Continuous Group-Key Agreement: Concurrent Updates without Pruning

Benedikt Auerbach* [1], Miguel Cueto Noval[2], Boran Erol†[3], and Krzysztof Pietrzak[2]

[1]PQShield
benedikt.auerbach@pqshield.com
[2]ISTA, Klosterneuburg, Austria
{mcuetono, pietrzak}@ista.ac.at
[3]UCLA, Los Angeles, USA
boranerol03@gmail.com

June 3, 2025

Abstract

Continuous Group Key Agreement (CGKA) is the primitive underlying secure group messaging. It allows a large group of N users to maintain a shared secret key that is frequently rotated by the group members in order to achieve forward secrecy and post compromise security. The group messaging scheme Messaging Layer Security (MLS) standardized by the IETF makes use of a CGKA called TreeKEM which arranges the N group members in a binary tree. Here, each node is associated with a public-key, each user is assigned one of the leaves, and a user knows the corresponding secret keys from their leaf to the root. To update the key material known to them, a user must just replace keys at $\log(N)$ nodes, which requires them to create and upload $\log(N)$ ciphertexts. Such updates must be processed sequentially by all users, which for large groups is impractical. To allow for concurrent updates, TreeKEM uses the "propose and commit" paradigm, where multiple users can concurrently propose to update (by just sampling a fresh leaf key), and a single user can then commit to all proposals at once.

Unfortunately, this process destroys the binary tree structure as the tree gets pruned and some nodes must be "blanked" at the cost of increasing the in-degree of others, which makes the commit operation, as well as, future commits more costly. In the worst case, the update cost (in terms of uploaded ciphertexts) per user can grow from $\log(N)$ to $\Omega(N)$.

In this work we provide two main contributions. First, we show that MLS' communication complexity is bad not only in the worst case but also if the proposers and committers are chosen at random: even if there's just one update proposal for every commit the expected cost is already over $\sqrt{N}$, and it approaches N as this ratio changes towards more proposals.

Our second contribution is a new variant of propose and commit for TreeKEM which for moderate amounts of update proposals per commit provably achieves an update cost of $\Theta(\log(N))$ assuming the proposers and committers are chosen at random.

*Benedikt Auerbach conducted part of this work at ISTA.
†Boran Erol conducted part of this work at ISTA.

Contents

1 Introduction

Messaging layer security. In recent years asynchronous secure messaging applications like the WhatsApp, Signal and iMessage have become an important part of every day life with user numbers ranging in the billions. While deployment of these applications has been a great success for practical cryptography, the used protocols do not scale well when used to secure the communication of large groups. For example, Signal limits the group size to 1000 users. Providing efficient secure messaging for large group sizes is the explicit goal of the IETF's *Messaging Layer Security* (MLS) Standard [BBR+23] of 2023. The standard was developed in a joined effort of industry players and the scientific community and aims to enable asynchronous and end-to-end encrypted messaging via an untrusted server, admitting group sizes up to 50.000.

TreeKEM [BBR18], the protocol at MLS core is a so called *continuous group-key agreement* [ACDT20] (CGKA). As a generalization of continuous key agreement [ACD19] to the group setting, it allows a group of users to agree on a shared, evolving secret that intuitively will be used to secure the communication within the group. CGKA is supposed to provide both forward secrecy (FS), meaning that compromising a user does not reveal past messages, and post-compromise security (PCS), i.e., that the group is able to recover from compromise by sampling new key material and exchanging protocol messages. MLS has been taken up with great interest by the community resulting in a vast amount of works analyzing its security [KPPW+21, ACJM20, CHK21, BCK21, AJM22, WPBB23, CGWZ25] or proving lower bounds on its communication complexity [BDR20, BDG+22, ANPPP23, AAB+24]. Further works develop protocol variants making use of an active server [HKP+21, AHKM22, AAN+22] (no longer simply forwarding protocol messages), being tailored to multiple overlapping groups [AAB+21], or enabling resilience against forking of users' protocol views [AMT23]. Yet another line of works is concerned with the security of alternative, not TreeKEM based, group-messaging protocols [WKHB21, BCG23, CEST24] or analyzes additional properties of CGKA like secure administration [BCV23] or meta data protection [HKP22].

Ratchet trees. While FS can be achieved essentially by means of symmetric cryptography, PCS requires the use of public key cryptography. TreeKEM makes use of so called ratchet trees, that can be seen as a public-key analog of logical key hierarchies [WHA99]. At a high level, a ratchet tree is a balanced binary tree in which each node has an associated public-key encryption (PKE) key-pair. Each group member is assigned to a leaf, the corresponding key-pair can be thought of as the user's personal key, and the one of the root as the shared group secret. To enable PCS the users will periodically rotate the secret key material known to them by exchanging protocol messages. While all users have a complete view on all public keys, throughout the execution of the protocol TreeKEM enforces the so-called *tree invariant* stating that every user has access to exactly the secret keys of nodes lying on their update path, i.e., the path from their leaf to the root.

To update their key material a user u replaces the keys of nodes on their update path by sampling a fresh key for their leaf, and deriving the remaining keys in a deterministic way from this key. Then every fresh secret key sk is encrypted under the public key of the node's co-child, i.e., the child not being part of u's update path. Now all users whose update path merges into u's at the node in question are able to recover sk by decrypting the ciphertext. They can then derive the remaining secret keys up to the root in a deterministic way. Note that in total u has to generate only logarithmically many ciphertexts (in the group size N), making the protocol particularly suited for large groups.

Proposals and Commits. TreeKEM, as described above, requires all updates to be executed in order (i.e., a user can only issue an update after it processed all previous updates), which might not be possible in a large group with frequent updates. To allow for concurrent issuing of protocol messages TreeKEM distinguishes between *proposal messages* and *commit messages*. In the former, a user issuing an update proposal indicates their goal of updating the key material known to them. However, the message only contains a new personal leaf public key and, in particular, does not establish a new shared group secret. Similarly, proposals of adding users to or removing users from the group are handled using proposal messages.

To establish a new group secret and move the group's state to the next epoch a user can issue a commit

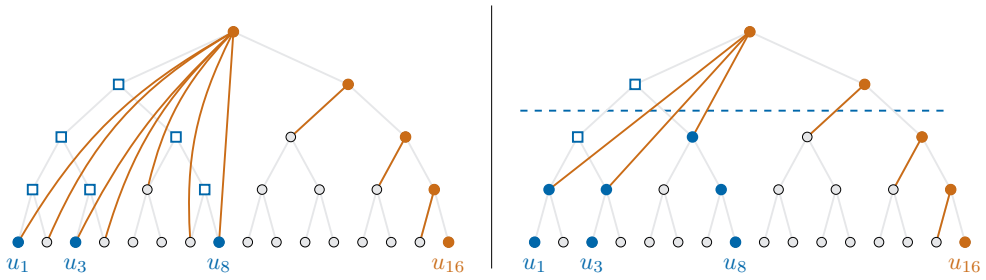


Figure 1: Working principle of update proposals and commits in MLS (left) and MLS-Cutoff (right). In both cases for a fully populated tree users u_1, u_3, u_8 issue an update proposal (depicted in blue) followed by a commit by user u_{16} (depicted in orange). Nodes being blanked are depicted as (empty) squares, and ciphertexts created as part of the commit with orange edges. In the depiction of MLS-Cutoff the blue dashed line marks the cutoff bound.

to a (potentially empty) set of proposal messages. Essentially, this implements all proposals and updates all keys on the committing user u's update path. Note that to preserve the tree-invariant u cannot refresh the keys on a proposing users' update paths unless they also lie on their own update path (as u must not learn the secret key of such nodes). Moreover, keys on u's co-path might no longer be considered secure, since the corresponding secret key might lie on the update path of a user issuing a proposal. Thus, TreeKEM handles update proposal by means of *blanking*. I.e., the key-pairs of all (non-leaf) nodes on the update path of proposing users which do not lie on the update path of the committing user are deleted from the ratchet tree. This has the effect of increasing the indegree of some nodes, and thus the commit operation as well as future commits require additional ciphertexts, negatively affecting the protocol's communication complexity. Removals of users from the group is also handled by means of blanking, while adding users employs the unmerged leaves technique. For a depiction of proposals and commits in MLS see Figure 1 (left).

Known results on the communication complexity of MLS. As discussed above, for a group of size N in the best case (in particular if no update proposals are issued at all) the size of a commit message is of order $O(\log(N))$. It is easy to see that in the worst case essentially the whole ratchet tree can end up blank, leading to a linear worst case communication complexity of order $\Omega(N)$. Unfortunately, the latter turns out to be inherent for CGKA schemes built from 'natural' primitives achieving fast PCS [BDR20, BDG+22, ANPPP23] (i.e., within a single propose-commit round). Accordingly, while different methods of implementing update proposals have been explored, they either exhibit the same worst-case behavior [KPPW+21] or suffer from achieving PCS slower [Wei19, AAN+22, AACN+24].

While the worst case performance of TreeKEM is a far shot from the advertised $\log(N)$, the known sequences causing it are very artificial. Thus, one might still hope that TreeKEM and in turn MLS perform well for 'natural' random sequences of operations, which would be sufficient for practical applications. However, up to our knowledge there has been very little analysis in this setting so far.

1.1 Our Contributions

The contribution of this work is twofold. We initialize the formal study of MLS's communication complexity for random sequences of operations, which models the lack of coordination between users given the asynchronous nature of CGKA schemes. To this end, we derive lower bounds on the size of its commit messages in an easy to understand model letting randomly chosen users issue update proposals and commits. Our results indicate that, unfortunately, even very small amounts of update proposals per commit operation have a substantial negative impact on MLS's communication complexity.

On the positive side, we propose an alternative way of handling update proposals that provably achieves a logarithmic communication complexity in our model. The changes required to handling proposals are not

too complex, which in our view is a very positive feature, as our proposed method is compatible with the complex mechanisms employed by MLS to ensure authenticity and consistency.

Lower bounds on MLS's Average Commit Size. Our main goal is to gain an understanding of the impact that blanking nodes in the ratchet tree due to update proposals has on the size of commit messages, for randomly generated sequences of operations. Ideally, one would analyze the communication complexity of MLS using real-world user data. As such data is unfortunately not available, we opt for a simple model of communication. The advantage of this being that it, on the one hand, makes the problem of deriving formal bounds approachable, and, on the other, leads to results that are understandable on an intuitive level.

Our experiment first initializes a group of size N. Then it repeatedly starts issuing update proposals and commits for rounds t as follows.

1. A set U_P consisting of $P \in \mathbb{N}$ group members is sampled uniformly at random.
2. All users in U_P issue an update proposal.
3. A user sampled uniformly at random issues a commit to all update proposals.
4. The size $\mathsf{Cost}(t)$ of the commit is recorded.

To also be able to handle the setting where update proposals are only issued every other round, the experiment is additionally parameterized by $C \in \mathbb{N}$. In the case $C > 1$, the commit to the P proposals is followed by $C - 1$ rounds of commits not implementing any proposals. We are interested in the expectation of $\mathsf{Cost}(t)$ after running the experiment for sufficiently many rounds. Informally, our main result regarding this is the following.

> **Theorem 4.2** (informal). *Let P, C be constant. Then the expected size of a commit generated according to the sequence described above is of order*
>
> $$\mathbb{E}[\mathsf{Cost}(t)] \geq \Omega(N^{\log_2(1+\frac{P}{P+C})}).$$
>
> Moreover, if $P = 1 = C$, it holds that
>
> $$\mathbb{E}[\mathsf{Cost}(t)] \geq \log^2(N)/4 - \log(N)/4,$$
>
> which is a more relevant bound for small values of N.

We provide some approximate values of the exponent $e = \log_2(1 + \frac{P}{P+C})$ in the table below. We use $C = 1$ for $P \geq 1$ and for ease of notation write $P = 1/c$ in the case $P = 1$, $C = c$.

P	$1/5$	$1/2$	1	2	5	10	50
e	0.22	0.42	0.58	0.74	0.87	0.93	0.99

Note that already if a single update proposal is issued per commit the MLS's communication complexity devolves to about $\sqrt{N}$, a far shot from $\log(N)$. Further, if the number of proposals increases the cost of commits approaches the worst case of a linear commit cost quite fast, with an exponent of 0.99 for $P = 50$.

This means that one either must perform almost all updates sequentially, or commits become almost linear in N. As MLS aims to support groups containing up to $N = 50000$ members, neither of these options is very appealing.

Before turning to our alternative proposal of handling update proposals, we give a bit of intuition on the lower bound's proof. As the cost of a commit is mainly determined by the blanks in the ratchet tree we first investigate the expected probability of a fixed node v being blank. We notice that the transition of v's state between blank and populated forms a Markov chain, allowing us to determine its stationary distribution. We obtain that the probability of v being blank, assuming v is of sufficient depth in the tree, is roughly $P/(P + C)$.

In MLS if a user's copath contains a node v, then how much v contributes to the communication complexity of a commit by that user depends on whether v is blank. If v is populated, then encrypting to it

only requires one ciphertext. If it is blank, a commit in MLS would include ciphertexts encrypted under the keys of its children and, due to the way MLS works, at least one child must be blank as well meaning that those ciphertexts would have to be encrypted under some of v's grandchildren and so on. Intuitively, this means that the number of ciphertexts needed to encrypt to a blank node v is $1 + P/(P + C)$ times the number of ciphertexts needed to encrypt to a child of v provided that both v and its child are blank, where 1 corresponds to the child that must be blank and $P/(P + C)$ to the probability that the other child is blank. This argument assumes independence (between the events that the left child of v is blank and the right child of v is blank), which is not justified. However, we show that in expectation the number of ciphertexts does indeed increase by a factor of $1 + P/(P + C)$.

An alternate way of performing update proposals. It is known that if too many users, say, for example, linearly many in the group size, issue concurrent update operations there is no possibility for efficient CGKA. In fact, Bienstock et al. [BDR20] prove that in this case the communication complexity must be linear in the group size. While the lower bound discussed above indicates that even a small number of update proposals per commit operation has a devastating effect on MLS's communication complexity, it leaves open the possibility of an alternative CGKA being more viable in this setting. Recall that update proposals introduce blanks in the tree, as all keys on the updating users' update paths are removed. A natural idea to prevent introducing blanks in the tree would be to not only have the user issuing a commit, but also updating users to re-key their complete update path. In fact this is the approach taken by the continuous group-key agreement scheme CoCoA [AAN+22] which in this setting has logarithmic communication complexity. However, proposals can be issued concurrently, meaning users make concurrent modifications to the ratchet tree. This leads to users encrypting new secrets to outdated keys, i.e., keys concurrently being replaced, and, as a second point, is not compatible with the mechanisms used by MLS to ensure integrity of the ratchet tree. As a consequence, CoCoA does achieve PCS slower than MLS, and does not achieve the same strong security notion of insider security.

The second main contribution of this work is a CGKA we denote MLS-Cutoff with the following features.

- For a constant number P of update proposals per commit operation and random operations as described above, the size of its proposal and commit messages is bounded by $O(\log(N))$
- It achieves the same strong security notion as the MLS protocol, namely insider security [AJM22] for the same safety predicate. In a nutshell, the latter says that if all compromised parties issue update proposals that are in turn committed to the group achieves PCS.
- It requires only small modifications to the MLS design. More precisely, it only modifies how the ratchet tree's nodes are blanked or populated by applying update proposals, but keeps other components required, for example, to ensure consistency and authenticity unchanged.

We provide some intuition on our design. Our main observation is that if the number of update proposals is not too large and the updating users are randomly distributed, then updating users' paths merge far up in the tree. Thus, while in MLS-Cutoff update proposals do rekey the issuing user's path, we stop the re-key operation i_{cut} steps before reaching the root, where i_{cut} is the cutoff parameter of the scheme, that is chosen as $\log(\log(N))$. If no update paths collide before the cutoff point, no secrets are encrypted to keys that are concurrently being changed. This on one hand, implies PCS at the same speed as MLS, and on the other is compatible with the tree-integrity mechanisms of tree-hash and parent-hash. In the case that we are unlucky and two updating user's paths collide before the cutoff point, the paths are blanked from (and including) the point where the paths meet. For a depiction on how MLS-Cutoff handles proposals and commits, see Figure 1 (right).

We point out that the worst-case communication complexity of MLS-Cutoff is $\Omega(N)$, as is the case for MLS. This, however, holds for any CGKA achieving fast PCS [BDR20] constructed from standard primitives. We consider MLS-Cutoff to be an attractive alternative to MLS in settings where while update proposals are required their number is not too large. This is a realistic scenario since MLS leaves the update policy to the implementation and, therefore, a rule which simply triggers an update with some probability for users that are online (or enforces it if too much time or communication happened since the last update) would

lead to a fairly random update pattern if there is no user-to-user coordination and thus good performance of MLS-Cutoff. We consider it an interesting question for future work to analyze its performance in more involved settings.

2 Preliminaries

2.1 Continuous Group-Key Agreement

We recall continuous key-agreement schemes (CGKA). As we aim for the security notion of insider security our syntax follows [AJM22].

Public-key infrastructure. We model the public-key infrastructure (PKI) as consisting of two services: an authentication service (AS) and a key service (KS). The former allows parties to register their signing keys and retrieve the signature verification keys of other users, while the latter allows users to register the key packages that are used by other users to add them to a group.

The authentication service stores pairs of users and signing verification keys, (u, svk), if the key svk has been register under user identity u. Any user u can register a new key and request to receive the secret signing keys that correspond to verification keys registered under them. Moreover, a user can also check if a pair (u, svk) has already been registered and a user can also delete their secrets. We consider the AS as not trusted and therefore in our security model we will give the adversary the possibility of registering keys as well as exposing keys registered by the users (provided that they have not been deleted).

The key service stores pairs of users and key packages, (u, kp), which are used by other parties to add them to a group. Similarly to the case of AS, users are able to register a new key package as well as retrieving the secret keys sk associated to their key packages and deleting their secret keys. A user can also request a key package for another party. In our model this is done by forwarding the request to the adversary which chooses the package received by the user. This reflects that our security model does not trust the KS. The adversary can also expose the secret keys of a user that have not been deleted.

Syntax. A CGKA scheme allows a group of users to agree on a shared secret evolving over epochs. To this end, the users exchange proposal messages, proposing changes to the group's state (as for example adding or removing users to or from the group), and commit messages, which implement proposals and advance the group's state to the next epoch by establishing a new shared secret.

Definition 2.1. *An asynchronous continuous group-key agreement protocol consists of a tuple of algorithms* $\mathsf{CGKA.Create}, \mathsf{CGKA.Prop}, \mathsf{CGKA.Com}, \mathsf{CGKA.Proc}, \mathsf{CGKA.Join}, \mathsf{CGKA.Key}$*, which can interact with a PKI. Additionally, each user maintains an internal state which we denote by* st *and is assumed to be an implicit input of all algorithms.*

- $\mathsf{CGKA.Create}(\mathsf{svk}_u)$ *is run by user* u *with signing verification key* svk_u *and creates a group with* u *as its only member.*
- $\mathsf{CGKA.Prop}(\mathsf{op}, \mathsf{ad}) \to \mathsf{pmsg}$ *is run by a member who wants to propose an operation* op*, where* $\mathsf{op} \in \{\texttt{'add'}, \texttt{'rem'}, \texttt{'upd'}\}$*,* ad *is some additional data that may depend on the operation. It outputs a proposal message* pmsg*.*
- $\mathsf{CGKA.Com}(\mathsf{PMSG}, \mathsf{ad}) \to (\mathsf{cmsg}, \mathsf{wmsg})$ *is run by a member* u *to commit to a list of proposals* $\mathsf{PMSG} = (\mathsf{pmsg}_i)_i$ *and it also takes as input some additional data* ad*. It outputs a commit message* cmsg*, and a (potentially empty) welcome message* wmsg*.*
- $\mathsf{CGKA.Proc}(\mathsf{cmsg}, \mathsf{PMSG}) \to (\mathsf{PropSemantics}, v)$ *is run by a client* u *to process a commit message* cmsg *and the corresponding proposal messages* PMSG*. It outputs some information about the proposals (which we denote* $\mathsf{PropSemantics}$*) and the party* v *that generated the commit* c*.*

$\mathsf{CGKA.Join}(\mathsf{wmsg}) \to (\mathsf{roster}, v)$ *is run by user* u *to process a welcome message and join the respective group. It outputs the set of group members and their signing verification keys (which we refer to as* roster*) and the party that generated the commit* c

$\mathsf{CGKA.Key} \to \mathsf{k}$ *is run by a member* u *to obtain the group key associated to their state.*

Correctness and security in the UC framework. We follow the work of Alwen et al. ([AJM22] and [ACJM20]). They define a notion of security based on the UC (universal composability) framework, which was first introduced by Canetti in [Can01]. We defer the formal description of the ideal CGKA functionality and the PKI functionalities to Appendix A. In the UC framework one formalizes security of a protocol as the indistinguishability between the 'real world' and an 'ideal world'. We will refer to the distinguisher as the environment.

In the real world the environment $\mathcal{Z}$ gets to choose the inputs (e.g., an instruction to Alice to add Bob) of the machines (e.g., the parties in an execution of a CGKA protocol) of the protocol π and receives their intermediate subroutine-outputs (e.g., a commit generated by Alice). The environment also interacts freely with an adversary $\mathcal{A}$. $\mathcal{A}$ also interacts with the machines of π and $\mathcal{A}$ has control over the network in the communication between the machines of π. The interaction between $\mathcal{A}$ and π also captures information leakage from the protocol execution (e.g., $\mathcal{A}$ may have the possibility of corrupting the parties).

In the ideal world the protocol π is substituted by an ideal functionality $\mathcal{F}$ and a collection of dummy machines that relay their inputs to $\mathcal{F}$ and the outputs of $\mathcal{F}$ to the corresponding machines. The adversary in the ideal world is denoted by $\mathcal{S}$ because it plays the role of a simulator (of the interaction between $\mathcal{A}$ and $\mathcal{Z}$ in the real world) and we refer to it as an ideal adversary. The adversary $\mathcal{S}$ now communicates with $\mathcal{F}$ and does not interact with the dummy machines. The environment $\mathcal{Z}$ interacts with the dummy machines and the adversary $\mathcal{S}$ just as in the real world.

