SUMAC: an Efficient Administrated-CGKA Using Multicast Key Agreement

Nicolas Bon
DIENS, École normale supérieure,
CNRS, PSL University, Inria
Paris, France
Crypto Experts
Paris, France
nicolas.bon@ens.fr

Céline Chevalier
DIENS, École normale supérieure,
CNRS, PSL University, Inria
Paris, France
CRED, Paris-Panthéon-Assas University
Paris, France
celine.chevalier@ens.fr

Guirec Lebrun
DIENS, École normale supérieure,
CNRS, PSL University, Inria
Paris, France
ANSSI
Paris, France
guirec.lebrun@ens.fr

Ange Martinelli
ANSSI
Paris, France
ange.martinelli@ssi.gouv.fr

Abstract

Since the standardization of the Secure Group Messaging protocol Messaging Layer Security (MLS) [4], whose core subprotocol is a Continuous Group Key Agreement (CGKA) mechanism named TreeKEM, CGKAs have become the norm for group key exchange protocols. However, in order to alleviate the security issue originating from the fact that all users in a CGKA are able to carry out sensitive operations on the member group, an augmented protocol called Administrated-CGKA (A-CGKA) has been recently created [2].

An A-CGKA includes in the cryptographic protocol the management of the administration rights that restrict the set of privileged users, giving strong security guarantees for the group administration. The protocol designed in [2] is a plugin added to a regular (black-box) CGKA, which consequently add some complexity to the underlying CGKA and curtail its performances. Yet, leaving the fully decentralized paradigm of a CGKA offers the perspective of new protocol designs, potentially more efficient.

We propose in this paper an A-CGKA called SUMAC, which offers strongly enhanced communication and storage performances compared to other A-CGKAs and even to TreeKEM. Our protocol is based on a novel design that modularly combines a regular CGKA used by the administrators of the group and a Tree-structured Multicast Key Agreement (TMKA) [9] – which is a centralized group key exchange mechanism administrated by a single group manager – between each administrator and all the standard users. That TMKA gives SUMAC an asymptotic communication cost logarithmic in the number of users, similarly to a CGKA. However, the concrete performances of our protocol are much better than the latter, especially in the post-quantum framework, due to the intensive use of secret-key cryptography that offers a lighter bandwidth than the public-key encryption schemes from a CGKA.

In practice, SUMAC improves the communication cost of TreeKEM by a factor 1.4 to 2.4 for admin operations and a factor 2 to 38 for user operations. Similarly, its storage cost divides that of TreeKEM by a factor 1.3 to 23 for an administrator and 3.9 to 1,070 for a standard user.

Our analysis of SUMAC is provided along with a ready-to-use open-source rust implementation that confirms the feasibility and the performances of our protocol.

Keywords

SGM, MLS, TreeKEM, CGKA, Administrated-CGKA, MKA

1 Introduction

1.1 Motivations

In July 2023, the IETF releases RFC 9420 [4] that standardizes a Secure Group Messaging (SGM) protocol called Messaging Layer Security (MLS). This protocol has been designed to ensure the security properties of point-to-point Secure Messaging (SM), while being more efficient – particularly communication-wise – than the *ad-hoc* protocols that were historically used by Secure Messaging applications, such as the Pairwise or the Sender Keys protocols. To do so, the core sub-protocol of MLS, a group key-agreement mechanism called TreeKEM [8] – that was abstracted by [1] as a Continuous Group Key Agreement (CGKA) – relies on a binary Ratchet Tree (RT) that gives the protocol an average[1] communication cost that is logarithmic in the number n of users ($O(\log(n))$), even when the handshake messages are broadcast to the entire user group.

1.1.1 The Administration of a CGKA. In MLS – and more generally, in any CGKA –, all users are considered equal and are therefore all entitled to carry out group operations (adding a new user to the group, removing a user from that group and updating the state of a user). However, in real-life group conversations, one may not want all group members to be able to change the group membership or

[1]In practice, the communication cost of a TreeKEM handshake depends on the structure of the Ratchet Tree. When this one is destructured, which happens regularly in the group history, due to security considerations, the communication cost increases. The logarithmic cost evoked for TreeKEM is therefore a *fair-weather* lower-bound.

to control the users' refreshment process. This is particularly true in large groups – which are precisely the target of MLS – where it becomes difficult to trust all users. Consequently, in practice, a subset of the member group (whose members are called *administrators*) is granted the administration rights for this group, whereas the other users (named in this paper *standard users*[2]) are restricted to communicating and possibly proposing changes to the group. Surprisingly, the administration of a user group is not considered in the MLS standard, which states in [6] that this matter is left to the application level.

The issue with that paradigm is that the partitioning between administrators and standard users offers no security guarantee from a cryptographic point of view, since it is excluded from the cryptographic protocol. The recent work of [2] includes the administration process into the CGKA in order to remedy the situation, creating a enhanced type of CGKA named *Administrated-CGKA* (A-CGKA). In their protocol, administrators belong to a subgroup of the user group and sign all messages exchanged within the CGKA by an individual or a shared admin signature key, so that every group member is able to check that the action demanded during a commit comes from an administrator.

[15] extends this work in the particular context of a single group manager, by granting some of the administration rights to the standard users. This protocol is called a CGKA with Flexible Authorizations (CGKA-FA, cf. Appendix A), and appears to be a mix of an A-CGKA and a Multicast Key Agreement (cf. below).

These studies achieve their purpose of including the group administration to the cryptographic protocol, but this security enhancement comes at some computational and communication cost. This is due to the fact that this process is carried out by a subprotocol, seen as a plugin that can be applied to any CGKA in order to transform it into an A-CGKA. But no advantage is taken from this new paradigm in order to improve the performances of that protocol, even if administrating a group key agreement protocol may permit to use a much more efficient architecture. This is the case notably with the historical protocol of Multicast Key Agreement (MKA).

1.1.2 Multicast Key Agreement: an Efficient Group Key Agreement Protocol. Multicast Key Agreement, also known as Broadcast Encryption [11], is a group key agreement protocol where, contrary to a CGKA, an omniscient user named *group manager* (*gm*) oversees the key agreement process by generating and distributing most of – if not all – the secrets necessary for all users to agree on a common key. This centralized paradigm is the natural design in many use cases (such as a videoconferencing session set up by a single organizer), which is why it has been studied since the 90's. The seminal concurrent works of [12] and [17] propose to use a key graph in order to enhance the distribution process and thus decrease the communication cost of the protocol. This concept, called *Logical Key Hierarchy*, consists in using intermediary keys, common to a subset of users; the information necessary to update the group key must therefore only be encrypted under a reduced number of these intermediary keys instead of the keys of *all* users. The most common key graph is a rooted tree; [9] uses such a tree structure when it refreshes the MKA construction with modern concepts from SGM protocols, especially:

- the use of black-box primitives, allowing crypto-agility in a time where the looming threat of the quantum computer forces public-key cryptographic schemes to evolve quickly;
- strong and well-formalized security properties for group key agreements, in particular Forward Secrecy (FS) and Post-Compromise Security (PCS).

The Tree-based MKA (TMKA) of [9], which is detailed in Section 4.1, not only yields a communication cost logarithmic in the number of users ($O(\log(n))$), similarly to what is achieved with tree-based CGKAs like TreeKEM. It also offers two great advantages that CGKAs are unable to provide:

- While the communication cost of TreeKEM-like protocols is often higher than the lower bound of $O(\log(n))$, because the group history and the decentralized setting do not permit to have a well-structured Ratchet Tree, a centralized TMKA can have a tree structure always close to the optimal one thanks to the omniscient group manager.
- More importantly, a TMKA relies almost exclusively on a secret-key cryptosystem instead of a public-key one[3], which greatly decreases the bandwidth of the key agreement process (besides limiting the exposure of the protocol to the quantum threat, since secret-key cryptography is only
  marginally affected by it).

Despite all its advantages, MKA is limited to use cases where a single organizer has all privileges. This framework does not correspond to that of Secure Group Messaging protocols – whether they include regular CGKAs or administrated-CGKAs – that make no assumption on the number of administrators.

1.2 Our Contributions

We propose in this paper SUMAC (**S**tandard **U**ser **M**ulticast **A**dministrated-**C**GKA), a novel Administrated-CGKA protocol – as defined by [2] –, which greatly enhances the communication and storage costs of existing A-CGKAs and even of TreeKEM-like CGKAs. To do so, SUMAC uses both the efficient but single-administrated TMKA of [9] (cf. Section 3.1) and a regular CGKA. All these protocols and their underlying cryptosystems are used as black-boxes, which makes SUMAC a versatile protocol independent of specific cryptographic assumptions.

The high level design of SUMAC is the following (cf. Figure 1):

- As in a standard A-CGKA, the member group $\mathcal{G}$ is divided into two disjoint subgroups: the administrator group ($\mathcal{G}_a$) and the one of the standard users ($\mathcal{G}_u$).
- Each administrator $a_i \in \mathcal{G}_a$ is a group manager of a TMKA instance – associated with a user tree $\mathcal{T}_{u_i}$ –, which includes all the standard users $u_j \in \mathcal{G}_u$ as that tree's leaves. Every administrator thus has an omniscient view of a single TMKA tree connected to all the standard users, whereas every standard user only has a partial view (its direct path) of as many TMKA trees as there are administrators in $\mathcal{G}_a$.

[2]By misuse of language, we sometimes call these standard users "users". Standard users and administrators together constitute the "group members".

[3]The use of public-key cryptography in a TMKA is indeed limited to sending, in unicast, new leaf secrets to new or updating users.

Each TMKA tree $\mathcal{T}_{u_i}$ yields a key I_{u_i} known by all standard users and the administrator a_i that manages it.

- The administrators are linked together through a regular CGKA, that outputs an admin key I_a known by all administrators but none of the standard users.
- Additionally, a group key I_g, common to all group members, is derived from the CGKA or the committer's TMKA, depending on the group operation carried out.

This architecture thus replaces a CGKA by a TMKA when the hierarchy makes it possible (administrators over standard users) while maintaining a CGKA otherwise (between the administrators, which are all equal and consequently must stay in the decentralized paradigm). The use of separate TMKA instances for each administrator comes from the need to partition secret information between these administrators, without which no operation of update or remove of an administrator could safely occur.

As shown in Table 4 and Table 1, SUMAC yields communication and storage costs far lower than that of the A-CGKAs proposed by [2] and even than TreeKEM. The bandwidth of SUMAC for user operations is indeed 42 to 97% lower than TreeKEM's, whereas in the slightly less efficient admin operations, the gain varies between 27 and 60%. The enhancement in memory is even more pronounced: administrators gain 26 to 96% compared to TreeKEM, and standard users gain 74 to more than 99%.

1.3 Outline of the Paper

We firstly discuss in Section 3 the links between the CGKA, MKA and A-CGKA protocols, and we show that CGKA and MKA are particular cases of the A-CGKA, depending on the number of administrators in the member group.

Section 4 describes the way SUMAC works for each group operation. In addition, Appendix E describes in pseudocode the TMKA and SUMAC algorithms, whereas the related figures are recalled in large size, for readability purpose, in Appendix F.

We then analyze the correctness and security of our protocol in Section 5 and Appendix B.

Finally, we provide in Section 6 the performance results of SUMAC, with a focus on its communication cost, whereas the computations leading to these results and the detailed communication and storage costs, as well as a preliminary survey on the computational cost, are dispatched in Appendix D.

In parallel, a ready-to-use rust implementation of SUMAC is available at https://anonymous.4open.science/r/SUMAC-5F5A, forked from the open-source implementation of OpenMLS [16].

2 Preliminaries

2.1 Notations and Terminology

The output of a probabilistic algorithm is represented by $\leftarrow$ and the one of a deterministic algorithm is given by $:=$. Sampling from a space $\mathcal{S}$ with uniform distribution is represented by $x \leftarrow \$ \mathcal{S}$.

$\cdot||\cdot$ is used for the concatenation operation. $|\cdot|$ denotes the cardinality of a set of elements or the bit-length of a bit-string. [.] denotes optional elements that may be left empty in some cases.

In our key agreement protocol, whose evolution over time is caught with the notion of "epoch" (cf. below), we note for instance x_j^i the data x (which may represent a key, a secret...) associated with

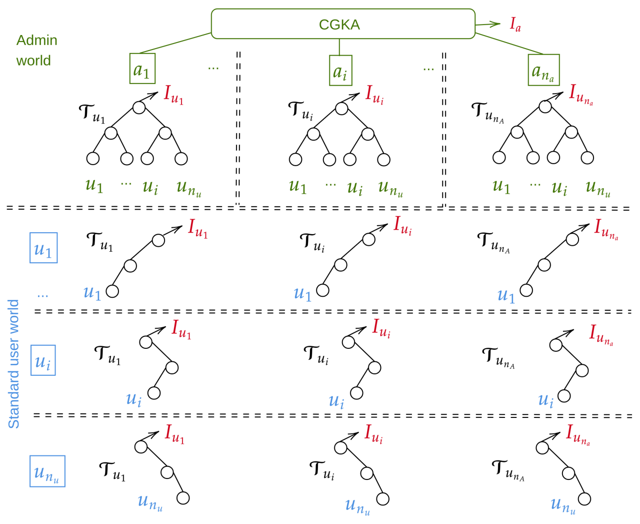

Figure 1: High-level design of SUMAC, composed of a CGKA between the administrators and an instance of a TMKA between each admin and all the standard users. Dashed lines indicate the partitioning between the admin world (green), in which every admin knows the admin key I_a and the entire TMKA tree it manages, and user world (blue), where every standard user knows (only) its direct path and the common key I_{u_i} in *all* the user trees.

user u_j in epoch t^i. All our algorithms, by convention, consider the passage from epoch t^i to t^{i+1}; consequently the updated data all receive the exponent $i+1$.

2.2 Ratchet Tree Basics

2.2.1 Ratchet Tree. A Ratchet Tree is a rooted tree used as key graph in protocols using the Logical Key Hierarchy paradigm. Therefore, this is a tree whose:

- leaves $\ell_{i\in[\![1,n]\!]}$ are associated with a set of n group members and that have states comprising notably signature and encryption keys;
- internal nodes v_i have states with "intermediate" encryption keys;
- root secret is known by all group members and yields the common key I that is used for group communication.

2.2.2 Parents, Children and Siblings. For any node in a Ratchet Tree, the node located immediately above is called its *parent* whereas the two nodes[4] underneath are its children. The nodes issued from the same parent are called siblings.

2.2.3 Direct Path and Copath. The (direct) path in a tree $\mathcal{T}$ of a leaf associated with some user u_j, noted $\mathcal{P}(u_j, \mathcal{T})$, is the set of ancestor nodes of that leaf, i.e. of the internal nodes linked to that leaf up to the tree root.

We define the *extended* (direct) path of a leaf ℓ_j (belonging to some user u_j) as the union of that leaf and its direct path:

[4]We only consider in this paper the most common case of a binary Ratchet Tree, even if [10] has shown that ternary trees could be more efficient for some parameter ranges of the CGKA.

$$\mathcal{P}_x(u_j, \mathcal{T}) := \{\ell_j\} \cup \mathcal{P}(u_j, \mathcal{T}) \quad (1)$$

The copath $\mathcal{CP}(u_j, \mathcal{T})$ of a user u_j in a tree $\mathcal{T}$ is the set of the siblings of all the nodes in its extended direct path – except for the root that has no sibling.

3 The World of Group Key Agreement Schemes

We detail in this part the main types of group key agreement protocols – the Multicast Key Agreement (MKA), the Continuous Group Key Agreement (CGKA) and the Administrated-CGKA (A-CGKA) –, that have been designed for different use cases and therefore have evolved separately. As a side contribution of this paper, we generalize the definitions of these protocols, unify their syntax and show that the A-CGKA can be seen as a generalization of the two other protocols, as discussed in Section 3.4.

General Changes in the Definitions. Group key exchange constructions have initially been designed so that every group operation is performed separately of the others (in what we call the *separate-operations* paradigm). Subsequently, the definition of a CGKA has been adapted to the Propose & Commit (P&C) paradigm, after this concept was adopted by the MLS IETF working group for the version 8 of its draft [3]. In a P&C protocol, group members are allowed to ask for group operations through *proposals*. The accepted changes are then validated altogether in an operation named *commit*, which enhances the protocol efficiency.

In order to be as generic as possible, we have modified the original definitions of the protocols below so that both the separate-operations and the propose & commit paradigms remain compatible. To do so, the commit algorithm takes as inputs a vector $\overrightarrow{P}$ of proposals sent by other users and a vector $\overrightarrow{op}$ of group operations initiated by the committer itself. $\overrightarrow{P}$ is empty in the separate-operations setting, whereas in the P&C framework, both vectors may be filled[5].

3.1 Multicast Key Agreement

In addition to these general changes, we have adapted the definition of an MKA from [9] regarding the matter of out-of-band communication. Indeed, an MKA needs unicast private communication between the group manager and a user one wants to add or update, in order to send it its new leaf secret generated by the group manager. [9] considers this leaf secret as an out-of-band value that must be sent separately through a secure point-to-point channel between the group manager and that user. In our definition below, we include this process within the MKA protocol, in order not to depend on a parallel Secure Messaging protocol[6].

