$.

Security. 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. More precisely, we say that skuPKE is *secure* with respect to an upper bound L on the number of key updates, if it satisfies the following security guarantees:

Definition 2. *Let $(pk_0, sk_0) \leftarrow \mathsf{skuPKE.Gen}(1^\lambda)$ and also let $(\Delta_0, \dots, \Delta_{Q-1})$, $(\delta_0, \dots, \delta_{Q-1})$ with $(\Delta_i, \delta_i) \leftarrow \mathsf{skuPKE.Sam}(1^\lambda)$, and let s and s_i denote the random coins used by* skuPKE.Gen *and* skuPKE.Sam*, respectively. For $i \in [Q-1]_0$ define $pk_{i+1} = \mathsf{skuPKE.UpdP}(pk_i, \Delta_i)$ and $sk_{i+1} = \mathsf{skuPKE.UpdS}(sk_i, \delta_i)$. Then,* skuPKE *is* IND-CPA *secure, if for any choice ρ, j^-, j^+ with $-1 \leq j^- < \rho \leq j^+ \leq Q$ and messages m_0, m_1 it holds that*

$$\mathsf{skuPKE.Enc}(pk_\rho, m_0) \approx_c \mathsf{skuPKE.Enc}(pk_\rho, m_1)$$

even given access to $(pk_i)_{i \in [L]_0}$, $(sk_i)_{i \in [Q]_0 \setminus [j^- + 1, j^+]}$, $(\Delta_i)_{i \in [Q-1]_0}$, $(\delta_i)_{i \in [Q-1]_0 \setminus \{j^-, j^+\}}$, as well as random coins s if $j^- \geq 0$, and $(s_i)_{i \in [Q-1]_0 \setminus \{j^-, j^+\}}$.

Our variant of IND-CPA is incomparable than the one required for two party ratcheting [JMM19]; in this work the update information can be generated using adversarially chosen randomness, and the challenge ciphertext encrypts a message, that contains secret update information, giving the security notion a circular flavor. On the other hand, only one secret key is ever exposed to the adversary, while in our notion several are. Compared to [ACDT20] our security notion is stronger; in this work the authors use skuPKE mainly to achieve improved forward secrecy. Accordingly, their variant of IND-CPA roughly requires that access to updated secret keys does not allow to compromise encryption to previous keys, as long as the update information used to generate the corrupted key remains secure.

Instantiations. A very efficient instantiation of skuPKE can be constructed in prime order groups $(\mathbb{G}, g, p)$. The scheme is essentially the Hashed ElGamal scheme [ABR01], where update information is of the form $(\Delta = g^\delta, \delta)$ with $\delta \in \mathbb{Z}_p$ uniformly random, and key pairs $(X = g^x, x)$ are updated as $x + \delta$ and $X \cdot \Delta$ respectively. A formal description of the scheme and a proof of its security under the CDH-assumption in the ROM is given in Appendix 6.

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

$$v.res = \begin{cases} \{v\} & \text{if } v \text{ not blank,} \\ \emptyset & \text{elseif } v \text{ is a blank leaf,} \\ \bigcup_{v' \in v.par} v'.res & \text{else.} \end{cases}$$

3 Protocol description

We now describe Coffee in detail. Section 3.1 describes how the protocol proceeds in epochs determined by the blockchain's blocks, Section 3.2 describes the contents of a user's state, Section 3.3 how the structure of the ratchet tree is modified when handling changes to the group membership, and Section 3.4 how update information for a path in the ratchet tree is samped and applied. Finally, in Section 3.5 we give the formal description of the protocol's algorithms.

T	ratchet tree at the beginning of the current epoch
T_{next}	working copy of the ratchet tree for the next epoch
O_{next}	dynamic operations to be implemented before next epoch
U_{pending}	pending update
e_{ctr}	epoch counter
K	the epoch's group key
K_{next}	working copy of next epoch's group key
(pk_c, sk_c)	dummy key pair

Table 1: User *id*'s state.

3.1 Blocks and Epochs

Coffee proceeds in epochs consisting of k blocks. More precisely the ith epoch corresponds to blocks $i+1, \ldots, i+k$ 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 round. 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 another round 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 rounds is now dictated by the block arrival and typical inclusion times of the underlying blockchain, while FS is (marginally) affected by the confirmation time of blocks.

3.2 Users' States

User *id*'s state $id.st$ contains 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. For an overview see Table 3.2.

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 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.

3.3 Implementing Dynamic Operations

As a result of dynamic operations, the tree structure will change. In this subsection we describe this change, in preparation for the protocol description.

To add parties we use the *unmerged leaves* technique, introduced in TreeKEM v9. Note that a new user might not be able to receive the keys for all nodes in their path to 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 rounds. More in detail, whenever *id*, whose path shares a node with that of a new party id', generates an update in a follow-up round, 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 for this 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 in a subsequent epoch.

All these changes are executed once at the end of each epoch – though note that 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. This seems to be inherent if we want to allow for concurrency: the author of an operation concurrent with a dynamic one will be oblivious to the latter, thus not being able 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 current epoch (their exact definition is given in Section 3.4 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, 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. 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 needs to be added to accommodate for the new parties, this will be given the dummy key-pair until it gets updated at the end of the next epoch. Then, for each of the newly added parties 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 tree resulting from applying these changes.

Both blanks and the 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.

3.4 Updating the States of an Update Path

During the initialization of a group and when updating, users will frequently update the keys along some path. Before turning to the description of 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 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

$$z_j \in w_{i-1}.res \cup v_i.unm_1 \setminus v_{i-1}.unm_1$$

for $i = m+1, \ldots, r$ and

$$z_j \in (v_i.lpar).res \cup (v_i.rpar).res \cup v_i.unm_1 \setminus \{id\}$$

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 and thus has no associated public key, 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 the resolution of v_i^{n} 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.