In the UC framework one says that a protocol π securely realizes a functionality $\mathcal{F}$ if for all efficient adversaries $\mathcal{A}$ there exists an efficient ideal adversary $\mathcal{S}$ such that the output distribution of any efficient environment $\mathcal{Z}$ after interacting with π and $\mathcal{A}$ is indistinguishable from the output distribution of $\mathcal{Z}$ after interacting with $\mathcal{F}$ and $\mathcal{S}$. This can be understood as representing the idea that an adversary $\mathcal{A}$ attacking the protocol π does not learn more than the ideal adversary (sometimes called simulator) $\mathcal{S}$ learns from interacting with the ideal functionality.

In the work of Alwen et al. [AJM22], whose security notion we use, one has to consider some additional notions since parties have access to a PKI and certain primitives are modeled as random oracles. We refer to these as setup functionalities. This is done by considering the so called hybrid models (first introduced in [CDPW07]). If $\mathcal{G}$ is a setup functionality, then the environment $\mathcal{Z}$ (as well as π, $\mathcal{A}$ and $\mathcal{S}$) is allowed to interact with $\mathcal{G}$. And one says that the a protocol π securely realizes a functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if the real world and ideal world are indistinguishable even if $\mathcal{Z}$ is allowed to interact with $\mathcal{G}$.

Ideal functionality for CGKA. The definition of the ideal functionality $\mathcal{F}_{\mathsf{CGKA}}$ already takes correctness into account. In order to do this certain checks are included in $\mathcal{F}_{\mathsf{CGKA}}$ and the environment rather than the adversary is given control over the network. Now we focus our attention on defining security for CGKA following [AJM22]. They consider two more constrains on the quantification of the notion of security described in the previous paragraph. We restrict ourselves to admissible environments, i.e., those that can only corrupt parties at certain times. This is defined as part of the ideal functionality which specifies some conditions that cannot be violated by the environment except with negligible probability. The second condition consists in not considering all possible adversaries but instead only corruption preserving adversaries, i.e., adversaries that trigger a corruption if and only if prompted by the environment.[1] This is done to prevent the ideal adversary from triggering a disallowed corruption since this would make security trivial in the sense that all environments would be disqualified.

In order to define the ideal functionality $\mathcal{F}_{\mathsf{CGKA}}$ we consider the history graph which keeps track of the evolution of the group. Intuitively, the nodes of the graph correspond to the operations done by the users

[1] Recall that an arbitrary adversary can choose when to corrupt a party without this being determined by the environment.

and the edges capture the order in which they are issued. The history graph [ACDT21] is an acyclic directed graph which consists of two kinds of nodes, proposal nodes and commit nodes. The edges of the graph capture the notion of how new epochs are derived. Additionally, each node is associated with a collection of attributes ($\mathsf{Node}[c]$ for commit nodes c and $\mathsf{Prop}[p]$ for proposal nodes p) related to the operations of a CGKA protocol. We list these attributes in the following table (divided into three groups: those that all nodes have, those that only proposal nodes have and those that only commit nodes have).

.orig	the party that triggers the creation of the node
.par	the parent commit node
.stat	`good` for secure nodes, `bad` for nodes created with adversarially chosen randomness (the adversary knows the secrets associated to this node) or `adv` for nodes that correspond to messages injected by the adversary
$\mathsf{Prop}[p].\mathsf{act}$	indicates the kind of action and some additional information $(\mathsf{upd}, \mathsf{svk})$ if it is an update performed by a party with key svk $(\mathsf{add}, v, \mathsf{svk}_v)$ if party v with key svk_v is added (rem, v) if party v is being removed
$\mathsf{Node}[c].\mathsf{pro}$	the list of proposals associated to commit c
$\mathsf{Node}[c].\mathsf{mem}$	list of pairs of members and their signing verification keys
$\mathsf{Node}[c].\mathsf{chall}$	`true` if the corresponding application secret is chosen at random and `false` if chosen adversarially
$\mathsf{Node}[c].\mathsf{exp}$	pairs $(u, \texttt{true/false})$ of corrupted parties u and a boolean value expressing whether the application secret also leaked to adversary

The ideal functionality just keeps track of the evolution of the history graph. It starts building the history graph with just one node $root_0$ when the group is created. Additionally $\mathcal{F}_{\mathsf{CGKA}}$ stores a value called pointer $\mathsf{Ptr}[u]$ for each party u that maps u to the last commit it processed. When a party u makes a proposal p, $\mathcal{F}_{\mathsf{CGKA}}$ creates a proposal node $\mathsf{Prop}[p]$ whose parent in the history graph is $\mathsf{Node}[\mathsf{Ptr}[u]]$ and u is the origin of the proposal $\mathsf{Prop}[p].\mathsf{orig}$. The functionality also stores information about the kind of operation associated to p in $\mathsf{Prop}[p].\mathsf{act}$. If u makes a commit c to some proposals P, $\mathcal{F}_{\mathsf{CGKA}}$ creates a commit node $\mathsf{Node}[c]$ whose parent in the history graph is $\mathsf{Node}[\mathsf{Ptr}[u]]$, $\mathsf{Node}[c].\mathsf{pro}$ is set to P and $\mathsf{Node}[c].\mathsf{orig}$ is set to u. And if a party v processes c, the functionality just moves v's pointer to c.

As mentioned before, the ideal functionality imposes certain restrictions on the behaviour of the environment. This is done by including two predicates, namely, $\mathsf{safe}(c)$ and $\mathsf{inj\text{-}allowed}(c, u)$ that regulate when an adversary can corrupt a user and when it can inject a commit on behalf of a user, respectively.

In order to guarantee that the only application secrets that can be learned by an adversary in the real world are the ones trivially derived from secrets of exposed users, we let the simulator choose these 'unsafe' secrets in the ideal world while the remaining ones are chosen at random by the ideal functionality. Thus, predicate $\mathsf{safe}(c)$ evaluates to `false` if and only if at least one of the following is true:

(a) $\mathsf{Node}[c].\mathsf{exp} \neq \emptyset$ and the application secret has been leaked to the adversary, or

(b) the adversary can process c using the secrets of exposed users and the exposed signing keys (in particular this captures the secret keys associated to key packages generated with exposed signing keys or corrupted randomness) of its ancestors.

Analogously, predicate $\mathsf{inj\text{-}allowed}(c, u)$ guarantees that in the real world authenticity is only broken when the adversary can trivially forge messages on behalf of a user by restricting when injections are possible in the ideal world. Predicate $\mathsf{inj\text{-}allowed}(c, u)$ evaluates to `true` if and only if u's signing key in $\mathsf{Node}[c]$ has been exposed (i.e., $\mathsf{Node}[c].\mathsf{mem}[u] \in \mathsf{Exposed}$) and at least one of the following two holds:

(a) $\mathsf{Node}[c].\mathsf{exp} \neq \emptyset$,

(b) the adversary can process c using the secrets of exposed users and the exposed signing keys (in particular this captures the secret keys associated to key packages generated with exposed signing keys or corrupted randomness) of its ancestors.

The reader may find the precise definition of the predicates $\mathsf{safe}(c)$ and $\mathsf{inj\text{-}allowed}(c, u)$ in Figure 17.

2.2 Ratchet Trees

We introduce some basic terminology. We use the word tree to refer to a directed acyclic graph $\mathsf{T} = (\mathsf{V}, \mathsf{E})$ with a root, i.e., a node r such that for every node v there exists a unique directed path between r and v. Said path is denoted by $\mathsf{path}(v)$. As in the MLS specification we use the convention that $\mathsf{path} = (v_1, \ldots, v_\ell = r)$ is ordered ascending to the root, i.e., in reverse direction of the edges, and does not include v.

Let u and v be two different nodes of some tree $\mathsf{T} = (\mathsf{V}, \mathsf{E})$. We say that u is a parent of v if there exists an edge $(u, v) \in \mathsf{E}$. We say that u is a child of v if and only if v is a parent of u. We denote the set of children by $\mathsf{child}(v)$. We say that u is an ancestor of v if there exists a directed path from u to v. The set of ancestors is denoted by $\mathsf{ancs}(u)$. We say that v is an descendant of u if and only if u is an ancestor of v. We say that u and v are siblings if they are children of the same node. We say that a node is a leaf if it has no children. Thus we have three different kinds of nodes; the root, the leaves and the remaining elements in V which we call intermediate nodes.

In this work we consider complete binary trees with $N = 2^n$ many nodes. Thus each non-leaf node v has exactly two children which we refer to as the left child, denoted by $\mathsf{left}(v)$, and the right child, $\mathsf{right}(v)$. Every node v other than the root has one parent, denoted by $\mathsf{par}(v)$, and a sibling, $\mathsf{sib}(v)$. The depth of a node v is the length of the path from the root to v and we denote it by $\mathsf{depth}(v)$.

A notion closely related to how MLS re-keys users' paths is that of co-path of a node. The co-path of a node v is defined as the set of children of the nodes in v's path that do not belong to v's path, namely,

$$\mathsf{co\text{-}path}(v) = \cup_{u \in \mathsf{path}(v)} \{w \mid w \in \{\mathsf{left}(u), \mathsf{right}(u)\} \setminus (\mathsf{path}(v) \cup \{v\})\}.$$

Node states. CGKA schemes like MLS make use of a *ratchet tree*, i.e., a complete binary tree $\mathsf{T} = (\mathsf{V}, \mathsf{E})$ with each group member being associated to one of the tree's leaves. Every node $v \in \mathsf{E}$ has as associated data a so called public and private state. Its public state contains

- a PKE public key $v.\mathsf{pk}$,
- a list $v.\mathsf{unm}$ of so called unmerged leaves,
- a parent-hash value $v.\mathsf{pHash}$,
- a signature verification key $v.\mathsf{svk}$ (only if the node is a leaf), and
- a credential $v.\mathsf{cred}$ (only if the node is a leaf).

The private state of the node is $v.\mathsf{sk}$ the secret key associated to $v.\mathsf{pk}$. MLS, as well as the variant defined in this work, enforce the so called tree-invariant stating that $v.\mathsf{sk}$ is known by exactly the user whose leaves have v as an ancestor. Further, a node can be *blank*, meaning that all associated values are empty. We use function $\mathsf{blank}(v)$ to blank a node, i.e., to set all its associated data to $\perp$.

Resolution and filtered path. The existence of blank nodes leads to the notion of resolution $\mathsf{Res}(v)$ of a node v which, intuitively, corresponds to the smallest covering of the set of descendants of v by non-blank nodes. We define it formally as follows;

- If v is a populated node, its resolution is $\mathsf{Res}(v) = \{v\}$.
- If v is a blank leaf, its resolution is $\mathsf{Res}(v) = \emptyset$.
- If v is a blank internal node, its resolution is defined recursively as

$$\mathsf{Res}(v) = \cup_{u \in \mathsf{child}(v)} \mathsf{Res}(u).$$

If X is a set of nodes, we use the notation $\mathsf{Res}(X)$ to denote $\cup_{v \in X} \mathsf{Res}(v)$.

A leaf v's filtered path $\mathsf{fil\text{-}path}$ is the update path after removing all nodes the co-child of which has an empty resolution (and unmerged leafs set). I.e., if $\mathsf{path}(v) = (v_1, \ldots, v_\ell = r)$ then using the convention $v_0 = v$ we define

$$\mathsf{fil\text{-}path}(v) = (v_i, i \in \{1, \ldots, \ell\} \mid (\mathsf{Res}(\mathsf{sib}(v_{i-1})) \cup \mathsf{sib}(v_{i-1}).\mathsf{unm}) \neq \emptyset).$$

Finally, we sometimes have to identify the node in which two leaves' filtered paths meet. Consider leaves $u, v \in \mathsf{V}$ of the same depth with filtered paths $(u_1, \ldots, u_\ell = r) \leftarrow \mathsf{fil\text{-}path}(u)$ and $(v_1, \ldots, v_{\ell'} = r) \leftarrow \mathsf{fil\text{-}path}(v)$, respectively. We define their least common ancestor $\mathsf{lca}(u, v)$ as the node u_i in $\mathsf{path}(u)$ where i is minimal such that $u_i \in \mathsf{path}(v)$. Counting from the root we define the index $\mathsf{ind\text{-}lca}$ of their least common ancestor as

$$\mathsf{ind\text{-}lca}(u, v) = \max(a \mid u_{\ell-a} = v_{\ell'-a}, a \in 0, \ldots, \ell - 1).$$

2.3 Auxiliary Results

Lemma 2.2 (Bernoulli's Inequality)**.** *For all $x \geq -1$ and $r \in \mathbb{R} \setminus (0, 1)$ it holds that*

$$(1 + x)^r \geq 1 + rx.$$

3 The Messaging Layer Security Protocol

We defer the formal description of the MLS protocol[2] [BBR+23] to Appendix B.1. In this section we provide an overview on its working principle. As the goal of this work is to investigate the protocol's communication complexity which is largely determined by the distribution of blank nodes in the underlying ratchet tree, our exposition mainly focuses on the mechanisms leading to the addition of blanks to the tree, or removal of blanks, respectively. For ease of exposition in this section we treat the key-packages kp (consisting of the corresponding user's PKE pk, signature verification key svk, and leaf credential cred) assigned to leaves in the ratchet tree as public key pk, and, similarly, assign the tree's root a key pair which is not the case in the full protocol (see Figure 19).

Protocol state. The cryptographic state st each user of MLS keeps track of is conceptually divided as follows.

- The underlying ratchet tree T. In this simplified exposition we identify users u with their associated leaf, simply writing v_u. As discussed in Section 2 every user u has a complete view of the tree's public part, but only has access to the secret keys $v.\mathsf{sk}$ of nodes that lie on their update path v_u. Sometimes when operating on ratchet trees we require the identity of the user running the algorithm, which we denote by u-own.
- The key schedule collects a number of secrets used for different purposes. These include, for example, the shared group secret referred to as the application secret, the initialization secret that is combined with the randomness obtained from re-keying a path to advance the key schedule to the next epoch, and a membership key used in combination with the user's signature key to authenticate protocol messages.
- Further, the protocol employs consistency mechanism ensuring that all users have the same view of the protocol's state. This includes the transcript hash that essentially encodes the complete history of the group including all operations processed so far, as well as so-called tree-hash and parent-hash values that ensure consistency of the ratchet tree. We will discuss the latter two in a bit more detail at the end of this section.

Generating proposal and commit messages. Users issue proposal messages to add or remove user to or from the group, respectively, as well as to update the personal key pair located at their leaf. In more detail, proposal messages

$$\mathsf{pmsg} = (\texttt{'type'}, \mathsf{ad})$$

consist of `'type'` indicating the type of proposal, as well as some associated data. In case of an update proposal `'upd'` the latter contains a new public key to be included at the issuing user's leaf, in case of

[2]Formally, in this work we consider the CGKA underlying MLS. Thus, our exposition ignores the secret tree scheduling keys used to encrypt and authenticate payload messages.

an addition 'add' the associated data ad = pk is the added party's personal key, and finally in case of a removal 'rem' it indicates the removed user's leaf. In case of an update proposal the issuing user stores the secret key sk corresponding to pk as a pending update in a list pendUpd.

To prevent unauthorized parties from issuing proposals pmsg has to be *framed*. Essentially, this means the proposal message and group context are signed by the issuing user as well as MACed using a key only available to group members. When processing proposal messages users unframe the message by verifying the authenticity of pmsg before recovering (('type', ad), u-snd). Here u-snd denotes the user issuing the message.

A user u can commit to a (potentially empty) list PMSG of proposals, implementing the proposed changes, advancing the group to the next epoch, and establishing a new application secret. This is done by executing the following steps.

1. Verify every proposal pmsg $\in$ PMSG.
2. Create a copy T' of ratchet tree st.T and
   (a) call apply-props(PMSG) to apply the proposed changes to T',
   (b) call rekey-path($v_{u\text{-own}}, 0$) to rekey u-own's update path in T' and generate an UpdPath object used to communicate the changes to the other group members.
3. Compute the new key schedule based on T'.
4. Authenticate UpdPath and bind it to PMSG.
5. Construct a welcome message if necessary.
6. Store the resulting state st′ (containing T') as a pending commit and return the corresponding, framed, commit message cmsg, which, in particular, contains UpdPath.

The largest component of cmsg is UpdPath which itself is comprised of public keys and ciphertexts. The number of ciphertexts that have to be generated depends on the blanks present in T'.

To process a commit message issued by u users essentially follow the same sequence of operations. More precisely, they recover T' by first applying the proposals PMSG to their current view of the tree, and then recovering the new keys on the committing user's update path from UpdPath. They update the key schedule and verify that all changes were properly authenticated. If the latter is the case, the new key schedule and ratchet tree replace the ones stored in the user's state. If the processing user is the one issuing the commit operation, they can simply update their state to the one stored as a pending commit.

Applying proposals and path rekeying. When issuing (or processing) a commit operation, blanks are introduced in the ratchet tree by calling apply-props in step 2a and removed from it when calling rekey-path (or applying the re-key operation) in step 2b. For an overview of the two functions see Figures 2 and 3.

apply-props on input a copy T' of the current ratchet tree and list PMSG of proposals applies changes to the ratchet tree as follows. For pmsg $\in$ PMSG of type 'upd' the issuing user u-snd's leaf key is replaced by the one contained in pmsg. Then u-snd's update path is blanked. If pmsg proposes removing user u-rem, leaf $v_{u\text{-rem}}$ as well as the removed user's update path are blanked. If the removal results in the right half of the tree not having any populated leaves its size is halved using function truncate. If pmsg proposes adding user u-add to the group, they get assigned the leftmost unpopulated leaf in the tree, doubling the tree size in the process (by adding blank nodes) if no such leaf exists. The leaf is assigned the public key contained in ad, and added to the set of unmerged leaves for every node in its update path.

To replace the keys on their path the committing user u calls rekey-path. It takes as input T', u's leaf v_u, and, in our presentation, an additional index i_{cut} that allows stopping the rekeying i_{cut} steps early. While MLS uses $i_{\text{cut}} = 0$, looking ahead, it will be used in proposed alternative method of issuing update proposals. Denoting $v_0 = v_u$ and $(v_1, \ldots, v_\ell) =$ fil-path(v_0) the function samples seed s_0 and iteratively generates seed s_i using a key-derivation function, in our presentation written as hash function H_1. It then generates key pairs $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Pke.Gen}(\mathsf{H}_2(s_i))$, H_2 being another hash function, to replace the keys associated to v_i.

```
Algorithm apply-props(T′, PMSG)
00 for pmsg ∈ PMSG:
01   parse ((‘type’, ad), u-snd) ← pmsg
02   if (‘type’, ad) = (‘upd’, pk):
03     assign-kp(T′, v_{u-snd}, pk)      \\ replace u-snd’s leaf pk
04     for v ∈ fil-path(v_{u-snd})       \\ blank u-snd’s path
05       blank(v)
06     if u-own = u-snd                  \\ own update proposal
07       fetch sk ← pendUpd(pmsg)
08       v_{u-own}.sk ← sk
09   else if (‘type’, ad) = (‘rem’, u-rem):
10     blank-leaf(T′, v_{u-rem})         \\ blank u-rem’s leaf
11     for v ∈ path(v_{u-rem})           \\ blank u-rem’s path
12       blank(v)
13     T′ ← truncate(T′)                 \\ truncate tree if necessary
14   else if (‘type’, ad) = (‘add’, (u-add, pk)):
15     add-leaf(T′, u-add, pk)           \\ assign u-add a leaf
16     for v ∈ path(v_{u-add})           \\ set v_{u-add} as unmerged
17       v.unm ←∪ {v_{u-add}}
18 return T′
```

Figure 2: High-level overview on function apply-props for MLS. The function's formal description is in Figure 23.

In order to enable the other group members to recover the secret keys they should have access to, for all $j \in \{1, \ldots, \ell\}$ the co-child v_{sib} with respect to v_u's update path is computed. Then the seed s_j is encrypted under all public keys associated to nodes contained in $\mathsf{Res}(v_{\mathsf{sib}})$, as well as the ones of unmerged leaves resulting in a collection C_j of ciphertexts. The update path object consists of

$$\mathsf{UpdPath} \leftarrow (\mathsf{pk}_0, (\mathsf{pk}_1, C_1), \ldots, (\mathsf{pk}_\ell, C_\ell)).$$

A user processing a commit message issued by u-snd replaces the public keys on u-snd's path by the ones contained in UpdPath. Then, by computing where their own and u-snd's update paths meet (say for index j) they are able to determine node v_{sib} relevant to them, and using the secret key in $\mathsf{Res}(v_{\mathsf{sib}}) \cup v_{\mathsf{sib}}.\mathsf{unm}$ they have access to decrypt one of the ciphertexts contained in C_j to recover s_j. In turn they are able to recover the secret keys $\mathsf{sk}_j, \ldots, \mathsf{sk}_\ell$ that they should have access to according to the tree invariant by applying H_1, H_2, and Pke.Gen.

Consistency mechanisms w.r.t the ratchet tree. MLS employs tree-hashes and parent-hashes (see Figure 21) to ensure consistent views of the tree and provide improved security guarantees for users joining an existing group. The former essentially is a commitment to the entire public state of the ratchet tree, and is computed as a Merkle commitment. I.e., a node's tree hash corresponds to a hash of the node's public values and the tree-hashes of its children.

The parent hash captures information on how the tree's nodes were populated during the protocol execution. Parent hash values are computed as part of a user's commit, along the user's update path, however in reverse order, i.e., from root to leaf. Essentially, a node's parent hash attests to its parent's public key, as well as the tree hash of the sub-tree rooted at its co-child. It plays an important role in preventing malicious insiders from being able to successfully invite users to malformed groups.

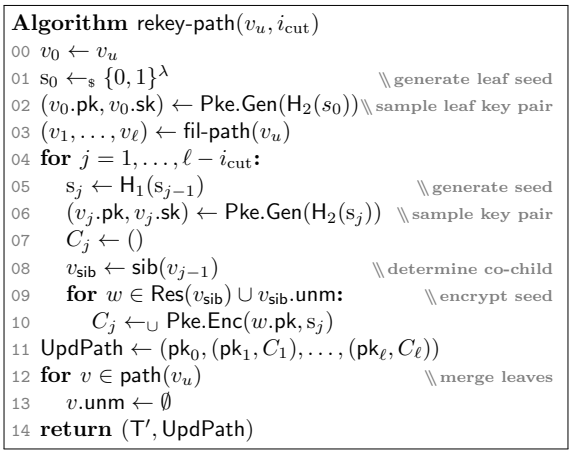


Figure 3: High-level overview on function rekey-path for MLS. The function's formal description is in Figure 22.

4 The Communication Cost of MLS for Random Sequences of Operations

We investigate the communication complexity of MLS. Section 4.1 defines a simple experiment used to generate random sequences of operations. We then proceed to compute bounds on the amount of blank nodes the MLS ratchet tree will have on average for such sequences (in Section 4.2) before computing a lower bound on the communication cost of commit operations (in Section 4.3).