Definition 3.1 (Multicast Key Agreement (MKA) (adapted from [9])). An MKA scheme is a group key agreement protocol consisting of the following algorithms:

- **Initialize**: group manager gm or user u_i creates its initial state, respectively γ and δ_i:
  $\boxed{\gamma \leftarrow \textbf{init}(gm) \quad / \quad \delta_i \leftarrow \textbf{init}(u_i)}$
- **Create Group**: group manager gm creates an initial member group $\mathcal{G} = \left(\text{gid}, \left(gm, (u_k)_{k=1}^{n}\right)\right)$ from a list of n users and a unique group ID gid, updating in accordance its initial state γ into γ', yielding the new group $\mathcal{G}$, an initial group key I and an associated commit message[7] C:
  $\boxed{(\gamma', \mathcal{G}, I, C) \leftarrow \textbf{create-group}(\gamma, (u_k)_{k=1}^{n})}$
- **Propose**: user u_i, with state δ_i, proposes a group operation $op \in O$ to apply on user u_j[8] within $\mathcal{G}$, generating a proposal message P and an updated state δ_i':
  $\boxed{(\delta_i', P) \leftarrow \textbf{propose}(\delta_i, \text{gid}, op(u_j))}$
- **Commit**: the group manager, with state γ, takes as input a *possibly empty* vector of proposals $\overrightarrow{P}$ sent by users, and a *possibly empty* vector of group operations $\overrightarrow{op}$ that it has generated itself. After updating its state accordingly, it generates a new group key I' and sends a commit message C to the existing and to the potential new users:
  $\boxed{(\gamma', I', C) \leftarrow \textbf{commit}(\gamma, \text{gid}, \overrightarrow{P}, \overrightarrow{op})}$
- **Process**: user u_i, with state δ_i, processes a commit message C it has received. After checking this message was released by the group manager, it updates its state accordingly and computes the new group key I' resulting from these changes. In case of failure, and notably if the commit message does not originate from the group manager, u_i aborts the ongoing process and returns a failure value $\perp$:
  $\boxed{(\delta_i', I')/\perp := \textbf{process}(\delta_i, \text{gid}, C)}$

The set O of group operations in an MKA is the following:

- **add** (u_j): new user u_j is added to the member group $\mathcal{G}$;
- **remove** (u_j): user u_j is removed from the member group $\mathcal{G}$;
- **update** (u_j): user u_j has its state updated;

3.2 Continuous Group Key Agreement

At a high level, a Continuous Group Key Agreement (CGKA) is a protocol between an arbitrary number n of equal users, which generates a group key I evolving over time – through the concept of *epoch* – in order to maintain the security properties of Forward Secrecy (FS) and Post-Compromise Security (PCS) (cf. below) despite the changes in the group membership and the deprecation of the users' keying material.

3.2.1 Definition. Originating from [1], the definition of a CGKA has been adapted to the propose & commit paradigm in subsequent works. Our definition relies on that of [2].

[5]Indeed, in many P&C protocols, every commit is associated with the automatic path update of the committer, which should be seen as a group operation from $\overrightarrow{op}$.

[6]In practice, we allow unicast exchanges between users, as usually considered in the architectural designs of CGKAs (such as MLS [6]). Therefore, when a PKI is used and users have public encryption keys associated with them, this exchange consists in sending the group manager a fresh public encryption key, and the latter then sends the user its new leaf secret, encrypted under this public key.

[7]This commit message is also called *control message* and noted T in several papers on group key agreements. It may be divided into a broadcast component C_{bdct}, one or several part(s) C_g sent to groups of users and messages C_{ind} sent in unicast to individual users.

[8]In practice, MKAs could limit users' proposals to specific operations and targets, for instance allowing users to only propose to update or remove themselves.

Definition 3.2 (Continuous Group Key Agreement (adapted from [2])). A CGKA is a group key agreement protocol consisting of the following algorithms:

- **Initialize**: user u_i creates its initial state δ_i: $\boxed{\delta_i \leftarrow \textbf{init}(u_i)}$
- **Create Group**: initial user u_1 creates an initial member group $\mathcal{G} = \left(\text{gid}, (u_k)_{k=1}^{n}\right)$ from a list of n users and a unique group ID gid, updating in accordance its initial state δ_1 into δ_1', yielding the new group $\mathcal{G}$, an initial group key I and an associated commit message C: $\boxed{(\delta_1', \mathcal{G}, I, C) \leftarrow \textbf{create-group}(\delta_1, (u_k)_{k=1}^{n})}$
- **Propose**: user u_i, with state δ_i, proposes a group operation $op \in O$ to apply on user u_j within $\mathcal{G}$, generating a proposal message P and an updated state δ_i': $\boxed{(\delta_i', P) \leftarrow \textbf{propose}(\delta_i, \text{gid}, op(u_j))}$
- **Commit**: user u_i, with state δ_i, takes as input a *possibly empty* vector of proposals $\overrightarrow{P}$ sent by other users, and a *possibly empty* vector of group operations $\overrightarrow{op}$ that it has generated itself. After updating its state accordingly, it generates a new group key I' and sends a commit message C to the existing and to the potential new users: $\boxed{(\delta_i', I', C) \leftarrow \textbf{commit}(\delta_i, \text{gid}, \overrightarrow{P}, \overrightarrow{op})}$
- **Process**: user u_i, with state δ_i, processes a commit message C it has received. After checking the validity of this message, it updates its state accordingly and computes the new group key I' resulting from these changes. In case of failure, u_i aborts the ongoing process and returns a failure value $\perp$: $\boxed{(\delta_i', I')/\perp := \textbf{process}(\delta_i, \text{gid}, C)}$

The set O of group operations in a CGKA is the following:

- **add** (u_j): new user u_j is added to the member group $\mathcal{G}$;
- **remove** (u_j): user u_j is removed from the member group $\mathcal{G}$;
- **update** (u_j): user u_j has its state updated;

3.3 Administrated-CGKA

An administrated-CGKA (A-CGKA) is a variant of CGKA, originally described by [2], in which group members, belonging to the member group $\mathcal{G}$, are split into two subgroups:

- a first one ($\mathcal{G}_a$) composed of n_a privileged users called administrators, that have the right to perform group operations;
- another one ($\mathcal{G}_u$) whose n_u members, named *standard users*, have no privilege and therefore are solely allowed to take part in the group conversation, to propose group operations and to initialize their own state.

We adapt beneath the definition of an A-CGKA from [2] in order to be more generic regarding the processes of adding and removing administrators. Indeed, in the original definition of [2], it is not possible to add administrators directly from the outside or to remove them directly from the member group. Instead, administrators are standard users that are granted administration rights; removing administrators revokes their rights but keeps them in the group conversation, as standard users. Yet, some protocols are more adapted to directly adding or removing administrators from the outside. This is precisely the case of SUMAC, for which the partitioning between the admin and the user worlds implies that a standard user upgraded to the admin status is in fact removed from the member group and then added as a new administrator (cf. Section 4).

Consequently, we consider two types of operations that change the membership of the admin group: in the **add-admin** and **remove-admin** operations, administrators are added to (removed from) the admin group directly from (to) the outside, without temporarily being standard users. In parallel, we define the migration operations of **upgrade-user** and **downgrade-admin**, which change the status of standard users into administrators, and conversely.

Definition 3.3 (Administrated-CGKA (adapted from [2])). An Administrated Continuous Group Key Agreement (A-CGKA) is a group key agreement protocol where group members are divided between n_a administrators a_i and n_u standard users u_i, and which consists of the following algorithms:

- **Initialize**: group member $m_i \in \{a_i, u_i\}$ creates its initial state γ_i (for an administrator) or δ_i (for a standard user): $\boxed{\gamma_i \leftarrow \textbf{init}(a_i) \quad / \quad \delta_i \leftarrow \textbf{init}(u_i)}$
- **Create Group**: initial administrator a_1 creates an initial member group $\mathcal{G}$ from a list $L_a = (a_k)_{k=1}^{n_a}$ of n_a administrators, a list $L_u = (u_k)_{k=1}^{n_u}$ of n_u users and a unique group ID gid, updating in accordance its initial state γ_1 into γ_1', yielding the new member group $\mathcal{G}$ (composed of the admin group $\mathcal{G}_a = (\text{gid}, L_a)$ and user group $\mathcal{G}_u = (\text{gid}, L_u)$), an initial group key I and an associated commit message C: $\boxed{(\gamma_1', (\mathcal{G}_a, \mathcal{G}_u), I, C) \leftarrow \textbf{create-group}(\gamma_1, L_a, L_u)}$
- **Propose**: group member m_i, with state γ_i or δ_i, proposes a group operation $op \in O$ to apply on group member m_j within $\mathcal{G}$, generating a proposal message P and an updated state γ_i' or δ_i': $\boxed{(\gamma_i'/\delta_i', P) \leftarrow \textbf{propose}(\gamma_i/\delta_i, \text{gid}, op(m_j))}$
- **Commit**: administrator a_i, with state γ_i, takes as input a *possibly empty* vector of proposals $\overrightarrow{P}$ sent by other group members and a *possibly empty* vector of group operations $\overrightarrow{op}$ that it has generated itself. It updates its state accordingly, generates a new group key I' and sends a commit message C to the existing and potentially new users: $\boxed{(\gamma_i', I', C) \leftarrow \textbf{commit}(\gamma_i, \text{gid}, \overrightarrow{P}, \overrightarrow{op})}$
- **Process**: group member m_i, with state γ_i or δ_i, processes a commit message C it has received. After checking that the message comes from an administrator, it updates its state accordingly and computes the new group key I' resulting from these changes. In case of failure, and notably if the commit message does not originate from an administrator, m_i aborts the ongoing process and returns a failure value $\perp$: $\boxed{(\gamma_i'/\delta_i', I')/\perp := \textbf{process}(\gamma_i/\delta_i, \text{gid}, C)}$

  The set O of group operations in an A-CGKA is the following:

  - **add-admin** (a_j): new admin a_j is added to the admin group $\mathcal{G}_a$ directly while joining the member group $\mathcal{G}$;
  - **remove-admin** (a_j): admin a_j is removed from the entire member group $\mathcal{G}$;

– **update-admin** (a_j): admin a_j has its state updated;
– **add-user** (u_j): new standard user u_j is added to the user group $\mathcal{G}_u$;
– **remove-user** (u_j): standard user u_j is removed from the member group $\mathcal{G}$;
– **update-user** (u_j): standard user u_j has its state updated;
– **upgrade-user** (u_j): standard user u_j becomes an administrator and leaves $\mathcal{G}_u$ to join $\mathcal{G}_a$;
– **downgrade-admin** (a_j): admin a_j becomes a standard user and leaves $\mathcal{G}_a$ to join $\mathcal{G}_u$.

3.4 A Unified View of Group Key Agreements

It emerges, from the above definitions with an harmonized syntax, that the Administrated-CGKA protocol is a generalization of both the regular CGKA and the MKA, depending on the number of administrators within the member group $\mathcal{G}$:

- A CGKA is an A-CGKA where all group members are administrators: $\mathcal{G}_a = \mathcal{G}$ and $\mathcal{G}_u = \emptyset$.
- An MKA is an A-CGKA with a single administrator gm: $\mathcal{G}_a = \{gm\}$.

Consequently, the functionalities and – above all – the security of all these protocols can be studied solely within the framework of the A-CGKA.

3.4.1 Epochs. The notion of epoch represents the evolution over time of a member group in a group key agreement scheme. An epoch t denotes the period between two consecutive group operations in the member group, and thus corresponds to a single group key[9]. Each commit therefore changes the epoch, in the view of a user processing it.

3.4.2 Security Considerations. A group key agreement must enforce the following security properties, that are informally stated below and that are captured in the security game from Section 5.1.1.

- **Privacy**: the group key is pseudorandom for any adversary that has access to the whole transcript of the handshake.
- **Forward Secrecy (FS)**: the corruption of one or several group members does not leak the secrets of past epochs. More specifically, forward secrecy for an epoch t^1 is achieved at an epoch $t^2 > t^1$ when any group member corruption occurring from epoch t^2 does not leak any secret up to t^1.
- **Post-Compromise Security (PCS)**: it represents the ability of the protocol to heal from group member corruption. It is achieved at an epoch t^2, after a single or multi-member corruption occurred at epoch $t^1 < t^2$, provided that the adversary remains passive between these two epochs.

4 SUMAC Protocol

We describe in this part our SUMAC protocol, which uses as core-components a TMKA and a CGKA. As the specific features of the tree structure of a TMKA are essential for SUMAC, we firstly detail that protocol, as well as the regeneration and derivation algorithms used as subroutines in SUMAC.

[9]In an A-CGKA, it is possible to have a more fine-grained time perspective by distinguishing admin and user epochs (between two consecutive changes in respectively the admin and user groups), in addition to the global epoch. This distinction is not made in our paper.

4.1 Tree-Based MKA

Definition 4.1 (Tree-Based Multicast Key Agreement (TMKA)). A tree-based multicast key agreement is an MKA (cf. Definition 3.1) that follows the logical key hierarchy from [17], in which the key graph is a rooted tree named Ratchet Tree, whose leaves are associated with the standard users from the group and where the group manager *gm* is located at the root.

Nota 1: We focus in this paper on TMKAs using exclusively secret-key encryption schemes. The augmented TMKA from [9, section 7], that uses public-key cryptography instead of the secret-key one in order to achieve a one-round PCS for the group manager, is not considered here as it looses the main advantage of being a centralized protocol.

Nota 2: The version of TMKA we use in SUMAC differs from the original one [9] on three points:

- For simplicity sake, we have not considered updatable secret-key encryption schemes that permit FS and PCS to occur more quickly than with regular encryption schemes. SUMAC remains nevertheless fully compatible with these primitives, as it uses TMKA almost as a black-box protocol.
- In the **add** and **update** operations, unicast conversation between a user and the group manager is needed. [9] ensures this exchange by considering separate secure channels between the group manager and each user. In particular, the leaf secret associated with u_j, that the group manager must transmit to that user, is sent through that channel and is therefore only considered in [9] as an out-of-band value. That channel removes the need to exchange the user's public key (step 1 of the **add** and **update** operations), which decreases – in appearance only – the protocol's communication cost[10]. Moreover, as the channel is authenticated, it is useless as well to use a Public Key Infrastructure (PKI) and to transmit public signature keys to authenticate the group members. However, in that case again, the communication cost of the authentication does not decrease and is solely reported to the separate Secure Messaging protocol that builds these point-to-point channels. In our paper, all these mechanisms are included in the TMKA, in order to be able to compare that protocol – and SUMAC afterwards – with others that are self-sufficient, such as the CGKAs.
- In the original TMKA, similarly as in TreeKEM, an internal node that has no descendants coming from one of its children is blanked (which means that all its state is emptied) since that node appears redundant with its filled child. However, the regeneration process used in SUMAC (cf. Section 4.2) needs that all internal nodes in the TMKA's Ratchet Tree remain filled. Therefore, the operations from SUMAC avoid that blanking process and instead update these nodes.

[10]In practice, the exchange of that public key is included in the Secure Messaging protocol managing the secure channels between the group manager and the users.

We describe hereunder and in Figure 2 the group operations of a TMKA: **add**, **remove** and **update**. The associated algorithms are given in Appendix E (Figure 14 p.26 and Figure 15 p.30).

4.1.1 Add User / Update User. User u_j is added to the member group $\mathcal{G}$ or updated with the following steps:

(1) u_j sends to the group manager a public encryption key pk_j^{i+1} that it has generated itself (potentially with a public signature key, according to the policy of credentials rotation).

(2) The group manager locally updates the Ratchet Tree by associating u_j to an empty leaf (in case of an **add-user**). In order to maintain the properties of forward secrecy and post-compromise security, the path $P(u_j)$ of u_j in the tree must be updated; to do so:

- The group manager *gm* randomly draws a new secret from the key space $\mathcal{K}$, called *leaf secret*, known only by *gm* and (ultimately) by u_j:
$$ls_j^{i+1} \overset{\$}{\leftarrow} \mathcal{K} \tag{2}$$
- This leaf secret is derived along u_j's path (of length h_u), up to the tree root, generating intermediary *path secrets*:
$$ps_1^{i+1} := \mathbf{derive}(ls_j^{i+1}, \text{“path”}) \tag{3}$$
$$\forall \ell \in [\![2, h_u]\!], ps_\ell^{i+1} := \mathbf{derive}(ps_{\ell-1}^{i+1}, \text{“path”}) \tag{4}$$
From these leaf and path secrets derive symmetric keys associated with these nodes and used for encryption (k_e) and potentially other functionalities (integrity...):
$$k_{e_j}^{i+1} := \mathbf{derive}(ls_j^{i+1}, \text{“enc”}) \tag{5}$$
$$\forall \ell \in [\![1, h_u]\!], k_{e_\ell}^{i+1} := \mathbf{derive}(ps_\ell^{i+1}, \text{“enc”}) \tag{6}$$
- The path secret at the root of the tree is derived to yield a new group key:
$$I^{i+1} := \mathbf{derive}(ps_{root}^{i+1}, \text{“key”}) \tag{7}$$

(3) The group manager unicasts to u_j its leaf secret, encrypted under pk_j^{i+1}. In practice, public-key cryptography is used according to the HPKE paradigm[11] [5]. This ciphertext corresponds to c_1 in Figure 2.

(4) The group manager broadcasts to the group a ciphertext message composed of all the path secrets ps_ℓ previously generated, each encrypted under the secret encryption key $k_{e_{cp\text{-}ch(\ell)}}^i$ of its child node in u_j's copath (ciphertexts c_2 and c_3 in Figure 2).

4.1.2 Remove User. User u_j is removed from the member group with the following steps:

(1) The group manager locally updates the Ratchet Tree by blanking the leaf associated with the removed user and by updating its direct path $P(u_j)$, as for an **add** or **update** operation.

(2) The secret-key ciphertexts are sent to the relevant recipient in the Ratchet Tree, as previously. No HPKE encryption is used in this case.

[11]Also known as KEM-DEM, the Hybrid Public-Key Encryption (HPKE) framework consists in drawing a random symmetric encryption key and encrypting it under the recipient's public key. The content of the message is then encrypting under the symmetric key, using a secret-key cryptosystem.

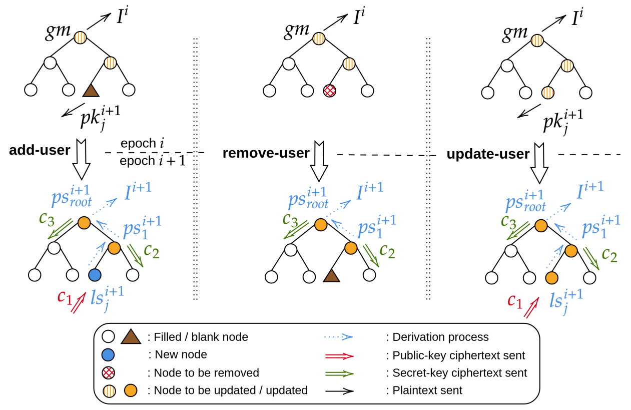

Figure 2: Overview of the group operations in a TMKA, as detailed in Section 3.1. All changes in the tree structure related to these operations (update of the direct path of the concerned user, encryption of the refreshed path secrets, generation of the new common secret) are similar to those of a TreeKEM-like CGKA [8] [4].