3.5 Protocol Algorithms

Initialization. To initialize a group for users $(id_1, \ldots, id_n)$, user id_1 first generates the dummy key-pair $(pk_c, sk_c) \xleftarrow{\$} \mathsf{skuPKE.Gen}(1^\lambda)$. They then set up a left-balanced binary ratchet tree $T = (V, E)$, where the ith leaf corresponds to user id_i. T is completely blanked except for the leaves, that are set to have the corresponding user's initialization public key as associated key and further contain their signature verification key. Further, $v_{id_1}.stsec$ contains id_1's secret decryption and signing key. id_1 incorporates (pk_c, sk_c), T, a copy T_{next} of T, and an empty list O_{next} in their state and then computes $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathsf{gen\text{-}path\text{-}upd}(id_1.st)$. $((\Delta_i, \delta_i)_i, \kappa)$ is added to id_1's state together with epoch counter $e_{\text{ctr}} = 1$ and K_{next} is set to the zero string. The resulting welcome message is $W = (T.stpub, (\Delta_i, C_i)_i, (pk_c, sk_c), \sigma, id_1)$, where σ is a signature of $(T.stpub, (\Delta_i, C_i)_i, (pk_c, sk_c))$ under ssk_{id_1}.

Update. To issue an update, party 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 $z_j \in v_i.unm_1$. The update message is given by $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\text{ctr}}, \sigma, id)$, where σ is a signature of $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\text{ctr}})$ under ssk_{id}.

Add. User id generates an add message $\tilde{A} = (A, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}}, \sigma, id)$ to add user id'. It contains an add request $\tilde{A}$ = "$\texttt{add.user}(id', id)$", 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 $(A, T.stpub, (pk_c, sk_c), U, e_{\text{ctr}})$ under ssk_{id}, and the identity id.

Remove. In order to remove user id' from the group, user id generated $\tilde{R} = (R =$ "$\texttt{remove.user}(id', id)''$, $e_{\text{ctr}}, \sigma, id)$, a removal message , where σ is a signature of (R, e_{ctr}) under ssk_{id}.

Processing 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 how users who already processed a block since joining the group, process B, before turning to newly added users. In this case $B = (U, \tilde{A}, \tilde{R})$. User id first processes the update messages in the order 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 $i \in [u_1, \ldots, u_t]$ The user uses the corresponding secret key to recover $v_{u_i}.sk$ and adds it to the node's state $v_{u_i}.st$ in T and T_{next} unless the state already contains a secret key. Then id calls $id.st \leftarrow \textsf{proc-path-upd}(id.st, (\Delta_i, C_i)_i)$, 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 that 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 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 $T_{\text{next}} \leftarrow \textsf{upd-tree}(T_{\text{next}}, A, R, M)$ to be used in the next epoch, update $T \leftarrow T_{\text{next}}$, increase the epoch counter to $e_{\text{ctr}} \leftarrow e_{\text{ctr}} + 1$, set O_{next} to the empty list, and update the group key to $K \leftarrow \mathsf{H}_1(\text{“}\texttt{key}\text{”}, K_{\text{next}})$, and afterwards $K_{\text{next}} \leftarrow \mathsf{H}_1(\text{“}\texttt{next}\text{”}, K_{\text{next}})$.

In the case that id processes their first block since joining the group we consider the two cases (a) that they were added in an add operation or (b) in the group initialization. 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 $A = \text{“}\texttt{add.user}(id', id)\text{”}$ being the add request for user id. The user, after validating 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., $B = (W)$ 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 $K \leftarrow \mathsf{H}_1(\text{“}\texttt{key}\text{”}, K_{\text{next}})$. 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 $M \leftarrow \textsf{proc-path-upd}(id'.st, (\Delta_i, C_i)_i)$ to update T_{next}. K is set to $\mathsf{H}_1(\text{“}\texttt{key}\text{”}, K_{\text{next}})$, O_{next} is initialized as empty list, as there are no merge, add, or remove operations yet, and $e_{\text{ctr}} \leftarrow 1$.

Protocol	Concurrent	Rounds to heal	Sender comm. (cumulative)	Recipient comm. (per user)	Update cost after healing
TreeKEM I [BBR18]	No	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
TreeKEM II [BBR18]	Yes	2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Causal TreeKEM [Mat19]	Yes	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
Bienstock *et al.* [BDR20]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log(n))^*$
Weidner *et al.* [WKHB20]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
CoCoA [AAC+22]	Yes	$\lceil\log(n)\rceil+1$	$\mathcal{O}(n\log^2(n))$	$\mathcal{O}(\log^2(n))$	$\mathcal{O}(\log(n))$
Coffee (this work)	Yes	$\lfloor\log(t)\rfloor+1$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(\log(n))$

Table 2: 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). 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. * [BDR20] only achieves weak PCS, obtaining PCS guarantees similar to the rest would need $\mathcal{O}(n)$ cost after healing, due to extensive tainting.

Retrieving the Group Key. To extract the current group key, a user *id* fetches K from its state, and deletes this value afterwards.

Sending a Transaction. To send a protocol message, *id* simply uses the underlying blockchain protocol to send it as a transaction to the blockchain.

Fetching new Blocks. To download the last blocks of operations, *id* uses the underlying blockchain protocol to retrieve the blocks added to it since it last did.

4 Efficiency

In this section we discuss the efficiency of the protocol in healing a group with t compromises, and how it compares to related ones. Throughout we refer to Table 2. 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 big amount of blanks, as illustrated on the last column of Table 2. Thus, the data shown for the communication complexity of the latter mode of TreeKEM during healing is only short term.

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 rounds 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 that 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 rounds, and thus reduce the communication complexity accordingly; but also for TreeKEM II, [BDR20] and [WKHB21], for which the number of rounds 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 rounds. 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.

5 Security

5.1 Security model and safe predicate

To analyze the modified protocol, we essentially use the security model from [KPPW$^+$21], 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. Regarding 2) it is however possible that messages get lost and hence, even if a user generated an update it might not process this update.