4.1 Considered Scenario

As discussed in the introduction MLS exhibits a communication complexity that is logarithmic (in the number of users) in the best case, but can devolve to being linear in the worst case. In fact, the latter must be the case for any protocol that is built from standard primitives and achieves PCS as fast as MLS [BDR20, ANPPP23, BDG+22]. Beyond this, very little is known about MLS's performance in practice.

We initiate the study of MLS's communication cost under randomized sequences of operations. While ideally, one would evaluate the performance for sequences of operations obtained from real-world user data, such data is unfortunately, up to our knowledge, not publicly available. As a consequence, in this work we take a pragmatic approach and consider sequences determined by a simple experiment. Essentially, we generate a fixed number of update proposals per commit operation, the users performing the operations being sampled uniformly at random. While we do not claim this realistically reflects user behavior, on the one hand, it makes the problem of computing lower bounds approachable, and, on the other, provides simple intuition on the impact that blanking has that is not obscured by technical details of a more complicated model.

We define experiment Exp-Prop-Com in Figure 4. It is parameterized by the group size $N = 2^n \in \mathbb{N}$, integers P, C, and the number of rounds t. Essentially, it initializes a group of N users and then for t rounds generates update-proposal and commit operations as follows. First, in every round P users U_P are chosen uniformly at random from the group $[N]$ and issue an update proposal. This is followed by an uniformly random user U_C issuing a commit, which leads to the group moving to the next epoch and ends the round. To allow for the more general scenario that proposals are only issued every couple epochs, we allow for a number of epochs in between sending update proposal in which a commit, but no proposals, is issued. This is done by instead of sampling a single committer to end the round, to sample C committers uniformly at random. To analyze the communication cost of operation sequences generated in this way we record the size

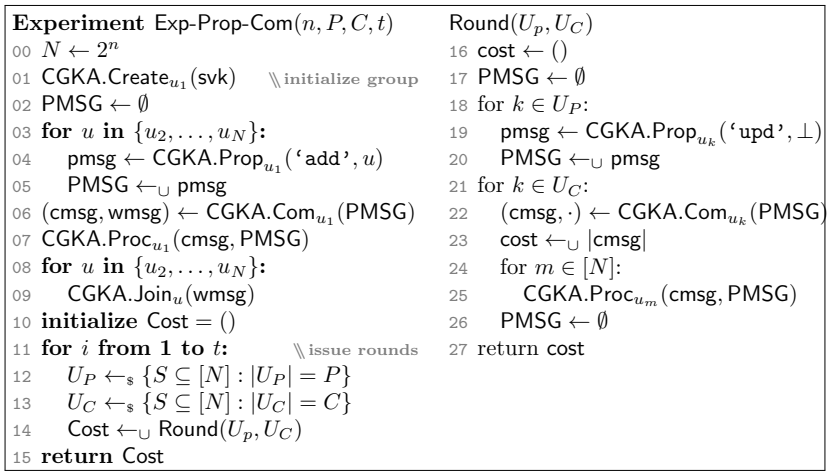


Figure 4: Experiment Exp-Prop-Com with respect to group size $N = 2^n$ and numbers of proposals P and commits C executed over t rounds. We use subscripts u to denote that an algorithm is executed by user u, and omit algorithm outputs not relevant to the experiment.

of each commit message generated during the experiment in an array Cost.

Our goal is to compute a lower bound on the expected value of $\mathsf{Cost}(t)$ after running the experiment for sufficiently many rounds. Recall that the amount of MACs, signatures, and public keys contained in commit messages is at most logarithmic in N. Thus, $\mathsf{Cost}(t)$ is dominated by the number of ciphertexts that have to be generated when calling rekey-path (see Figure 2) to communicate the new secret keys on the committer u-snd's path to the other users. Accordingly, we can bound the communication cost from below by

$$\mathsf{Cost}(t) \geq \sum_{v_i \in \mathsf{co\text{-}path}(v_{u\text{-}\mathrm{snd}})} |\mathsf{Res}(v_i)|,$$

the sum of resolution sizes on the committer's co-path, and in our analysis it suffices to derive a bound for this expression.

4.2 Expected Number of Blanks

Recall that the resolution of a node is determined by which of its descendants are blank. In this section we compute the expected probability of nodes at a certain depth being blank in the ratchet tree generated by experiment Exp-Prop-Com.

Expressing blanks as a Markov chain. Consider the sequence $(\mathsf{T}_i)_{i=1}^t$ of ratchet trees generated by experiment Exp-Prop-Com, where T_i is the tree all users store as part of their state after the ith call to round function Round. For a fixed internal node v we can define a random variable $(B_i(v))_{i=0}^t$ taking values in $\{0, 1\}$ where 0 corresponds to being a populated node and 1 to v being blank. We write B_i instead of $B_i(v)$ when there is no ambiguity. We observe that $(B_i)_{i=0}^t$ forms a Markov chain. Indeed, v transitioning from blank to populated or vice versa only depends on the choice of the user U_P and U_C issuing proposals or commits in the current round. Since the chain is clearly irreducible, it has a unique stationary distribution. Denoting by $p_{b \to b'}$ the probability of the node switching from state $B_i = b$ to $B_{i+1} = b'$, the transition matrix of $(B_i)_{i=0}^t$ with respect to state vector $(0, 1)^\top$ is given by

$$M = \begin{pmatrix} p_{0\to 0} & p_{1\to 0} \\ p_{0\to 1} & p_{1\to 1} \end{pmatrix}.$$

The eigenvalues of M are given by $p_{0\to 0} - p_{1\to 0}$ and 1 and an eigenvector of one-norm 1 with eigenvalue 1, corresponding to the stationary distribution, is given by

$$\left(\frac{p_{1\to 0}}{p_{1\to 0}+p_{0\to 1}}, \frac{p_{0\to 1}}{p_{1\to 0}+p_{0\to 1}}\right)^\top. \tag{1}$$

Analyzing the stationary distribution of $(B_i)_{i=0}^t$. Consider the state B_i of node v at the end of round i. If $B_i = 1$ (i.e., v is blank at the end of round i), then after the proposals U_P and commits U_C of round $i+1$ have happened B_{i+1} will remain blank exactly if none of the users in U_C have v as an ancestor. Further, if $B_i = 0$ (i.e., v is a populated node at the end of round i), then B_{i+1} will remain non-blank unless at least one of the users in U_P has v as an ancestor while none of the users in U_C do. Thus, if we denote

$$p_P := \Pr[v \in \mathsf{ancs}(U_P)] \quad \text{and} \quad p_C := \Pr[v \in \mathsf{ancs}(U_C)]$$

we have that $p_{1\to 0} = p_C$ and by independence of the choice of proposer and committer that $p_{0\to 1} = (1-p_C)p_P$. By Equation 1 we obtain that the stationary distribution of a node being blank is given by

$$\text{blanks}_{\text{MLS}} := \left(\frac{p_C}{p_C + p_P(1-p_C)}, \frac{p_P(1-p_C)}{p_C + (1-p_C)p_P}\right)^\top \in [0,1]^2. \tag{2}$$

We now compute the probability of v being an ancestor of a proposing or committing leaf, respectively. Assume v is at depth $\mathsf{depth}(v) = d$. Then T has $N2^{-d}$ many leaves that are in $\mathsf{ancs}(v)$. Therefore there are $N(1-2^{-d})$ many leaves that are not in $\mathsf{ancs}(v)$ which implies that

$$p_P = \Pr[v \in \mathsf{ancs}(U_P)] = 1 - \Pr[v \notin \mathsf{ancs}(U_P)] = 1 - \frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}.$$

Analogously,

$$p_C = \Pr[v \in \mathsf{ancs}(U_C)] = 1 - \Pr[v \notin \mathsf{ancs}(U_C)] = 1 - \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}.$$

Using Equation 2 we have shown the following.

Lemma 4.1. *For an execution of experiment* Exp-Prop-Com *and internal node v of depth $\mathsf{depth}(v) = d$ let $(B_i)_i = (B_i(v))_i$ be defined as above. Then the probability that v is blank in the stationary distribution $B = \text{blanks}_{\text{MLS}}$ is*

$$\mathbb{E}[B(v) = 1] = \frac{\left(1 - \frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}\right)\frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}}{1 - \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}\frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}}. \tag{3}$$

Before turning to computing a lower bound on the expected commit size in experiment Exp-Prop-Com we look at some simple cases:

When $P = 1 = C$, we have that $p_P = 2^{-d} = p_C$. Therefore,

$$\mathbb{E}[B(v) = 1] = (1-2^{-d})/(2-2^{-d}) \approx 1/2 \text{ for } d \text{ large enough.}$$

If $P + C = O(1)$,

$$\begin{aligned}\mathbb{E}[B(v) = 1] &\approx \frac{(1-(1-2^{-d})^P)(1-2^{-d})^C}{1-(1-2^{-d})^P(1-2^{-d})^C} \\ &\approx \frac{(1-(1-P2^{-d}))(1-C2^{-d})}{1-(1-(P+C)2^{-d})} \\ &\approx \frac{P}{P+C} \text{ for } d \text{ large enough.}\end{aligned} \tag{4}$$

In both examples we see a constant probability that a node is blank which, intuitively, means that a commit should have a high upload cost. We formalize this in the next section.

4.3 Lower Bound on Sent Ciphertexts

In this section we prove a lower bound on MLS's communication cost for sequences of operations generated by Exp-Prop-Com. As it turns out, already a constant number of proposals per commit leads to commits being exponential in $n = \log(N)$, a far shot from the best case communication complexity. More precisely we will prove the following.

Theorem 4.2. *If $P+C = O(1)$ and $t > 2N^2 \log(N)$, then there exists a constant $\varepsilon > 0$ such that the expected cost of a commit after Experiment* Exp-Prop-Com(n, P, C, t) *(Figure 4) satisfies the following inequality:*

$$\mathbb{E}[\mathsf{Cost}(t)] \geq \Omega(N^{\log_2(1+(1-\varepsilon)\frac{P}{P+C})}).$$

Before proving this result, we show a weaker result that already gives some of the main ideas used in the proof of Theorem 4.2. We start by studying the size of the resolution of a node v_d at depth $d \in \{1, \ldots, \log(N) - 1\}$:

$$\begin{aligned}
\mathbb{E}[|\mathsf{Res}(v_d)|] &\geq \mathbb{E}[|\mathsf{Res}(v_d)| \mid B(v_d) = 0] \Pr[B(v_d) = 0] \\
&\quad + \mathbb{E}[|\mathsf{Res}(v_d)| \mid B(v_d) = 1] \Pr[B(v_d) = 1] \\
&= \Pr[B(v_d) = 0] \\
&\quad + \mathbb{E}[|\mathsf{Res}(\mathsf{right}(v_d))| + |\mathsf{Res}(\mathsf{left}(v_d))| \mid B(v_d) = 1] \Pr[B(v_d) = 1] \\
&= \Pr[B(v_d) = 0] \\
&\quad + 2\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v_d))| \mid B(v_d) = 1] \Pr[B(v_d) = 1].
\end{aligned}$$

One can show that

$$\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v_d))| \mid B(v_d) = 1] \geq \mathbb{E}[|\mathsf{Res}(\mathsf{left}(v_d))|]. \tag{5}$$

This implies that for a node v_d at depth $d \in \{1, \ldots, \log(N) - 1\}$,

$$\mathbb{E}[|\mathsf{Res}(v_d)|] \geq \Pr[B(v_d) = 0] + 2\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v_d))|] \Pr[B(v_d) = 1].$$

A recursive application of this inequality yields

$$\mathbb{E}[|\mathsf{Res}(v_d)|] \geq 1 + \sum_{i=0}^{\log(N)-d-1} 2^i \prod_{j=0}^{i} \Pr[B(v_{d+j}) = 1].$$

Now let's assume that $P = 1 = C$. We know that for $d \geq 3$, $\mathbb{E}[B(v) = 1] \approx 1/2$. Therefore we obtain that

$$\begin{aligned}
\mathbb{E}[\mathsf{Cost}(t)] &\geq \sum_{d=3}^{\log(N)} (1 + \sum_{i=0}^{\log(N)-d-1} 1/2) \\
&= \log(N) - 2 + \sum_{d=3}^{\log(N)} (\log(N) - d)/2 \\
&= \log(N) - 2 + (\log(N) - 3)(\log(N) - 2)/4 \\
&\approxeq \log^2(N)/4 - \log(N)/4.
\end{aligned}$$

A similar computation when $P + C = O(1)$ and $P > C$ shows $\mathbb{E}[\mathsf{Cost}(t)] = \Omega(N^{\log_2(\frac{2P}{P+C})})$.

The gap between these bounds and the one stated in Theorem 4.2 is due to Equation 5 being non-tight. This is solved by studying how the size of the resolution of a node grows when several of its ancestors are blank.

Proof of Theorem 4.2. In order to prove the theorem we first establish some helpful results. Let's consider the random variable $|\mathsf{Res}(v)|$ that denotes the size of the resolution of a node v. This corresponds to the number of ciphertexts that have to be uploaded when a commit includes a fresh key for v's parent. The central idea of the lower bound is to relate the expected resolution size of a node to the expected resolution size of its children, and iterate this expectation in order to get a lower bound that's exponential in $\log(N)$. In the proof, we keep conditioning on an increasingly complicated event, and only consider the probability of a node being blank conditioned on this event. This allows us to argue using a local property whereas the resolution size of a node depends on the entire subtree of the node.

We now introduce some notation. Let $\text{Anc-bl}_t(d, v)$ be the event that all ancestors of v up to depth d are blank after the experiment ends. We usually drop the subindex t as we do with the other random variables considered in this section. When the node v is clear from context, we write $\text{Anc-bl}(d)$. Let v_a denote the ancestor of v such that $\mathsf{depth}(v_a) = d$. We use T_{sub} to denote the full binary subtree with root v_a and leaves the nodes in T that have v_a as an ascendant and have the same depth as v. Let n' be the number of leaves in T_{sub}. Notice that $n' = 2^{\mathsf{depth}(v)-d}$.

Lemma 4.3. *Let $X \sim \text{Geo}(q)$ be distributed according to the geometric distribution with $q \in (0, 1)$ and let $Z = X \mod (P+C)$. Let $i, j \in \{1, 2, \ldots, P+C\}$. Let s be an integer multiple of $P+C$, i.e. $s = m(P+C)$ for some $m > 0$. Then,*

$$\Pr[Z = i \mid X \leq s] \geq (1-q)^{P+C-1} \Pr[Z = j \mid X \leq s].$$

Proof. This is an almost immediate consequence of some elementary computation with finite geometric series. Let $i, j \in \{1, 2, \ldots, P+C\}$. Notice that

$$\begin{aligned}\Pr[Z = i \mid X \leq s] &= \frac{1}{1-(1-q)^s} \sum_{h=0}^{m-1} (1-q)^{(P+C)h+i-1} q \\ &= \frac{1}{1-(1-q)^s} (1-q)^{i-1} q \sum_{h=0}^{m-1} (1-q)^{(P+C)h}.\end{aligned}$$

Now, notice that the only factor that depends on i is $(1-q)^i$. The result follows from the observation that $(1-q)^{i-j} \geq (1-q)^{P+C-1}$. □

We now introduce the event All-act. Intuitively, All-act simply states that all users in T_{sub} acted relatively recently. We'll later show that All-act happens with overwhelming probability. More precisely, All-act simply states that if we look at the last $2N^2 \log N$ actions in T, ordered from the most recent to the least recent, there's a new user from T_{sub} in every at most $2N \log N$ steps.

In other words, we're conditioning on the following event: Let X_i be the random variable denoting how many actions it takes for the ith user to appear after the $(i-1)$ previous users already did. All these users must be different. We're assuming that $X_i \leq 2N \log N$ for every $i \in \{1, 2, \ldots, n'\}$. More formally, we'll condition on X_i being less than some $s \geq 2N \log N$ that is a multiple of $P + C$, but we omit this since it doesn't affect the proof and introduces unnecessary notation.

Now, we analyze that the probability of v being blank conditioned on all of its ancestors up to depth d being blank and All-act.

Lemma 4.4. *Let $\varepsilon > 0$ and $d \in \mathbb{N}$ such that $\varepsilon \geq 2^{\mathsf{depth}(v)-d+1}(P+C-1)/N$ where v is a node at depth $\mathsf{depth}(v) > d$. Then,*

$$\Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d)] \geq \frac{1}{2}(1 + (1-\varepsilon)\frac{P}{P+C}).$$

Proof. We describe an experiment that will help us determine the probabilities of possible states of T_{sub}.

For $i \in [n']$, let $X_i \sim \text{Geo}(q_i)$ where $q_i = \frac{n'-i+1}{N} \leq 2^{\mathsf{depth}(v)-d}/N$. Intuitively, X_i corresponds to the number of actions that occurred after the action corresponding to X_{i-1} until a "new" member acted on the tree.

Define $Y_i := \sum_{k=1}^{i} X_k \mod (P+C)$. By linearity, we have that $Y_i = X_i + Y_{i-1} \mod (P+C)$. Intuitively, the Y_i's help us track whether this action was a commit or a proposal. More formally, notice that if $Y_i \leq C$, then the user that acted ith from last committed, and if $C+1 \leq Y_i \leq P+C$ the user proposed. We also have that the Y_i's form a Markov chain, i.e. Y_i conditioned on Y_{i-1} is independent of the previous Y_k's.

Moreover, we sample a uniformly random string of length n' elements in order to assign each leaf node to a position. For example, $\sigma = 312$ ($\sigma(1) = 3, \sigma(2) = 1, \sigma(3) = 2$) would mean that the leftmost user was the first to act, the 'central' user was last to act and the third user acted second. Notice that $\sigma(1)$ tells us the position in which v acted and $\sigma(2)$ tells us the position in which $\mathsf{sib}(v)$ acted. For example, $\sigma(1) = 2$ would imply that the action on node v was second to last on T_{sub}. Notice that σ cannot repeat elements. Now we study the probability that v is blank conditioned on the events All-act and Anc-bl(d):

$$\begin{aligned}
&\Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d)] \\
&= \Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d), \sigma(1) < \sigma(2)] \Pr[\sigma(1) < \sigma(2) \mid \text{All-act}, \text{Anc-bl}(d)] \\
&+ \Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d), \sigma(1) > \sigma(2)] \Pr[\sigma(1) > \sigma(2) \mid \text{All-act}, \text{Anc-bl}(d)].
\end{aligned}$$

We have that $\Pr[\sigma(1) < \sigma(2) \mid \text{All-act}, \text{Anc-bl}(d)] = \frac{1}{2}$, as which node comes later is clearly independent of All-act, Anc-bl(d) due to symmetry. Moreover, if $\sigma(1) < \sigma(2)$, i.e. v acts later than $\mathsf{sib}(v)$, we have that v is blank since $p(v)$ is blank as we're conditioning on Anc-bl(d). Thus, we have that

$$\begin{aligned}
&\Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d)] \\
&\quad = \frac{1}{2}(1 + \Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d), \sigma(1) > \sigma(2)]).
\end{aligned}$$

For the rest of the proof, we'll prove that

$$\Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d), \sigma(1) > \sigma(2)] \geq (1 - \varepsilon)\frac{P}{P+C}$$

Before we get into the details, let's explain the intuition behind the proof. The proof rests on the observation that if $X_k \sim \text{Geo}(q)$ for small q, $X_k \mod (P+C)$ is close to a uniform random variable. Moreover, X_k and Y_{k-1} are independent, so $Y_k = X_k + Y_{k-1} \mod (P+C)$ is also close to being a uniform distribution. Thus, no matter what complicated event we're conditioning on, the probability that $1 + C \leq Y \leq P + C$ will be approximately $\frac{P}{P+C}$.