4.2 Derivation and Regeneration Subroutines

4.2.1 Derivation. Our protocol derives a secret with a Key Derivation Function (KDF) H taking as input that secret and a context-dependent label label and yielding a pseudorandom output mapping the key space $\mathcal{K}$: $s' := \mathbf{derive}(s, \text{label}) = H(s, \text{label})$.

This function H is modeled in the standard model as a pseudorandom function (PRF) using the secret s as its key and the label as its input.

4.2.2 Regeneration. A central process in SUMAC is what we call the regeneration of secrets. Post-compromise security implies to regularly refresh all the secrets known by any group member, with the help of fresh entropy – contrary to forward secrecy that can be more easily achieved with a simple derivation of these secrets. In SUMAC, the number of secrets to refresh is large, considering the admin CGKA and the n_a user TMKAs, and it is consequently useful to carry out this operation efficiently w.r.t. the communication cost.

To do so, our regeneration algorithm uses a regeneration set[12] $\mathcal{R}$, which comprises the entropy needed for the refreshment of the set $\mathcal{S}$ of secrets. It combines together these two sets in such a way that the set $\mathcal{S}'$ of regenerated secrets can be retrieved only by knowing both the original secrets and the regeneration values.

This secure combination of two secrets s_1 and s_2 is carried out by a key combiner F taking as inputs these two secrets and yielding a pseudorandom output mapping the key space $\mathcal{K}$. This function F is modeled in the standard model as a dual pseudorandom function (dPRF).

The algorithms of derivation and regeneration are detailed in Figure 16 (Appendix E, p.31).

[12]This regeneration set does not necessarily have the same size as the set of secrets to be regenerated; in fact a single regeneration value would suffice to refresh an arbitrary number of secrets. Nevertheless, in practice, the security model associated with SUMAC, in which any combination of administrators and standard users may be corrupt, implies to associate a different regeneration value with each input secret.

4.3 Description of SUMAC

SUMAC is an administrated-CGKA (A-CGKA), in the sense of Definition 3.3, that uses as sub-protocols a CGKA and n_a instances of a TMKA (with n_a the number of administrators).

- Administrators from the admin group $\mathcal{G}_a$ are linked together through the CGKA, which yields an admin key I_a known by the administrators only.
- In parallel, each administrator $a_{i \in [\![1,n_a]\!]}$ is appointed group manager of a TMKA instance, with a user tree $\mathcal{T}_{u_i}$ where all standard users from the user group $\mathcal{G}_u$ are located at the leaves. Each tree generates a user key I_{u_i} known by all standard users and by the single admin a_i managing it.
- The various group operations from SUMAC derive a group key I_g, known by every group members and that serves as a shared secret to establish the group conversation.

We describe beneath how SUMAC performs the group operations of an A-CGKA, considering separately admin and user operations that are implemented quite differently. These operations are described in pseudo-code in Figure 17 (Appendix E, p.33).

4.3.1 Admin Group Operations. Committer a_c (associated with a proposing admin a_p for an **add-admin**) performs an operation $op \in \{$**add-admin**, **remove-admin**, **update-admin**$\}$ on administrator a_j as follows:

(1) For all these three operations, a_c carries out in the administration CGKA the operation corresponding to op. Each administrator therefore receives from the committer a_c the information needed to compute the common CGKA secret, from which are derived both the new admin key I_a and the new group key I_g. The latter is subsequently broadcast by a_c to the standard users (a single message suffices, encrypted under the user key $I_{u_c}^{i+1}$ which is known both by a_c and all the standard users).

(2) In a **remove-admin**, the state of the removed administrator a_j – including notably its whole user tree $\mathcal{T}_{u_j}$ – is simply erased (cf. Figure 5).

(3) In an **update-admin**, a regeneration process is carried out on the whole user tree $\mathcal{T}_{u_c}$ of the committer. a_c therefore generates a regeneration set $\mathcal{R}$ derived from its own user tree ($\mathcal{R}$:= **derive**($\mathcal{T}_{u_c}^{i+1}$, "regen")), that it sends *in unicast* to the updated admin a_j. The latter uses that set to regenerate its whole user tree $\mathcal{T}_{u_j}$. In parallel, standard users regenerate their direct path in $\mathcal{T}_{u_j}$, using regeneration sets derived from their path in $\mathcal{T}_{u_c}^{i+1}$ (cf. Figure 3).

(4) In an **add-admin** (with at least two already existing administrators), the process is complicated by the fact that the user tree $\mathcal{T}_{u_j}$ of the new administrator is initially empty, which prevents any regeneration. Consequently, the commit for an **add-admin** operation must be preceded by a proposal, compulsorily by another administrator a_p. In that proposal, a_p includes a temporary user tree $\overline{\mathcal{T}_{u_j}^{i+1}}$ for a_j that is derived from its own tree: $\overline{\mathcal{T}_{u_j}^{i+1}}$:= **derive**($\mathcal{T}_{u_p}^{i}$, "regen"). A regeneration process may then be applied on that temporary tree, with the regeneration set $\mathcal{R}$ yielded by the committer a_c as for an **update-admin** operation. Figure 4 details these steps.

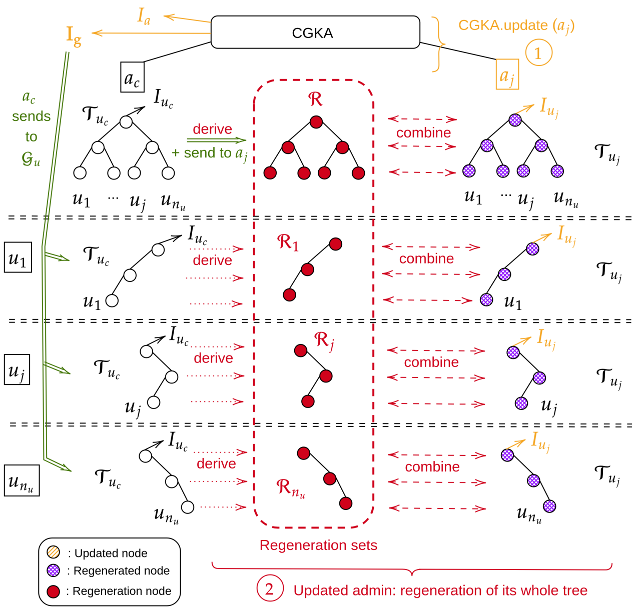

Figure 3: Description of the update-admin operation in SUMAC.

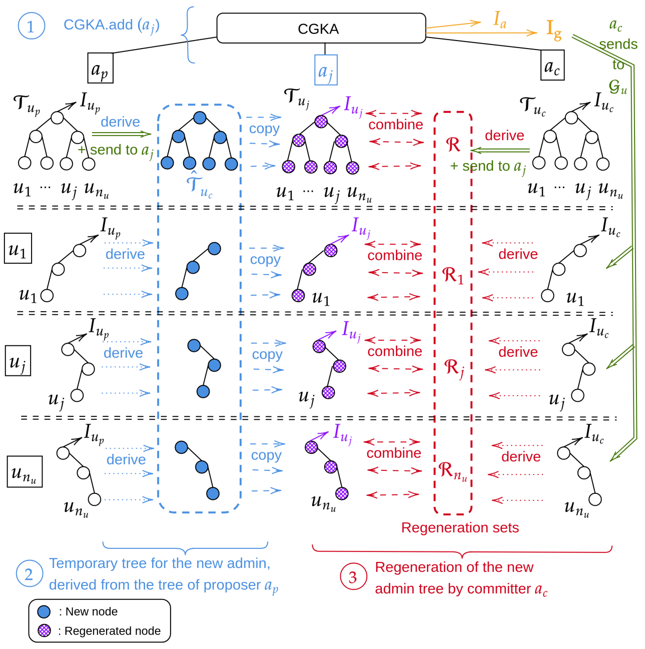

Figure 4: Description of the add-admin operation in SUMAC.

4.3.2 User Group Operations. Committer a_c performs an operation $op \in \{$**add-user**, **remove-user**, **update-user**$\}$ on standard user u_j as follows (cf. Figures 6, 7 and 8):

(1) For all these three operations, a_c carries out, in the tree $\mathcal{T}_{u_c}$ that it manages, the TMKA operation corresponding to

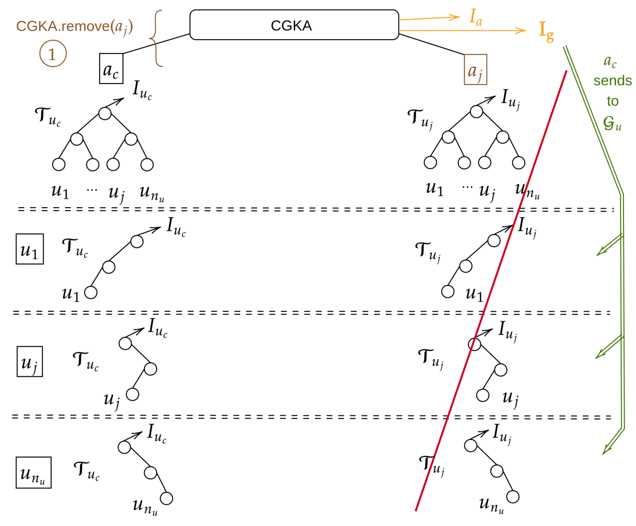

Figure 5: Description of the remove-admin operation in SUMAC.

op (cf. Section 4.1). Each standard user therefore receives from a_c the information needed to update the nodes from its direct path in $\mathcal{T}_{u_c}$ that it has in common with the added/ removed/ updated standard user u_j.

(2) A regeneration process spreads the changes from $\mathcal{T}_{u_c}$ to the other user trees $\mathcal{T}_{u_{i\neq c}}$:
- For an **add-user** or a **remove-user**, a regeneration set $\mathcal{R}$ is created by a_c by derivation of the updated path of u_j in $\mathcal{T}_{u_c}$: $\mathcal{R} :=$ **derive**$(\mathcal{P}(u_j, \mathcal{T}^{i+1}_{u_c}), \text{"regen"})$. It is then broadcast by a_c to the admin group $\mathcal{G}_a$, and the other administrators regenerate u_j's path in their own user tree. In an **update-user**, the same process happens on the *extended* path of u_j (thus also including u_j's leaf, cf. Equation (1)):
$\mathcal{R} :=$ **derive**$(\mathcal{P}_x(u_j, \mathcal{T}^{i+1}_{u_c}), \text{"regen"})$.
- Each standard user u_i, that has a partial view of *all* the user trees, is able to locally compute its own regeneration set $\mathcal{R}_i$, derived from the nodes in $\mathcal{T}_{u_c}$ that were updated by the TMKA in the previous step. It then regenerates these nodes in the trees $\mathcal{T}_{u_{i\neq c}}$, with that same regeneration set.
- The new group key I_g is derived from the top regeneration node, common to all the regeneration sets since it is derived from the root of the user tree $\mathcal{T}_{u_c}$. The new group key is thus naturally known by the committer and all the standard users, as well as by the other administrators as soon as they receive the committer's regeneration set $\mathcal{R}$.

The main advantage of such a regeneration process, in terms of communication cost, is to update $n_a - 1$ user trees with only one broadcast message of size logarithmic in the number of standard users.

(3) In an **add-user**, contrary to the other two user operations, the leaf associated with u_j in user trees $\mathcal{T}_{u_{i\neq c}}$ is still empty at that point. Moreover, u_j does not know (and must not, due to forward secrecy) its direct path in these trees. Therefore, all administrators $a_{i\neq c}$ must subsequently draw a random leaf secret related to u_j's leaf in their user tree, and transmit to u_j (possibly in unicast) that leaf secret and the previously regenerated direct path of u_j in that tree. In practice, this step is carried out progressively: temporarily, u_j is attached to the root of the trees $\mathcal{T}_{u_{i\neq c}}$ and every communication sent to standard users in these trees must be encrypted under that user's own public key[13]. Then, each time an administrator commits, it checks beforehand whether it has standard users straightly attached to the root of its user tree, and the case being, it sends them the secrets of their direct paths[14].

4.3.3 *Upgrade User / Downgrade Admin.*

Due to the strict partitioning between administrators and standard users in SUMAC, imposed by security constraints, it is not possible to change in one step the status of a group member. Indeed, each type of group members knows secret information that is not compatible with the other category: an administrator has a full view of its user tree, whereas a standard user only knows its path in this tree but also knows its path on all the other user trees. Consequently, SUMAC changes the status of a group member in two steps, by firstly removing it from the member group – which updates the secret elements this member had knowledge of – and then adding it as a new member of the other type.

We note that in the propose & commit paradigm, instead of carrying out this operation in two separate commits – and therefore, two different epochs, which would imply some unavailability time for the migrating group member, in between these two commits –, the two steps mentioned above can be executed in a row. In that case, group members still perform the two operations composing this migration process; nevertheless, the intermediary group key $\overline{I_g}$ yielded by the first operation is not used.

These steps are detailed in Figure 17 for the separate-operations framework, thus within two different commits.

4.3.4 *Limit Case: Adding a Second Administrator.*

A limit case of SUMAC is the operation of adding a second administrator to the admin group. Indeed, the regular **add-admin** operation uses a pair of administrators (a_p, a_c), without which the partitioning between administrators is not respected (and may be hard to achieve, even after several updates of that new administrator, depending on the corrupt standard users in the considered security game). When there exists only one administrator in $\mathcal{G}_a$, it is therefore impossible to implement that **add-admin** operation as mentioned above, and the new administrator a_j must create its own user tree by itself and distribute to every standard user its direct path (hence a cost of $n_u(\log(n_u) + 1)$ secrets to transmit, encrypted in HPKE). This unwanted use case may happen in two situations:

- At the beginning of the group history, when the admin group $\mathcal{G}_a$ is progressively built. In this inevitable case (unless the group sticks to a single administrator), the induced

[13]This process is similar to the mechanism of *merged leaves* in MLS, where new users are attached higher in the Ratchet Tree, initially directly to the root, as long as the nodes in their direct path have not been updated.

[14]This operation can also be optimized, for instance by updating at the same time that users' siblings.

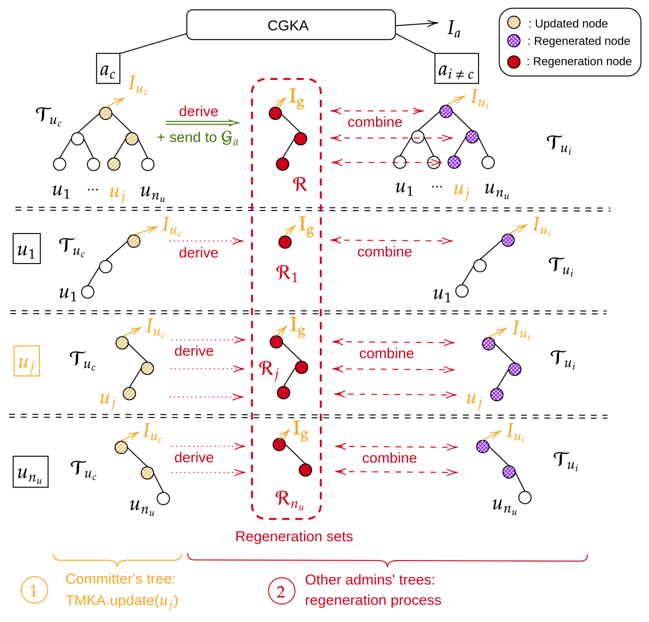

Figure 6: Description of the update-user operation in SUMAC.

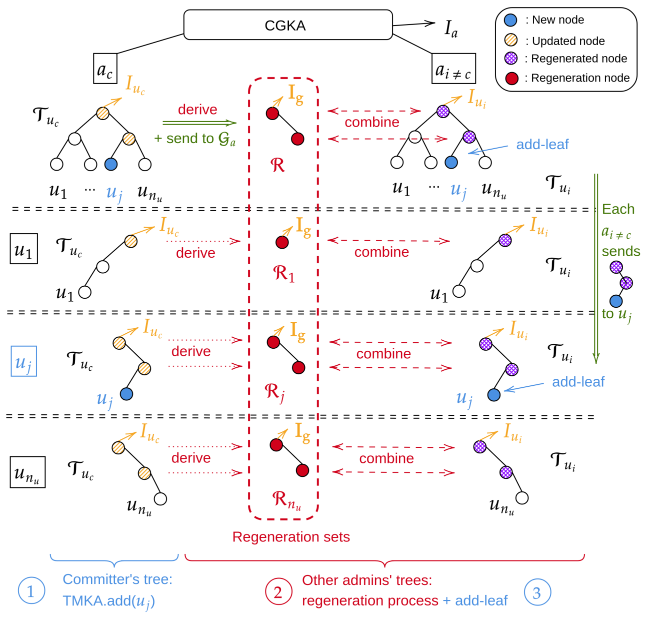
CGKA
: New node
: Updated node
: Regenerated node
: Regeneration node
derive
+ send to $\mathcal{G}_a$
combine
add-leaf
Each $a_{i\neq c}$ sends to u_j
Regeneration sets
Committer's tree: TMKA.add(u_j)
Other admins' trees: regeneration process + add-leaf

Figure 7: Description of the add-user operation in SUMAC.

cost is very small since the number of existing standard users remains low at that point.