Definition 3 (Asynchronous baCGKA Security)**.** *The security for baCGKA 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 initialises the group* G *with identities* $(id_1, \dots, id_{n'})$*. 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* **store-on-blockchain** *and* **process** *allow 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.*

1. **add-user**(id, id')*: a user id requests to add another user* id' *to the group.*
2. **remove-user**(id, 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. **store-on-blockchain**$(q_1, \dots, q_l)$*: for queries* $q_1, \dots, q_l$*, all of which must be actions of the form* $\mathsf{a}_i \in \{$**create-group**, **add-user**, **remove-user**, **update**$\}$ *by some users* id_i *(for* $i \in [l]$*), this action stores the outputs of the queries in the next block of the blockchain.*

5. **process**(ℓ', id)*: for* $(B_1, \dots, B_\ell) \leftarrow \mathsf{baCGKA.Fetch}(id.st)$ *and* $\ell' \in [\ell]$*, this action forwards all blocks* $B_1, \dots, B_{\ell'}$ *to id, who immediately processes them.*

6. **start-corrupt**(id)*: from now on the entire internal state and randomness of id is leaked to the adversary, with the exception of* ssk_{id}*.*

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

8. **challenge**(ℓ^*)*:* A *picks a block* B_{ℓ^*}*. Let* K_0 *denote the group key that is established by processing the first* ℓ^* *blocks* $B_1, \dots, B_{\ell^*}$ *in the blockchain and* K_1 *be a fresh random key; if there is no group key established after block* B_{ℓ^*}*,*[3] *then set* $K_0 = K_1 := \perp$*. 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 baCGKA scheme* (ϵ, t, Q)*-baCGKA-secure if for any adversary* A *making at most Q queries of the form* **update**$(\cdot)$ *and running in time t it holds*

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

We define the safe predicate to rule out all trivial winning strategies, such as challenging a block while some current group member is corrupted.

Definition 4 (Critical window, safe user)**.** *Let L be the length of the blockchain, C the number of users* A *corrupts throughout the security game, and* $\ell^* \in [L]$*. For user id, define* $q^-_{id} \in [Q]_0$ *to be maximal such that the following holds:*

- *There exist* $c := \lfloor \log(C) \rfloor + 1$ *blocks* $B_{\ell^1_{id}}, \dots, B_{\ell^c_{id}}$ *in distinct epochs within the first* ℓ^* *blocks in the blockchain such that each contains an update query* $\mathsf{a}^i_{id} :=$ **update**(id) $(i \in [c])$ *that*

  1. *was generated by id in or after query* q^-_{id}*,*
  2. *is* successful*, i.e. refers to block* $B_{\bar{\ell}^i_{id}}$ *with* $\bar{\ell}^i_{id} = \ell^i_{id} - (\ell^i_{id} \mod k)$*.*[4]

  If there do not exist c such blocks then we set $q^-_{id} = 0$*, the first query.*

- *There exists a block* $B_{\ell^-_{id}}$ *with* $\ell^-_{id} \leq \ell^*$ *that contains an update* $\mathsf{a}^-_{id} :=$ **update**(id) *for user id for which 1) and 2) hold, but the entire epoch does not contain any more successful updates for corrupted users. We call such an update a* single *update.*

[3]This could happen if the root of the tree is blanked, e.g. if no update was stored on the blockchain yet.

[4]Recall, by definition of the process operation in our protocol, condition 2) is necessary for the update a^i_{id} in block $B_{\ell^i_{id}}$ to be indeed processed by users processing block $B_{\ell^i_{id}}$.

Furthermore, let q_{id}^+ be the first query that invalidates id's current keys, i.e., in query q_{id}^+, id processes an initial block $B_{\ell_{id}^+}$ of some subsequent epoch[5] *(i.e. $\ell_{id}^+/k = \lfloor \ell_{id}^+/k \rfloor > \lfloor \ell^*/k \rfloor$) such that one of the blocks $B_{\ell^*+1}, \ldots, B_{\ell_{id}^+}$ contains an update $\mathsf{a}_{id}^+ := \mathbf{update}(id)$ referring to block $B_{\ell_{id}^+-k}$. If id does not process any such query then we set $q_{id}^+ = Q$, the last query.*
We say that the window $[q_{id}^-, q_{id}^+]$ is critical *for id with respect to challenge ℓ^*. Moreover, if the user id is* not corrupted *at any time point in the critical window, we say that id is* safe *w.r.t. ℓ^*.*

In Section 5.3 we discuss an strenghtening of this definition, that our protocol would also satisfy, but which we omit for now for the sake of simplicity. Similar to [KPPW$^+$21], we define a group key as *safe* if all the users in the group are individually safe, i.e., not corrupted in their critical windows.

Definition 5 (Safe predicate)**.** *Let K^* be a group key established by processing the first ℓ^* blocks of the blockchain and let G^* be the set of users which end up in the group after block B_{ℓ^*} was processed. Then the key K^* is considered* safe *if for all users $id \in G^*$ we have that id is safe w.r.t. ℓ^* (as per Definition 4).*

5.2 Security of the protocol

Theorem 1. *If the secretly key-updatable public key encryption scheme used in Coffee is $(\epsilon_{\text{Enc}}, t)$-IND-CPA-secure and the used hash functions are modeled as random oracles, then Coffee is $(O(\epsilon_{\text{Enc}} \cdot 2(nQ^2)^2), t, Q)$-baCGKA-secure.*

In order to prove Theorem 1, we first argue that a *safe* group key is not leaked to the adversary via corruption. We make this formal in the following definition and Lemma 2. In fact, we define leakage of arbitrary secret information which the adversary could potentially learn through corruption.

Definition 6 (Secure keys, update information, and seeds)**.** *For a seed s we say s is* leaked *if it is sampled by a user while this user is corrupted, or it is encrypted to the public key associated to a leaked secret key, or s was derived through $s := \mathsf{H}_1(s^-)$ and s^- is leaked.*
A key K_{next} that was derived through $K_{\text{next}} := K_{\text{next}}^- \oplus \kappa$ is leaked *if it is contained in a user's state while this user is corrupted, or K_{next}^- and κ are both leaked. If K_{next} was derived through $K_{\text{next}} := \mathsf{H}_1(\text{“}\texttt{next}\text{”}, K_{\text{next}}^-)$ then it is* leaked *if it is contained in a user's state while this user is corrupted, or K_{next}^- is leaked.*
A group key K that was derived through $K \leftarrow \mathsf{H}_1(\text{“}\texttt{key}\text{”}, K_{\text{next}})$ is leaked *if K is contained in a user's state while this user is corrupted, or K_{next} is leaked.*
Let δ be secret update information that was generated by first sampling a seed s, then computing $s' := \mathsf{H}_1^i(s)$ for some $i \in [\lceil \log(n) \rceil]_0$, and then computing $(\Delta, \delta) \leftarrow \mathsf{skuPKE.Sam}(\mathsf{H}_2(s'))$. The secret update information δ is leaked *if δ is contained in a user's state while this user*