Let $f(y_1, y_2, \ldots, y_{n'}, \sigma)$ be a predicate that returns 1 if the assignment is compatible with Anc-bl(d), $\sigma(1) > \sigma(2)$ and 0 otherwise. Let $Z = \Pr[\text{All-act}, \text{Anc-bl}(d), \sigma(1) > \sigma(2)]$. For notational simplicity, we'll write $\sum$ instead of

$$\sum_{k=1}^{n'} \sum_{\substack{\sigma \in S_{n'} \\ \sigma(1)=k}} \sum_{y_1=1}^{P+C} \sum_{y_2=1}^{P+C} \cdots \sum_{y_{k-1}=1}^{P+C} \sum_{y_{k+1}=1}^{P+C} \cdots \sum_{y_{n'}=1}^{P+C}$$

in what follows. Also for notational convenience, we omit the conditioning on All-act, even though it's implicit in every probability we write. Then, we have that $\Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d), \sigma(1) > \sigma(2)]$ is equal to

$$\frac{1}{Z} \sum \sum_{y_k=1+C}^{P+C} f(y_1, y_2, \ldots, y_{n'}, \sigma) \frac{1}{(n-1)!} \Pr[Y_1 = y_1] \prod_{l=2}^{n'} \Pr[Y_l = y_l \mid Y_{l-1} = y_{l-1}], \tag{6}$$

where we used the Markov property of the Y_k's. Also notice that since we're interested in the probability of v being blank, we're only summing up the values in the support of Y_k that start from $C+1$, so we are ignoring the first C values. Again, for notation simplicity, we introduce a new variable

$$\rho := \Pr[Y_1 = y_1] \prod_{l>1, l\neq k, l\neq k+1} \Pr[Y_l = y_l \mid Y_{l-1} = y_{l-1}].$$

Rearranging Equation 6, we get

$$\frac{1}{Z}\sum \frac{1}{(n-1)!}\rho f(y_1, y_2, \ldots, y_{n'}, \sigma) \cdot \sum_{y_k=1+C}^{C+P} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = y_k] \Pr[Y_k = y_k \mid Y_{k-1} = y_{k-1}]$$

Let's now argue why it was possible to take $f(y_1, y_2, \ldots, y_{n'}, \sigma)$ out of the sum, i.e. why it's independent of $Y_{\sigma(1)}$. If $\sigma(1) < \sigma(2)$, $f(y_1, y_2, \ldots, y_{n'}, \sigma) = 0$ no matter what Y_k is. If $\sigma(1) > \sigma(2)$, $\sigma(1)$ has no effect on its ancestors, so Y_k doesn't matter for Anc-bl(d). Now, notice that if we instead sum all values in the support of Y_k, we have by the law of total probability that

$$1 = \frac{1}{Z}\sum \frac{1}{(n-1)!}\rho f(y_1, y_2, \ldots, y_{n'}, \sigma) \cdot \sum_{y_k=1}^{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = y_k] \Pr[Y_k = y_k \mid Y_{k-1} = y_{k-1}]$$

Moreover, using the Markov property of the Y_i, we have that

$$\begin{aligned}
&\Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}] \\
&= \sum_{y_k=1}^{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = y_k, Y_{k-1} = y_{k-1}] \Pr[Y_k = y_k \mid Y_{k-1} = y_{k-1}] \\
&= \sum_{y_k=1}^{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = y_k] \Pr[Y_k = y_k \mid Y_{k-1} = y_{k-1}] \qquad (7)
\end{aligned}$$

Thus, it suffices to show that

$$\begin{aligned}
&\sum_{y_k=1+C}^{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = y_k] \Pr[Y_k = y_k \mid Y_{k-1} = y_{k-1}] \\
&\geq (1-\varepsilon)\frac{P}{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}].
\end{aligned}$$

Let $t_{max} := \arg\max_t \Pr[Y_{k+1} = y_{k+1} \mid Y_k = t] \Pr[Y_k = t \mid Y_{k-1} = y_{k-1}]$. Notice that

$$\begin{aligned}
&\Pr[Y_{k+1} = y_{k+1} \mid Y_k = t_{max}] \Pr[Y_k = t_{max} \mid Y_{k-1} = y_{k-1}] \\
&\geq \frac{1}{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}] \qquad (8)
\end{aligned}$$

which follows from Equation 7. Then, by Lemma 4.3 (observe that X_k is bounded because of the event All-act), for any $1 + C \leq t \leq P + C$, we have that

$$\begin{aligned}
&\Pr[Y_{k+1} = y_{k+1} \mid Y_k = t] \Pr[Y_k = t \mid Y_{k-1} = y_{k-1}] \\
&\geq \Pr[Y_{k+1} = y_{k+1} \mid Y_k = t_{max}] \Pr[Y_k = t_{max} \mid Y_{k-1} = y_{k-1}](1-q)^{2(P+C-1)} \\
&\geq (1 - 2q(P+C-1))\frac{1}{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}] \\
&\geq (1-\varepsilon)\frac{1}{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}]
\end{aligned}$$

where the second inequality follows from Lemma 2.2 and Equation 8. The last inequality follows from the assumption that $\varepsilon \geq 2^{\mathsf{depth}(v)-d+1}(P+C-1)/N$ and the fact that $q \leq 2^{\mathsf{depth}(v)-d}/N$. Thus,

$$\sum_{t=1+C}^{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_k = t] \Pr[Y_k = t \mid Y_{k-1} = y_{k-1}]$$
$$\geq (1-\varepsilon)\frac{P}{P+C} \Pr[Y_{k+1} = y_{k+1} \mid Y_{k-1} = y_{k-1}].$$

We thus conclude the proof. □

We have two more lemmata left to prove before moving onto combining our results.

Lemma 4.5. *Assuming at least $2N^2 \log(N)$ actions took place,* $\Pr[\text{All-act}] \geq 1 - \frac{1}{N}$.

Proof. Notice that for any $i \in \{1, \ldots, N\}$,

$$\Pr[X_i > 2N \log N] \leq (1 - \frac{1}{N})^{2N \log N} \leq \exp(-2\log(N)) = N^{-2}.$$

Then, $\Pr[\text{All-act}] \geq 1 - \frac{1}{N}$ by the union bound over all the X_i . □

Lemma 4.6. *Let $\varepsilon > 0$ and $d \in \mathbb{N}$ such that $\varepsilon \geq 2^{-d}(P+C-1)$ and consider a node v such that $\log(N) - 1 \geq \mathsf{depth}(v) > d$. Then, we have that*

$$\mathbb{E}[|\mathsf{Res}(v)| \mid \text{All-act}, \text{Anc-bl}(d)]$$
$$\geq (1 + (1-\varepsilon)\frac{P}{P+C})\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v))| \mid \text{All-act}, \text{Anc-bl}(d, \mathsf{left}(v))].$$

Proof. Observe that $\varepsilon \geq 2^{\mathsf{depth}(v)-d+1}(P+C-1)/N$ since $\log(N) - 1 \geq \mathsf{depth}(v)$. By Lemma 4.4, we have that

$$\begin{aligned}
&\mathbb{E}[|\mathsf{Res}(v)| \mid \text{All-act}, \text{Anc-bl}(d, v)] \\
&\geq \mathbb{E}[|\mathsf{Res}(v)| \mid \text{All-act}, \text{Anc-bl}(d, v), B(v) = 1] \cdot \\
&\qquad \cdot \Pr[B(v) = 1 \mid \text{All-act}, \text{Anc-bl}(d, v)] \\
&\geq \mathbb{E}[|\mathsf{Res}(v)| \mid \text{All-act}, \text{Anc-bl}(d, v), B(v) = 1]\frac{1}{2}(1 + (1-\varepsilon)\frac{P}{P+C}) \\
&= 2\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v))| \mid \text{All-act}, \text{Anc-bl}(d, v), B(v) = 1]\frac{1}{2}(1 + (1-\varepsilon)\frac{P}{P+C}) \\
&= \mathbb{E}[|\mathsf{Res}(\mathsf{left}(v))| \mid \text{All-act}, \text{Anc-bl}(d, \mathsf{left}(v))](1 + (1-\varepsilon)\frac{P}{P+C}).
\end{aligned}$$

□

We are now ready to prove the section's main theorem.

Proof of Theorem 4.2. Let v be any node at depth $d(v) = d$ and let $\varepsilon > 0$ be a small constant such that $\varepsilon \geq 2^{-d}(P+C-1)$. The fact that such d and ε exist follows from the assumption that $P + C = O(1)$. We show that the expected resolution size of v is $\Omega(N^{\log_2(1+(1-\varepsilon)\frac{P}{P+C})})$. Since any commit will have a node of said depth in its copath, this is sufficient for the proof. We start with the simple observation that

$$\begin{aligned}
\mathbb{E}[|\mathsf{Res}(v)|] &\geq \mathbb{E}[|\mathsf{Res}(v)| \mid \text{All-act}, B(v) = 1] \Pr[\text{All-act}, B(v) = 1] \\
&= 2\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v))| \mid \text{All-act}, \text{Anc-bl}(d, \mathsf{left}(v))] \Pr[\text{All-act}, B(v) = 1].
\end{aligned}$$

We use Lemma 4.5 in order to obtain that

$$\begin{aligned}\Pr[\text{All-act}, B(v)=1] &\geq \Pr[B(v)=1] - \Pr[\neg\text{All-act}] \\ &\geq \Pr[B(v)=1] - \frac{1}{N}.\end{aligned}$$

It follows from repeatedly applying Lemmata 4.6 and 4.1 that

$$\begin{aligned}&\mathbb{E}[|\mathsf{Res}(v)|] \\ &\geq 2(\Pr[B(v)=1] - \frac{1}{N})\mathbb{E}[|\mathsf{Res}(\mathsf{left}(v))| \mid \text{All-act}, \text{Anc-bl}(d, \mathsf{left}(v))] \\ &\geq 2(\Pr[B(v)=1] - \frac{1}{N})(1+(1-\varepsilon)\frac{P}{P+C})^{\log_2 N - d(\mathsf{left}(v))} \\ &\geq \Omega\left(\left(\frac{\left(1-\frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}\right)\frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}}{1-\frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}\frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}} - \frac{1}{N}\right)\left(1+(1-\varepsilon)\frac{P}{P+C}\right)^{\log_2 N - d}\right).\end{aligned}$$

We make use of Equation 4 and the expression above simplifies to $\mathbb{E}[\mathsf{Cost}(t)] \geq \Omega(N^{\log_2(1+(1-\varepsilon)\frac{P}{P+C})})$, as desired. □

The proof above does not explicitly state the constant that appears in the lower bound of $\mathbb{E}[\mathsf{Cost}(t)] = \Omega(N^{\log_2(1+(1-\varepsilon)\frac{P}{P+C})})$. However, one can obtain an explicit constant from the proof. For instance if $P = 1 = C$, one can take $\varepsilon = 1/8$ and then for a node v_3 at depth 3 and a node v_4 at depth 4, the proof shows that

$$\begin{aligned}\mathbb{E}[|\mathsf{Res}(v_3)|] &\geq 2 \cdot 1/2 \cdot (1+(1-1/8)\cdot 1/2)^{\log_2 N - 3 - 1} \\ \mathbb{E}[|\mathsf{Res}(v_4)|] &\geq 2 \cdot 1/2 \cdot (1+(1-1/8)\cdot 1/2)^{\log_2 N - 4 - 1}.\end{aligned}$$

This implies that

$$\begin{aligned}\mathbb{E}[\mathsf{Cost}(t)] &\geq \mathbb{E}[|\mathsf{Res}(v_3)|] + \mathbb{E}[|\mathsf{Res}(v_4)|] \\ &\approxeq 0.39 \cdot N^{0.52}.\end{aligned}$$

For small values of N the previously shown bound of $\log^2(N)/4 - \log(N)/4$ is more relevant.

5 MLS-Cutoff: an Alternative Method for Update Proposals

We introduce MLS-Cutoff a protocol variant of MLS that achieves better communication complexity under random sequences of operations as defined in experiment Exp-Prop-Com. As discussed in the previous section, blanking the path of users issuing an update proposals already for a small constant number of proposals leads to a constant fraction of the tree's node being blank on expectation, which in turn substantially negatively impacts the size of commit messages.

To prevent introducing blanks in the tree, one could have users perform a rekey-path operation with every update proposal. In fact, this is the approach taken by the continuous group-key agreement scheme CoCoA [AAN+22]. However, taking this approach has two downsides, both due to the fact that proposals can be issued concurrently, meaning users make concurrent modifications to the ratchet tree.

1. If two users u_1, u_2 rekey their path concurrently, u_1 will encrypt a path secret s_j under a public known to u_2 that is supposed to be replaced by u_2's rekey operation. As a consequence CoCoA suffers from slower PCS. In MLS, if every compromised user issues an update proposal and another user commits to the proposals, then the new application secret will be secure. In CocoA, on the other hand, it can take up to $\mathsf{depth}(\mathsf{T})$ rounds to recover from compromise.

2. Concurrently rekeying paths prevents users to compute and sign parent hashes, as they cannot anticipate which keys will be placed in the sub-trees rooted at their co-path nodes. (Recall that the parent-hash value of node v is computed with respect to the tree-hash of its sibling $\mathsf{sib}(v)$.) As a consequence, CoCoA only achieves semi-active but neither active [ACJM20] nor insider security [AJM22].

We observe that the issues above only occur from the point on where u_1 and u_2's update paths meet. Further, if the number of issued update proposals is small, with high probability the users' update paths merge high up in the tree. Our protocol MLS-Cutoff makes use of this observation by having update proposals only re-key the path up to a certain point. Looking ahead, proposals do not affect the $\log(\mathsf{depth}(\mathsf{T}))$ topmost levels of the tree. Still, when applying the proposals it can be the case that a node is affected by more than one rekey operation if the issuing users are situated close enough in the tree. In this case we have to resort to blanking the node. This turns out to be sufficient to prevent the two issues outlined above.

5.1 Protocol Description

In this section we discuss the modifications MLS-Cutoff makes to the MLS protocol and defer its formal description to Appendix B.2. MLS-Cutoff differs in two aspects from MLS, namely how update proposal messages pmsg are generated, and how proposals are applied to the ratchet tree using apply-props.

The protocol is parameterized by a cutoff parameter i_{cut}. CGKA.Prop works the same as before for removals or additions of users. When issuing an update, however, the user creates a copy T' of the ratchet tree and calls the path rekeying function (see Figure 3) to obtain

$$(\mathsf{T}', \mathsf{UpdPath}) \leftarrow \mathsf{rekey\text{-}path}(v_u, i_{\text{cut}}).$$

The proposal message $\mathsf{pmsg} = (\texttt{'upd'}, \mathsf{UpdPath})$ is framed and communicated to the rest of the group, and T' stored as a pending update.

Commit operations work in the same way as in MLS, except for using a modified subroutine apply-props in step 2a. For an overview on apply-props see Figure 5). The function modifies the copied tree T' and processes removals and additions in the same way as MLS, i.e., by blanking the leaf and path of removed users and adding added users as unmerged leaves. Update proposals $(\texttt{'upd'}, \mathsf{UpdPath})$ from issuing user u-snd are processed by first updating the keys of affected nodes. More precisely, the public keys $\mathsf{pk}_0, \ldots, \mathsf{pk}_{\ell - i_{\text{cut}}}$ contained in UpdPath are used to replace the ones on u-snd's update path $(v_1, \ldots, v_\ell) = \mathsf{path}(v_0)$, where $v_0 = v_{u\text{-snd}}$. Further, the processing user u recovers all secret keys they should have access to, i.e., the ones from the least common ancestor v_j of v_u and $v_{u\text{-snd}}$ up to $v_{\ell - i_{\text{cut}}}$ (if the v_j is above $v_{\ell - i_{\text{cut}}}$ no secret keys are recovered), and assigns them to the corresponding nodes. This is abstracted as helper function recover-sks in Figure 5 that takes as input T' and ciphertext collection C_j and returns the updated view of T' with replaced secret keys.

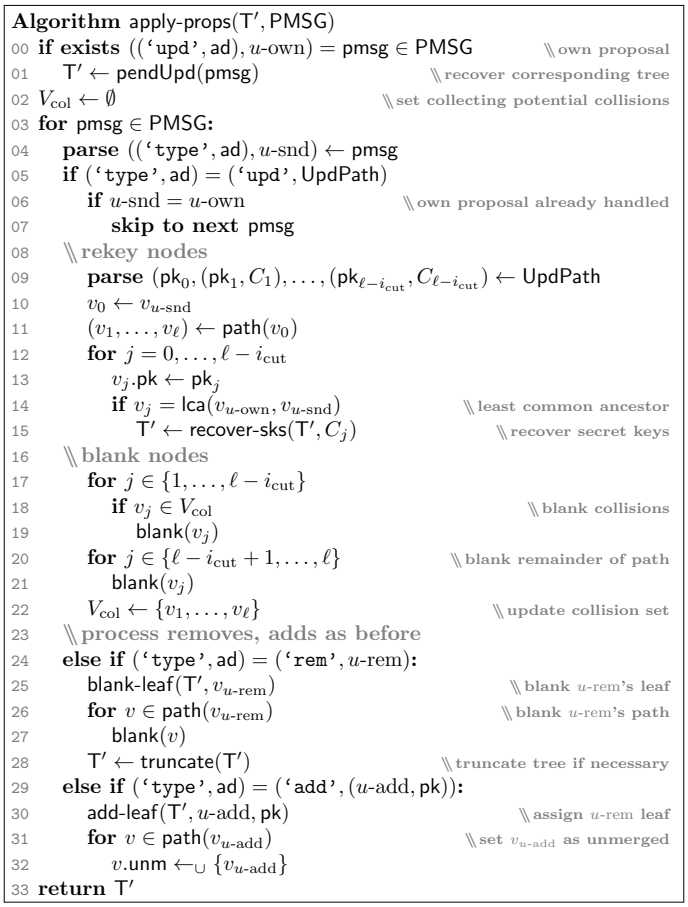
After the keys of nodes affected by pmsg have been updated, apply-props blanks nodes if necessary. I.e., all nodes $(v_{\ell - i_{\text{cut}}+1}, \ldots, v_\ell)$ on u-snd's update path after the cutoff point are blanked. Further, nodes affected by at least two update proposals are blanked as well. To this end the function keeps track of a set V_{col} containing all nodes the keys of which have been replaced so far, that is updated at the end of processing pmsg.

After having executed the alternative version of apply-props in step 2a the commit algorithm CGKA.Com proceeds in the same way as MLS. In particular, in step 2b the committing user issues a *full* path rekey operation $\mathsf{rekey\text{-}path}(u, 0)$, which establishes a new root secret.

Commits to a collection of proposals are processed in the analog way to MLS,i.e, the only difference being the use of the modified apply-props function.

5.2 Security

MLS-Cutoff achieves the same level of security as MLS. More formally in Appendix B.2 we prove the following theorem



Algorithm apply-props($\mathsf{T}', \mathsf{PMSG}$)
00 **if exists** $((\texttt{'upd'}, \mathsf{ad}), u\text{-own}) = \mathsf{pmsg} \in \mathsf{PMSG}$ \\ own proposal
01 $\quad \mathsf{T}' \leftarrow \mathsf{pendUpd}(\mathsf{pmsg})$ \\ recover corresponding tree
02 $V_{\text{col}} \leftarrow \emptyset$ \\ set collecting potential collisions
03 **for** $\mathsf{pmsg} \in \mathsf{PMSG}$**:**
04 $\quad$ **parse** $((\texttt{'type'}, \mathsf{ad}), u\text{-snd}) \leftarrow \mathsf{pmsg}$
05 $\quad$ **if** $(\texttt{'type'}, \mathsf{ad}) = (\texttt{'upd'}, \mathsf{UpdPath})$
06 $\quad\quad$ **if** $u\text{-snd} = u\text{-own}$ \\ own proposal already handled
07 $\quad\quad\quad$ **skip to next** pmsg
08 $\quad$ \\ **rekey nodes**
09 $\quad\quad$ **parse** $(\mathsf{pk}_0, (\mathsf{pk}_1, C_1), \ldots, (\mathsf{pk}_{\ell - i_{\text{cut}}}, C_{\ell - i_{\text{cut}}}) \leftarrow \mathsf{UpdPath}$
10 $\quad\quad$ $v_0 \leftarrow v_{u\text{-snd}}$
11 $\quad\quad$ $(v_1, \ldots, v_\ell) \leftarrow \mathsf{path}(v_0)$
12 $\quad\quad$ **for** $j = 0, \ldots, \ell - i_{\text{cut}}$
13 $\quad\quad\quad$ $v_j.\mathsf{pk} \leftarrow \mathsf{pk}_j$
14 $\quad\quad\quad$ **if** $v_j = \mathsf{lca}(v_{u\text{-own}}, v_{u\text{-snd}})$ \\ least common ancestor
15 $\quad\quad\quad\quad$ $\mathsf{T}' \leftarrow \mathsf{recover\text{-}sks}(\mathsf{T}', C_j)$ \\ recover secret keys
16 $\quad$ \\ **blank nodes**
17 $\quad\quad$ **for** $j \in \{1, \ldots, \ell - i_{\text{cut}}\}$
18 $\quad\quad\quad$ **if** $v_j \in V_{\text{col}}$ \\ blank collisions
19 $\quad\quad\quad\quad$ $\mathsf{blank}(v_j)$
20 $\quad\quad$ **for** $j \in \{\ell - i_{\text{cut}} + 1, \ldots, \ell\}$ \\ blank remainder of path
21 $\quad\quad\quad$ $\mathsf{blank}(v_j)$
22 $\quad\quad$ $V_{\text{col}} \leftarrow \{v_1, \ldots, v_\ell\}$ \\ update collision set
23 $\quad$ \\ **process removes, adds as before**
24 $\quad$ **else if** $(\texttt{'type'}, \mathsf{ad}) = (\texttt{'rem'}, u\text{-rem})$**:**
25 $\quad\quad$ $\mathsf{blank\text{-}leaf}(\mathsf{T}', v_{u\text{-rem}})$ \\ blank u-rem's leaf
26 $\quad\quad$ **for** $v \in \mathsf{path}(v_{u\text{-rem}})$ \\ blank u-rem's path
27 $\quad\quad\quad$ $\mathsf{blank}(v)$
28 $\quad\quad$ $\mathsf{T}' \leftarrow \mathsf{truncate}(\mathsf{T}')$ \\ truncate tree if necessary
29 $\quad$ **else if** $(\texttt{'type'}, \mathsf{ad}) = (\texttt{'add'}, (u\text{-add}, \mathsf{pk}))$**:**
30 $\quad\quad$ $\mathsf{add\text{-}leaf}(\mathsf{T}', u\text{-add}, \mathsf{pk})$ \\ assign u-rem leaf
31 $\quad\quad$ **for** $v \in \mathsf{path}(v_{u\text{-add}})$ \\ set $v_{u\text{-add}}$ as unmerged
32 $\quad\quad\quad$ $v.\mathsf{unm} \leftarrow_{\cup} \{v_{u\text{-add}}\}$
33 **return** T'

Figure 5: Applying proposal messages in MLS-Cutoff. The function's formal description is in Figures 23. We assume PMSG is ordered with update proposals followed by removals, followed by adds.

Theorem 5.1. *If* Pke *is IND-CCA2 secure,* Sig *is SEUF-CMA secure and calls to* $\mathsf{H}_1, \mathsf{H}_2$, H *and* Mac *are replaced by calls to a random oracle* $\mathcal{G}$*, then* MLS-Cutoff *securely realizes* $(\mathcal{F}_{\mathsf{AS}}^{\mathsf{I}}, \mathcal{F}_{\mathsf{KS}}^{\mathsf{I}}, \mathcal{F}_{\mathsf{CGKA}})$ *in the* $(\mathcal{F}_{\mathsf{AS}}, \mathcal{F}_{\mathsf{KS}}, \mathcal{G})$*-hybrid model.*

We provide some intuition on why the result holds. As MLS-Cutoff closely resembles MLS we are able to largely rely of the security proof of [AJM22]. The recent work of Cremers et al. [CGWZ25] shows that the signature scheme used must be SEUF-CMA secure rather than just EUF-CMA secure as originally stated in [AJM22], though their security proof already made use of SEUF-CMA security implicitly. We note that while in MLS-Cutoff update proposals lead to modifications to ratchet tree nodes beyond users' personal leaves, it preserves the following property. For an update issuing user u denote their leaf by v_0, consider their update path $(v_1, \ldots, v_\ell)$, and assume that the keys of nodes $v_1, \ldots, v_k$ have been replaced after processing a commit to the update proposal and $v_{k+1}, \ldots, v_\ell$ have been blanked. Then all sub-trees rooted at the nodes $\mathsf{sib}(v_0), ..., \mathsf{sib}(v_k)$ must be in the same state as they were when the update proposal was issued. Indeed, would this not be the case for one of the nodes, then it would have caused a collision of update proposals and been blanked. This has the consequence that, on the one hand, keys being added to the tree cannot be encrypted under a public key that was concurrently being replaced, and, on the other hand, that the added nodes are compatible with the parent hash mechanism.

5.3 Upper Bounds on the Update Cost

We compute an upper bound on the communication complexity of MLS-Cutoff for sequences of operations as produces in experiment Exp-Prop-Com. We focus on the case $C = 1$, as the performance for $C \geq 1$ is even better.