- When a large group is built in MKA (with a single administrator), before switching to the multi-administrator setting. This case may imply an important overhead. However, we esteem that this situation is unlikely to occur frequently in real use-cases, since the single-administrator or multi-administrator setting is generally determined at the beginning of a group history, and even frequently depends on the application using the A-CGKA protocol. The applicative

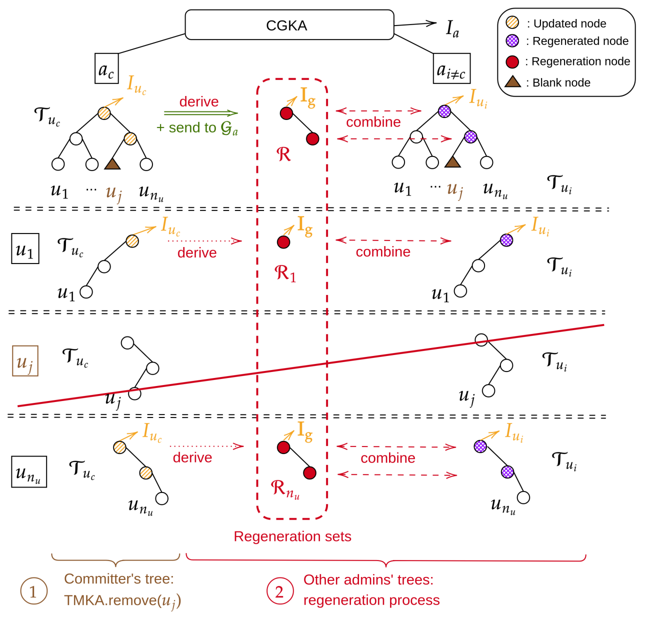
CGKA
: Updated node
: Regenerated node
: Regeneration node
: Blank node
derive
+ send to $\mathcal{G}_a$
combine
Regeneration sets
Committer's tree: TMKA.remove(u_j)
Other admins' trees: regeneration process

Figure 8: Description of the remove-user operation in SUMAC.

layer may also incite the first administrator to quickly add at least another peer in order to avoid such an inconvenient overhead (as well as to avoid the case where the single administrator leaves the conversation).

We underline that a group with two administrators works fine as long as none of them leaves prematurely the admin group, in which case we fall back on the previous limit case.

4.3.5 Authentication of the Messages. In an A-CGKA, the messages sent by the administrators (similarly to those sent by the group manager in an MKA) must be authenticated so that any group member processing a commit message can check that it indeed originates from a privileged group member. No authentication is required by default for the standard users, since any change proposed by a (potentially impersonated) standard user must be subsequently validated by an administrator.

However, in SUMAC, we require that any proposal or commit message is authenticated with a digital signature. This implies that any group member, including standard users, must generate its own signature key-pair, refresh it regularly and distribute its fresh public key to the whole group. In addition, any administrator processing a proposal message and any group member processing a commit message must check its authenticity and the legitimacy of its sender before taking it into account.

5 Correctness and Security

The correctness and security analyses of SUMAC as an A-CGKA rely on the framework of [2]. This work is itself based on [1] – for the security of a regular CGKA – upon which it adds, as additional constraint inherent to the administrated feature of an A-CGKA, the potential of an adversary to win the security game by forging a commit message on behalf of an administrator, instead of breaking the indistinguishability property of the group key.

We display here the high level description of our security model and the security results. The correctness analysis and the security proof are given in Appendix B.

5.1 Security Model

5.1.1 A-CGKA Security Game. Originating from the model of [2], the security of SUMAC is assessed trough a key indistinguishability security game KIND between a challenger C and an active and adaptive adversary $\mathcal{A}$. It nevertheless differs from its original model on two points: firstly, the concept of key indistinguishability in SUMAC does not concern a single group key but a tuple of admin, user and group keys. Moreover, the A-CGKA security against impersonation attacks is assessed in this paper regarding administrators and standard users (and therefore against the forgery of both proposal and commit messages), whereas [2] only considers the case of admin impersonation (and the forgery of a commit).

The challenger starts by tossing a coin b in order to determine if the security game occurs in the real ($b = 0$) or in the random ($b = 1$) world. Then, the adversary can make a sequence of queries to some oracles, in any arbitrary order, in order to simulate the evolution of the member group over time (*create* – at the beginning of the experiment –, *propose* and *commit* oracles) or the potential attacks carried out during the group history (*reveal*, *corrupt* and *inject* oracles). A *challenge* oracle must also be queried during the experiment in order to test the ability of the adversary to break the key indistinguishability of the protocol. These oracles are:

(1) $O^{create}(a_c, (a_\ell)_\ell, (u_\ell)_\ell)$: at the beginning of the security game, administrator a_c creates a member group $\mathcal{G}$ with admin and user subgroups $\mathcal{G}_a = (\text{gid}, (a_\ell)_{\ell=1}^{n_a})$ and $\mathcal{G}_u = (\text{gid}, (u_\ell)_{\ell=1}^{n_u})$.
(2) $O^{propose}(m_i, m_j, P)$: group member m_i proposes to implement a proposal P, corresponding to a group operation as given in the list of definition 3.3, for group member m_j.
(3) $O^{commit}(a_c, \overrightarrow{P}, \overrightarrow{op})$: administrator a_c implements group operations issued from a vector $\overrightarrow{P}$ of legitimate proposals and/or a vector of self-initiated operations $\overrightarrow{op}$.
(4) $O^{deliver}(m_j, t^i, C)$: sends to group member m_j the commit message C at epoch t^i and makes m_j process that commit.
(5) $O^{expose}(m_j, t^i)$[15]: leaks to the adversary the private state of group member m_j at the epoch t^i of the query.
(6) $O^{reveal}(t^i)$: leaks to the adversary the tuple of common keys $(I_a^i, (I_{u_\ell}^i)_{\ell=1}^{n_a}, I_g^i)$ for the epoch t^i of the query.
(7) $O^{inject}(m_j, msg \in \{C_{adm}, C_{usr}, C_{ind}, P\}, t^i)$: the adversary tries to forge either a commit or a proposal message at epoch t^i and to send it to group member m_j (for a commit) or admin a_j (for a proposal) for processing. If that member successfully processes it, the forgery succeeds (which is represented by the forgery predicate **forge** below) and the challenge bit b is given to the adversary. Otherwise the oracle returns $\perp$. As prerequisite, the admin impersonation-safe $\mathbf{safe_{adm}}$ and the user impersonation-safe $\mathbf{safe_{usr}}$ predicates (explained beneath) must be satisfied.
(8) $O^{challenge}(t^i)$: Let $k_a^0 = I_a^i$, $(k_{u_\ell}^0)_\ell = (I_{u_\ell}^i)_{\ell=1}^{n_a}$ and $k_g^0 = I_g^i$ be respectively the real admin key, user keys and group key at the epoch t^i of that challenge query and k_a^1, $(k_{u_\ell}^1)_\ell$ and k_g^1 be fresh random keys from the key space $\mathcal{K}$. Then, if either the cgka-safe $\mathbf{safe_{cgka}}$ or the forgery predicate **forge** is satisfied, the tuple of keys $(k_a^b, (k_{u_\ell}^b)_\ell, k_g^b)$ – with b the coin tossed by the challenger at the beginning of the security game – is given to the adversary. If none of these predicates is satisfied, the adversary gets nothing.

At the end of the game, the adversary outputs a bit $\hat{b}$ and wins if $\hat{b} = b$ (as stated above, the adversary knows the challenge bit in case of a successful impersonation of an administrator or a standard user). We call the A-CGKA scheme (t, q, ϵ)-A-CGKA-secure if for any adversary $\mathcal{A}$ making at most q queries to the oracles and running in time t, its advantage advKIND in winning the key indistinguishability game, with respect to the safe and forgery predicates $\mathbf{safe_{cgka}}$, $\mathbf{safe_{adm}}$, $\mathbf{safe_{usr}}$ and **forge** , remains bounded by ϵ:

$$\mathsf{advKIND}^{\text{A-CGKA}}_{\mathbf{safe_{cgka}},\mathbf{safe_{adm}},\mathbf{safe_{usr}},\mathbf{forge}}(\mathcal{A}^O) = \left| \Pr\left[\hat{b} = b\right] - \frac{1}{2} \right| \leq \epsilon$$

5.1.2 Safe Predicates. In order to disregard in the A-CGKA security game trivial attacks that cannot be avoided, we use predicates that guarantee that none of these unwanted attacks occur at the time of query of the challenge oracle. The concept of cgka-safe predicate – noted here $\mathbf{safe_{cgka}}$ – originates from the regular CGKA world [1]; it prevents the trivial leakage of the group key. [2] extends this notion to the trivial impersonation of an administrator in their A-CGKA security model, with the admin impersonation-safe predicate[16] $\mathbf{safe_{adm}}$.

As we extend in our paper the scope of potential adversarial impersonations by considering that of a standard user, we additionally determine a user impersonation-safe predicate $\mathbf{safe_{usr}}$, which is similar to its admin counterpart and which prevents trivial forgery of a proposal message by an impersonated standard user. As explained above, the reason for this new predicate is that even if an A-CGKA controls the changes in the member group by granting commit rights to admins only, standard users still have the ability to propose changes through possibly malicious proposals that could then be naively validated by an honest committer. In particular, a self-removal forged proposal sent by an impersonated standard user would be most probably validated by the committer and thus would lead to a dishonest change in the group membership.

In practice, the forgery of proposals is already (informally) taken into account by some CGKAs such as in MLS, where every handshake message – proposals included – is signed by its emitter.

5.1.3 Forgery Predicate. In addition to the safe predicate, the forgery predicate, noted **forge** and also issued from [2], formalizes the fact that the forgery of either a commit or a proposal message, represented by an adversarial query to the *inject* oracle, has been accepted by a processing group member, making it successful. In that case, the adversary wins the security game, provided that the admin impersonation-safe predicate $\mathbf{safe_{adm}}$ (for both a forged commit and a forged proposal) and the user impersonation-safe predicate $\mathbf{safe_{usr}}$ (only for a forged proposal) are also respected.

[15]Corresponds to the $O^{corrupt}$ oracle in some related works.

[16]Stated in [2] as the "admin predicate".

5.2 Security Results

We prove the security of SUMAC by relying on that of the underlying admin CGKA cgka (that matches the framework of [2, Figure 3]), of the user TMKA tmka (according to [9, Figure 2]) and of the encryption schemes and key derivation functions, whose security models are recalled in Figures 10 and 11 (Appendix A, p.15). Our security proof, given in Appendix B, is established in the standard model. It lies in the separate-operations framework, for sake of simplicity, but could easily be extended in a future work to the propose & commit paradigm.

Theorem 5.1 (Security of SUMAC). *Let* cgka *be a $(t_{cgka}, q, \epsilon_{cgka})$-secure CGKA and* tmka *be a $(t_{tmka}, q, \epsilon_{tmka})$-secure tree-based MKA. Let $\mathcal{E}_1$ be a $(t_{hpke}, \epsilon_{hpke})$-IND-CPA secure HPKE mechanism, $\mathcal{E}_2$ a $(t_{aead}, \epsilon_{aead})$-IND-CPA secure AEAD*[17] *scheme, $\mathcal{S}$ a $(t_{sig}, \epsilon_{sig})$-SUF-CMA secure signature scheme and H and F key derivation functions modeled respectively as a (t_H, ϵ_H)-secure pseudorandom function and a (t_F, ϵ_F)-secure dual-PRF.*

Then SUMAC, implemented with cgka *as its admin CGKA,* tmka *as its user MKA and F, H, $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{S}$ as its auxiliary functions, is $(t_{sumac}, q, \epsilon_{sumac})$-A-CGKA secure, with $t_{sumac} \approx t_{cgka} \approx t_{tmka}$ and ϵ_{sumac} s.t.:*

$$\begin{aligned}\epsilon_{sumac} \leq q^2\epsilon_{sig} + \epsilon_{cgka} + \epsilon_{tmka} + q \cdot \big(&2\epsilon_{aead} + (n_u + 1) \cdot \epsilon_{hpke} \\ &+ (2n_u - 1 + (n_u - 1)(\log_2(n_u) + 1)) \cdot \epsilon_F \\ &+ (5n_u + \log_2(n_u) + 2) \cdot \epsilon_H\big)\end{aligned} \tag{8}$$

6 Performances

This section presents the performances of SUMAC with respect to the traditional metrics of communication and storage (memory) costs. We underline that the main purpose of our algorithm is to decrease the communication cost of an A-CGKA, hence this factor is dealt with more details than the other one. These costs are assessed for several group sizes, with numbers of standard users and administrators of $(2^4, 2^2)$, $(2^8, 2^4)$ and $(2^{16}, 2^8)$. We have set the number of administrators to the square root of the number of standard users (the need for administrators indeed does not grow linearly with the number of standard users). This choice, that we esteem realistic, is arbitrary; however our simulations have shown that this ratio does not anyway strongly influence the performances of our protocol.

We state beneath the main results of our performance analysis; the related computations are detailed in Appendix D.

6.1 Storage Cost

Table 1 (Appendix D.1, p.21) displays the results of our analysis on the storage cost of the aforementioned protocols. It shows a great improvement brought by SUMAC in that matter, with a gain w.r.t. TreeKEM and IAS-TK (that have the same storage cost) between 25% and 96% for administrators, and ranging from 74% to more than 99% for standard users.

This improvement comes from the fact that in SUMAC, the public states of the user trees are no longer composed of public keys – very cumbersome in the post-quantum framework – but of much more compact secret keys. Moreover, even if standard users have a transversal view on all n_a user Ratchet Trees, they do no need to keep a complete view of these trees, but instead only record their direct path in these trees, of length logarithmic in the number of users.

[17] Authenticated Encryption scheme with Associated Data.

6.2 Communication Cost

6.2.1 Framework of the Analysis. We study here the communication cost induced in SUMAC by the eight group operations of an A-CGKA, and we compare these costs with those of the two A-CGKAs from [2], implemented upon TreeKEM: IAS-TK and DGS-TK. We also include TreeKEM in this performance analysis; since the CGKA operations do not precisely match that of an A-CGKA, we compare the **add**, **remove** and **update** operations of TreeKEM with both the admin and the user operations of the A-CGKAs.

In order to avoid any bias in this comparison, we consider the *separate-operations* paradigm, where group operations are carried out independently the one from another, contrary to what is done in the propose & commit setting. Indeed, because the latter enables to group any combination of any number of group operations within a commit, the performances of a protocol in that framework greatly depends on the arbitrary choice of proposals to be implemented, which makes difficult to impartially compare several protocols.

Regarding the message distribution method (cf. Appendix D.3.1), our analysis is made in the *broadcast-only* distribution setting. The idea is indeed to remain generic by considering the most basic Delivery Service. We nevertheless underline that SUMAC also allows the *group-distribution* method – contrary to TreeKEM, DGS-TK and IAS-TK –, which is far more efficient since messages addressed to administrators or standard users are only distributed to their corresponding subgroups instead of the entire member group.

6.2.2 Results. The results depicted in Table 4 (Appendix D.3, p.28) show a significant reduction in communication cost with SUMAC, compared with TreeKEM and the IAS-TK and DGS-TK A-CGKAs, especially in the post-quantum setting due to the use of secret-key cryptography.

In all the admin and user operations, the communication cost of SUMAC appears always lower than that of TreeKEM, IAS-TK and DGS-TK. SUMAC performs particularly well in user operations (**add-user**, **remove-user**, **update-user**), in which the communication cost of SUMAC is between 42 and 97% lower than that of TreeKEM and IAS-TK. The best performances of SUMAC occur in a large group in the PQ setting, where the cost is divided by a factor 17 to 37. Even for the admin operations (**add-admin**, **remove-admin**, **update-admin**), SUMAC has a communication cost 27% to 60% lower than TreeKEM, depending on the operation considered.

On the other hand, SUMAC is less efficient than IAS-TK and DGS-TK[18] for the two – less frequent – migration operations of **upgrade** and **downgrade**, due to the partitioning between administrators and standard users in SUMAC.

The overall performances of SUMAC therefore depend on the sequence of group operations that are carried out during the member group history. In particular, SUMAC would be best used by privileging adding new administrators straightly from outside the member group, instead of upgrading existing standard users. Since

[18] No comparison is possible with TreeKEM for these migration operations, because this one is not administrated.

the most frequent operations that are expected to be performed (update of the administrators and any operation on the numerous standard users) induce a lower communication cost with SUMAC than its competitors, we can infer that in general, our protocol behaves much more efficiently than DGS-TK, IAS-TK and even than TreeKEM. The comparison with TreeKEM underlines that the administrated setting, far from being a costly burden, can be turned into an advantage in terms of performances by using an adequate protocol design.

References

[1] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2020. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging. In *Advances in Cryptology – CRYPTO 2020, Part I (Lecture Notes in Computer Science, Vol. 12170)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Cham, Switzerland, Santa Barbara, CA, USA, 248–277. https://doi.org/10.1007/978-3-030-56784-2_9

[2] David Balbás, Daniel Collins, and Serge Vaudenay. 2023. Cryptographic Administration for Secure Group Messaging. In *USENIX Security 2023: 32nd USENIX Security Symposium*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, Anaheim, CA, USA, 1253–1270.

[3] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2019. *The Messaging Layer Security (MLS) Protocol.* Internet-Draft draft-ietf-mls-protocol-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/08/ Work in Progress.

[4] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. 2023. The Messaging Layer Security (MLS) Protocol. RFC 9420. https://doi.org/10.17487/RFC9420

[5] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. 2022. Hybrid Public Key Encryption. RFC 9180. https://doi.org/10.17487/RFC9180

[6] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. 2024. *The Messaging Layer Security (MLS) Architecture.* Internet-Draft draft-ietf-mls-architecture-15. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/15/ Work in Progress.

[7] Benjamin Beurdouche, Eric Rescorla, Emad Omara, Srinivas Inguva, and Alan Duric. 2024. *The Messaging Layer Security (MLS) Architecture.* Internet-Draft draft-ietf-mls-architecture-13. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/13/ Work in Progress.