[5] Recall, in order to be able to process messages in the current epoch, a user keeps the keys of the first round of the current epoch in its state and will only release these keys once it proceeded to the next epoch.

is corrupted, or s' is leaked.
The secret key sk_c of the dummy key pair (pk_c, sk_c) is always considered leaked. For a user's initial key pair (pk, sk), sk is leaked if sk was in the user's state while the user was corrupted. Let sk' be a secret key that was generated as $sk' \leftarrow \mathsf{skuPKE.UpdS}(sk, \delta)$. The key sk' is leaked *if sk' is contained in a user's state while this user is corrupted, or sk and δ are both leaked.*
A secret key/secret update information/seed is called secure *if it is not leaked. We say that a corruption of some user id does not leak key sk, if leakage of sk is independent of that corruption of id.*

Remark. *Note that the above definition only defines security for honestly generated secret keys/secret update information/seeds. This is enough for our purpose, since in our security model the adversary can only act through honest users. Furthermore, the definition might look circular at first sight; however, this is not the case since any seed associated with some node in the tree is only encrypted to keys that are associated with nodes* lower *in the tree.*

Lemma 2. *Assume there are no collisions among seeds, update information and keys throughout the security experiment. If a group key K^* is* safe *as per Definition 5 then it is secure as per Definition 6.*

In order to prove Lemma 2, we rely on the fact that the users who can derive the challenge key K^* are exactly those in G^*, where the set of group members G^* is defined to be the users for which either an **add-user**$(\cdot, id)$ operation was included in block $\ell^a \leq \ell^* - (\ell^* \mod k)$, or $id \in G$ for the initial group set up by **create-group**(G) (in which case we let $\ell^a = 0$); and such that no **remove-user**$(\cdot, id)$ was included in block ℓ^r, with $\ell^a + k - (\ell^a \mod k) \leq \ell^r \leq \ell^* - k - (\ell^* \mod k)$.

Note that, on the one hand, any operation included in a block and accepted by users must come from a user itself, as the adversary is not allowed to create messages itself. On the other hand, since all users share a common view of the blockchain, they will accept the same operations and have the same view of the group members set.

Lemma 3. *Assume there are no collisions among seeds, update information and keys throughout the security experiment. Then corruption of users not in G^* does not leak K^*.*

Proof. Assume K^* is leaked. We show that K^* must have been leaked through corruption of some user $id \in G^*$. By definition, either a user who had K^* in its state was corrupted or the key K^*_{next} used to derive K^* was leaked. In the first case, since all users share a common view of the blockchain and a user holding K^* must have processed the update in which K^* was generated, clearly this user must be in G^* and hence leakage of K^* is independent of any further corruptions of users outside G^*. Now, consider the second case. Similarly, a user holding K^*_{next} in its state must be in G^*, and the same is true for a user holding K^-_{next} if K^*_{next} was derived as $K^*_{\text{next}} := \mathsf{H}_1(\text{“}\texttt{next}\text{”}, K^-_{\text{next}})$. Hence we consider the case where K^* is leaked because for some K_{next}, which was derived as $K_{\text{next}} = K'_{\text{next}} \oplus \kappa$, both K'_{next} and κ were leaked.

Let $id \notin G^*$ and assume for contradiction that id during the game learns a seed that was used to derive κ. Clearly, since $id \notin G^*$, id cannot have produced κ itself. Let $\ell \leq \ell^*$ be the last block index such that $\ell \equiv 0 \mod k$, and let $\ell^- = \ell - k$. We must have that either no **add-user**$(\cdot, id)$ operation was included in any block before time ℓ, or that a block $\ell^r \leq \ell^-$ contained a **remove-user**$(\cdot, id)$ operation. Now, if there was never an **add-user**$(\cdot, id)$ before or at time ℓ (for convenience, here we count time in blocks on the blockchain), no seed was ever encrypted to an initkey of id at any time before ℓ. Moreover, if id is added to the group after ℓ, it will not be sent any key or new seed until it belongs to the set $v.unm_1$ for some v on the update path of the user generating κ, meaning that at least one update affecting the v took place after ℓ, thus updating its key at this time. Similarly, if such an operation was included in a block in $[\ell + 1, \ell^*]$ (if such an interval exists), id will still not receive any encryption by block ℓ^*, and will thus learn no seeds used to derive κ either.

Assume, thus, that id was removed in block ℓ^r. Since the group key K^* is generated w.r.t. time ℓ, there must have been an entire epoch between $[\ell^r, \ell]$ (the first following the epoch to which ℓ^r belongs to, and where any updates took place), where all new secret update information values were encrypted under keys outside the then blanked path of id. In particular, id cannot have learnt a seed that was used to derive κ.

This implies that κ was leaked through corruption of a user in G^* at a time when it did not yet process the update generating K^*. By correctness of the scheme, this user must be able to derive K'_{next}, hence K'_{next} is leaked through the same corruption and, hence, leakage of K^* is independent of any corruption of users outside G^*. $\square$

Proof (of Lemma 2). By Lemma 3 leakage of the challenge key K^* is independent of corruption of users outside G^*, hence we only have to consider users $id \in G^*$ in the following. Since the challenge group key K^* is safe, all users $id \in G^*$ are safe, i.e. not corrupted during their respective critical windows. This implies for every user $id \in G^*$ that 1) id is not corrupted during the current epoch; 2) either id was not corrupted before it processed B_{ℓ^*}, or id successfully updated in at least $c := \lfloor \log(C) \rfloor + 1$ epochs before the current one and after it's last corruption (where C denotes the number of corrupted parties), or id had a successful single update in some previous epoch; and 3) after it processed B_{ℓ^*}, either id was never corrupted again, or an update for id gets included into a block after B_{ℓ^*} and id processed the initial block of the subsequent epoch before it's next corruption started.