Expected number of blanks in MLS-Cutoff**'s ratchet tree.** Let $N = 2^n$, $t \in \mathbb{N}$, $C = 1$, and let $P \in \mathbb{N}$ be constant. Consider an execution of Exp-Prop-Com(n, P, C, t) with respect to CGKA MLS-Cutoff. As in Section 4.2 consider the sequence $(\mathsf{T}_i)_{i=1}^t$ of ratchet trees generated after the ith application of Exp-Prop-Com's round function. For a fixed internal node v we can define a random variable $(B_i(v))_{i=0}^t$ taking values in $\{0, 1\}$ where 0 corresponds to being a populated node and 1 to v being blank. We write B_i instead of $B_i(v)$ when there is no ambiguity. We note that all users have filtered paths of length n.

Now consider a node v below the cutoff bound i_{cut}, meaning that v's dept $d = \mathsf{depth}(v) \geq i_{\text{cut}}$, which implies that v is re-keyed if one of its descendant leaves issues an update proposal. We observe that, again, $(B_i)_i$ forms a Markov chain, since the node switching from blank or vice versa only depends on the question, whether it is affected by update proposals and/or commits.

More precisely v moves from blank to populated if it is affected by the commit or by exactly one update proposal, and from populated to blank if it is not affected by the commit and at least two update proposals.

$$\begin{aligned} p_{1\to 0} &= \Pr[v \in \mathsf{ancs}(U_C) \vee |\{u \in U_P : v \in \mathsf{ancs}(v_u)\}| = 1] \\ p_{0\to 1} &= \Pr[v \notin \mathsf{ancs}(U_C) \wedge |\{u \in U_P : v \in \mathsf{ancs}(v_u)\}| \geq 2]. \end{aligned}$$

Thus, we have

$$\begin{aligned} p_{1\to 0} &= \Pr[v \in \mathsf{ancs}(U_C)] + \Pr[|\{u \in U_P : v \in \mathsf{ancs}(v_u)\}| = 1] \\ &\quad - \Pr[v \in \mathsf{ancs}(U_C) \wedge |\{u \in U_P : v \in \mathsf{ancs}(v_u)\}| = 1] \\ &= 1 - \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}} + P(1-2^{-d})^{P-1}2^{-d} - (1 - \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}})P(1-2^{-d})^{P-1}2^{-d} \\ &= 1 + (P(1-2^{-d})^{P-1}2^{-d} - 1)\frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}, \end{aligned}$$

and

$$\begin{aligned} p_{0\to 1} &= \Pr[v \notin \mathsf{ancs}(U_C)] \Pr[|\{u \in U_P : v \in \mathsf{ancs}(v_u)\}| \geq 2] \\ &= \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}(1 - (1-2^{-d})^P - P2^{-d}(1-2^{-d})^{P-1}). \end{aligned}$$

Observe that for $C = 1$ and d sufficiently large this simplifies to

$$p_{1\to 0} \approxeq 1 - (1 - 2^{-d}) = 2^{-d}$$

and

$$p_{0\to 1} \approxeq 1 - (1-2^{-d})^P - P2^{-d}(1-2^{-d})^{P-1} \approxeq P(P-1)2^{-2d}$$

By Equation 1 we obtain that the probability of v being blank in the stationary distribution (when $d = \mathsf{depth}(v) \geq i_{\text{cut}}$) is given by

$$\mathbb{E}[B(v) = 1] = \frac{p_{0\to 1}}{p_{1\to 0} + p_{0\to 1}} \approxeq \frac{P^2 2^{-2d}}{2^{-d} + P^2 2^{-2d}} \approxeq \frac{P^2 2^{-d}}{1 + P^2 2^{-d}} \in O\left(\frac{P^2}{2^d}\right). \tag{9}$$

As a consequence, the ratchet tree exhibits on expectation a constant amount of about P^2 blank nodes on level d.

Now consider a node v above the cutoff bound i_{cut}, i.e., $d = \mathsf{depth}(v) < i_{\text{cut}}$ This implies that v changes from populated to blank if and only if (at least) one of its descendant leaves issues an update proposal and none of them issue a commit, whereas v changes from populated to blank if and only if at least one of them issues a commit. Using similar arguments to previous ones, we obtain that

$$p_{1\to 0} = \Pr[v \in \mathsf{ancs}(U_C)] = 1 - \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}$$

and

$$p_{0\to 1} = \Pr[v \notin \mathsf{ancs}(U_C) \wedge v \in \mathsf{ancs}(U_P)] = \frac{\binom{N(1-2^{-d})}{C}}{\binom{N}{C}}\left(1 - \frac{\binom{N(1-2^{-d})}{P}}{\binom{N}{P}}\right).$$

If $C = 1$ this simplifies to

$$\begin{aligned} p_{1\to 0} &= 2^{-d}, \\ p_{0\to 1} &\approxeq (1-2^{-d})(1-(1-2^{-d})^P) \approxeq (1-2^{-d})P2^{-d}. \end{aligned}$$

By Equation 1 we obtain that the probability of v being blank in the stationary distribution (when $d = \mathsf{depth}(v) < i_{\text{cut}}$) is given by

$$\mathbb{E}[B(v) = 1] = \frac{p_{0\to 1}}{p_{1\to 0} + p_{0\to 1}} \approxeq \frac{(1-2^{-d})P2^{-d}}{2^{-d} + (1-2^{-d})P2^{-d}} \in \left[0, \frac{P}{1+P}\right]. \tag{10}$$

Bounding $|\mathsf{Res}(v)|$. The size of the resolution of a node v can be bounded by the number of blank nodes in the subtree rooted at v plus one. If $C = 1$, we use this simple fact and Equations 9 and 10 in order to obtain that for nodes of $d = \mathsf{depth}(v) \geq i_{\text{cut}}$

$$\begin{aligned} \mathbb{E}[|\mathsf{Res}(v)|] &\leq 1 + \sum_{d'=\log(N)}^{d+1} 2^{d'-d}\frac{P(P-1)2^{-d'}}{1+P(P-1)2^{-d'}} \\ &\leq 1 + P(P-1)2^{-d}(\log(N) - d) \end{aligned}$$

and for nodes of $d = \mathsf{depth}(v) < i_{\text{cut}}$

$$\begin{aligned} \mathbb{E}[|\mathsf{Res}(v)|] &\leq 1 + \sum_{d'=\log(N)}^{i_{\text{cut}}} 2^{d'-d}\frac{P(P-1)2^{-d'}}{1+P(P-1)2^{-d'}} + \sum_{d'=i_{\text{cut}}-1}^{d+1} 2^{d'-d}\frac{P}{1+P} \\ &\leq 1 + P(P-1)2^{-d}(\log(N) - i_{\text{cut}}) + 2^{-d}\frac{2^d - 2^{i_{\text{cut}}}}{1-2} \\ &\leq P(P-1)2^{-d}(\log(N) - i_{\text{cut}}) + 2^{i_{\text{cut}}-d} \end{aligned}$$

Bounding $|\mathsf{pmsg}|$ and $|\mathsf{cmsg}|$. To compute a bound on the communication complexity of MLS-Cutoff for random sequences of operations, recall that by definition of $\mathsf{rekey\text{-}path}(\mathsf{T}', \mathsf{id}, i)$, if the $v_0 = v_{\mathsf{id}}$ and $(v_1, \ldots, v_\ell) = \mathsf{path}(v_0)$, the amount of ciphertexts is given by

$$|\mathsf{CTXT}| = \sum_{j=0}^{\ell - i_{\text{cut}} - 1} |\mathsf{Res}(\mathsf{sib}(v_i))|.$$

We make use the previous bounds on the size of the resolution of a node and obtain that

$$\begin{aligned}\mathbb{E}[|\mathsf{CTXT}|] &= \sum_{d=1}^{\log(N)} \mathbb{E}[|\mathsf{Res}(v_d)|] \\ &\leq \sum_{d=1}^{i_{\text{cut}}-1} (P(P-1)2^{-d}(\log(N) - i_{\text{cut}}) + 2^{i_{\text{cut}}-d}) \\ &+ \sum_{d=i_{\text{cut}}}^{\log(N)} (1 + P(P-1)2^{-d}(\log(N) - d)) \\ &\leq P(P-1)(\log(N) - i_{\text{cut}}) + 2^{i_{\text{cut}}} + \log(N) - i_{\text{cut}} + 1.\end{aligned}$$

If we set $i_{\text{cut}} = \log(n) = \log(\log(N))$, we obtain that

$$\mathbb{E}[|\mathsf{CTXT}|] \leq \log(N) + (P(P-1)+1)\log(N/\log(N)) + 1.$$

We have shown the following.

Lemma 5.2. *Let $C = 1$, and let $P \in \mathbb{N}$ be constant. Let t be sufficiently large. For $N = 2^n$ set $i_{\text{cut}} = \log(n)$. Then, on expectation, proposal messages* pmsg *and commit messages* cmsg *output by* MLS-Cutoff *for sequences of operations as defined by* Exp-Prop-Com *satisfy*

$$\mathbb{E}[|\mathsf{pmsg}|] \leq \mathbb{E}[|\mathsf{cmsg}|] \in O(\log(N)).$$

We point out that the amount of proposals per commit needs to be small compared to the group size N in order to guarantee a logarithmic communication complexity. Observe that Lemma 5.2 gives a bound that is asymptotic in N while P is assumed to be constant. If P becomes too large compared to the group size, the probability of blanks being introduced in the ratchet tree grows. In the extreme case of all users issuing update proposals, for example, the whole tree is blanked just as in MLS. We point out, that a linear communication complexity is, however, inherent for this type of 'massive' concurrency (Compare, e.g., the lower bounds of [BDR20]). As a second point, experiment Exp-Prop-Com does not remove users from the group, which would introduce blanks as well. However, observe that these blanks would be removed at a faster pace in MLS-Cutoff compared to MLS due to the additional path rekeying in update proposals.

Summing up, we see MLS-Cutoff as an attractive alternative to how update proposals are handled in MLS, that provides the same security properties, but has an improved communication complexity for moderate numbers of concurrent update proposals. We consider it an interesting open question to evaluate its performance in more realistic models.

6 Acknowledgment

We are very grateful to Guillermo Pascual Perez for the helpful conversations on MLS, its different versions and its consistency mechanisms.

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, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 222–253. Springer, Cham, November 2021.

[AAB+24] Michael Anastos, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Matthew Kwan, Guillermo Pascual-Perez, and Krzysztof Pietrzak. The cost of maintaining keys in dynamic groups with applications to multicast encryption and group messaging. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part I*, volume 15364 of *LNCS*, pages 413–443. Springer, Cham, December 2024.

[AACN+24] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzyzstof Pietrzak. Decaf: Decentralizable cgka with fast healing. In Clemente Galdi and Duong Hieu Phan, editors, *Security and Cryptography for Networks*, pages 294–313, Cham, 2024. Springer Nature Switzerland.

[AAN+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, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Cham, May / June 2022.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Cham, May 2019.

[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, Cham, 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 Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, November 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, Cham, November 2020.

[AHKM22] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, November 2022.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Cham, August 2022.

[AMT23] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 396–429. Springer, Cham, August 2023.

[ANPPP23] Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the cost of post-compromise security in concurrent continuous group-key agreement. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part III*, volume 14371 of *LNCS*, pages 271–300. Springer, Cham, November / December 2023.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. https://mailarchive.ietf.org/arch/attach/mls/pdf1XUH6o.pdf, May 2018.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[BCG23] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the MLS RFC, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021.

[BCV23] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1253–1270. USENIX Association, August 2023.

[BDG+22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Cham, November 2022.

[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, Cham, November 2020.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Berlin, Heidelberg, February 2007.

[CEST24] Kelong Cong, Karim Eldefrawy, Nigel P. Smart, and Ben Terner. The key lattice framework for concurrent group messaging. In Christina Pöpper and Lejla Batina, editors, *ACNS 24International Conference on Applied Cryptography and Network Security, Part II*, volume 14584 of *LNCS*, pages 133–162. Springer, Cham, March 2024.

[CGWZ25] Cas Cremers, Esra Günsay, Vera Wesselkamp, and Mang Zhao. ETK: External-operations TreeKEM and the security of MLS in RFC 9420. Cryptology ePrint Archive, Paper 2025/229, 2025.

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1847–1864. USENIX Association, August 2021.

[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 Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide metadata in mls-like secure group messaging: Simple, modular, and post-quantum. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1399–1412, New York, NY, USA, 2022. Association for Computing Machinery.

[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.

[Wei19] Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.

[WHA99] D. Wallner, E. Harder, and R. Agee. Key management for multicast: Issues and architectures. Request for Comments: 2627, Internet Engineering Task Force, 1999.

[WKHB21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, November 2021.

[WPBB23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1217–1233. USENIX Association, August 2023.

Supplementary material

A Omitted Formal Security Definition

A.1 Public Key Encryption

Definition A.1. *A public-key encryption (PKE) scheme is a tuple of efficient algorithms* $\mathsf{Pke} = (\mathsf{Pke.Gen}, \mathsf{Pke.Enc}, \mathsf{Pke.Dec})$ *formed by*

- *a probabilistic key generation algorithm* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Pke.Gen}()$ *that outputs a pair formed by a public key and a secret key,*
- *a probabilistic encryption algorithm that on input a public key* pk *and a message* m *outputs a ciphertext* $c \leftarrow \mathsf{Pke.Enc}(\mathsf{pk}, m)$*,*
- *a deterministic decryption algorithm that on input a ciphertext* c *and a secret key* sk *outputs a message* $m \leftarrow \mathsf{Pke.Dec}(\mathsf{sk}, c)$ *or an error value* $\perp$*.*

We say it is correct if for all $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Pke.Gen}()$ *and all messages* m *it holds that*

$$\mathsf{Pke.Dec}(\mathsf{sk}, \mathsf{Pke.Enc}(\mathsf{pk}, m)) = m.$$

Now we proceed to define security. For an adversary $\mathcal{A}$ and a PKE scheme Pke we consider the following experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Pke}}$:

1. sample $b \in\leftarrow_{\$} \{0, 1\}$,
2. sample $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{Pke.Gen}()$ and give pk to $\mathcal{A}$,
3. $\mathcal{A}$ is allowed to make decryption queries in which it sends a ciphertext c and receives $\mathsf{Pke.Dec}(\mathsf{sk}, c)$,
4. $\mathcal{A}$ sends a pair of messages (m_0, m_1) and receives a challenge ciphertext $c^* \leftarrow \mathsf{Pke.Enc}(\mathsf{pk}, m_b)$
5. $\mathcal{A}$ is allowed to make decryption queries in which it sends a ciphertext c with the condition that $c \neq c^*$ and receives $\mathsf{Pke.Dec}(\mathsf{sk}, c)$,
6. $\mathcal{A}$ outputs a bit b',
7. the output of the experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{PKE}}$ is 1 if $b = b'$ and 0 otherwise.

Definition A.2. *We say that a PKE scheme* Pke *is IND-CCA2 secure if for all efficient adversaries* $\mathcal{A}$

$$|\Pr[\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Pke}} = 1] - \frac{1}{2}|$$

is negligible.

A.2 Digital Signatures

Definition A.3. *A digital signature is a tuple of efficient algorithms* $\mathsf{Sig} = (\mathsf{GenS}, \mathsf{Sign}, \mathsf{Vrfy})$ *formed by*

- *a probabilistic key generation algorithm* $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{GenS}()$ *that outputs a pair formed by a public verification key and a secret signing key,*
- *a probabilistic encryption algorithm that on input a secret signing key* ssk *and a message* m *outputs a signature* $\sigma \leftarrow \mathsf{Sign}(\mathsf{ssk}, m)$*,*
- *a deterministic verification algorithm that on input a message signature pair* (m, σ) *and a verification key* svk *outputs a boolean value* $b \leftarrow \mathsf{Vrfy}(\mathsf{svk}, m, \sigma)$*.*

We say it is correct if for all $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{GenS}()$ *and all messages* m *it holds that*

$$\mathsf{Vrfy}(\mathsf{svk}, \mathsf{Sign}(\mathsf{ssk}, m)) = 1.$$

For an adversary $\mathcal{A}$ and a signature scheme Sig we consider the following experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sig}}$:

1. sample $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{GenS}()$ and give svk to $\mathcal{A}$,
2. $\mathcal{A}$ is allowed to make signing queries in which it sends a message m_i and receives $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{ssk}, m_i)$,
3. $\mathcal{A}$ outputs a message-signature pair (m^*, σ^*),
4. the output of $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sig}}$ is 1 if for all queries i it holds that $(m^*, \sigma^*) \neq (m_i, \sigma_i)$ and $\mathsf{Vrfy}(\mathsf{svk}, m^*, \sigma^*) = 1$. Otherwise the output is 0.

Definition A.4. *We say that a signature scheme* Sig *is SEUF-CMA secure if for all efficient adversaries* $\mathcal{A}$

$$|\Pr[\mathsf{Exp}^{\mathcal{A}}_{\mathsf{Sig}} = 1] - \frac{1}{2}|$$

is negligible.

A.3 PKI and CGKA

For the definition of the PKI functionalities see Figures 6 and 7. The reader may find a pseudo-code description of the ideal functionality $\mathcal{F}_{\mathsf{CGKA}}$ in Figures 8, 9, 10 and 11. The helpers used to describe $\mathcal{F}_{\mathsf{CGKA}}$ can be found in Figures 12, 13, 14, 15 and 16. The reader may find the definition of the predicates $\mathsf{safe}(c)$ and $\mathsf{inj\text{-}allowed}(c, u)$ in Figure 17.

B Omitted Formal Protocol Descriptions

B.1 Formal description of MLS

In this section we provide pseudocode for the CGKA underlying MLS. Our protocol description closely follows the one of [AJM22], more precisely the one of protocol ITK, a CGKA introducing fixes to a prior draft of the MLS protocol that were later adapted into the standard. As the focus on this work is on the impact of blanking on the protocol's communication complexity, makes the same simplifying assumptions. I.e., we focus on the CGKA part of the protocol and do not include its secure messaging layer, assume a fixed protocol version and cipher suite, and omit features like meta data protection and the additional proposal types PreSharedKey, ReInit, ExternalInit, GroupContextExtensions and external proposals. As a minor difference we use the parent hash mechanism as defined in the IETF standard [BBR+23] that slightly differs from the one used by ITK.

The protocol is defined with respect to public-key encryption scheme $\mathsf{Pke} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with key pairs $(\mathsf{pk}, \mathsf{sk})$, signature scheme $\mathsf{Sig} = (\mathsf{GenS}, \mathsf{Sign}, \mathsf{Vrfy})$ with key pairs $(\mathsf{svk}, \mathsf{ssk})$, MAC $\mathsf{Mac} = (\mathsf{Mac.Tag}, \mathsf{Mac.Ver})$, key-derivation function the use of which is modeled as hash functions $\mathsf{H}_1, \mathsf{H}_2$, and hash function H.

Protocol state. We list the content of users' states, as well as the intuition behind additional values computed throughout the protocol execution in Table 1.

Interaction with KS and AS. Users interact with the key service and authentication service via the following.

fetch-ssk-if-nec is used to retrieve (and potentially update) a user's secret signing key (see Figure 18).

`get-sks` is used to fetch the secret keys of key packages when joining a group (see Figure 7).

`fetch-kp`: fetches a user's key package from the KS (See Figure 7).

validate-kp is used to validate key packages (See Figure 18).

Gen-kp generates a key package (Figure 18).

state	
st.groupID	the group identifier
st.ep	the current epoch
st.id	the user's leaf identifier
st.T	the user's view of the ratchet tree
st.tHash	tree hash of st.T
st.ssk	the user's current secret signing key
st.cert-SPks	the signature verification keys associated to each user's id
st.pendUpd	a list of pending updates
st.pendComm	a list of pending commits
key schedule stored in state	
st.appSec	the current epoch's CGKA group key
st.interim-tranHash	the interim transcript hash for the following epoch
st.memKey	membership key used to frame protocol messages
st.initSec	the next epoch's initialization secret
additional values	
comSec	commit secret generated by rekey-path
joinerSec	secret for added group members, derived from initSec, comSec
confKey	key to tag confirmed transcript hash
conf-TransHash	hash encoding operations performed so far

Table 1: Users' state in MLS.

Ratchet tree state and modifications. The state of nodes in the protocol's underlying ratchet tree is listed in Table 2. The table further list functions returning particular properties of the tree, as well as functions implementing changes to the ratchet tree.

Protocol algorithms. The group creation, commit, process, join and key-recovery algorithms of MLS can be found in Figure 19. Its proposal algorithm, contrasted against the one of MLS-Cutoff, is in Figure 20. The algorithms make use of helper functionalities rekey-path, apply-rekey (both in Figure 22), and apply-props. The latter is in Figure 23, contrasted against its MLS-Cutoff variant.