[8] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. 2018. *TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS).* Research Report. Inria Paris. https://inria.hal.science/hal-02425247

[9] Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. 2022. Multicast Key Agreement, Revisited. In *Topics in Cryptology – CT-RSA 2022 (Lecture Notes in Computer Science, Vol. 13161)*, Steven D. Galbraith (Ed.). Springer, Cham, Switzerland, Virtual Event, 1–25. https://doi.org/10.1007/978-3-030-95312-6_1

[10] Céline Chevalier, Guirec Lebrun, Ange Martinelli, and Jérôme Plût. 2024. The Art of Bonsai: How Well-Shaped Trees Improve the Communication Cost of MLS. Cryptology ePrint Archive, Report 2024/746. https://eprint.iacr.org/2024/746

[11] Amos Fiat and Moni Naor. 1994. Broadcast Encryption. In *Advances in Cryptology – CRYPTO'93 (Lecture Notes in Computer Science, Vol. 773)*, Douglas R. Stinson (Ed.). Springer Berlin Heidelberg, Germany, Santa Barbara, CA, USA, 480–491. https://doi.org/10.1007/3-540-48329-2_40

[12] Eric J. Harder and Debby M. Wallner. 1999. Key Management for Multicast: Issues and Architectures. RFC 2627. https://doi.org/10.17487/RFC2627

[13] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, Virtual Event, Republic of Korea, 1441–1462. https://doi.org/10.1145/3460120.3484817

[14] Kathrin Hövelmanns, Eike Kiltz, Sven Schäge, and Dominique Unruh. 2020. Generic Authenticated Key Exchange in the Quantum Random Oracle Model. In *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part II (Lecture Notes in Computer Science, Vol. 12111)*, Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas (Eds.). Springer, Cham, Switzerland, Edinburgh, UK, 389–422. https://doi.org/10.1007/978-3-030-45388-6_14

[15] Kaisei Kajita, Keita Emura, Kazuto Ogawa, Ryo Nojima, and Go Ohtake. 2022. Continuous Group Key Agreement with Flexible Authorization and Its Applications. Cryptology ePrint Archive, Report 2022/1768. https://eprint.iacr.org/2022/1768

[16] Raphael Robert. 2024. OpenMLS v0.6. https://github.com/openmls/openmls.

[17] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. 1998. Secure Group Communications Using Key Graphs. In *Proceedings of ACM SIGCOMM*. Vancouver, BC, Canada, 68–79.

A Omitted Preliminaries

We state hereunder some primitives that are either at the edge of our study (such as the A-CGKA-FA below) or that are preliminaries omitted from the main body, for lack of space.

A.1 Administrated-CGKA with Flexible Authorizations

Informally, an Administrated-CGKA with Flexible Authorizations (A-CGKA-FA) is a variant of the A-CGKA from Definition 3.3, conceptualized by [15], where the administration rights are given to the administrators and the standard users according to a fine-grained policy, captured by our notion of authorization sets. These ones depict the group operations allowed for each type of group member, considering separately the cases of a proposal and of a commit.

Definition A.1 (A-CGKA-FA). An Administrated Continuous Group Key Agreement with Flexible Authorizations (A-CGKA-FA) is a group key agreement protocol where group members are divided between administrators and standard users, and consisting of a set O of group operations, associated with authorizations, and of the algorithms beneath.

- **Initialize**: group member $m_i \in \{a_i, u_i\}$ creates its initial state γ_i (for an administrator) or δ_i (for a standard user):
  $\boxed{\gamma_i \leftarrow \textbf{init}(a_i) \quad / \quad \delta_i \leftarrow \textbf{init}(u_i)}$
- **Create Group**: initial group member m_1 creates an initial member group $\mathcal{G}$ from a list $L_a = (a_k)_{k=1}^{n_a}$ of n_a administrators, a list $L_u = (u_k)_{k=1}^{n_u}$ of n_u users and a unique group ID gid, updating in accordance its initial state γ_1/δ_1 into γ_1'/δ_1', yielding the new member group $\mathcal{G}$ (composed of the admin group $\mathcal{G}_a = (\text{gid}, L_a)$ and user group $\mathcal{G}_u = (\text{gid}, L_u)$), an initial group key I and an associated commit message C:
  $\boxed{(\gamma_1'/\delta_1', (\mathcal{G}_a, \mathcal{G}_u), I, C) \leftarrow \textbf{create-group}(\gamma_1/\delta_1, L_a, L_u)}$
- **Propose**: group member m_i, with state γ_i or δ_i, proposes a group operation $op \in \mathcal{A}_{adm/usr}^{prop}$ to apply on group member m_j within $\mathcal{G}$, generating a proposal message P and an updated state γ_i' or δ_i':
  $\boxed{(\gamma_i'/\delta_i', P) \leftarrow \textbf{propose}(\gamma_i/\delta_i, \text{gid}, op(m_j))}$
- **Commit**: group member m_i, with state γ_i or δ_i, takes as input a *possibly empty* vector of proposals $\overrightarrow{P}$ sent by other group members and a *possibly empty* vector of group operations $\overrightarrow{op}$ that it has generated itself, s.t. $\forall op \in \overrightarrow{op}, op \in \mathcal{A}_{adm/usr}^{com}$. Considering only the legitimate[19] operations from these two vectors, m_i updates its state accordingly, generates a new group key I' and sends a commit message C to the existing and potentially new users:
  $\boxed{(\gamma_i'/\delta_i', I', C) \leftarrow \textbf{commit}(\gamma_i/\delta_i, \text{gid}, \overrightarrow{P}, \overrightarrow{op})}$
- **Process**: group member m_i, with state γ_i or δ_i, processes a commit message C it has received. After checking the legitimacy of this message[19], it updates its state accordingly and computes the new group key I' resulting from these changes. In case of failure, and notably if the commit message is not legitimate, m_i aborts the ongoing process and returns a failure value $\perp$:
  $\boxed{(\gamma_i'/\delta_i', I')/\perp := \textbf{process}(\gamma_i/\delta_i, \text{gid}, C)}$

The set O of group operations in an A-CGKA-FA is the following:

[19] A group operation from a Proposal or a Commit is legitimate if it belongs to the authorization set $\mathcal{A}_{adm/usr}^{prop/com}$ of the group member at its origin.

- **add-admin** (a_j): new admin a_j is added to the admin group $\mathcal{G}_a$ directly while joining the member group $\mathcal{G}$;
- **remove-admin** (a_j): admin a_j is removed from the entire member group $\mathcal{G}$;
- **update-admin** (a_j): admin a_j has its state updated;
- **add-user** (u_j): new standard user u_j is added to the user group $\mathcal{G}_u$;
- **remove-user** (u_j): standard user u_j is removed from the member group $\mathcal{G}$;
- **update-user** (u_j): standard user u_j has its state updated;
- **upgrade-user** (u_j): standard user u_j becomes an administrator and leaves $\mathcal{G}_u$ to join $\mathcal{G}_a$;
- **downgrade-admin** (a_j): admin a_j becomes a standard user and leaves $\mathcal{G}_a$ to join $\mathcal{G}_u$.

We define $\mathcal{A}^{prop}_{adm}$, $\mathcal{A}^{com}_{adm}$, $\mathcal{A}^{prop}_{usr}$ and $\mathcal{A}^{com}_{usr}$ as subsets of O corresponding to the authorized operations for administrators and standard users in a proposal and in a commit.

A.2 Signature Security Game

Definition A.2 (SUF-CMA Security for a Digital Signature). A digital signature scheme $\mathcal{S}$ is said to be (t, ϵ)-secure against strong existential forgery under chosen message attacks (SUF-CMA) if, for any adversary $\mathcal{A}$ running in polynomial-time t, the probability that this adversary wins the security game (given by $\Pr[\mathcal{A}^{suf\text{-}cma}_{\mathcal{S}} = 1]$) is bounded by ϵ, where the probability is taken over the choice of the challenger and adversary's random coins.

```
SUF-CMA$^{\mathcal{A}}_{\mathcal{S}}$
1 : $(spk, ssk) \leftarrow$ key-gen$()$
2 : $Q := \emptyset$
3 : $(m, \sigma) \leftarrow \mathcal{A}^{O^{sign}}(spk)$
4 : require$(m, \sigma) \notin Q$
5 : if verif$(spk, \sigma, m) = \top$ then :
6 :     return 1    / successful forgery
7 : else :
8 :     return 0

$O^{sign}(m)$
1 : $\sigma \leftarrow$ sign$(ssk, m)$
2 : $Q := Q \cup \{(m, \sigma)\}$
3 : return $\sigma$
```

Figure 9: SUF-CMA security game for a digital signature scheme.

A.3 Security and Correctness of an Encryption Scheme

A.3.1 IND-CPA Security of an Encryption Scheme. The security game corresponding to the IND-CPA security of public-key and secret-key encryption schemes, with two experiments $\mathsf{IND\text{-}CPA}^{\mathcal{A}}_{\mathcal{E}}(b)$, $b \in \{0, 1\}$, is detailed in Figure 10.

We say the the encryption scheme $\mathcal{E}$ is (t, ϵ)-IND-CPA secure if for any adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ running in polynomial-time t, its advantage in winning the security game by guessing the experiment bit b is bounded by ϵ:

$$\mathsf{adv}^{\mathsf{IND-CPA}}_{\mathcal{E}}(\mathcal{A}) = \left| \Pr\left[\hat{b} = b\right] - \frac{1}{2} \right| \leq \epsilon \tag{9}$$

```
IND-CPA$^{\mathcal{A}}_{\mathsf{PKE}}(b)$
1 : $(pk, sk) \leftarrow$ key-gen$()$
2 : $(m^0, m^1, st) \leftarrow \mathcal{A}_1(pk)$
3 : $c^* \leftarrow$ enc$(pk, m^b)$
4 : $\hat{b} \leftarrow \mathcal{A}_2(pk, c^*, m^0, m^1, st)$
5 : return $\hat{b}$

IND-CPA$^{\mathcal{A}}_{\mathsf{SKE}}(b)$
1 : $k \leftarrow$ key-gen$()$
2 : $(m^0, m^1, st) \leftarrow \mathcal{A}_1^{O^{enc}}()$
3 : $c^* \leftarrow$ enc$(k, m^b)$
4 : $\hat{b} \leftarrow \mathcal{A}_2^{O^{enc}}(c^*, m^0, m^1, st)$
5 : return $\hat{b}$

$O^{enc}(m)$
1 : $c \leftarrow$ enc$(k, m)$
2 : return $c$
```

Figure 10: IND-CPA security games for a public-key encryption scheme (left) – including HKPE constructions – and a secret-key encryption scheme (right), comprising the AEAD mechanisms.

A.3.2 Worst-Case Correctness. We say that a secret-key encryption scheme $\mathcal{E}$ is *perfectly* correct if we have:

$$\forall m \in \mathcal{M}, \forall k \leftarrow \mathbf{key\text{-}gen}(),\ \mathbf{dec}(k, \mathbf{enc}(k, m)) = m \tag{10}$$

Perfect correctness for a public-key encryption scheme (PKE) is defined similarly. However, many PKEs, especially in the post-quantum framework, do not reach this property and it appears useful to statistically bound their correctness error.

Definition A.3 (Worst-Case Correctness (from [14])). The worst-case correctness of a public-key encryption scheme PKE, with a message space $\mathcal{M}$ and a randomness space $\mathcal{R}$, is defined as:

$$\delta^{wc} := \underset{(pk,sk)\leftarrow \mathbf{key\text{-}gen}()}{\mathbb{E}} \left(\underset{m \in \mathcal{M}}{\mathrm{Max}} \left\{ \underset{r \in \mathcal{R}}{\Pr} \left[\mathbf{dec}\big(sk, \mathbf{enc}(pk, m; r)\big) \neq m \right] \right\} \right) \tag{11}$$

A.4 PRF and Dual-PRF Security

Informally, a pseudorandom function (PRF) is a keyed function $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ whose outputs are indistinguishable from a random distribution from the output space $\mathcal{Y}$. A dual-PRF (dPRF) is a function that behaves as a PRF even when reversing the roles of its key $k \in \mathcal{K}$ and its input $x \in \mathcal{X}$.

The security of a PRF is established with a real-random security game described in Figure 11. We say that the function F is a (t, ϵ)-secure PRF if, for any adversary $\mathcal{A}$ running in polynomial-time t, the probability that this adversary wins the security game (by guessing the experiment bit b) is bounded by ϵ.

$$\mathsf{adv}^{\mathsf{PRF}}_{\mathsf{F}}(\mathcal{A}) = \left| \Pr\left[\hat{b} = b\right] - \frac{1}{2} \right| \leq \epsilon \tag{12}$$

Beforehand, we define $\mathsf{Fun}(\mathcal{X}, \mathcal{Y})$ as the set of all functions defined over $\mathcal{X}$ and $\mathcal{Y}$.

```
PRF_F^A(b)                          O^{prf}(x)
1 :  if $b = 0$ then :              1 :  $y := f(x)$
2 :      $k \leftarrow \$ \mathcal{K}$              2 :  return $y$
3 :      $f := F(k, \cdot)$
4 :  else :
5 :      $f \leftarrow \$ \mathsf{Fun}(\mathcal{X}, \mathcal{Y})$
6 :  $\hat{b} \leftarrow \mathcal{A}^{O^{prf}}()$
7 :  return $\hat{b}$
```

Figure 11: Security game for a pseudorandom function.

B Security Analysis

This appendix details and proves the security results from Section 5.2, firstly by analyzing – in the framework of SUMAC – the safe and impersonation predicates used in the security game, and then by proving our main security result: Theorem 5.1.

B.1 Safe Predicates

This part is dedicated to assessing the three safe predicates $\mathbf{safe}_{\mathbf{cgka}}$, $\mathbf{safe}_{\mathbf{adm}}$ and $\mathbf{safe}_{\mathbf{usr}}$ for SUMAC.

B.1.1 CGKA Safe Predicate of SUMAC. [2] distinguishes in their study the part of the cgka-safe predicate that is intrinsic to any CGKA – that they call the optimal safe predicate $\mathbf{safe}_{\mathbf{cgka\text{-}opt}}$ – and the additional part of that predicate that originates from a given protocol Π – noted $\mathbf{safe}^{\Pi}_{\mathbf{cgka\text{-}add}}$ –, s.t.:

$$\mathbf{safe}^{\Pi}_{\mathbf{cgka}} = \mathbf{safe}_{\mathbf{cgka\text{-}opt}} \wedge \mathbf{safe}^{\Pi}_{\mathbf{cgka\text{-}add}} \tag{13}$$

According to their analysis, $\mathbf{safe}_{\mathbf{cgka\text{-}opt}}$ covers the following unavoidable attacks:

- querying the *challenge* oracle right after a *reveal* oracle query or another *challenge* query;
- corrupting a group member, with a query to the *expose* oracle, at the challenge epoch or before it updates its states after that epoch.

$\mathbf{safe}_{\mathbf{cgka\text{-}opt}}$ therefore ensures that only one *reveal* or *challenge* query may be performed per epoch, and that for an *expose* query on a group member m_j:

- this one was outside the member group at the challenge time,
- or it had updated its keys before the challenge epoch (post-compromise security),
- or the exposure occurred after the challenge epoch and the update of the member's keys (forward secrecy).

We now determine the additional features of the cgka-safe predicate in SUMAC.

Proposition B.1 (SUMAC cgka-Safe Predicate). *Let us consider SUMAC implemented with an administration CGKA* cgka *and a Tree-based MKA* tmka. *Then, its cgka-safe predicate* $\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{cgka}}$ *can be expressed as:*

$$\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{cgka}} = \mathbf{safe}_{\mathbf{cgka\text{-}opt}} \wedge \mathbf{safe}^{\mathbf{cgka}}_{\mathbf{cgka\text{-}add}} \wedge \mathbf{safe}^{\mathbf{tmka}}_{\mathbf{cgka\text{-}add}} \tag{14}$$

Proof. The proof comes straightly from the design of SUMAC. Let us assume that the admin CGKA cgka has an additional cgka-safe predicate $\mathbf{safe}^{\mathbf{cgka}}_{\mathbf{cgka\text{-}add}}$ associated with it and that is not respected. Then, by definition of that predicate, nothing prevents the adversary from carrying out a trivial attack – other than the attacks captured by the optimal cgka-safe predicate – resulting in leaking the challenge admin key k^*_a (after derivation of the related cgka key I_{cgka}). Since the security game of SUMAC is built on the indistinguishability of the tuple of keys $(I_a, (I_{u_\ell})^{n_a}_{\ell=1}, I_g)$, leaking that admin key suffices to make the adversary win the security game with an overwhelming advantage. Similarly, if the TMKA tmka has an additional cgka-safe predicate $\mathbf{safe}^{\mathbf{tmka}}_{\mathbf{cgka\text{-}add}}$ that is violated – and since all user trees have the same structure –, the adversary is able to recover the challenge user keys $(k^*_{u_\ell})^{n_a}_{\ell=1}$ and win in consequence the security game with a very high probability.

Conversely, as the admin key and user keys only depend (respectively) on the admin CGKA and the TMKA algorithms[20], they are not trivially leaked to the adversary if the two cgka-safe predicates $\mathbf{safe}^{\mathbf{cgka}}_{\mathbf{cgka}}$ and $\mathbf{safe}^{\mathbf{tmka}}_{\mathbf{cgka}}$ are simultaneously respected. Moreover, the challenge group key k^*_g, which is output indirectly from the CGKA – if the previous group operation queried by the adversary is an admin operation – or from the TMKA – if the last operation is a user one –, is not leaked either to the adversary when these additional predicates are all respected. □

B.1.2 Impersonation Safe Predicates. Similarly to the cgka-safe predicate, the impersonation-safe predicates of an A-CGKA – $\mathbf{safe}_{\mathbf{adm}}$ regarding the safety of the administrators and $\mathbf{safe}_{\mathbf{usr}}$ for the standard users– can be each expressed as the logical conjonction of an intrinsic general impersonation predicate $\mathbf{safe}_{\mathbf{imp\text{-}opt}}$, independent of the protocol and of the type of group member considered, and an additional predicate $\mathbf{safe}^{\Pi}_{\mathbf{adm\text{-}add}}$ (resp. $\mathbf{safe}^{\Pi}_{\mathbf{usr\text{-}add}}$) that depends on the features of the protocol Π and the type of member it is applied on:

$$\mathbf{safe}^{\Pi}_{\mathbf{adm}} = \mathbf{safe}_{\mathbf{imp\text{-}opt}} \wedge \mathbf{safe}^{\Pi}_{\mathbf{adm\text{-}add}} \tag{15}$$

$$\mathbf{safe}^{\Pi}_{\mathbf{usr}} = \mathbf{safe}_{\mathbf{imp\text{-}opt}} \wedge \mathbf{safe}^{\Pi}_{\mathbf{usr\text{-}add}} \tag{16}$$

In particular, $\mathbf{safe}_{\mathbf{imp\text{-}opt}}$ excludes any impersonation on a previously corrupt group member that has not refreshed yet its signature key-pair.