We will first argue that due to 3), corruption of safe users after they already processed B_{ℓ^*} does not leak the challenge key K^*. To this aim, note that through successfully updating and processing the initial block of the subsequent epoch, a user completely refreshes its state and, in particular, does not have any of the keys associated with the tree established in block B_{ℓ^*} or with any previous tree state in its state, neither does it have any seeds used to derive such keys in its state. Furthermore, all the seeds used to derive the keys in the tree established in B_{ℓ^*} were encrypted to tree states associated with blocks *before* block B_{ℓ^*}, and the seed used for the successful update was freshly sampled after processing block B_{ℓ^*} and deleted when processing the initial block of the subsequent epoch. On the other hand, if for some node on the update path the associated seed derived during such a successful update is leaked through another user, then also the key associated to that node in the beginning of the respective epoch is already leaked through that user. In other words, while leakage of some update information could allow an adversary who is given the new key to reverse that update and derive the old key, this old key is already leaked through the same corruption that leaked the update information. This proves that corruption of safe users after they processed B_{ℓ^*} does not leak K^*.

Now, consider a node v in the tree established in block B_{ℓ^*} and assume that every party under v, that was corrupted before it processed B_{ℓ^*}, since corruption ended successfully updated in at least i previous epochs or had a successful single update in some previous epoch, and furthermore every party under v, that was corrupted after it processed B_{ℓ^*}, successfully updated after it processed B_{ℓ^*} and processed the initial block of the subsequent epoch before its next corruption starts. We will show by induction on i that if the secret key, which is associated to v (resp. the challenge key in case v is the root) after block B_{ℓ^*} was processed, is leaked, then at least 2^i of the corrupted parties $\{id_1, \ldots, id_C\}$ have update paths through v. Since for $i = \lfloor \log(C) \rfloor + 1$ we have that $2^i > C$, it follows that the key associated to node v cannot be leaked. Hence, for $v = v_{root}$ we obtain that K^* is secure.

For the inductive argument, note that for $i = 0$ the statement is true since if the key associated to v is leaked there must be at least $1 = 2^0$ corrupted parties with an update path through v. Now, let $i \geq 1$ and assume that the statement holds for all integers smaller than i. Let l be the epoch in which the last of the corrupted parties with update paths through v updates for the ith time or had a successful single update. During this epoch, key sk_v at node v is replaced with $\mathsf{skuPKE.UpdS}(\ldots \mathsf{skuPKE.UpdS}(\mathsf{skuPKE.UpdS}(\mathsf{sk}_v, \delta_1), \delta_2) \ldots, \delta_J)$, where the rerandomization terms δ_j and s_j stem from the J parties which update node v during epoch l. The group key K, on the other hand, which is associated with the root of the tree, is derived as $\mathsf{H}_1(\text{“}\mathtt{key}\text{”}, K_{\text{next}})$ where K_{next} is replaced with $K_{\text{next}} \oplus \bigoplus_{j \in [J]} \kappa_j$. Note that in order for sk_v (resp. K^* if v is the root of the tree) to be leaked it is necessary that the adversary learns all δ_j (resp. κ_j), which implies that for all $j \in [J]$ the seed used to derive δ_j (resp. κ_j) is leaked, i.e. was either derived from a leaked seed, or encrypted to a leaked key. We consider the three cases that after epoch $l-1$ (a) there are at least two nodes v_1, v_2 in the resolution of the parents of v whose associated keys are leaked, (b) there is exactly one node v' in the resolution of the parents of v whose associated key is leaked and at least one update path in epoch l goes through v', and (c) there is exactly one node v' in the resolution of the parents of v whose associated key is leaked and all of the update paths of epoch l do not go through v'. Note that one of the cases has to occur since otherwise the key associated to v would be secure after epoch l.

Consider case (a). After epoch $l-1$, by minimality of l, it must hold that either 1) every corrupted party under v_1 and v_2 has updated in at least $i-1$ epochs or had a successful single update, or 2) all but one corrupted party under v_1 and v_2 has updated in at least i epochs or had a successful single update. In case 1), we obtain by the induction hypothesis that at least 2^{i-1} corrupted parties have update paths through v_1 and v_2 respectively. In turn there are at least 2^i corrupted parties under v. In case 2), we have that all corrupted users under v_b for some $b \in \{1, 2\}$ have successfully updated in at least i epochs preceding $l-1$ or had a successful single update before epoch $l-1$. Furthermore, the number of corrupted users below v_b is strictly smaller than the number of corrupted parties below v. We denote by l' the epoch in which the last of the corrupted parties with update paths through v_b updates for the ith time or had a successful single update and can now do the same case distinction for epoch l' and node v_b.

In case (b), for every update path of epoch l which goes through v' the seed used to derive the δ_j is encrypted to secure keys. Thus, in order for sk_v to be leaked it is necessary that the seeds used to derive the key associated to node v' were leaked as well. This implies that the key associated to v' is leaked even after epoch l. Thus we can set $l' \leftarrow l$ and make the same case distinction for v'.

Now consider case (c) and let v' be the only node in the resolution of the parents of v that has a leaked associated key. Node v' is not part of the update paths of epoch l. Thus, every corrupted party with update path through v' must have updated in at least i epochs before epoch l or had a successful single update before epoch l, and further by definition of l the number of such parties is strictly smaller than the number of corrupted parties below v. Analogous to above let l' denote the epoch in which the last corrupted party under v' updated for the ith time. We can now make the same case distinction as above.

Summing up, if case (a)1) occurs, then at least 2^i of the corrupted parties $\{id_1, \ldots, id_C\}$ have update paths trough v. If, on the other hand, cases (a)2), (b) or (c) occur, then there exist a parent v' of v and an epoch l' such that all corrupted parties under v' updated at least i times or had a single update, and the last to do so did in epoch l'. Note that repeated application of the case distinction reduces the height of node v' in the tree. Thus if we assume that case (a)1) never occurs, at some point we end up with a leaf node v' such that the associated key is leaked and the user associated with that leaf either was not corrupted or updated at least once since its last corruption; in both cases the associated key would be secure. Thus, at some point case (a)1) has to occur, which implies the desired statement. $\square$

Lemma 2 in place, the proof of Theorem 1 follows the security proof from [KPPW+21]. The main difference here is that we reduce baCGKA security of Coffee to the IND-CPA security of the underlying secretly key-updatable public-key encryption scheme skuPKE as per Definition 2 (opposed to IND-CPA security of a simple public-key encryption scheme as in [KPPW+21]). Looking into the details of our protocol, another difference is that the update information for the group key is derived by hashing a seed associated to the root of the challenge tree, but this update information is never encrypted (as opposed to [KPPW+21], where the seed is directly applied to derive the new group key); this slight modification in our current protocol will allow for quite some simplification of the proof from [KPPW+21].