Further, the protocol employs several additional helper functionalities. While these play an important role in deriving the key schedule and ensuring consistency and authenticity of protocol messages, they do not play a role in modifications to ratchet tree's state, which is the main focus of this work. Thus for formal definitions of these functions we refer to [AJM22], providing only an overview on their roles below.

init-epoch on input a user's state st returns a copy st′ of the state with incremented epoch $\mathsf{st}'.\mathsf{ep} = \mathsf{st}.\mathsf{ep} + 1$ and empty list of pending updates and commits.

derive-keys on input state st, pending state st′ and commit secret comSec first computes the joiner secret joinerSec from comSec and initSec. From this it computes confirmation key confKey, application secret appSec, membership key memKey, and updated initialization secret initSec. The latter three are stored in st′, and the function's output is $(\mathsf{st}', \mathsf{confKey})$.

derive-epochKeys on input pending state st′ and joiner secret joinerSec computes confKey, appSec, memKey, and initSec in the same way as derive-keys and returns $(\mathsf{st}', \mathsf{confKey}, \mathsf{joinerSec})$.

sign-commit receives as input the state st and value C that contains hashes of the applied commits and the UpdPath object. It returns a signature σ on the concatenation of C with the group context, id, epoch, committer leaf id.

public state of node v	
$v.\mathsf{index}$	index of the node
$v.\mathsf{pk}$	public PKE key
$v.\mathsf{pHash}$	parent hash value
$v.\mathsf{unm}$	unmerged leaves
$v.\mathsf{svk}$	signature verification key (if v is leaf)
$v.\mathsf{cred}$	credential (if v is leaf)
$v.\mathsf{kp}$	key package $v.\mathsf{kp} = (v.\mathsf{pk}, v.\mathsf{svk}, v.\mathsf{cred})$ (if v is leaf)
private state of node v	
$v.\mathsf{sk}$	secret PKE key
tree state	
$\mathsf{root}(\mathsf{T})$	root of the tree
$\mathsf{leaves}(\mathsf{T})$ returns the leaves of T	
$\mathsf{T}.\mathsf{public}$	public part of the ratchet tree
v_{id}	leaf associated to leaf identity id
$\mathsf{id}(v)$	leaf identity of node v
$\mathsf{path}(v)$	direct path from v to the root
$\mathsf{fil\text{-}path}(v)$	filtered direct path from v_0 to the root
$\mathsf{lca}(u, v)$	least common ancestor w.r.t filtered paths of leaves u, v
$\mathsf{ind\text{-}lca}(u, v)$	index (from root) of the least common ancestor w.r.t filtered paths of leaves u, v
$\mathsf{Res}(v)$	resolution of node v
$\mathsf{roster}(\mathsf{T})$	returns list of leaf IDs
modifications to tree	
$\mathsf{init\text{-}tree}$	creates ratchet tree of size 1
$\mathsf{assign\text{-}kp}(\mathsf{T}, \mathsf{id}, \mathsf{kp})$	assigns leaf associated to id key package kp
$\mathsf{add\text{-}leaf}(\mathsf{T}, \mathsf{id})$	assign id leftmost free leaf, doubles size of T if necessary
$\mathsf{truncate}(\mathsf{T})$	removes right half of T if it contains no populated leaves
$\mathsf{blank\text{-}leaf}(\mathsf{T}, v)$	sets $\mathsf{blank}(v)$ for leaf v
$\mathsf{blank\text{-}path}(\mathsf{T}, v, i)$	calls $\mathtt{blank}(v_j)$ for $v_j \in \mathsf{path}(v)$ with $j \geq i$
$\mathsf{set\text{-}as\text{-}unmerged}(\mathsf{T}, v_0)$	sets $v.\mathsf{unm} \leftarrow_{\cup} \{v_0\}$ for all $v \in \mathsf{path}(v_0)$
$\mathsf{merge\text{-}leaves}(\mathsf{T}, v_0)$	sets $v.\mathsf{unm} \leftarrow \emptyset$ for all $v \in \mathsf{path}(v_0)$

Table 2: Ratchet tree state and functions modifying the ratchet tree.

set-int-transHash on input pending state st′ and confirmation tag confTag computes the interim transcript hash interim-tranHash as a hash of the confirmed transcript hash stored in st′ and confTag. It stores the resulting value in st′.

set-conf-trans-hash on input state st, pending state st′, sender id id-snd, C, and signature σ as above computes the confirmed transcript hash conf-TransHash as a hash of the latter three values and the interim transcript hash stored in st′. The confirmed transcript hash conf-TransHash is stored in st′.

compute-confTag on input pending state st′ and confirmation key confKey returns a MAC confTag of the confirmed transcript hash conf-TransHash contained in st′.

verify-confTag on input pending statest′, confirmation key confKey, and confirmation tag confTag verifies the tag.

frame-prop frames proposal $P = (\texttt{'type'}, \mathsf{ad})$ by signing and MACing (under the membership key memKey) P together with group context, group id, epoch, leaf id of the sender.

unframe-prop on input state st and framed proposal pmsg, verifies signature and MAC tag and returns P as well as the sender id.

frame-comm on input state st, confirmation tag confTag, signature σ (as in sign-commit), and membership tag memTag frames the commit by appending group id, epoch, and id of the committer.

unframe-comm on input state st and framed commit cmsg, verifies the contained signature and MAC tag. If both verify it returns the sender's id, C, the confirmation tag, signature, and the membership tag.

B.2 Formal description of MLS-Cutoff

We provide a formal description of protocol MLS-Cutoff. It is defined with respect to public key encryption scheme Pke, Signature scheme Sign, message authentication code Mac, hash function H, and a key-derivation function the use of which is denoted by H_1 and H_2. The protocol is parameterized by cutoff parameter i_{cut}. The protocol shares most algorithms with MLS differing only in two aspects:

MLS-Cutoff generates proposal messages in a different way. Its proposal algorithm CGKA.Propcan be found in Figure 20 (right).

MLS-Cutoff uses a different method of applying proposals to the ratchet tree. I.e., algorithms CGKA.Com and CGKA.Proc make use of the subroutine apply-props of Figure 23 (right).

All remaining aspects are handled as in MLS. I.e., the protocol has the same user state (except for storing the cutoff bound in $\mathsf{st}.i_{\text{cut}}$ that remains unchanged throughout the protocol execution), uses CGKA.Create, CGKA.Com, CGKA.Proc, CGKA.Join, and CGKA.Key as defined in Figure 19, and makes use of the subroutines listed in Appendix B.1 and Figures 22 and 18. Note that when rekeying paths while applying updates MLS-Cutoff calls apply-rekey with option $\texttt{copySt} = \texttt{True}$. This means the parent hashes contained in each update proposal are verified with respect to a tree resulting from applying the modifications to st.T instead of the pending tree T′. Thus, the contained key package verifies if the update message was honestly generated, which implies the parent hash was computed with respect to such a tree.

Security of MLS-Cutoff.

Proof of Theorem 5.1. We are able to closely follow the security proof for ITK in [AJM22]. The proof is divided into two parts. First a sequence of hybrids shows that a version of the MLS protocol (MLS*) that has a different parent hash securely realizes $\mathcal{F}_{\mathsf{CGKA}}$ with a slightly different version of the safe predicate denoted safe*. The second step consists in showing that using the actual parent hash guarantees and the actual safe predicate satisfy the following property: if $\mathsf{safe}(c) = \texttt{true}$, then no node in the ratchet tree of the epoch that corresponds to c contains an exposed key pk.

We argue that both steps can be adapted to the case of MLS-Cutoff. We consider the following hybrids:

- Hybrid $\mathcal{H}_0$: Real world.
- Hybrid $\mathcal{H}_1$: Instead of the protocol as in the real world, now we consider a dummy functionality $\mathcal{F}_{\mathsf{Dummy}}$ that simply routs its inputs and outputs through a simulator $\mathcal{S}_1$ that executes the protocol.
- Hybrid $\mathcal{H}_2$: $\mathcal{F}_{\mathsf{Dummy}}$ is substituted by the ideal functionalities $\mathcal{F}_{\mathsf{AS}}, \mathcal{F}_{\mathsf{KS}}$ and $\mathcal{F}_{\mathsf{CGKA}}$ with the only change that $\mathcal{F}_{\mathsf{CGKA}}$ uses as predicates $\mathsf{safe}(c) = \texttt{false}$ and $\mathsf{inj\text{-}allowed}(c, \mathsf{id}) = \texttt{true}$. The simulator $\mathcal{S}_2$ still executes the actual protocol.
- Hybrid $\mathcal{H}_3$: now $\mathcal{F}_{\mathsf{CGKA}}$ uses the actual predicate safe^*. The simulator $\mathcal{S}_3$ proceeds as before except that it only sets applications secrets when $\mathsf{safe}^*(c) = \texttt{false}$.
- Hybrid $\mathcal{H}_4$: Ideal world. The difference with the previous hybrid consists in also using the actual predicate $\mathsf{inj\text{-}allowed}$. The simulator $\mathcal{S}_4$ is the same as $\mathcal{S}_3$.

Clearly the first two hybrids are indistinguishable. Let's argue that $\mathcal{H}_1$ and $\mathcal{H}_2$ are indistinguishable. Despite the fact that in MLS-Cutoff update proposals are created differently, we still frame the proposals in the same way and this guarantees consistency by the same argument as in the proof for MLS.

Concerning $\mathcal{H}_2$ and $\mathcal{H}_3$ the only change in the argument that has to be made corresponds to the fact that $\mathsf{rekey\text{-}path}$ is also used in proposals. Therefore the GSD graph also contains nodes and edges corresponding to these new keys just as is the case for commits in [AJM22].

The indistinguishability of hybrids $\mathcal{H}_3$ and $\mathcal{H}_4$ follows from the argument made in [AJM22]. Therefore it just remains to show that: if $\mathsf{safe}(c) = \texttt{true}$, then no node in the ratchet tree of the epoch that corresponds to c contains an exposed key pk. Here we make use of the fact that keys ending up in the ratchet tree after processing commits to update proposals were never encrypted under keys being concurrently updated. In the inductive step we have to consider an additional case, namely, that in which the node v for which $v.\mathsf{pk}$ has been exposed is set as part of proposal that is included in some commit c and the user who generated the commit is a user outsider v's subtree (in the ratchet tree T that corresponds to that commit). We denote by id_c the leaf of the party that generated the commit c.

It may be the case that the parent hash of v, pHash_v, cannot be verified by using the parent hash of its parent $v.\mathsf{par}$ in T as it would be the case when $v.\mathsf{par}$ is blanked. However, all nodes contained in the ratchet tree are parent hash valid, as defined in the MLS standard, which uses a slightly modified parent hash definition compared to ITK.

However the node at which the path of v to the root and the path of id_c to the root merge contains a commitment to the value of pHash_v (by definition of $\mathsf{par\text{-}hash\text{-}cochild}$) which can be verified and this is done by $\mathsf{verify\text{-}treeState}$. □

Initialization

00 $\mathsf{Registered}, \mathsf{Exposed} \leftarrow \emptyset, \mathsf{SSK}[*,*] \leftarrow \perp, \mathsf{RndCor}[*] \leftarrow \texttt{good}$

Input `register-spk` **from a party** u

01 **if** $\mathsf{RndCor}[u] = \texttt{good}$ **then** $(\mathsf{svk}, \mathsf{ssk}) \leftarrow_{\$} \mathsf{GenS}()$

02 **else**

03 **send** `rnd` **to the adversary and receive** r

04 $(\mathsf{svk}, \mathsf{ssk}) \leftarrow \mathsf{GenS}(r)$

05 **send** $(\texttt{sample-ssk}, u)$ **to the adversary and receive** $(\mathsf{svk}, \mathsf{ssk})$

06 **if** $\mathsf{RndCor}[u] \neq \texttt{good}$ **then** $\mathsf{Exposed} \leftarrow_{\cup} \{\mathsf{svk}\}$

07 $\mathsf{SSK}[u, \mathsf{svk}] \leftarrow \mathsf{ssk}$

08 $\mathsf{Registered} \leftarrow_{\cup} \{(u, \mathsf{svk})\}$

09 **send** $(\texttt{register-spk}, u, \mathsf{svk})$ **to the adversary**

10 **send** svk **to** u

Input $(\texttt{verify-cert}, v, \mathsf{svk})$ **from a party** u

11 **Send** $(v, \mathsf{svk}) \in \mathsf{Registered}$ **to** u

Input $(\texttt{get-ssk}, \mathsf{svk})$ **from a party** u

12 **Send** $\mathsf{SSK}[u, \mathsf{svk}]$ **to** u

Input $(\texttt{delete-ssk}, \mathsf{svk})$ **from a party** u

13 $\mathsf{SSK}[u, \mathsf{svk}] \leftarrow \perp$

Input $(\texttt{register-spk}, u, \mathsf{svk})$ **from the adversary**

14 **if** $(*, \mathsf{svk}) \notin \mathsf{Registered}$ **then** $\mathsf{Exposed} \leftarrow_{\cup} \{\mathsf{svk}\}$

15 $\mathsf{Registered} \leftarrow_{\cup} \{(u, \mathsf{svk})\}$

Input $(\texttt{expose}, u)$ **from the adversary**

16 $\mathsf{Exposed} \leftarrow_{\cup} \{\mathsf{svk} \mid \mathsf{SSK}[u, \mathsf{svk}] \neq \perp\}$

17 **Send** $\mathsf{SSK}[u, *]$ **to the adversary**

Input $(\texttt{CorrRand}, u, b)$ **from the adversary where** $b \in \{\texttt{good}, \texttt{bad}\}$

18 $\mathsf{RndCor}[u] \leftarrow b$

Input $(\texttt{exposed}, u, \mathsf{svk})$ **from** $\mathcal{F}_{\mathsf{KS}}$

19 $\mathsf{Exposed} \leftarrow_{\cup} \{\mathsf{svk}\}$

20 **Send** $\mathsf{SSK}[u, \mathsf{svk}]$ **to the adversary**

Input $(\texttt{has-ssk}, \mathsf{svk}, u)$ **from** $\mathcal{F}_{\mathsf{CGKA}}$

21 **Send** $\mathsf{SSK}[u, \mathsf{svk}] \neq \perp$ **to** $\mathcal{F}_{\mathsf{CGKA}}$

Figure 6: Functionality $\mathcal{F}_{\mathsf{AS}}$ (parametrized by a key generation algorithm for a signature scheme GenS) and the ideal functionality $\mathcal{F}_{\mathsf{AS}}^{\mathsf{I}}$. Yellow lines (●) are only executed by $\mathcal{F}_{\mathsf{AS}}$ while cyan lines (●) are only executed by $\mathcal{F}_{\mathsf{AS}}^{\mathsf{I}}$.

Initialization

00 $\mathsf{SK}[*,*], \mathsf{SVK} \leftarrow \bot, \mathsf{RndCor}[*] \leftarrow \mathtt{good}$

Input $(\mathtt{register\text{-}kp}, \mathsf{svk}, \mathsf{ssk})$ **from a party** u

01 **if** $\mathsf{RndCor}[u] = \mathtt{good}$ **then** $(\mathsf{kp}, \mathsf{sk}) \leftarrow_{\$} \mathsf{Gen\text{-}kp}()$

02 **if** $\mathsf{kp} \neq \bot$ **then return**

03 **else**

04 **send** $\mathtt{rnd}$ **to the adversary and receive** r

05 $(\mathsf{kp}, \mathsf{sk}) \leftarrow \mathsf{Gen\text{-}kp}(r)$

06 **if** $\mathsf{kp} \neq \bot$ **then return**

07 **send** $(\mathtt{exposed}, u, \mathsf{svk})$ **to** $\mathcal{F}_{\mathsf{AS}}$

08 **send** ssk **to the adversary**

09 **send** $(\mathtt{sample\text{-}sk}, u, \mathsf{svk}, \mathsf{ssk})$ **to the adversary and receive** $(\mathsf{kp}, \mathsf{sk}, ack)$

10 **if** $\neg ack$ **then return**

11 **if** $\mathsf{RndCor}[u] \neq \mathtt{good}$ **then**

12 **send** $(\mathtt{exposed}, u, \mathsf{svk})$ **to** $\mathcal{F}_{\mathsf{AS}}$

13 $\mathsf{SK}[u, \mathsf{kp}] \leftarrow \mathsf{sk}, \mathsf{SVK}[u, \mathsf{kp}] \leftarrow \mathsf{svk}$

14 **send** $(\mathtt{register\text{-}kp}, u, \mathsf{svk}, \mathsf{kp})$ **to the adversary**

15 **send** kp **to** u

Input $\mathtt{get\text{-}sks}$ **from a party** u

16 **Send** $\{(\mathsf{kp}, \mathsf{SK}[u, \mathsf{kp}]) \mid \mathsf{SK}[u, \mathsf{kp}] \neq \bot\}$ **to** u

Input $(\mathtt{fetch\text{-}kp}, v)$ **from a party** u

17 **send** $(\mathtt{fetch\text{-}kp}, u, v)$ **to the adversary and receive** kp

18 **send**kp **to** u

Input $(\mathtt{delete\text{-}sk}, \mathsf{svk})$ **from a party** u

19 $\mathsf{SK}[u, \mathsf{kp}] \leftarrow \bot$

Input $(\mathtt{expose}, u)$ **from the adversary**

20 **Send** $\mathsf{SK}[u, *]$ **to the adversary**

Input $(\mathtt{CorrRand}, u, b)$ **from the adversary where** $b \in \{\mathtt{good}, \mathtt{bad}\}$

21 $\mathsf{RndCor}[u] \leftarrow b$

Figure 7: Functionality $\mathcal{F}_{\mathsf{KS}}$ (parametrized by a key generation algorithm for key packages $\mathsf{Gen\text{-}kp}$) and the ideal functionality $\mathcal{F}_{\mathsf{KS}}^{\mathsf{I}}$. Yellow lines (●) are only executed by $\mathcal{F}_{\mathsf{KS}}$ while cyan lines (●) are only executed by $\mathcal{F}_{\mathsf{KS}}^{\mathsf{I}}$.

Initialization

00 $\mathsf{Ptr}[*], \mathsf{Node}[*], \mathsf{Prop}[*], \mathsf{Wel}[*] \leftarrow \perp$
01 $\mathsf{RndCor}[*] \leftarrow \texttt{good}$
02 $\mathsf{HasKey}[*] \leftarrow \texttt{false}$
03 $\mathsf{rootCtr} \leftarrow 0$
Input $(\texttt{create}, \mathsf{svk})$ **from** $u_{\mathbf{creator}}$ \\ Used only once
04 **req** $\mathsf{Node}[root_0] = \perp$
05 **req** $\mathsf{valid\text{-}vsk}(u_{\mathbf{creator}}, \mathsf{svk})$
06 $\mathsf{orig} \leftarrow u_{\mathbf{creator}}$
07 $\mathsf{mem} \leftarrow \{(u_{\mathbf{creator}}, \mathsf{svk})\}$
08 $\mathsf{Node}[root_0] \leftarrow \mathsf{create\text{-}root}(\mathsf{orig}, \mathsf{mem}, \mathsf{RndCor}[\mathsf{orig}])$
09 $\mathsf{HasKey}[u_{\mathbf{creator}}] \leftarrow \texttt{true}$
10 $\mathsf{Ptr}[u_{\mathbf{creator}}] \leftarrow root_0$

Figure 8: Initialization of $\mathcal{F}_{\mathsf{CGKA}}$.

Input $(\texttt{propose}, \mathsf{act})$ **from** u
00 **req** $\mathsf{Ptr}[u] \neq \bot$
01 **send** $(\texttt{propose}, u, \mathsf{act})$ **to the adversary and receive** (p, svk_v, ack)
02 **if** $\neg$req-correct-prop(u, act) **then req** ack
03 **if** $\mathsf{act} = (\mathsf{upd}, \mathsf{svk})$ **then assert** valid-vsk(u, svk)
04 **if** $\mathsf{act} = (\mathsf{add}, v)$ **then** $\mathsf{act} \leftarrow (\mathsf{add}, v, \mathsf{svk}_v)$
05 **if** $\mathsf{Prop}[p] = \bot$ **then**
06 $\quad \mathsf{Prop}[p] \leftarrow$ create-prop$(\mathsf{Ptr}[u], u, \mathsf{act}, \mathsf{RndCor}[u])$
07 **else** consistent-prop$(p, u, \mathsf{act}, \mathsf{RndCor}[u])$
08 **if** $\mathsf{RndCor}[u] = \texttt{bad}$ **then send** $(\texttt{exposed}, u, \mathsf{svk})$ **to** $\mathcal{F}_{\mathsf{AS}}$
09 **return** p
Input $(\texttt{commit}, P, \mathsf{svk})$ **from** u
10 **req** $\mathsf{Ptr}[u] \neq \bot$
11 **send** $(\texttt{commit}, u, P, \mathsf{svk})$ **to the adversary and receive** (c, w, ack, rt)
12 **if** $\neg$req-correct-commit(u, P, svk) **then req** ack
13 fill-props(u, P)
14 **assert** valid-vsk(u, svk)
15 $\mathsf{mem} \leftarrow$ members(c, u, P, svk)
16 **assert** $\mathsf{mem} \neq \bot \ \wedge (u, \mathsf{svk}) \in \mathsf{mem}$
17 **if** $\mathsf{Node}[c] = \bot \ \wedge \ rt = \bot$ **then**
18 $\quad \mathsf{Node}[c] \leftarrow$ create-child$(\mathsf{Ptr}[u], u, P, \mathsf{mem}, \mathsf{RndCor}[u])$
19 **else**
20 $\quad$ **if** $\mathsf{Node}[c] = \bot$ **then** $c' \leftarrow root_{rt}$
21 $\quad$ **else** $c' \leftarrow c$
22 $\quad$ consistent-commit(c', u, P, mem)
23 $\quad$ **if** $c \neq c'$ **then** attach(c, c', u, P)
24 **assert** $w \neq \bot$ **if and only if** $\exists p \in P$: $\mathsf{Node}[p].\mathsf{act} = (\mathsf{add}, *)$
25 **if** $w \neq \bot$ **then**
26 $\quad$ **assert** $\mathsf{Wel}[w] \in \{\bot, c\}$
27 $\quad \mathsf{Wel}[w] \leftarrow c$
28 **assert** cons-invariant $\wedge$ auth-invariant
29 **if** $\mathsf{RndCor}[u] = \texttt{bad}$ **then send** $(\texttt{exposed}, u, \mathsf{Node}[u].\mathsf{mem}[u])$ **to** $\mathcal{F}_{\mathsf{AS}}$
30 **return** (c, w)

Figure 9: Proposals, commits and process of $\mathcal{F}_{\mathsf{CGKA}}$.