Proposition B.2 (SUMAC Impersonation Safe Predicates). *Let us consider SUMAC implemented with an administration CGKA* cgka *and a Tree-based MKA* tmka*, respectively associated with impersonation-safe predicates*[21] $\mathbf{safe}^{\mathbf{cgka}}_{\mathbf{adm}}$ *and* $\mathbf{safe}^{\mathbf{tmka}}_{\mathbf{usr}}$. *Then, its admin and user impersonation-safe predicates* $\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{adm}}$ *and* $\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{usr}}$ *can be expressed as:*

[20]More specifically, the user key of the committer is straightly issued from the TMKA protocol, whereas the other user keys are output by the regeneration of their tree with the regeneration set, which itself solely depends on the TMKA protocol.

[21]The user impersonation-safe predicate of the TMKA captures trivial impersonation attacks carried out on standard users but does not cover impersonation attacks on the group manager. This is not an issue for the security of SUMAC, since the group managers of the user trees are the administrators, themselves protected from trivial attacks by the admin impersonation-safe predicate.

$$\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{adm}} = \mathbf{safe}^{\mathbf{cgka}}_{\mathbf{adm}} = \mathbf{safe}_{\mathbf{imp\text{-}opt}} \wedge \mathbf{safe}^{\mathbf{cgka}}_{\mathbf{adm\text{-}add}} \quad (17)$$

$$\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{usr}} = \mathbf{safe}^{\mathbf{tmka}}_{\mathbf{usr}} = \mathbf{safe}_{\mathbf{imp\text{-}opt}} \wedge \mathbf{safe}^{\mathbf{tmka}}_{\mathbf{usr\text{-}add}} \quad (18)$$

Proof. The proof is straightforward. The authentication of administrators, which aims to prevent the impersonation of these group members, is managed solely by the admin CGKA. Therefore, the scope of trivial impersonation attacks on the users of that CGKA considered by its impersonation-safe predicate $\mathbf{safe}^{\mathbf{cgka}}_{\mathbf{adm}}$ matches precisely that of the admin impersonation attacks in SUMAC, associated with the predicate $\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{adm}}$.

Similarly, the authentication of standard users in SUMAC is exclusively performed by the TMKA protocol which is applied on the user trees. Consequently, the trivial attacks on the users in the TMKA, captured by $\mathbf{safe}^{\mathbf{tmka}}_{\mathbf{usr}}$, correspond to that of the standard users in SUMAC, associated with the predicate $\mathbf{safe}^{\mathbf{sumac}}_{\mathbf{usr}}$. □

B.2 Proof of Theorem 5.1

We recall here the theorem stating the security of SUMAC:

Theorem 5.1 (Security of SUMAC). *Let* cgka *be a $(t_{cgka}, q, \epsilon_{cgka})$-secure CGKA and* tmka *be a $(t_{tmka}, q, \epsilon_{tmka})$-secure tree-based MKA. Let $\mathcal{E}_1$ be a $(t_{hpke}, \epsilon_{hpke})$-IND-CPA secure HPKE mechanism, $\mathcal{E}_2$ a $(t_{aead}, \epsilon_{aead})$-IND-CPA secure AEAD[22] scheme, $\mathcal{S}$ a $(t_{sig}, \epsilon_{sig})$-SUF-CMA secure signature scheme and H and F key derivation functions modeled respectively as a (t_H, ϵ_H)-secure pseudorandom function and a (t_F, ϵ_F)-secure dual-PRF.*

Then SUMAC, implemented with cgka *as its admin CGKA,* tmka *as its user MKA and F, H, $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{S}$ as its auxiliary functions, is $(t_{sumac}, q, \epsilon_{sumac})$-A-CGKA secure, with $t_{sumac} \approx t_{cgka} \approx t_{tmka}$ and ϵ_{sumac} s.t.:*

$$\begin{aligned}\epsilon_{sumac} \leq q^2\epsilon_{sig} + \epsilon_{cgka} + \epsilon_{tmka} + q \cdot \big(&2\epsilon_{aead} + (n_u+1)\cdot\epsilon_{hpke} \\ &+ (2n_u - 1 + (n_u-1)(\log_2(n_u)+1))\cdot\epsilon_F \\ &+ (5n_u + \log_2(n_u) + 2)\cdot\epsilon_H\big)\end{aligned} \quad (8)$$

Proof. Let us consider the event E_1 in which the *inject* oracle provides the adversary with the guessing bit b of the security game and the event E_2 in which, conversely, this oracle outputs $\perp$ (intuitively, these events respectively correspond to an impersonation attack and to an attack against the key indistinguishability of SUMAC). As we have $\Pr[E_1] + \Pr[E_2] = 1$, we can study these two events separately.

Our proof is structured as follows:

- in Appendix B.2.1, we determine the advantage of the adversary in winning the security game by forging a commit or a proposal (event E_1).
- Then, in Appendix B.2.2 and Appendix B.2.3, we assess the adversarial advantage against the key indistinguishability of SUMAC in the framework of event E_2, for respectively admin and user operations. To do so, we use an hybrid argument going from experiment $\mathsf{KIND}^0_{\mathsf{sumac}}$ of the A-CGKA security game – which corresponds to the real execution of the protocol and gives the adversary, during the challenge query, the real tuple of keys $(k^0_a, (k^0_{u_\ell})_\ell, k^0_g)$ – to experiment $\mathsf{KIND}^1_{\mathsf{sumac}}$ where the output values – especially the common keys – are randomized.

The overall advantage of an adversary against the KIND security of SUMAC is then bounded by the sum of the advantages found for these two events E_1 and E_2 above. □

[22] Authenticated Encryption scheme with Associated Data.

B.2.1 Event E_1: Impersonation Attack. Let us consider the event E_1 in which the inject oracle, requested upon a group member $m_j \in \{a_j, u_j\}$, outputs a guessing bit and makes the impersonating adversary $\mathcal{A}$ against SUMAC win the A-CGKA security game. Our analysis framework is based on that of [2].

We firstly detail beneath a lemma, whose syntax is adapted from [2], which states that a successful query to the *inject* oracle in the A-CGKA security game of SUMAC corresponds to a valid forgery of a signature key-pair.

Lemma B.3. *Let $\mathcal{A}$ be an impersonating A-CGKA adversary against SUMAC, trying to win the security game by successfully forging a message through the inject oracle. If the answer of that oracle to a query $O^{inject}(m_j \in \{a_j, u_j\}, msg \in \{C_{bdct}, C_{ind}, C_{adm}, C_{usr}, P\}, t^i)$ is a valid $v \neq \perp$, then $\mathcal{A}$ can parse the message $msg = (\widehat{msg}, \sigma_{msg})$ and efficiently derive a public signature key spk s.t.* $\mathbf{SIG.verif}(spk, \sigma_{msg}, \widehat{msg}) = \top$.

Proof. Let us consider a query of an adversary $\mathcal{A}$ to the *inject* oracle $O^{inject}(m_j, msg, t^i)$ of SUMAC in order to forge a message msg w.r.t. a group member m_j at epoch t^i, which results in a valid output $v \neq \perp$. Since the *inject* oracle in the A-CGKA security game (cf. Section 5.1.1) consists in making the group member m_j process the injected signed message msg, then according to the process algorithm of SUMAC (detailed in Figure 18), m_j parses that signed message into a $(\widehat{msg}, \sigma)$ pair (line 2) and verifies the associated signature (line 3) with one of the public signature keys associated with the group members, that were previously broadcast within commit messages (according to lines 2-5 of the **adm-com** algorithm in Figure 18 and line 1 of **usr-com**). Then, only a success to that signature verification leads to a valid output of the **process** algorithm (instead of the failure symbol $\perp$), therefore to the forgery predicate **forge** = 1 and the generation of a valid output $v \neq \perp$ to the inject oracle. □

Therefore, in order to carry out a valid simulation, our security reduction must guess correctly the query in which the forged signature key-pair has been generated and the query of the first successful injection of the adversary $\mathcal{A}$. Consequently, we divide the event E_1 into events $(E_{1,i,j})_{i,j \in [\![1,q]\!]}$, which correspond to a forgery realized at query q_j on a signature-message pair associated with a signature key-pair generated at query q_i. Then, by the union bound, we have:

$$\Pr[\mathrm{E_1}] \leq \sum_{i,j \in [\![1,q]\!]} \Pr[\mathrm{E_{1,i,j}}] \quad (19)$$

$$\mathsf{advKIND}_{E_1}(\mathcal{A}) \leq \sum_{i,j \in [\![1,q]\!]} \mathsf{advKIND}_{E_{1,i,j}}(\mathcal{A}) \quad (20)$$

We now consider that the event $E_{1,i,j}$ holds. Then, the (impersonating) adversary $\mathcal{A}$ against SUMAC can be used by an adversary $\mathcal{B}$

trying to break the SUF-CMA security of the signature scheme $\mathcal{S}$ used in our protocol (cf. Appendix A.2 for the SUF-CMA security game of a signature scheme).

To do so, $\mathcal{B}$ locally simulates the A-CGKA oracles for $\mathcal{A}$, during its $q_i - 1$ first queries. This simulation does not raise any issue, especially since for any query $q_{j'<j}$ to the *inject* oracle, $\mathcal{B}$ simulates for $\mathcal{A}$ by returning an output $v = \bot$.

Let now (spk^*, ssk^*) be the signature key-pair sampled by the challenger of $\mathcal{B}$ in the security game of the signature scheme. At query q_i (which corresponds to a query to either the *propose* or the *commit* oracle, associated with an **add-adm** or a **full-upd-adm** operation – if the inject oracle is requested on an administrator – or an **add-usr** or a **full-upd-usr** otherwise), $\mathcal{B}$ replaces the fresh signature key spk' of the concerned group member by its challenge public key spk^*.

Then, for each query between q_i and q_j, when the signature by ssk' is required, the adversary $\mathcal{B}$ uses its own signature oracle to provide a valid signature associated with the propose or commit messages needed to be signed, and returns that signature to $\mathcal{A}$ in order to keep the simulation going. Similarly, when the public signature key spk' needs to be forwarded, it is replaced by spk^*.

At query q_j made by $\mathcal{A}$ to the *inject* oracle, $\mathcal{B}$ parses the injected signed message $m = (\widehat{m}, \sigma_m)$ and returns to its own challenger the pair $(\widehat{m}, \sigma_m)$. Given the fact that:

- the injection query $q_j > q_i$ is related to the challenge signature key spk^* of $\mathcal{B}$;
- the oracle returns a valid output $v \neq \bot$ (according to the initial assumption $E_{1,i,j}$);

then, according to Lemma B.3, the message-signature pair issued from $\mathcal{A}$'s injection attempt is valid and associated with spk^* (i.e. we have $\mathbf{SIG.verif}(\mathbf{spk}^*, \sigma_\mathbf{m}, \widehat{\mathbf{m}}) = \top$), with a probability $\Pr = \mathsf{advKIND}_{E_{1,i,j}}(\mathcal{A})$. Consequently, in the framework of the event $E_{1,i,j}$, $\mathcal{B}$ wins its signature security game with an advantage ϵ_{sig} at least equal to that of the adversary $\mathcal{A}$ in the A-CGKA security game:

$$\mathsf{advKIND}_{E_{1,i,j}}(\mathcal{A}) \leq \epsilon_{sig} \tag{21}$$

$$\Rightarrow \epsilon_{sumac}^{E_1} = \mathsf{advKIND}_{E_1}(\mathcal{A}) \leq q^2 \epsilon_{sig} \tag{22}$$

We now focus on the event E_2 that corresponds to an attack against the key indistinguishability of SUMAC. Figure 12 represents the instructions carried out during either an admin operation (**add-adm**, **rem-adm**, **upd-adm**) or a user operation (**add-usr**, **rem-usr**, **upd-usr**) of SUMAC[23]. We consequently study separately, in the following parts, the hybrid arguments for each type of these operations. The security of our protocol is determined w.r.t. the cgka-safe predicate of SUMAC.

B.2.2 Event E_2: Randomization of an Admin Operation. Firstly, we study the implications of the cgka-safe predicate of SUMAC, which is based on that of its component CGKA and TMKA (cf. Proposition B.1). The optimal predicate $\mathbf{safe_{cgka\text{-}opt}}$ implies that none of the standard users nor the administrators is compromised – through a current or previous *expose* query that has not been healed with an update – at the challenge epoch $t^* = t^{i+1}$. Given that the group operation leading to that epoch is an admin operation performed on some a_j, that same optimal cgka-safe predicate states that no standard user and no administrator other than a_j was still compromised at the epoch t^i preceding t^*. Moreover, the user keys $I^i_{u_\ell} = I^{i+1}_{u_\ell},\ \ell \in [\![1, n_a]\!] \setminus \{j\}$ do not change between the epochs t^i and t^{i+1} and the optimal predicate $\mathbf{safe_{cgka\text{-}opt}}$ also requires the safety of these keys (no *reveal* query from epoch t^i). Finally, the design of a TMKA (cf. Figure 15) implies that the update of any standard user comes with the update of its direct path in its user tree. Therefore, according to $\mathbf{safe^{tmka}_{cgka\text{-}add}}$, all user trees in the group, except $\mathcal{T}^i_{u_j}$ and $\mathcal{T}^{i+1}_{u_j}$, are entirely safe (i.e. non-compromised) at the epochs t^i and $t^{i+1} = t^*$.

We now proceed to the randomization of the lines in the algorithms of SUMAC, that lead to the common keys evaluated in the security game:

- $\mathbf{G_0}$: This game corresponds to the experiment $\mathsf{KIND}^0_{\mathsf{sumac}}$ for an admin operation.
- $\mathbf{G_1}$: In this game, we replace the output of the admin CGKA cgka by random values, in particular the cgka key I^{i+1}_{cgka} (line 1 of Figure 12, left column). The structure and the content of the commit message are not studied here since we use the admin CGKA as a black-box. Nevertheless, the randomization of the encrypted secret elements is already taken into account in the security bound of that CGKA. Consequently, we can straightly deduce that the advantage ϵ_1 of any adversary $\mathcal{A}_{G_0\text{-}G_1}$ in distinguishing the two games G_0 and G_1 is bounded by that of an adversary $\mathcal{A}'$ against the CGKA security of cgka, considering the respect of the additional cgka-safe predicate of cgka: $\epsilon_1 \leq \epsilon_{cgka}$.
- $\mathbf{G_2}$: We replace in line 4 the input I^{i+1}_g of the AEAD scheme by a random value. The user key $I^i_{u_c}$ used as encryption key being secret due to the respect of the cgka-safe predicate, as stated above, then the advantage of any adversary in distinguishing game 1 from game 2 is bounded by that of an adversary against the CPA security of the AEAD scheme: $\epsilon_2 \leq \epsilon_{aead}$.
- $\mathbf{G_3}$: Similarly, we replace here the input of the HPKE scheme in lines 7 and 10 by random values. a_j's secret key sk^{i+1}_j being fresh, the adversarial advantage in distinguishing this game from the previous one is bounded by the security bound of the HPKE mechanism: $\epsilon_2 \leq 2\epsilon_{hpke}$.
- $\mathbf{G_4}$: We replace the output of the regeneration function from line 11 – conceptualized in the standard model as a dual-PRF (dPRF) and in the ROM as a random oracle, applied to each pair of values of its input sets – by a set of $2n_u - 1$ random coins. This replacement is made possible by the secrecy of the respective inputs $\overline{\mathcal{T}_{u_j}}^{i+1} := \mathbf{derive}(\mathcal{T}^i_{u_p}, \text{“regen”})$ and $\mathcal{R} := \mathbf{derive}(\mathcal{T}^i_{u_c}, \text{“regen”})$ due to the respect of the cgka-safe predicate of SUMAC. The advantage for an adversary trying to distinguish this game from the previous one is thus bounded by the dPRF security bound for each regenerated value: $\epsilon_4 \leq (2n_u - 1) \cdot \epsilon_F$, with ϵ_F the advantage of a dual-PRF adversary against the regeneration function.

[23]The migration operations **upgrade** and **downgrade** are decomposed into their component subroutines and therefore count both as an admin and a user operation.

- $\mathbf{G_5}$: At this final stage, we are able to replace in lines 2, 3, 6, 9 and 12 the output of the derivation function[24], which is formalized in the standard model as a pseudorandom function (PRF) and in the ROM as a random oracle. Considering that these derivations of lines 6 and 9 are applied on entire user trees with $2n_u - 1$ nodes (for binary trees), the overall advantage – in the standard model – of any adversary in distinguishing G_5 from G_4 is bounded by: $\epsilon_5 \le (4n_u+1)\cdot\epsilon_H$, with ϵ_H the advantage of a PRF adversary against the derivation function H.
  This game corresponds to the experiment $\mathsf{KIND}^1_{\text{sumac}}$ of the A-CGKA security game of SUMAC.

B.2.3 Event E_2: Randomization of a User Operation. We proceed similarly for a user operation, as described in Figure 12 (right column).

As the user operation leading to the challenge epoch $t^* = t^{i+1}$ concerns a single standard user u_j, the optimal cgka-safe predicate $\mathbf{safe}_{\textbf{cgka-opt}}$ requires all the other standard users and all the administrators to be non-compromised at the epochs t^i and $t^{i+1} = t^*$. In addition, that predicate states that the admin key $I_a^i = I_a^{i+1}$ is not revealed through a *reveal* query at epoch t^i or t^{i+1}.