Repeating the entire rather technical argument of [KPPW+21] would be outside the scope of this work; instead we give a high level overview on the proof of [KPPW+21] and discuss how the proof can be adapted.

Proof sketch (of Theorem 1). The main idea in [KPPW+21] is the following: If H_1 and H_2 are modeled as random oracles, then all the public-key pairs (pk, sk) sampled through $\mathsf{skuPKE.Gen}$ as well as the update information (Δ_i, δ_i) have the same distribution as if they were sampled independently (to ensure consistency, the random oracles can be programmed accordingly). Furthermore, by Lemma 2, the challenge key $K^* := \mathsf{H}_1(\text{“}\texttt{key}\text{”}, K_{\text{next}})$ is secure, i.e. K^* is not contained in a user's state while the user is corrupted and (the seed) K_{next} is secure.

Now, if the adversary never queries a secure seed to the random oracles H_1 and H_2, then the group key K^* is identically distributed to a uniformly random, independent string. Thus, any adversary that has advantage > 0 in breaking the security of Coffee must query the oracles H_1 or H_2 on some secure seed; we call this event E.[6] As long as E doesn't happen, every secure seed is information-theoretically hidden unless encrypted to some (secure) key. The idea for our (fully black-box) reduction R now is to embed an IND-CPA challenge (with two uniformly random seeds as messages) for skuPKE and hope that the query that makes E turn true will be the seed that was encrypted in the challenge ciphertext; when E turns true, the reduction stops the experiment. To see why this works, note that by Definition 6, for every secure key pair (pk^*, sk^*) there exist ρ, j^-, j^+ with $-1 \leq j^- < \rho \leq j^+ \leq Q$ such that

- (pk^*, sk^*) was derived by ρ times updating either some dummy key pair (pk_0, sk_0) or an init key of some user; we write $(pk_\rho, sk_\rho) := (pk^*, sk^*)$,
- secret keys $(sk_i)_{i\in[j^-+1,j^+]}$ as well as secret update information δ_{j^-}, δ_{j^+} are secure.

Now, as long as E does not happen, the secret update information δ_{j^-}, δ_{j^+} is identically distributed to freshly sampled, independent update information, hence, the reduction can indeed embed an IND-CPA challenge for skuPKE within the baCGKA security experiment.

To bound the security loss involved by our reduction, note that seeds associated to leaves are information-theoretically hidden unless compromised through corruption, and also the respective other message used in the IND-CPA security experiment is information-theoretically hidden as long as E did not happen[7]. Thus, except with negligible probability, whenever the reduction R correctly guessed ρ^*, j^-, j^+ and embedded the challenge key pair (pk_ρ, sk_ρ) of the skuPKE challenge and the two seeds at the right position in the challenge tree, then R succeeds in embedding its challenge and turning the adversary into an adversary against IND-CPA security of the skuPKE scheme. More precisely, before the game starts, R guesses uniformly at random the query q^* in which the seed s^* that makes event E turn true is generated. Furthermore, for the key pk^* to which s^* will be encrypted during the game, R guesses uniformly at random the position v^* in the tree as well as the number of updates ρ^* through which the key pair (pk^*, sk^*) was derived, as well as the indices j^-, j^+ for the skuPKE challenge. Thus, R succeeds with probability $1/(2nQ^4)$, and additionally taking into account unmerged leaves we end up with a security loss of roughly $2(nQ^2)^2$. $\square$

[6]In fact, this property of our scheme would allow us to prove security based on a weaker security assumption than IND-CPA security for skuPKE, where given an encryption of a random message the adversary has to compute the message.

[7]For simplicity of exposition, we ignore the issue of unmerged leaves here; the general case including unmerged leaves and therefore multiple encryptions of the same seed follows by a hybrid argument, losing another multiplicative factor n in security.

5.3 A stronger safe predicate

The safe predicate in the section above, or, in particular, the definition of critical window, is written with respect to the users corrupted by A since the beginning of the security game. Here, we will briefly argue that, while we presented it like this for simplicity, in practice one would want to consider a stronger version, that takes into account the users corrupted only from the last time a group key was safe. We will argue that such a strenghthening follows easily, if only at the cost of a more convoluted presentation.

Example: A safe group key not covered by the safe predicate. First, to see why the predicate defined above (Definitions 4 and 5) is suboptimal, observe that by defining it in such a fashion, we exclude several situations where a key is safe (but would be marked as unsafe by said predicate). This is because it ignores the possibility of healing at some point throughout the game execution, some time before the challenge query. For instance, consider the game execution where the adversary corrupts every user at some point, but does so by corrupting users two by two, in order from left to right, say. Further, A ends each pair of corruptions before starting the next and, moreover, in between each pair of corruptions, A has the last two corrupted users, concurrently, issue two updates each, thus healing their state. I.e., A first corrupts id_1 and id_2, ends the corruption of both of them, makes them issue updates q_1, q_2 respectively, calls **store-on-blockchain**(q_1, q_2), makes both users process this last block, then issue new updates q_1', q_2', and then process the block resulting from **store-on-blockchain**(q_1', q_2'). Done that, then A corrupts id_3 and id_4, stops the corruption, and proceeds in the same fashion as before, making these two users update twice, before corrupting id_5 and id_6, and so on. In this execution of the game, it is clear that the group key will be secure every time a pair of users execute their pair of concurrent updates. However, from the time the adversary has corrupted 4 or more users, the predicate above will consider any future group key insecure, as $C \geq 4$ corruptions would require either $c \geq 3$ concurrent updates or a single update, from each corrupted user. Since each user only ever updates twice, and those updates are concurrent, the safe predicate will indeed never be satisfied.