Input $(\texttt{process}, c, P)$ **from** u
00 **send** $(\texttt{process}, u, c, P)$ **to the adversary and receive** $(rt, \mathsf{orig}', \mathsf{svk}', ack)$
01 **if** $\neg$req-correct-proc(u, c, P) **then req** ack
02 fill-props(u, P)
03 **if** Node$[c] = \bot \ \wedge \ rt = \bot$ **then**
04 mem $\leftarrow$ members(Ptr$[u]$, orig$'$, P, svk$'$)
05 **assert** mem $\neq \bot \ \wedge$ inj-allowed(Ptr$[u]$, u)
06 **else**
07 **if** Node$[c] = \bot$ **then** $c' \leftarrow root_{rt}$
08 **else** $c' \leftarrow c$
09 $v \leftarrow$ Node$[c']$.orig
10 $\mathsf{svk}_v \leftarrow$ Node$[c']$.mem$[v]$
11 mem $\leftarrow$ members(Ptr$[u]$, v, P, svk_v)
12 **assert** mem $\neq \bot$
13 valid-succesor$(c', u, P,$ mem$)$
14 **if** $c \neq c'$ **then** attach(c, c', u, P)
15 **if** $\exists p \in P$: Prop$[p]$.act $= (\mathsf{rem}, u)$ **then** Ptr$[u] \leftarrow \bot$
16 **else**
17 **assert** $u \in$ Node$[c]$.mem
18 Ptr$[u] \leftarrow c$
19 HasKey$[u] \leftarrow \texttt{true}$
20 **assert** cons-invariant $\wedge$ auth-invariant
21 $(*,$ PropSemantics$) \leftarrow$ apply-props$(c,$ Node$[c]$.pro$)$
22 **return** (Node$[c]$.orig, PropSemantics)
Input $(\texttt{join}, w)$ **from** u
23 **send** $(\texttt{join}, u, w)$ **to the adversary and receive** $(c', \mathsf{orig}', \mathsf{mem}', ack)$
24 **req** ack
25 $c \leftarrow$ Wel$[w]$
26 **if** $c = \bot$ **then**
27 **if** Node$[c'] \neq \bot$ **then** $c \leftarrow c'$
28 **else**
29 rootCtr $\leftarrow$ rootCtr $+ 1$
30 $c \leftarrow root_{\mathsf{rootCtr}}$
31 Node$[c] \leftarrow$ create-root(orig$'$, mem$'$, adv)
32 Wel$[w] \leftarrow c$
33 Ptr$[u] \leftarrow c$
34 HasKey$[u] \leftarrow \texttt{true}$
35 **assert** $u \in$ Node$[c]$.mem $\wedge$ cons-invariant $\wedge$ auth-invariant
36 **return** (Node$[c]$.mem, Node$[c]$.orig)
Input key **from** u
37 **req** Ptr$[u] \neq \bot \ \wedge$ HasKey$[u]$
38 **if** Node$[u]$.key $= \bot$ **then** set-key(Ptr$[u]$)
39 HasKey$[u] \leftarrow \texttt{false}$
40 **return** Node[Ptr$[u]$].key

Figure 10: Process, join and key in $\mathcal{F}_{\mathsf{CGKA}}$.

Input $(\texttt{expose}, u)$ **from the adversary**
00 **This input is not allowed if** $\exists c$ **such that** $\mathsf{Node}[c].\mathsf{chall} \ \wedge \ \neg\mathsf{safe}(c)$
01 **if** $\mathsf{Ptr}[u] \neq \bot$ **then**
02 $\quad \mathsf{Node}[\mathsf{Ptr}[u]].\mathsf{exp} \leftarrow_\cup \{(u, \mathsf{HasKey}[u])\}$
03 $\quad$ update-stat-after-exp(u)
04 $\quad$ **send** $(\texttt{exposed}, u, \mathsf{Node}[u].\mathsf{mem}[u])$ **to** $\mathcal{F}_{\mathsf{AS}}$
05 **send** `get-sks` **to** $\mathcal{F}_{\mathsf{KS}}$ **and receive** SK **and** SVK**.**
06 **for each** kp **such that** $\mathsf{SK}[u, \mathsf{kp}] \neq \bot \ \wedge \ \mathsf{SVK}[u, \mathsf{kp}] = \mathsf{svk}$ **do**
07 $\quad$ **for each** c **such that** $\exists p \in \mathsf{Node}[c].\mathsf{pro}\colon \mathsf{Prop}[p].\mathsf{act} = (\mathsf{add}, u, \mathsf{svk})$ **do**
08 $\quad\quad \mathsf{Node}[c].\mathsf{exp} \leftarrow_\cup \{(u, \texttt{true})\}$
Input $(\texttt{CorrRand}, u, b)$ **from the adversary where** $b \in \{\mathsf{good}, \mathsf{bad}\}$
09 $\mathsf{RndCor}[u] \leftarrow b$

Figure 11: Corruptions in $\mathcal{F}_{\mathsf{CGKA}}$.

Helper create-root$(u, \mathsf{mem}, \mathsf{stat})$
00 **return new node with** $\mathsf{par} \leftarrow \bot, \mathsf{orig} \leftarrow u, \mathsf{pro} \leftarrow \bot, \mathsf{mem} \leftarrow \mathsf{mem}, \mathsf{stat} \leftarrow \mathsf{stat}$
Helper create-prop$(c, u, \mathsf{act}, \mathsf{stat})$
01 **return new proposal node with** $\mathsf{par} \leftarrow c, \mathsf{orig} \leftarrow u, \mathsf{act} \leftarrow \mathsf{act}, \mathsf{stat} \leftarrow \mathsf{stat}$
Helper create-child$(c, u, P, \mathsf{mem}, \mathsf{stat})$
02 **return new node with** $\mathsf{par} \leftarrow c, \mathsf{orig} \leftarrow u, \mathsf{pro} \leftarrow P, \mathsf{mem} \leftarrow \mathsf{mem}, \mathsf{stat} \leftarrow \mathsf{stat}$
Helper fill-props(u, P)
03 **for** $p \in P$ **such that** $\mathsf{Prop}[p] = \bot$ **do**
04 $\quad$ **send** $(\texttt{propose}, p)$ **to the adversary and receive** $(\mathsf{orig}, \mathsf{act})$
05 $\quad \mathsf{Prop}[p] \leftarrow$ create-prop$(\mathsf{Ptr}[u], \mathsf{orig}, \mathsf{act}, \mathsf{adv})$

Figure 12: Helpers for creating new nodes in $\mathcal{F}_{\mathsf{CGKA}}$.

Helper valid-vsk(u, svk')
00 $\mathsf{svk} \leftarrow \mathsf{Node}[\mathsf{Ptr}[u]].\mathsf{mem}[u]$
01 **if** $\mathsf{svk} \neq \bot \ \wedge \ \mathsf{svk} = \mathsf{svk}'$ **then return** `true`
02 **send** $(\texttt{has-ssk}, \mathsf{svk}', u)$ **to** $\mathcal{F}_{\mathsf{AS}}$ **and receive** ack
03 **return** ack
Helper set-key(c)
04 **if** $\neg\mathsf{safe}(c)$ **then**
05 **send** $(\texttt{key}, u)$ **to the adversary and receive** K
06 $\mathsf{Node}[c].\texttt{key} \leftarrow \mathrm{K}, \mathsf{Node}[c].\mathsf{chall} \leftarrow \texttt{false}$
07 **else** $\mathsf{Node}[c].\texttt{key} \leftarrow_{\$} \mathcal{K}, \mathsf{Node}[c].\mathsf{chall} \leftarrow \texttt{true}$
Helper update-stat-after-exp(u)
08 **for each** p **such that** $\mathsf{Prop}[p] \neq \bot$ **and**
09 $\mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[u]$ **and**
10 $\mathsf{Prop}[p].\mathsf{orig} = u$ **and**
11 $\mathsf{Prop}[p].\mathsf{act} = \mathsf{upd}$
12 **do** $\mathsf{Prop}[p].\mathsf{stat} \leftarrow \mathsf{bad}$
13 **for each** c **such that** $\mathsf{Node}[c] \neq \bot$ **and**
14 $\mathsf{Node}[c].\mathsf{par} = \mathsf{Ptr}[u]$ **and**
15 $\mathsf{Node}[c].\mathsf{orig} = u$
16 **do** $\mathsf{Node}[c].\mathsf{stat} \leftarrow \mathsf{bad}$
Helper members(c, u, P, svk)
17 $(G, *) \leftarrow$ apply-props(c, u, P, svk)
18 **if** $(G, *) = \bot$ **then return** $\bot$
19 **else return** G

Helper apply-props(c, u, P, svk)
20 req $\mathsf{Node}[c] \neq \bot \ \wedge \ (u, *) \in \mathsf{Node}[c].\mathsf{mem}$
21 req $\forall p \in P\colon \mathsf{Prop}[p] \neq \bot \ \wedge \ \mathsf{Prop}[p].\mathsf{par} = c$
22 $G \leftarrow \mathsf{Node}[c].\mathsf{mem}$
23 $G \leftarrow G \setminus \{(u, *)\}$
24 $G \leftarrow_{\cup} \{(u, \mathsf{svk})\}$
25 $L \leftarrow \{u\}$
26 for each $p \in P$ such that $\mathsf{Prop}[p].\mathsf{act} = (\mathsf{upd}, *)$ do
27 $(v, (\mathsf{upd}, \mathsf{svk}')) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
28 req $v \in G \setminus L$
29 $G \leftarrow G \setminus \{(v, *)\}$
30 $G \leftarrow_{\cup} \{(v, \mathsf{svk}')\}$
31 $L \leftarrow_{\cup} \{v\}$
32 for each $p \in P$ such that $\mathsf{Prop}[p].\mathsf{act} = (\mathsf{rem}, *)$ do
33 $(v, (\mathsf{rem}, v')) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
34 req $v \in G \ \wedge \ v' \in G \setminus L$
35 $G \leftarrow G \setminus \{(v', *)\}$
36 for each $p \in P$ such that $\mathsf{Prop}[p].\mathsf{act} = (\mathsf{add}, *)$ do
37 $(v, (\mathsf{add}, v', \mathsf{svk}_{v'})) \leftarrow (\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})$
38 req $v \in G \ \wedge \ v' \notin G$
39 $G \leftarrow_{\cup} \{(v', \mathsf{svk}_{v'})\}$
40 $\mathsf{PropSemantics} \leftarrow ((\mathsf{Prop}[p].\mathsf{orig}, \mathsf{Prop}[p].\mathsf{act})\colon p \in P)$
41 return $(G, \mathsf{PropSemantics})$

Figure 13: Other helpers for $\mathcal{F}_{\mathsf{CGKA}}$.

Helper consistent-prop$(p, u, \mathsf{act}, \mathsf{stat})$
00 **assert** $\mathsf{Prop}[p].\mathsf{orig} = u \ \wedge \ \mathsf{Prop}[p].\mathsf{act} = \mathsf{act} \ \wedge \ \mathsf{Prop}[p].\mathsf{par} = \mathsf{Ptr}[u]$
Helper valid-succesor(c, u, P, mem)
01 $\mathsf{Node}[c] \neq \bot \ \wedge \ \mathsf{Node}[c].\mathsf{mem} = \mathsf{mem} \ \wedge \ \mathsf{Node}[c].\mathsf{pro} \in \{\bot, P\} \ \wedge \ \mathsf{Node}[c].\mathsf{par} \in \{\bot, \mathsf{Ptr}[u]\}$
Helper consistent-commit(c, u, P, mem)
02 **assert** valid-succesor(c, u, P, mem)
03 **assert** $\mathsf{RndCor}[u] \neq \mathsf{good} \ \wedge \ \mathsf{Node}[c].\mathsf{orig} = u$
Helper attach(c, c', u, P)
04 **assert** $c' \neq root_0$
05 $\mathsf{Node}[c'].\mathsf{par} \leftarrow \mathsf{Ptr}[u], \mathsf{Node}[c'].\mathsf{pro} \leftarrow P, \mathsf{Node}[c] \leftarrow \mathsf{Node}[c'], \mathsf{Node}[c'] \leftarrow \bot$
06 **for each** w **such that** $\mathsf{Wel}[w] = c'$ **do** $\mathsf{Wel}[c] \leftarrow c$

Figure 14: Consistency helpers for $\mathcal{F}_{\mathsf{CGKA}}$.

```
Helper req-correct-prop(u, act)
00 if act = (rem, v) then
01    return v ∈ Node[Ptr[u]].mem
02 else if act = (upd, svk) then
03    return valid-vsk(u, svk)
04 else return false
Helper req-correct-commit(u, P, svk)
05 return apply-props(u, P, svk) ≠ ⊥ ∧ valid-vsk(u, svk)
Helper req-correct-proc(u, c, P)
06 return Node[c] ≠ ⊥ ∧ Node[c].par = Ptr[u] ∧ Node[c].pro = P ∧
07     Node[c].stat ≠ adv ∧ ∀p ∈ P: Prop[p].stat ≠ adv
```

Figure 15: Correctness helpers for $\mathcal{F}_{\mathsf{CGKA}}$.

```
Helper auth-invariant
00 return true iff
01    ∀c if Node[c].stat = adv then inj-allowed(Node[c].par, Node[c].orig) and
02    ∀p if Prop[p].stat = adv then inj-allowed(Prop[p].par, Prop[p].orig)
Helper cons-invariant
03 return true iff
04    ∀c such that Node[c].par ≠ ⊥ it holds that
05       Node[c].pro ≠ ⊥ and ∀p ∈ Node[c].pro: Prop[p].par = Node[c].par
06    and ∀u such that Ptr[u] ≠ ⊥: u ∈ Node[Ptr[u]].mem
07    and the history graph contains no cycles
```

Figure 16: Invariant helpers for $\mathcal{F}_{\mathsf{CGKA}}$.

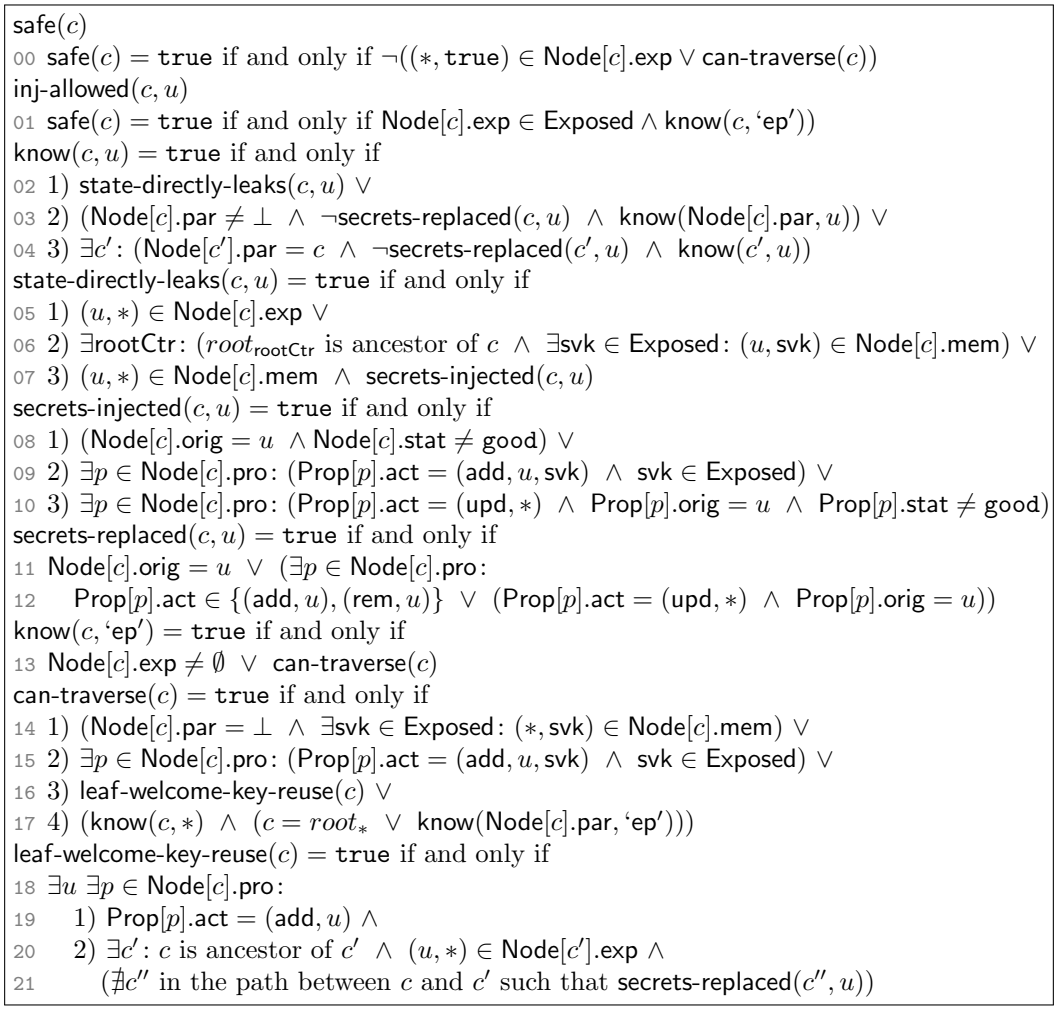


Figure 17: Predicates safe(c) and inj-allowed(c, u).

Helper fetch-ssk-if-nec(st, svk)
00 **if** $v_{\mathsf{st.id}}$.svk $\neq$ svk **then**
01 **send** (get-ssk, svk) **to** $\mathcal{F}_{\mathsf{AS}}$ **and receive** ssk
02 **else** ssk $\leftarrow$ st.ssk
03 **return** ssk

Helper validate-kp(st, kp, id, pHash)

04 **parse** $(\mathsf{id}', \mathsf{pk}, \mathsf{svk}, \mathsf{pHash}', \sigma) \leftarrow \mathsf{kp}$
05 **req** $\mathsf{id} = \mathsf{id}' \ \wedge \ \mathsf{pHash} = \mathsf{pHash}'$
06 **if** $\mathsf{svk} \notin \mathsf{st.cert\text{-}SPks[id]}$ **then**
07 **send** (verify-cert, id, svk) **to** $\mathcal{F}_{\mathsf{AS}}$ **and receive** b
08 **req** b
09 $\mathsf{st.cert\text{-}SPks[id]} \leftarrow_{\cup} \{\mathsf{svk}\}$
10 **req** $\mathsf{Vrfy}(\mathsf{svk}, \sigma, (\mathsf{id}', \mathsf{pk}, \mathsf{svk}, \mathsf{pHash}', \sigma))$
11 **return** st

Helper Gen-kp(id, svk, ssk)
12 $(\mathsf{pk}, \mathsf{sk}) \leftarrow_{\$} \mathsf{Pke.Gen}()$
13 $\sigma \leftarrow \mathsf{Sign}(\mathsf{ssk}, (\mathsf{id}, \mathsf{pk}, \mathsf{svk}, \mathsf{pHash}))$
14 $\mathsf{kp} \leftarrow (\mathsf{id}, \mathsf{pk}, \mathsf{svk}, \mathsf{pHash}, \sigma)$
15 return $(\mathsf{kp}, \mathsf{sk})$

Figure 18: Helper functions for the interaction with KS and AS.

Algorithm CGKA.Create(svk)
00 **require** st $= \bot \wedge u = u_{\mathbf{creator}}$
01 st.T $\leftarrow$ init-tree
02 st.groupID $\leftarrow_{\$} \{0,1\}^{\kappa}$
03 st.appSec $\leftarrow_{\$} \{0,1\}^{\kappa}$
04 st.ep $\leftarrow 0$
05 st.interim-tranHash $\leftarrow \varepsilon$
06 **try** st.ssk $\leftarrow$ fetch-ssk-if-nec(st, svk)
07 (kp, sk) $\leftarrow$ Gen-kp(svk, ssk)
08 id(leaves(T)) $\leftarrow$ H(pk)
09 assign-kp(T, v_{id}, kp)
10 v_{id}.sk $\leftarrow$ sk

Algorithm CGKA.Com(st, PMSG)
11 st′ $\leftarrow$ init-epoch(st)
12 T′ $\leftarrow$ st′.T
13 **try** (st′, upd, rem, add) $\leftarrow$ apply-props(st, PMSG)
14 **require** (st.id, ·, ·) $\notin$ rem $\wedge$ (st.id, ·, ·) $\notin$ upd
15 **try** (st′, comSec, UpdPath, pathSec)
$\leftarrow$ rekey-path(st′, st.id, v_{id}.svk, 0)
16 propIds $\leftarrow$ ()
17 **for** pmsg $\in$ PMSG**:**
18 propIds $\leftarrow_{\cup}$ H(pmsg)
19 $C \leftarrow$ (propIds, UpdPath)
20 $\sigma \leftarrow$ sign-commit(st, C)
21 st′ $\leftarrow$ set-conf-trans-hash(st, st′, st.id, C, σ)
22 (st′, confKey, joinerSec) $\leftarrow$ derive-keys(st, st′, comSec)
23 confTag $\leftarrow$ compute-confTag(st′, confKey)
24 **if** rem $\neq$ ()
25 memTag $\leftarrow$ Mac.Tag(st.memKey, C)
26 **else** memTag $\leftarrow \bot$
27 cmsg $\leftarrow$ frame-comm(st, confTag, σ, memTag)
28 **if** add $\neq$ ()
29 (st′, wmsg) $\leftarrow$ comp-wmsg(st′,
, add, joinerSec, pathSec, confTag)
30 **else** wmsg $\leftarrow \bot$
31 st′ $\leftarrow$ set-int-transHash(st′, confTag)
32 st.pendComm(cmsg) $\leftarrow$ (st′, PMSG, upd, rem, add)
33 **return** (cmsg, wmsg)

Algorithm CGKA.Key
34 $(k$, st.appSec$) \leftarrow$ (st.appSec, $\bot$)
35 **return** k

Algorithm CGKA.Proc(cmsg, PMSG)
36 (id-snd, C, confTag, σ, memTag) $\leftarrow$ unframe-comm(st, cmsg)
37 **if** id-com $=$ st.id
38 (st′, PMSG′, upd, rem, add) $\leftarrow$ st.pendComm(cmsg)
39 **require** PMSG′ $=$ PMSG
40 st $\leftarrow$ st′
41 **return**
42 **parse** (propIds, UpdPath) $\leftarrow C$
43 **require** propIds(i) $=$ H(PMSG(i)) **for all** i
44 st′ $\leftarrow$ init-epoch(st)
45 **try** (st′, upd, rem, add) $\leftarrow$ apply-props(st, PMSG)
46 **require** id-com $\notin$ rem $\wedge$ id-com $\notin$ upd
47 **if** st.id $\in$ rem
48 **require** Mac.Ver(st.memKey, memTag)
49 st $\leftarrow \bot$
50 **else**
51 (st′, comSec) $\leftarrow$ apply-rekey(st, st′, id-com, UpdPath, 0, `False`)
52 st′ $\leftarrow$ set-conf-trans-hash(st, st′, id-com, C, σ)
53 (st′, confKey, joinerSec) $\leftarrow$ derive-keys(st, st′, comSec)
54 **require** verify-confTag(st′, confKey, confTag)
55 st′ $\leftarrow$ set-int-transHash(st′, confTag)
56 st $\leftarrow$ st′
57 **return** ((upd, rem, add), id-snd)

Algorithm CGKA.Join(wmsg)
58 **parse** (encGrpSec, GrpInfo) $\leftarrow$ wmsg
59 **parse** (GrpInfoTBS, σ) $\leftarrow$ GrpInfo
60 **parse** (st.groupID, st.ep, st.tHash, st.conf-TransHash,
st.interim-tranHash, st.confTag, id-com) $\leftarrow$ GrpInfoTBS
61 **require** Sig.Vrfy($v_{\mathsf{id\text{-}com}}$.svk, σ, GrpInfoTBS)
62 **try** verify-treeState(st)
63 **try** fetch-ssk-if-nec(st, $v_{\mathsf{st.id}}$.svk)
64 kbs $\leftarrow$ `get-sks`
65 (joinerSec, pathSec) $\leftarrow (\bot, \bot)$
66 **for** $e \in$ encGrpSec
67 **parse** $(h, c) \leftarrow e$
68 **for** (kp, sk) $\in$ kbs
69 **if** $h =$ H(kp)
70 v.sk $\leftarrow$ sk
71 **require** $v_{\mathsf{st.id}}$.kp $=$ kp
72 **parse** (joinerSec, pathSec) $\leftarrow$ Pke.Dec(sk, c)
73 **if** $s_j =$ pathSec $\neq \bot$
74 $a \leftarrow$ st.T.ind-lca($v_{\mathsf{st.id}}, v_{\mathsf{id\text{-}com}}$)
75 $(v_1, \ldots, v_\ell) \leftarrow$ fil-path(v)
76 **for** $i \in \{\ell - a, \ldots, \ell - 1\}$
77 (pk, v_i.sk) $\leftarrow$ Pke.Gen(H$_2(s_i)$)
78 **require** pk $= v_i$.pk
79 $s_{i+1} \leftarrow$ H$_1(s_i)$
80 (st, confKey) $\leftarrow$ derive-epochKeys(st, joinerSec)
81 **require** verify-confTag(st, confKey, confTag)
82 **return** (roster(T), st.id)

Figure 19: Algorithms CGKA.Create, CGKA.Com, CGKA.Proc, CGKA.Join, CGKA.Key shared between MLS and MLS-Cutoff. For subroutines rekey-path, apply-props, comp-wmsg, see Figure 22, 23, 18, respectively.