- The initial game G_0 corresponds to the experiment $\mathsf{KIND}^0_{\text{sumac}}$ of the A-CGKA security game of SUMAC.
- Game G_1 corresponds to the replacement of the committer's user key $I_{u_c}^{i+1}$ yielded by the **commit** algorithm of the TMKA in line 1, by a random key from the key space. Since the safety predicate of the TMKA is respected, we have: $\epsilon_1 \le \epsilon_{tmka}$.
- In game G_2, we replace the regeneration set $\mathcal{R}$ (of maximum size $\log_2(n_u) + 1$) output by the derivation function in lines 2 and 5 by a random value. This replacement is made possible by the designs of SUMAC and of a TMKA: any user operation, such as the one undergone by standard user u_j from epoch t^i to epoch t^{i+1}, indeed updates the (extended) direct path of that user in the committer's tree. Consequently, distinguishing this game from the previous one can be bounded as follows: $\epsilon_2 \le (\log_2(n_u) + 1)\epsilon_H$.
- In game G_3, the input to the AEAD scheme (line 7) is randomized. The encryption key I_a^i being safe due to the optimal cgka-safe predicate, we can rely on the CPA-security of the AEAD scheme et therefore have: $\epsilon_3 \le \epsilon_{aead}$.
- In game G_4, the input to the HPKE scheme (line 14) is randomized. The encryption is carried out with fresh encryption key-pairs belonging to u_j, which leads to: $\epsilon_4 \le (n_u - 1)\cdot\epsilon_{hpke}$ (since the HPKE encryption algorithm is called in each of the $n_u - 1$ user trees other than the committer's).
- Thanks to the previous randomization of the $\mathcal{R}$ in game G_2, game G_5 replaces the output of the regeneration algorithm in lines 10 and 12, of maximum size $\log_2(n_u) + 1$, by a random value, for each of the $n_u - 1$ calls to that function. The advantage in distinguishing this game from the previous one is: $\epsilon_5 \le (n_u - 1)(\log_2(n_u) + 1)\cdot\epsilon_F$.
- Finally, in game G_6, the n_u outputs of the derivation function used in lines 6 and 15 are also randomized, giving an advantage $\epsilon_6 \le n_u \cdot \epsilon_H$.

Therefore, for an adversary issuing q queries in the A-CGKA security game that lead to q_a admin operations and q_u user operations (s.t. $q_a + q_u \le q$), we bound its advantage as follows[25], considering that the event E_2 is occurring:

$$\begin{aligned}\epsilon_{sumac}^{E_2} \le\ & \epsilon_{cgka} + \epsilon_{tmka} + (q_a + q_u)\cdot\epsilon_{aead} \\ & + (2q_a + q_u(n_u - 1))\cdot\epsilon_{hpke} \\ & + (q_a(2n_u - 1) + q_u(n_u - 1)(\log_2(n_u) + 1))\cdot\epsilon_F \\ & + (q_a(4n_u + 1) + q_u(\log_2(n_u) + n_u + 1))\cdot\epsilon_H \qquad (23) \\ \le\ & \epsilon_{cgka} + \epsilon_{tmka} + q\cdot\big(2\epsilon_{aead} + (n_u + 1)\cdot\epsilon_{hpke} \\ & + (2n_u - 1 + (n_u - 1)(\log_2(n_u) + 1))\cdot\epsilon_F \\ & + (5n_u + \log_2(n_u) + 2)\cdot\epsilon_H\big)\end{aligned}$$

C Correctness Analysis

C.1 Correctness Model

Similarly to the security study, the correctness of SUMAC as an A-CGKA is analyzed in the framework set up by [2, 3.3]. We recall beneath the general ideas of this framework and invite the reader to refer directly to that seminal work (including its Figure 2 that details the correctness game) for details.

Correctness is studied in that work as a game against an adversary $\mathcal{A}$ that performs q queries to some oracles – similar to that of the security game – in order to make the member group evolve over time. Correctness is ensured if all group members that have processed the commit messages related to these queries end up with identical views of the group key(s) and of the group membership (including the membership of its admin and user subgroups).

In each of the oracles, the correctness of the A-CGKA is assessed through "reward" conditions, which make the adversary win the correctness game ($\mathsf{CORR}_{\text{a-cgka}}(\mathcal{A}) = 1$) if at least one of them is true. More specifically, a **check-same-group-state** function is evaluated at each oracle, which checks that for two different states given in argument, their views of the common keys and of the membership of the admin and user groups are identical. Some additional reward conditions are also present in the oracles.

An A-CGKA is said to be $(q, \delta_{a\text{-}cgka})$-correct if, for any adversary $\mathcal{A}$ issuing q oracle queries in the correctness game, we have:

$$\Pr\left[\mathsf{CORR}_{\text{a-cgka}}(\mathcal{A}) = 1\right] \le \delta_{a\text{-}cgka}. \qquad (24)$$

C.2 Correctness of SUMAC

The correctness of SUMAC is based on those of the underlying CGKA and TMKA – themselves assessed through the same correctness game as an A-CGKA [2, 3.3] – and on the worst-case correctness of a PKE recalled in Appendix A.3.2.

[24]Once again, this replacement is possible because the inputs to the derivation function are secret, thanks to the cgka-safe predicate of SUMAC.

[25]The advantages of the CGKA and TMKA protocols are already given considering q adversarial queries.

Admin operation

```
$(I^{i+1}_{cgka}, C_{adm}) \leftarrow$ CGKA.commit()   / $G_0$
$I^{i+1}_a :=$ derive($I^{i+1}_{cgka}$, "path")   / $G_0 - G_4$
$I^{i+1}_g :=$ derive($I^{i+1}_{cgka}$, "key")   / $G_0 - G_4$
$C_{usr} \leftarrow$ AEAD.enc($I^i_{u_c}, I^{i+1}_g$)   / $G_0 - G_1$
if add-adm then :
    $\overline{\mathcal{T}_{u_j}}^{i+1} :=$ derive($\mathcal{T}^i_{u_p}$, "regen")   / $G_0 - G_4$
    $C_{ind} \leftarrow$ HPKE.enc($pk^{i+1}_j, \overline{\mathcal{T}_{u_j}}^{i+1}$)   / $G_0 - G_2$
if add-adm or upd-adm then :
    $\mathcal{R} :=$ derive($\mathcal{T}^i_{u_c}$, "regen")   / $G_0 - G_4$
    $C'_{ind} \leftarrow$ HPKE.enc($pk^{i+1}_j, \mathcal{R}$)   / $G_0 - G_2$
    $\mathcal{T}^{i+1}_{u_j} :=$ regen($\overline{\mathcal{T}_{u_j}}^{i+1}, \mathcal{R}$)   / $G_0 - G_3$
    $I^{i+1}_{u_j} :=$ derive($\mathcal{T}^{i+1}_{u_j}$.root, "key")   / $G_0 - G_4$
```

User operation

```
$(I^{i+1}_{u_c}, C_{usr}) \leftarrow$ TMKA.commit()   / $G_0$
if upd-usr then :
    $\mathcal{R} :=$ derive($\mathcal{P}_x(u_j, \mathcal{T}^{i+1}_{u_c})$, "regen")   / $G_0 - G_1$
else :
    $\mathcal{R} :=$ derive($\mathcal{P}(u_j, \mathcal{T}^{i+1}_{u_c})$, "regen")   / $G_0 - G_1$
$I^{i+1}_g :=$ derive($\mathcal{R}$.root, "key")   / $G_0 - G_5$
$C_{adm} \leftarrow$ AEAD.enc($I^i_a, \mathcal{R}$)   / $G_0 - G_2$
for $\ell \neq c$ do :
    if upd-usr then :
        $\mathcal{P}_x(u_j, \mathcal{T}^{i+1}_{u_\ell}) :=$ regen($\mathcal{P}_x(u_j, \mathcal{T}^i_{u_\ell}), \mathcal{R}$)   / $G_0 - G_4$
    else :
        $\mathcal{P}(u_j, \mathcal{T}^{i+1}_{u_\ell}) :=$ regen($\mathcal{P}(u_j, \mathcal{T}^i_{u_\ell}), \mathcal{R}$)   / $G_0 - G_4$
        if add-usr then :
            $C_{ind} \leftarrow$ HPKE.enc($pk^{i+x}_j, \mathcal{P}_x(u_j, \mathcal{T}^{i+x}_{u_\ell})$)   / $G_0 - G_3$
    $I^{i+1}_{u_\ell} :=$ derive($\mathcal{T}^{i+1}_{u_\ell}$.root, "key")   / $G_0 - G_5$
```

Figure 12: Admin and user group operation in SUMAC, in the experiment $\mathsf{KIND}^0_{\mathsf{sumac}}$ (real world) of the A-CGKA security game. In the experiment $\mathsf{KIND}^1_{\mathsf{sumac}}$ (random world), all these lines leading to the challenge values need to be randomized.

Theorem C.1 (Correctness of SUMAC). *Let SUMAC be implemented with a (q, δ_{cgka})-correct admin CGKA* cgka*, a (q, δ_{tmka})-correct TMKA* tmka*, an HPKE encryption scheme relying on a δ_{pke}-correct PKE and a perfectly correct AEAD mechanism. Then, w.r.t. an adversary $\mathcal{A}$ against the correctness of the A-CGKA and issuing, in the correctness game, q_a queries to the commit oracle related to admin operations and q_u queries to that oracle related to user operations s.t. $q_a + q_u \leq q$, SUMAC is a $(q, \delta_{a\text{-}cgka})$-correct A-CGKA protocol, with:*

$$\delta_{a\text{-}cgka} = n_a \cdot \delta_{cgka} + n_u \cdot \delta_{tmka} + (2q_a + n_a q_u) \cdot \delta_{pke} \qquad (25)$$

Proof. We now prove the correctness of SUMAC w.r.t. the correctness game mentioned above, based on the correctness of the underlying primitives. To do so, we analyze the oracles at disposal to the adversary $\mathcal{A}$, in order to assess if their use in SUMAC could break the correctness property.

According to the definition of an A-CGKA, only a *commit* or a *create-group* oracle may change either the group membership or the common key[26], knowing that the create-group operation is itself a wrapper around a commit. All the information comprised in a proposal message output by the *propose* oracle is recalled in the associated commit message. The changes related to that proposal and that commit are then taken into account by the other group members *via* the process operation, which is part of the *deliver* oracle.

Therefore, all misinterpretation by a processing group member – which leads to breaking the correctness of the protocol – comes from the processing of a commit. The only exception to that statement with SUMAC is for an **add-adm** operation, where an encrypted temporary user tree is sent directly by the proposing administrator to the new one, without being recalled by the committer. Therefore, a bad processing of that specific **add-adm** proposal by the new admin may also break the correctness of SUMAC.

Therefore, the correctness of SUMAC is assessed by studying:

- the processing of an **add-adm** proposal;
- the processing of a commit related to any operation.

The inspection of SUMAC algorithms of **prop-process**, **adm-com-process** and **usr-com-process** (cf. Figure 18, p.36) shows that the correctness of these processing operations relies on the processing algorithms of the underlying CGKA and TMKA as well as on the public-key decryption algorithm from the HPKE scheme – since we consider perfectly correct secret-key encryption schemes, the symmetric part of the HPKE mechanism as well as the AEAD construction are not taken into account in this correctness analysis.

- **add-adm proposal:** the processing of such a proposal corresponds to a single HPKE decryption (line 1 of **prop-process** in Figure 18) by a single administrator.
- **Commit for an admin operation:** in this type of operation, all administrators (n_a for an **add-adm** or a **upd-adm**, $n_a - 1$ for a **rem-adm**) run the process algorithm of the admin CGKA cgka (line 4 of **adm-com-process**). In addition, in a **upd-adm** or in an **add-adm**, the updated/new administrator uses the decryption algorithm of the HPKE scheme (line 8).
  On the other hand, the standard users do not run any algorithm that influences the correctness of the protocol.
- **Commit for a user operation:** in a **upd-usr** or a **rem-usr**, all standard users (n_u and $n_u - 1$, respectively) run the process algorithm of the user TMKA (**usr-com-process**,

[26]Which is, in the case of SUMAC, the tuple $(I_a, (I_{u_\ell})_{\ell=1}^{n_a}, I_g)$ composed of the admin key, all the user keys and the group key.

lines 8 and 20). In an **add-usr**, the $n_u - 1$ already existing users run the TMKA process algorithm (line 8), while the new user runs an HPKE decryption for each of the n_a TMKA trees (line 17).
The administrators do not use any algorithm with an impact on the correctness of SUMAC.

Consequently, for an adversary that queries q_a times the *deliver* oracle of SUMAC in the correctness game after an admin operation and q_u times after a user operation (with the number q of queries s.t. $q \geq q_a + q_u$), the number x of calls to the CGKA, TMKA and HPKE algorithms with an impact on the correctness of the protocol is bounded as follows:

$$\begin{aligned} x &\leq q_a(n_a \cdot \text{CGKA.process} + 2 \cdot \text{HPKE.dec}) \\ &\qquad + q_u(n_u \cdot \text{TMKA.process} + n_a \cdot \text{HPKE.dec}) \\ &\leq n_a q \cdot \text{CGKA.process} + n_u q \cdot \text{TMKA.process} \\ &\qquad + (2q_a + n_a q_u) \cdot \text{HPKE.dec} \end{aligned} \tag{26}$$

As the CGKA and TMKA protocols are respectively δ_{cgka} and δ_{tmka}-correct w.r.t. an adversary issuing q requests, whereas the PKE scheme within the HPKE construction is δ_{pke}-correct w.r.t. to a single decryption query, we have:

$$\delta_{a\text{-}cgka} \leq n_a \cdot \delta_{cgka} + n_u \cdot \delta_{tmka} + (2q_a + n_a q_u) \cdot \delta_{pke} \tag{27}$$

which concludes the proof. □

D Details on the Performances of SUMAC

We detail in this section the analyses of the storage, computational and communication costs whose results are displayed in Section 6.

D.1 Storage Cost

We consider here simplified protocols where the state of a node simply consists in the keying material used by the protocol (encryption and signature key-pairs and group key). In this setting, the storage cost of the main group key agreement schemes can be approximated as follows:

D.1.1 TMKA. Since a TMKA relies on secret-key cryptography, a node in the Ratchet Tree only has a private state, comprising a secret encryption key (other keys derived from the leaf or path secret used to generate the encryption key may also be used for other purpose (derivation...), but are not considered here since their existence highly depends on the precise TMKA protocol). All these values have a size of $|sec|$, depending on the security parameter.

A standard user u_j stores the (private) states of its leaf and of all the nodes in its direct path (storage cost of $(h+1)\,|sec|$) and additionally records its own signature key-pair and the public signature key of the group manager. In parallel, the group manager must maintain a complete view of the whole Ratchet Tree ($2n-1$ encryption keys, one group key derived from the root and n public signature keys for the standard users) and also has its own signature key-pair. The storage cost of a TMKA in a tree of height $h = \log(n)$ is therefore of:

Table 1: Compared storage costs per user of DGS-TreeKEM, TreeKEM and SUMAC, in the classical and post-quantum frameworks. This table underlines the high performances of our protocol, especially in the PQ framework – due to the use of secret-key cryptography – and for the standard users, which no longer need to record a complete view of the member group.

Setting	Number of		Storage Cost per User (kB)			Gain
	std users	admins	DGS-TK	(IAS-) TreeKEM	SUMAC	SUMAC / TK (%)
Administrator						
Class.	16	4	2.4	2.0	1.5	25.6
	256	16	27.6	26.4	18.1	31.3
	65,536	256	6,333	6,317	4,219	33.2
PQ	16	4	70.4	56.1	13.7	75.6
	256	16	717	670	70.3	89.5
	65,536	256	158,562	157,937	6,923	95.6
Standard user						
Class.	16	4	2.0	2.0	0.5	74.1
	256	16	26.4	26.4	2.9	89.0
	65,536	256	6,317	6,317	78	98.8
PQ	16	4	56.1	56.1	0.9	98.5
	256	16	670	670	5.2	99.2
	65,536	256	157,937	157,937	148	99.9

$$\left|\delta_j^{tmka}\right| = 2\,|spk| + |ssk| + (h+1)\,|sec| \tag{28}$$

$$\left|\gamma^{tmka}\right| = (n+1)|spk| + |ssk| + 2n\,|sec| \tag{29}$$

D.1.2 TreeKEM. In a TreeKEM-like CGKA, each user u_j stores in its public state $pub(\gamma_j)$ a complete view of the Ratchet Tree, which comprises the public state of all nodes in that tree. Moreover, it stores in its private state $priv(\gamma_j)$ the private state of its leaf and of all the nodes in its direct path.

In our simplified setting, the public (resp. private) state of an internal node in TreeKEM, root excepted, simply consists in a public (resp. private) encryption key, that of a leaf includes public (resp. private) encryption and signature keys and that of the tree root is empty (resp. comprises the group key of size $|sec|$). In this framework, the storage cost of user u_j in a tree of height $h = \log(n)$ is:

$$\begin{aligned} \left|\gamma_j^{treekem}\right| &= \left|pub(\gamma_j)\right| + \left|priv(\gamma_j)\right| \\ \left|pub(\gamma_j)\right| &= |\text{RT}| = (2n-2)|pk| + n|spk| \\ \left|priv(\gamma_j)\right| &= h\,|sk| + |ssk| + |sec| \end{aligned} \tag{30}$$

D.1.3 SUMAC. In SUMAC, an administrator stores an entire TMKA Ratchet Tree (as a regular group manager in a TMKA protocol) as well as a CGKA user state in the admin CGKA, plus a group key of size $|sec|$. When implemented with a TreeKEM-like CGKA, an administrator storage cost in SUMAC, in a group with n_u standard users and n_a administrators, in an admin Ratchet Tree of height $h_a = \log(n_a)$, is:

$$\begin{aligned}\left|\gamma_j^{sumac}\right| &= \left|\gamma^{tmka}\right| + \left|\delta_j^{treekem}\right| + |sec| \\ &= (n_u+1)|spk| + |ssk| + (2n_u)\,|sec| \\ &\quad + (2n_a-2)|pk| + n_a|spk| + h_a\,|sk| + |ssk| + |sec| \\ &= (2n_a-2)|pk| + (n_u+n_a+1)|spk| + h_a\,|sk| \\ &\quad + 2\,|ssk| + (2n_u+1)\,|sec| \qquad (31)\end{aligned}$$

A standard user in SUMAC needs to store the state of a standard user in a TMKA, for n_a TMKA trees (one for each administrator), along with the group key. However, it only generates a single encryption key-pair for all the TMKA trees, which reduces the storage cost.

$$\begin{aligned}\left|\delta_j^{sumac}\right| &= n_a\left|\delta_j^{tmka}\right| - (n_a-1)(|spk|+|ssk|) + |sec| \\ &= (n_a+1)|spk| + |ssk| + (n_a(h_u+1)+1)\,|sec| \qquad (32)\end{aligned}$$

D.1.4 IAS and DGS A-CGKAs. Straightforwardly, IAS has the same storage cost as the underlying user CGKA provided that the latter already uses signatures to authenticate its users.