A stronger safe predicate. This issue, however, can be solved rather easily by introducing a slightly modified, recursive definition of the safe predicate $\mathit{safe}(\ell^*)$ associated to block ℓ^* (equivalently, to its corresponding epoch). For this, to ℓ^* we associate $\ell^-(\ell^*) < \ell^*$, the last block before ℓ^* that satisfied $\mathit{safe}(\ell^-)$, where we set $\ell^-(\ell^*) = 0$ if no such block before ℓ^* exists. Now, *safe* can be defined as in Section 5.1, the only difference being that in Definition 4 the number of corrupted users $C(\ell^*)$ is defined as the number of users A corrupts between $\ell^-(\ell^*)$ and ℓ^* (instead of the number of all users corrupted up to ℓ^*).

In order to see that the proof would carry over to this new predicate, note that we would only need to ensure that Lemma 2 still holds. Namely, that if the stronger safe predicate holds for key K^*, then K^* is not leaked. This can indeed be showed through an inductive argument on the sequence of secure epochs. Note that the base case, i.e. $\ell^-(\ell^*) = 0$ corresponds to the already existing predicate and is taken care of by the current proof. For the inductive step, one would need to show that key K^* is secure (as per Definition 6) given that the group key defined by $\ell^-(\ell^*)$ is secure. This follows from the existing proof together with two observations, which we will briefly argue in the paragraphs below. On the one hand, the fact that the ratchet tree defined by processing blocks up to the safe one $\ell^-(\ell^*)$ exclusively contains keys that have not leaked. On the other, the fact that if a seed set by any update included in any block after $\ell^-(\ell^*)$ is encrypted under a key pk belonging to a tree associated to some block $\tilde{\ell} < \ell^-(\ell^*)$, then pk also belongs to the tree associated to $\ell^-(\ell^*)$. These two observations ensure that the leakage of any key generated during the period between $\ell^-(\ell^*)$ and ℓ^* can be traced back to a corruption taking place during that same period. This, in turn, allows to use esentially the same proof of Lemma 2 to argue for the inductive step.

To see why the first observation is true, one can look at the simpler case: if u and v are two nodes in the ratchet tree, with u being the child of v, then it is not possible for the secret key at v to be leaked, while the secret key for u is secure (since, by assumption, the group key at $\ell^-(\ell^*)$ is secure, the statement follows). Indeed, let sk_v be leaked and q_v be the time at which A first learnt the value of a secret key at v (and such that from q_v to the present there was no time when A did not have knowledge of the secret key at v). At this time, A must have learnt this key through a corruption, and so must have also learnt the secret key at u at the time. However, since A has knowledge of the key at v throughout the interval from q_v to the time sk_v was set, they, in particular, must also have learnt all seeds used to derive secret update informations updating the key at v during that time. Consider now the different secret update informations evolving the key at u. Any such δ that comes from an update by a user below v is derived from a seed, itself derived by a hash evaluation of a seed that A learnt. For the other δ coming from the other sub-tree under u, the corresponding seed gets encrypted to a key at v, which A also knows, by assumption. This shows that A would also know the key at u, i.e. it is leaked.

The second observation follows easily from the consistency properties that the blockchain ensures, in particular the agreement of all users on the transcript of the execution so far. Indeed, for the statement of the above observation to not be true, an update consistent with the transcript so far up to some block $\hat{\ell} \leq \tilde{\ell}$ would have needed to be included and processed by users in some block between $\tilde{\ell}$ and $\ell^-(\ell^*)$, which is not possible.

References

[AAB+21] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 222–253. Springer, 2021.

[AAC+22] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent Continuous Group Key Agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022*, Lecture Notes in Computer Science. Springer, To appear, 2022. `https://ia.cr/2022/251`.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, April 2001.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 1463–1483. ACM, 2021.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.

[AHKM21] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. Cryptology ePrint Archive, Report 2021/1456, 2021. `https://ia.cr/2021/1456`.

[AJM20] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Report 2020/1327, 2020. `https://eprint.iacr.org/2020/1327`.

[BBM+20] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-09, Internet Engineering Task Force, March 2020. Work in Progress.

[BBN19] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. May 2018.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. `https://eprint.iacr.org/2021/137`.

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.

[CCG$^+$18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why Cross-Group effects matter. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1847–1864. USENIX Association, August 2021.

[Coi] XX Coin. Elixxir architecture brief v2.0. `https://xx.network/elixxir-architecture-brief-v1.0.pdf`.

[DDF21] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. Mls group messaging: How zero-knowledge can secure updates. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security – ESORICS 2021*, pages 587–607, Cham, 2021. Springer International Publishing.

[HKP$^+$21] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient pkes. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1441–1462, New York, NY, USA, 2021. Association for Computing Machinery.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.

[KPPW$^+$21] K. Klein, G. Pascual-Perez, M. Walter, C. Kamath, M. Capretto, M. Cueto, I. Markov, M. Yeo, J. Alwen, and K. Pietrzak. Keep the Dirt: Tainted

TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[Mat19] Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.

[PM16] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 148–176. Springer, Heidelberg, May 2021.

[WKHB20] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. Cryptology ePrint Archive, Report 2020/1281, 2020. `https://eprint.iacr.org/2020/1281`.

[WKHB21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2024–2045, New York, NY, USA, 2021. Association for Computing Machinery.

6 Appendix: skuPKE from Hashed ElGamal

In this appendix we give a proof of the IND-CPA security (in our model) of the secretly key-updatable public-key encryption scheme presented in [JMM19]. The construction is based on the Hashed ElGamal scheme and we recall it here.

The key-generation, encryption and decryption algorithms work as in the Hashed ElGamal scheme. That is, $\mathsf{skuPKE.Gen}(1^\lambda)$ outputs a pair $(pk, sk) = ((\mathbb{G}, p, g, g^x, \mathsf{H}), (\mathbb{G}, p, g, x, \mathsf{H}))$, where $\mathbb{G}$ is a group of prime order p (the bit length of p is λ), g is a generator of $\mathbb{G}$, x is sampled at random from $\mathbb{Z}_p$ and H is a hash function that takes elements in $\mathbb{G}$ as input and outputs strings in $\{0,1\}^\lambda$. An encryption of a message $m \in \{0,1\}^\lambda$ using the public key g^x is a pair $(g^y, \mathsf{H}((g^x)^y) \oplus m)$ where y is sampled at random from $\mathbb{Z}_p$. The decryption algorithm takes as input a ciphertext (c_1, c_2) and a private key x and outputs $\mathsf{H}((c_1)^x) \oplus c_2$.