Algorithm CGKA.Prop(op, ad) **(used in** MLS)

```
00 T ← st.T
01 if op = 'upd':
02    try st.ssk ← fetch-ssk-if-nec(st, svk)
03
04    (kp, sk) ← Gen-kp(svk, ssk)
05    pmsg ← frame-prop(st, ('upd', kp))
06    st.pendUpd(pmsg) ← (ssk, sk)
07 if op = ('add'):
08    parse u ← ad
09    require u ∉ roster(T)
10    send (fetch-kp, u) to F_KS and get kp_a
11    try validate-kp(kp_a)
12    pmsg ← frame-prop(st, ('add', kp_a))
13 if op = 'rem':
14    parse u ← ad
15    require u ∈ roster(T)
16    pmsg ← frame-prop(st, ('rem', id-rem))
17 return pmsg
```

Algorithm CGKA.Prop(op, ad) **(used in** MLS-Cutoff**)**

```
18 T ← st.T
19 if op = 'upd':
20    try st.ssk ← fetch-ssk-if-nec(st, svk)
21    st' ← init-epoch(st)
22    (id-own, svk) ← (st'.id-own, v_id-own.svk)
23    try (st', ·, UpdPath, pathSec) ← rekey-path(st', id-own, svk, st.i_cut)
24    pmsg ← frame-prop(st, ('upd', UpdPath))
25    st.pendUpd(pmsg) ← (ssk, v'_id-own.sk, pathSec)
26 if op = ('add'):
27    parse u ← ad
28    require u ∉ roster(T)
29    send (fetch-kp, u) to F_KS and get kp_a
30    try validate-kp(kp_a)
31    pmsg ← frame-prop(st, ('add', kp_a))
32 if op = 'rem':
33    parse u ← ad
34    require u ∈ roster(T)
35    pmsg ← frame-prop(st, ('rem', id-rem))
36 return pmsg
```

Figure 20: Generating proposals in MLS (left) and protocol variant MLS-Cutoff (right). Differences between the protocols are marked in yellow (•) and cyan (•). In line 25 $v'_{\text{id-own}}$ is to be understood as the issuing user's leaf with respect to st′.

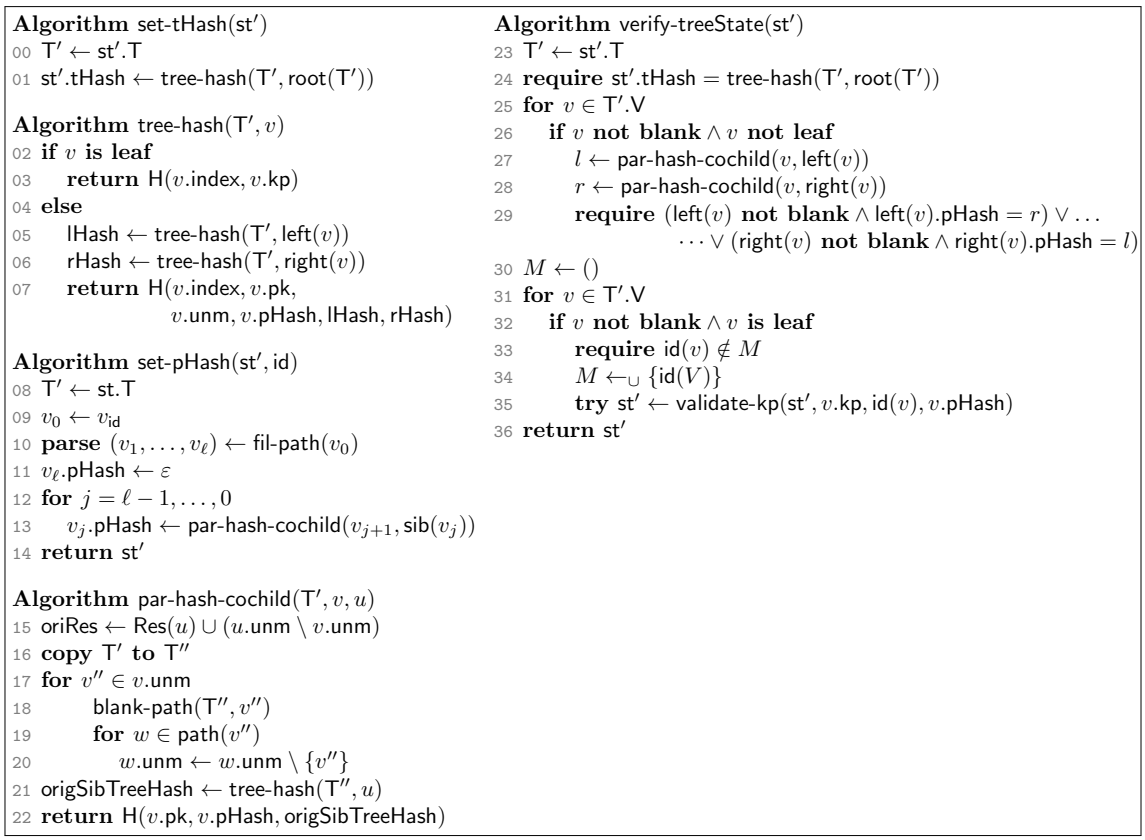


Figure 21: Helper functions set-tHash, set-pHash, and verify-treeState, used authenticate the ratchet tree's state.

Algorithm rekey-path($\mathsf{st}', \mathsf{id}, \mathsf{svk}, i$)

$\mathsf{T}' \leftarrow \mathsf{st}'.\mathsf{T}$
$\mathsf{UpdPathNodes} \leftarrow ()$
$\mathsf{pathSec} \leftarrow ()$
$(v_1, \ldots, v_\ell) \leftarrow \mathsf{fil\text{-}path}(v_{\mathsf{id}})$
$\mathrm{s}_0 \leftarrow_{\$} \{0,1\}^\lambda$
for $j = 1, \ldots, \ell - i$**:**
  $\mathrm{s}_j \leftarrow \mathsf{H}_1(\mathrm{s}_{j-1})$
  if $j < \ell$**:**
    $(v_j.\mathsf{pk}, v_j.\mathsf{sk}_j) \leftarrow \mathsf{Gen}(\mathsf{H}_2(\mathrm{s}_j))$
  $C \leftarrow ()$
  $v_{\mathsf{sib}} \leftarrow \mathsf{sib}(v_{j-1})$
  for $w \in \mathsf{Res}(v_{\mathsf{sib}}) \cup v_{\mathsf{sib}}.\mathsf{unm}$**:**
    $C \leftarrow_\cup c_w := \mathsf{Pke.Enc}(w.\mathsf{pk}, \mathrm{s}_j)$
  $\mathsf{UpdPathNodes} \leftarrow_\cup (v_j.\mathsf{pk}, C)$
  $\mathsf{pathSec} \leftarrow_\cup (s_j)$
if $i = 0$
  $\mathsf{comSec} \leftarrow s_\ell$
else $\mathsf{comSec} = \bot$
$\mathsf{merge\text{-}leaves}(\mathsf{T}', v_{\mathsf{id}})$
$\mathsf{set\text{-}pHash}(\mathsf{T}', v_{\mathsf{id}})$
try $\mathsf{fetch\text{-}ssk\text{-}if\text{-}nec}(\mathsf{T}', \mathsf{svk})$
$(\mathsf{kp}, \mathsf{sk}) \leftarrow \mathsf{Gen\text{-}kp}(\mathsf{id}, \mathsf{svk}, \mathsf{ssk}, v.\mathsf{pHash}; s_0)$
require $v_{\mathsf{H(kp)}} = \bot$
$\mathsf{id}(v_{\mathsf{id}}) \leftarrow \mathsf{H}(\mathsf{kp})$
$\mathsf{assign\text{-}kp}(\mathsf{T}', \mathsf{H}(\mathsf{kp}), \mathsf{kp})$
$v_{\mathsf{id}}.\mathsf{sk} \leftarrow \mathsf{sk}$
$\mathsf{set\text{-}tHash}(\mathsf{T}')$
$\mathsf{UpdPath} \leftarrow (\mathsf{UpdPathNodes}, \mathsf{kp})$
return $(\mathsf{T}', \mathsf{comSec}, \mathsf{UpdPath}, \mathsf{pathSec})$

Algorithm comp-wmsg($\mathsf{st}', \mathrm{add}, \mathsf{joinerSec}, \mathsf{pathSec}, \mathsf{confTag}$)

$\mathsf{tbs} \leftarrow (\mathsf{st}'.\mathsf{groupID}, \mathsf{st}'.\mathsf{ep}, \mathsf{st}'.\mathsf{tHash},$
  $\mathsf{st.conf\text{-}TransHash}, \mathsf{st}'.\mathsf{interim\text{-}tranHash},$
  $\mathsf{st}'.\mathsf{T}.\mathsf{public}, \mathsf{st}'.\mathsf{confTag}, \mathsf{st}'.\mathsf{id})$
$\sigma \leftarrow \mathsf{Sign}(\mathsf{st}'.\mathsf{ssk}, \mathsf{tbs})$
$\mathsf{GrpInfo} \leftarrow (\mathsf{tbs}, \sigma))$
$\mathsf{encGrpSec} \leftarrow ()$
for $(\cdot, \mathsf{id\text{-}add}, \mathsf{svk}) \in \mathrm{add}$
  $a \leftarrow \mathsf{ind\text{-}lca}(v_{\mathsf{st}'.\mathsf{id}}, v_{\mathsf{id\text{-}add}})$
  $c \leftarrow \mathsf{Pke.Enc}(v_{\mathsf{id\text{-}add}}.\mathsf{pk}, (\mathsf{joinerSec}, \mathsf{pathSec}(\ell - a)))$
  $\mathsf{encGrpSec} \leftarrow_\cup (\mathsf{H}(v_{\mathsf{id\text{-}add}}.\mathsf{kp}), c)$
return $(\mathsf{st}', \mathsf{wmsg})$

Algorithm apply-rekey($\mathsf{st}, \mathsf{st}', \mathsf{id\text{-}snd}, \mathsf{UpdPath}, i, \mathtt{copySt}$)

$\mathsf{T}' \leftarrow \mathsf{st}'.\mathsf{T}$
parse $(\mathsf{UpdPathNodes}, \mathsf{kp}) \leftarrow \mathsf{UpdPath}$
$(v_1, \ldots, v_\ell) \leftarrow \mathsf{fil\text{-}path}(\mathsf{T}', v_{\mathsf{st.id}})$
$(v'_1, \ldots, v'_{\ell'}) \leftarrow \mathsf{fil\text{-}path}(\mathsf{T}', v_{\mathsf{id\text{-}snd}})$
parse $((\mathsf{pk}_1, C_1), \ldots, (\mathsf{pk}_m, C_m)) \leftarrow \mathsf{UpdPathNodes}$
for $i \in \{1, \ldots, m\}$
  $v'_i.\mathsf{pk} \leftarrow \mathsf{pk}_i$
$a \leftarrow \mathsf{ind\text{-}lca}(v_{\mathsf{st.id}}, v_{\mathsf{id\text{-}snd}})$
if $i > a$**:**
  $\mathsf{comSec} \leftarrow \bot$
else
  parse $(\mathsf{c}_w)_w \leftarrow C_{\ell - a}$
  if $v'_0 \in v'_{\ell' - a - 1}.\mathsf{unm}$**:**
    $w \leftarrow v'_0$
  else:
    $m \leftarrow \max(m' \in \{0, \ldots, \ell' - a\} \mid v'_{m'} \text{ not blank})$
    $w \leftarrow v'_m$
  $s_{\ell - a} \leftarrow \mathsf{Pke.Dec}(\mathsf{sk}_w, \mathsf{c}_w)$
  for $j = \ell' - a + 1, \ldots, \ell' - i$**:**
    $\mathrm{s}_j \leftarrow \mathsf{H}_1(\mathrm{s}_{j-1})$
    if $j < \ell$
      $(\mathsf{pk}_j, \mathsf{sk}_j) \leftarrow \mathsf{Gen}(\mathsf{H}_2(\mathrm{s}_j))$
      $v'_j.\mathsf{sk} \leftarrow \mathsf{sk}_j$
      require $v'_j.\mathsf{pk} = \mathsf{pk}_j$
if $i = 0$
  $\mathsf{comSec} \leftarrow s_\ell$
else $\mathsf{comSec} = \bot$
$\mathsf{merge\text{-}leaves}(\mathsf{T}', v_{\mathsf{id}})$
$\mathsf{st}' \leftarrow \mathsf{set\text{-}pHash}(\mathsf{T}', v_{\mathsf{id}})$
if $\mathtt{copySt} = \mathtt{False}$
  try $\mathsf{validate\text{-}kp}(\mathsf{st}'', \mathsf{kp}, \mathsf{id\text{-}snd}, \mathsf{st}''.\mathsf{pHash})$
else
  $\mathsf{st}'' \leftarrow \mathsf{init\text{-}epoch}(\mathsf{st})$
  $\mathsf{T}'' \leftarrow \mathsf{st}''.\mathsf{T}$
  $(v''_1, \ldots, v''_{\ell'}) \leftarrow \mathsf{fil\text{-}path}(\mathsf{T}'', v_{\mathsf{id\text{-}snd}})$
  for $i \in \{1, \ldots, m\}$
    $v''_i.\mathsf{pk} \leftarrow \mathsf{pk}_i$
  $\mathsf{merge\text{-}leaves}(\mathsf{T}'', v_{\mathsf{id}})$
  $\mathsf{st}'' \leftarrow \mathsf{set\text{-}pHash}(\mathsf{T}'', v_{\mathsf{id}})$
  try $\mathsf{validate\text{-}kp}(\mathsf{st}'', \mathsf{kp}, \mathsf{id\text{-}snd}, \mathsf{st}''.\mathsf{pHash})$
require $v_{\mathsf{H(kp)}} = \bot$
$\mathsf{id}(v_{\mathsf{id\text{-}snd}}) \leftarrow \mathsf{H}(\mathsf{kp})$
$\mathsf{assign\text{-}kp}(\mathsf{T}', \mathsf{H}(\mathsf{kp}), \mathsf{kp})$
$\mathsf{set\text{-}tHash}(\mathsf{T}')$
return $(\mathsf{st}', \mathsf{comSec})$

Figure 22: Helper functions rekey-path and apply-rekey used to rekey users' paths.

Algorithm apply-props(st, st′, PMSG) **(used in** MLS**)**

00 $(\text{upd}, \text{rem}, \text{add}) \leftarrow ((), (), ())$
01 **fetch** $\mathsf{T}' \leftarrow \mathsf{st}'.\mathsf{T}$
02 **for** pmsg $\in$ PMSG**:**
03 $((\texttt{`type'}, \mathsf{ad}), \text{id-snd}) \leftarrow$ unframe-prop(st, pmsg)
04 **if** `type' = `upd'**:**
05 **require** $(\text{id-snd}, \cdot) \notin \text{upd} \wedge \text{rem} = () \wedge \text{add} = ()$
06 **try** validate-kp(st′, ad, id-snd, ε)
07 assign-kp($\mathsf{T}', v_{\text{id-snd}}$, ad)
08 **if** st′.id = id-snd
09 **parse** (ssk, sk) $\leftarrow$ pendUpd(pmsg)
10 $v_{\text{id-snd}}.\mathsf{sk} \leftarrow \mathsf{sk}$
11 st′.ssk $\leftarrow$ ssk
12 blank-path($\mathsf{T}', v_{\text{id-snd}}, 1$)
13 svk $\leftarrow$ leaf(T', id-snd).svk
14 **require** $v_{\mathsf{H}(\mathsf{kp})} = \bot$
15 $\mathsf{id}(v_{\text{id-snd}}) \leftarrow \mathsf{H}(\mathsf{kp})$
16 $\text{upd} \leftarrow_\cup (\text{id-snd}, \texttt{`upd'}, \mathsf{svk})$
17 **else if** `type' = `rem'**:**
18 **require** id-rem := ad $\neq$ id-snd $\wedge\, v_{\text{id-rem}} \neq \bot$
19 **require** $(\mathsf{ad}, \cdot) \notin \text{upd} \wedge \text{add} = ()$
20 blank-leaf($\mathsf{T}', v_{\text{id-rem}}$)
21 blank-path($\mathsf{T}', v_{\text{id-rem}}, 1$)
22 $\mathsf{T}' \leftarrow$ truncate(T')
23 $\text{rem} \leftarrow_\cup (\text{id-snd}, \texttt{`rem'}, \text{id-rem})$
24 **else if** `type' = `add'**:**
25 **try** st′ $\leftarrow$ validate-kp(st′, ad, $u_{\text{new}}, \varepsilon$)
26 id-add $\leftarrow \mathsf{H}(\mathsf{ad})$
27 add-leaf(T', id-add)
28 set-as-unmerged(T', id-add)
29 $\text{add} \leftarrow_\cup (\text{id-snd}, \texttt{`add'}, v_{\text{id-add}}.\mathsf{svk})$
30 **else return** $\bot$
31 **return** $(\mathsf{st}', \text{upd}, \text{rem}, \text{add})$

Algorithm apply-props(st, st′, PMSG) **(used in** MLS-Cutoff**)**

32 $(\text{upd}, \text{rem}, \text{add}) \leftarrow ((), (), ())$
33 **fetch** $\mathsf{T}' \leftarrow \mathsf{st}'.\mathsf{T}$
34 **for** pmsg $\in$ PMSG**:**
35 $((\texttt{`type'}, \mathsf{ad}), \text{id-snd}) \leftarrow$ unframe-prop(st, pmsg)
36 **if** `type' = `upd'**:**
37 **require** $(\text{id-snd}, \cdot) \notin \text{upd} \wedge \text{rem} = () \wedge \text{add} = ()$
38 **if** st′.id $\neq$ id-snd
39 $(\mathsf{st}', \cdot) \leftarrow$ apply-rekey(st, st′, id-snd, ad, $\mathsf{st}.i_{\text{cut}}$, `True')
40 **if** st′.id = id-snd
41 **parse** $((\mathsf{pk}_1, \cdot), \ldots, (\mathsf{pk}_{\ell - i_{\text{cut}}}, \cdot), \mathsf{kp}) \leftarrow \mathsf{ad}$
42 assign-kp($\mathsf{T}', v_{\text{id-snd}}$, kp)
43 **parse** (ssk, sk, pathSec) $\leftarrow$ pendUpd(pmsg)
44 $v_{\text{id-snd}}.\mathsf{sk} \leftarrow \mathsf{sk}$
45 st′.ssk $\leftarrow$ ssk
46 **parse** $(v_1, \ldots, v_\ell) \leftarrow$ path(id-snd)
47 **for** $j = 1, \ldots, \ell - \mathsf{st}.i_{\text{cut}}$
48 $(v_j.\mathsf{pk}, v_j.\mathsf{sk}) \leftarrow (pk_j, \mathsf{pathSec}(j))$
49 blank-path($\mathsf{T}', v_{\text{id-snd}}, \ell - \mathsf{st}.i_{\text{cut}} + 1$)
50 **parse** $(v_1, \ldots, v_\ell) \leftarrow \mathsf{path}(v_{\text{id-snd}})$
51 **for** $j = 1, \ldots, \ell - \mathsf{st}.i_{\text{cut}}$
52 **if** $v_j \in V_{\text{col}}$
53 blank(v_j)
54 $V_{\text{col}} \leftarrow_\cup \{v_j\}$
55 svk $\leftarrow$ leaf(T', id-snd).svk
56 **require** $v_{\mathsf{H}(\mathsf{kp})} = \bot$
57 $\mathsf{id}(v_{\text{id-snd}}) \leftarrow \mathsf{H}(\mathsf{kp})$
58 $\text{upd} \leftarrow_\cup (\text{id-snd}, \texttt{`upd'}, \mathsf{svk})$
59 **else if** `type' = `rem'**:**
60 **require** id-rem := ad $\neq$ id-snd $\wedge\, v_{\text{id-rem}} \neq \bot$
61 **require** $(\mathsf{ad}, \cdot) \notin \text{upd} \wedge \text{add} = ()$
62 blank-leaf($\mathsf{T}', v_{\text{id-rem}}$)
63 blank-path($\mathsf{T}', v_{\text{id-rem}}, 1$
64 $\mathsf{T}' \leftarrow$ truncate(T')
65 $\text{rem} \leftarrow_\cup (\text{id-snd}, \texttt{`rem'}, \text{id-rem})$
66 **else if** `type' = `add'**:**
67 **try** st′ $\leftarrow$ validate-kp(st′, ad, $u_{\text{new}}, \varepsilon$)
68 id-add $\leftarrow \mathsf{H}(\mathsf{ad})$
69 add-leaf(T', id-add)
70 set-as-unmerged(T', id-add)
71 $\text{add} \leftarrow_\cup (\text{id-snd}, \texttt{`add'}, v_{\text{id-add}}.\mathsf{svk})$
72 **else return** $\bot$
73 **return** $(\mathsf{st}', \text{upd}, \text{rem}, \text{add})$

Figure 23: Helper function apply-props for MLS (left) and MLS-Cutoff (right). Differences between the protocols are marked in yellow (•) and cyan (•).