The sampling algorithm $\mathsf{skuPKE.Sam}(1^\lambda)$ outputs a pair $(\Delta = g^\delta, \delta)$ where δ is sampled from the uniform distribution over $\mathbb{Z}_p$. Public-key-update algorithm $\mathsf{skuPKE.UpdP}$ gets as input (g^x, Δ) and outputs $g^x\Delta$, while $\mathsf{skuPKE.UpdS}$ takes (x, δ) as input and outputs $x + \delta$.

The security proof is based on a standard IND-CPA security proof of Hashed ElGamal like the one that can be found on textbooks and it is provided for completeness. It relies on the hardness of the computational Diffie-Hellman (CDH) problem and uses the random oracle model.

We say that the CDH problem is hard with respect to $\mathsf{skuPKE.Gen}$ if for every PPT algorithm A there exists a negligible function $\epsilon(n)$ such that $\Pr[\mathsf{A}(\mathbb{G}, q, g, g^x, g^y) = g^{xy}] \leq \epsilon(n)$ where the probabilities are taking over the randomness used by $\mathsf{skuPKE.Gen}$ to generate $(\mathbb{G}, q, g)$ and x and y are sampled uniformly from $\mathbb{G}$.

Theorem 4. *If the CDH problem is hard with respect to* $\mathsf{skuPKE.Gen}$ *and* H *is modeled as a random oracle, the Hashed ElGamal skuPKE scheme is IND-CPA secure.*

Proof. Let ρ, j^-, j^+ be a set of indices such that $-1 \leq j^- < \rho \leq j^+ \leq L$ and A be a PPT adversary trying to distinguish

$$\mathsf{skuPKE.Enc}(pk_\rho, m_0) \approx_c \mathsf{skuPKE.Enc}(pk_\rho, m_1)$$

as in Definition 2.

Let $(g^y, \mathsf{H}((pk_\rho)^y) \oplus m_b)$ denote a ciphertext. As the hash function is modeled as a random oracle, A cannot distinguish the ciphertexts with probability greater than 1/2 unless it makes a query to the random oracle on $(pk_\rho)^y$. Let E denote the event that such a query is made. Therefore the probability that A is able to distinguish the two distributions is bounded by $1/2 + \Pr[E]$.

We now show that $\Pr[E]$ is negligible. We define an algorithm B that takes as input a CDH challenge $(\mathbb{G}, p, g, g^x, g^y)$ and uses A as a subroutine. It samples $b \leftarrow \{0, 1\}$ and $(\Delta_i, \delta_i) \leftarrow \mathsf{skuPKE.Sam}(1^\lambda)$ for $i \in \{0, \ldots, j^- - 1\} \cup \{j^- + 1, \ldots, j^+ - 1\} \cup \{j^+ + 1, \ldots, L - 1\}$. It chooses g^x as the ρ-th public key, $(pk_{j^-}, sk_{j^-}) = (g^{r^-}, r^-)$ and $(pk_{j^+ + 1}, sk_{j^+ + 1}) = (g^{r^+}, r^+)$ where r^-, r^+ are uniformly chosen in $\mathbb{Z}_p$. It computes $\Delta_{j^-} = g^x (\prod_{i=j^-+1}^{\rho-1} \Delta_i)^{-1} g^{-r^-}$ and $\Delta_{j^+} = g^{-x} (\prod_{i=\rho}^{j^+-1} \Delta_i)^{-1} g^{r^+}$. The remaining public and private keys are chosen accordingly, that is,

$$\begin{aligned}
pk_i &= pk_{i+1} \cdot \Delta_i^{-1} && \text{for } i \in \{\rho - 1, \ldots, 0\} \\
pk_i &= pk_{i-1} \cdot \Delta_{i-1} && \text{for } i \in \{\rho + 1, \ldots, L\} \\
sk_i &= sk_{i+1} - \delta_i && \text{for } i \in \{j^- - 1, \ldots, L\} \\
sk_i &= sk_{i-1} + \delta_{i-1} && \text{for } i \in \{j^+ + 2, \ldots, L\}
\end{aligned}$$

Then B sends to A $(pk_i)_{i \in [L]_0}$, $(sk_i)_{i \in [L]_0 \setminus [j^- + 1, j^+]}$, $(\Delta_i)_{i \in [L-1]_0}$, $(\delta_i)_{i \in [L-1]_0 \setminus \{j^-, j^+\}}$ as well as the random coins used by $\mathsf{skuPKE.Gen}$ and $\mathsf{skuPKE.Sam}$ as specified in Definition 2.

As an observation, B can actually compute those secret keys because it first chooses sk_{j^-} and $sk_{j^+ + 1}$, and then it proceeds recursively using the δ_i that it sampled before. The construction also guarantees that the pairs (pk_i, sk_i) satisfy $g^{sk_i} = pk_i$.

When A makes a random oracle query $u \in \mathbb{G}$, B sends a random string s_u and keeps a list of pairs (u, s_u). When A sends two messages m_0, m_1, B replies with a ciphertext $(g^y, k \oplus m_b)$ where k is sampled uniformly at random.

Finally, B chooses a random pair in the list of random oracle queries A made and outputs the first component.

Since the view of A as an IND-CPA adversary and when run as a subroutine of B before it makes a query to the random oracle on $(pk_\rho)^y$ is the same, the probability that E happens is the same in both cases. This is because if A does not make said query then B perfectly simulates the IND-CPA game. Let Q denote the number of random oracle queries. By construction, when A is run as a subroutine of B, $\Pr[E]/Q \leq \Pr[\mathsf{B}(\mathbb{G}, q, g, g^x, g^y) = g^{xy}] \leq \epsilon(\lambda)$ for some negligible function by hypothesis. Hence the probability that A is able to distinguish the two distributions is bounded by $1/2 + Q \cdot \epsilon(\lambda)$, i.e., the Hashed ElGamal skuPKE scheme is IND-CPA secure. $\square$