Regarding DGS-TK (implemented using TreeKEM both as its user and admin CGKAs), the storage cost of a standard user is identical to that of the user CGKA (in a tree bearing $n_t = n_a + n_u$ leaves, of height $h_t = \log(n_t)$). An administrator, on the other hand, needs to record not only its state from the user CGKA but also that from the admin CGKA (except for the signature keys, that are already recorded in the user CGKA state). Moreover, instead of storing the group key as in the user CGKA, the administrator stores the admin signature key-pair generated from the admin common secret.

$$\begin{aligned}\left|\gamma_j^{dgs\text{-}tk}\right| &= (2(n_t+n_a)-2)|pk| + (n_t+1)|spk| + (h_t+h_a)\,|sk| \\ &\quad + 2\,|ssk| + |sec| \qquad (33)\end{aligned}$$

$$\left|\delta_j^{dgs\text{-}tk}\right| = (2n_t-2)|pk| + n_t|spk| + h_t\,|sk| + |ssk| + |sec| \qquad (34)$$

D.1.5 Results. The storage cost of the studied protocols are detailed, for the same parameters as with the communication cost, in Table 1.

D.2 Computational Cost

In addition to our main studies on the communication and storage costs, we have started an on-going experiment to compare the computation costs of TreeKEM and SUMAC, using our implementation of SUMAC (from https://anonymous.4open.science/r/SUMAC-5F5A) on a laptop with a processor Intel 12th Gen Intel(R) Core(TM) i5-1245U at 4.4 GHz and 16 GB of RAM.

As any slight change in the implementations of these protocols may change drastically their performances, it is impossible to run directly the existing implementations of MLS (for instance the open-source implementation of OpenMLS) and we need to use our own version of TreeKEM that we have implemented for the admin CGKA of SUMAC.

The preliminary results of our survey, limited for the moment to the computational cost of the generation of a commit, for the **add-user** operation in the classical setting, are stated in Table 2. They show that with that operation, SUMAC does not suffer from any computational overhead w.r.t. TreeKEM and on the contrary appears much lighter than that CGKA. These early and partial results seem consistent with the architectures of these two protocols. Indeed, the cost of a commit grows with the number of encryptions to perform in order to transmit the group key to the entire group. Both TreeKEM and SUMAC have an asymptotic logarithmic communication cost, hence an asymptotically equal number of ciphertexts to generate. However:

- The centralized setting of a TMKA allows user trees in SUMAC to remain more structured than the Ratchet Tree in TreeKEM (due to the mechanism of *unmerged leaves* used by the latter at each group operation, whereas SUMAC straightly updates the paths of the concerned users instead of blanking them). Therefore, the number of ciphertexts for a commit in SUMAC is much closer to the logarithmic lower bound than it is in TreeKEM.
- Moreover, SUMAC uses a secret-key encryption scheme to generate these ciphertexts, computationally more efficient than classical public-key encryption mechanisms such as the ones based on elliptic curves.

A full study of the computational costs will be included in the final version of this paper.

D.3 Communication Cost

D.3.1 Message Distribution. Our metric considers the average communication cost per user of the protocol, both upwards (from the emitter of the message to the Delivery Service (DS), i.e. the Central Server) and downwards (from the DS to the recipient(s) of the message). The way the DS distributes messages within the group depends on the functionalities offered by the Delivery Service (cf. Figure 13). Some works have considered an advanced server capable of distributing to the relevant recipient pieces of broadcast messages; this setting is called *server-aided* and has led to modifying the design of the group key exchange protocol in order to take advantage of this advanced functionality (cf. [13] for instance). Nevertheless, most works, and in particular the MLS standard [7], consider a basic DS in a framework named *broadcast-only*, that is solely capable of relaying broadcast messages to the whole group. Despite its name, this basic server can also distribute unicast messages to a single recipient.

The design of an A-CGKA, where administrators and standard users belong to separate groups $\mathcal{G}_a$ and $\mathcal{G}_u$, also allows a distribution setting that is an in-between of the broadcast-only and the server-aided settings: we call it the *group-distribution* framework. In that case, the DS knows which group members belong to which subgroup of the member group $\mathcal{G}$ and therefore distinguishes broadcast messages sent to all group members from the ones addressed either to the administrators or to the standard users[27]. This setting, as it stands, unfortunately causes privacy issues since the DS knows the identity of the administrators in the member group; consequently, we have mainly studied the performances of our protocol in the broadcast-only setting. Nevertheless, we provide below the general formulas of the communication costs, that fit both the broadcast-only and the group-distribution settings.

[27]The server is also able, in this group-distribution setting, to deliver unicast messages.

Table 2: Compared computational costs of a commit generation in SUMAC and TreeKEM, in the classical framework, for an add-user operation.

Operation	Number of std users	Computational cost (ms) of	
		TreeKEM	SUMAC
add-usr	16	2	0.50
	256	11	0.77
	1,024	117	1.06

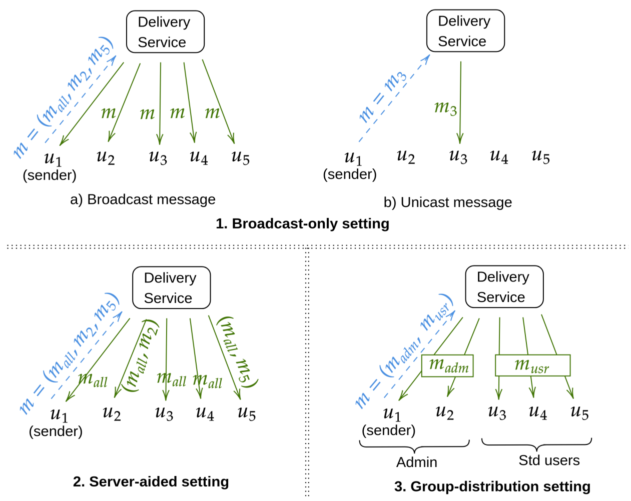

Figure 13: Message distribution modes in a Secure Group Messaging protocol. The setting historically considered by default is the *broadcast-only* one, where the Delivery Service ensured by the Central Server is only capable of forwarding collective messages to the whole group, while being also able of transmitting unicast messages to a single recipient. In the most sophisticated *server-aider*, each part of a broadcast message is sent only to the relevant recipient, enhancing the delivery efficiency. Our *group-distribution* setting is a trade-off between these two modes, which enables the Delivery Service to forward broadcast messages to subgroups of the member group, in our case $\mathcal{G}_a$ and $\mathcal{G}_u$.

To any message m sent within SUMAC, we associate the following distribution coefficients:

- message to/from the Authentication Service (AS): the coefficient $g_{as} := \frac{1}{n_t}$;
- unicast message: the coefficient $g_{ind} := \frac{2}{n_t}$;
- message sent to the admin group $\mathcal{G}_a$ or to the user group $\mathcal{G}_u$, of resp. cardinalities n_a and n_u: $g_a := \frac{n_a+1}{n_t}$ / $g_u := \frac{n_u+1}{n_t}$;
- message broadcast to the member group $\mathcal{G}$: $g := \frac{n_t+1}{n_t}$.

Then, in the broadcast-only setting, we have: $g_a = g_u := g$, as any collective message is broadcast to all group members.

D.3.2 Notations. In addition to the general notations of Section 2.1, we detail below some notations used in our performance analysis. We recall notably that $[\cdot]$ denotes optional fields or lines of algorithm that may be left empty.

In this section, we denote the size (i.e. the bit-length) of secret and public encryption keys by $|sk|$ and $|pk|$; that of a signature and of a signature key-pair are noted $|\sigma|$, $|spk|$ and $|ssk|$. The size of a group member's index is given by $|id|$. The size of the credentials used to authenticate group members – that comprise their public signature key – is given by $|spk|$.

Finally, $|ct^{aead}(x)|$ and $|ct^{hpke}(x)|$ corresponds to the size of the ciphertext associated with a plaintext composed of x secrets, each respecting the security parameter (therefore, in our case, of 32 byte-long each), for either an AEAD or an HPKE encryption.

D.3.3 Proposals and Signatures. The communication costs assessed in this part correspond to that of a SGM protocol operating within the separate-operations paradigm, where each group operation is performed independently of the other. In practice, most recent SGM protocols use the propose & commit framework, that groups together several operations and thus permits to save some bandwidth. However, this paradigm makes it more difficult to compare several protocols, since any combination of proposals can be emitted during a group lifetime; we could compare some of these combinations but the arbitrary choice of the selected sets would make the comparison less objective as the separate-operations setting.

We consequently consider operations carried out by a committer on its own initiative, without any previous proposal. The only exception to this is the **add-admin** operation of SUMAC, that must comprise a preliminary proposal where a temporary user tree is given to the joining administrator.

As in regular CGKAs, all handshake messages exchanged between group members (even those sent by standard users) are signed in order to prevent any user impersonation. The rotation of the group members' signature keys is independent of that of the encryption keys; the underlying idea is that the private signature keys can be protected in tamper-proof modules and are therefore harder to corrupt than the other secrets from a member's private state. In order to remain generic, our algorithms for a TMKA and SUMAC (Figures 14, 15, 17 and 18) consider two types of update operations: in the partial one **part-upd**, only the encryption key-pair is refreshed (this corresponds to the standard update in TreeKEM) whereas in a full-update **full-upd**, both the encryption and the signature key-pairs are renewed.

D.3.4 Key Delivery. The key delivery of group members to the group administrators is carried out by two separate mechanisms.

- The first one corresponds to the **add** of a new group member. In its **init** algorithm (**register-keys** auxiliary function), this one locally generates new encryption and signature key-pairs and push them to the Authentication Service (AS), which is a Public Key Infrastructure (PKI), hosted by the central server and that we assume honest and incorruptible. This message is not signed by the emitter. Subsequently, when an administrator needs to get that member's keys, it demands these keys to the AS through the **get-keys** function that consists in sending to the AS a signed request regarding the group member, to which the AS responds with a signed message comprising the requested keys.
- On the other hand, when a group member refreshes its keys after an **update** operation, its new public key(s) may

be transmitted to the administrators in the proposal message that initiated that operation. However, when the group operation is initiated without any proposal – which is precisely the case considered in our performance analysis –, the key delivery process is modeled with another auxiliary function named **request-keys**. In that sub-routine, the administrator performing the commit directly contacts in unicast the relevant group member to request its new key(s) and the latter answers with a signed message with the requested key(s). These functions have the following communication cost:

$$|\mathbf{regist}(\mathrm{pk}, \mathrm{spk})| = g_{as}(|pk| + |spk|) \tag{35}$$
$$|\mathbf{get}(\mathrm{pk},\ [\mathrm{spk}])| = g_{as}(|pk|\ [+|spk|] + 2|\sigma|) \tag{36}$$
$$|\mathbf{req}(\mathrm{pk}, [\mathrm{spk}])| = g_{ind}(|pk|\ [+|spk|] + 2|\sigma|) \tag{37}$$

We are now able to compute the communication costs of the protocols we want to compare : the TMKA and the TreeKEM-like CGKA that are both used by SUMAC as sub-routines, SUMAC itself, as well as the two administrated-CGKAs from [2] applied on TreeKEM: IAS-TreeKEM and DGS-TreeKEM.

D.3.5 Communication Cost of a TMKA. The exchanges between group members that occur during the various group operations of a TMKA are highlighted in blue in Figures 14, 15 and 16. They permit to compute the communication cost associated with that protocol.

The communication cost of a TMKA, in a user tree of height h_u, is the following:

$$\begin{aligned} c^{tmka}_{part\text{-}upd} &= g_{ind}(|\mathbf{req}(pk)| + |C_{ind}|) + g_u\,|C_{usr}| \\ &= g_u(h_u|ct^{aead}(1)| + |\sigma|) \\ &\quad + g_{ind}(|pk| + |ct^{hpke}(1)| + 3|\sigma|) \end{aligned} \tag{38}$$

$$\begin{aligned} c^{tmka}_{full\text{-}upd} &= g_{ind}(|\mathbf{req}(pk, spk)| + |C_{ind}|) + g_u\,|C_{usr}| \\ &= c_{part\text{-}upd} + g_{ind}|spk| \\ &= g_u(h_u|ct^{aead}(1)| + |\sigma|) + g_{ind}(|pk| + |spk| \\ &\quad + |ct^{hpke}(1)| + 3|\sigma|) \end{aligned} \tag{39}$$

$$\begin{aligned} c^{tmka}_{add} &= g_{as}(|\mathbf{regist}(pk, spk)| + |\mathbf{get}(pk, spk)|) \\ &\quad + g_{ind}\,|C_{ind}| + g_u\,|C_{usr}| \\ &= g_u(h_u|ct^{aead}(1)| + |\sigma|) + g_{ind}(|ct^{hpke}(1)| + |\sigma|) \\ &\quad + g_{as}(2|pk| + 2|spk| + 2|\sigma|) \end{aligned} \tag{40}$$

$$c^{tmka}_{rem} = g_u\,|C_{usr}| = g_u(h_u|ct^{aead}(1)| + |\sigma|) \tag{41}$$

D.3.6 Communication Cost of a TreeKEM-like CGKA. Similarly, the group operations of a TreeKEM-like CGKA, performed in a Ratchet Tree of height h, induce the costs beneath. We consider here commits realized without any preliminary proposal (as in the following analysis for SUMAC). When fresh encryption (and possibly signature) material is required, the exchanges are therefore realized during the commit trough the unicast communication of the **register-keys**, **get-keys** and **request-keys** auxiliary functions[28].

For all these operations, a path update of the committer is carried out, which implies to broadcast to all group members (since they all have the same status), within the commit message, the h updated public encryption keys of the committer and of its path (except for the tree root, that has no public key) and the h encrypted path secrets from that user's direct path.

Finally, in an **add** operation, the arriving user is provided (in unicast) with a copy of the Ratchet Tree, mainly composed of the encryption public keys of every node in that tree (root excepted) and the public signature keys of the existing users: $|\mathrm{RT}| = (2n - 2)|pk| + n|spk|$.

$$\begin{aligned} c^{treekem}_{part\text{-}upd} &= |\mathbf{req}(pk)| + g((h+1)|pk| + h|ct^{hpke}(1)| + |\sigma|) \\ &= g((h+1)|pk| + h|ct^{hpke}(1)| + |\sigma|) + g_{ind}(|pk| + 2|\sigma|) \end{aligned} \tag{42}$$

$$\begin{aligned} c^{treekem}_{full\text{-}upd} &= |\mathbf{req}(pk, spk)| + g((h+1)|pk| + |spk| \\ &\quad + h|ct^{hpke}(1)| + |\sigma|) \\ &= g((h+1)|pk| + |spk| + h|ct^{hpke}(1)| + |\sigma|) \\ &\quad + g_{ind}(|pk| + |spk| + 2|\sigma|) \end{aligned} \tag{43}$$

$$\begin{aligned} c^{treekem}_{add} &= |\mathbf{regist}(pk, spk)| + |\mathbf{get}(pk, spk)| + g((h+1)|pk| \\ &\quad + |spk| + h|ct^{hpke}(1)| + |\sigma|) + g_{ind}\,|\mathrm{RT}| \\ &= g((h+1)|pk| + |spk| + h|ct^{hpke}(1)| + |\sigma|) \\ &\quad + g_{ind}(|\mathrm{RT}|) + g_{as}(2|pk| + 2|spk| + 2|\sigma|) \end{aligned} \tag{44}$$

$$c^{treekem}_{rem} = g(h(|pk| + |ct^{hpke}(1)|) + |\sigma|) \tag{45}$$

D.3.7 Communication Cost of SUMAC. We detail in Table 3 the generic communication costs associated with each group operation in SUMAC. We note that these costs notably depend on the underlying TMKA and CGKA protocols. The costs depicted in our analysis are based on a TreeKEM-like admin CGKA. We underline that when the number of administrators is low, it may appear easier and even more efficient to use a more naive CGKA for these administrators, such as the Pairwise protocol.

D.3.8 Communication Costs of DGS-TreeKEM and IAS-TreeKEM. IAS (Individual Admin Signature) and DGS (Dynamic Group Signature) are two plugins developed in [2], which turn any CGKA into an A-CGKA. The high-level idea of these mechanisms is given below (cf. that paper to dive into more details).

[28] In practice, in TreeKEM, this fresh material is provided by the concerned user through a proposal. However, for the fairness of our study, we did not consider this process, as proposals are broadcast and consequently would have increased the communication cost of that CGKA.

IAS. In this protocol, each group member in the CGKA has an (up-to-date) list of the administrators in the member group, along with their associated public signature keys. Every proposal related to an administration task must be signed by one of the administrators, except for self-removals that can be signed by a fresh signature key of the concerned users. The committer – which must be an administrator – checks that the proposals are correctly signed and then broadcasts a control message T_{ias} and, if needed, a Welcome message W_{ias} based on these of the underlying CGKA, respectively additionally comprising its new public signature key spk'_c and the updated list of administrators adminList.

For the sake of the performance comparison with SUMAC, we slightly modify IAS as follows:

- We do not take into account the administrators list sent to during the commit (since it only comprises the IDs of the administrators, but not their signature public keys that are already transmitted *via* the regular CGKA, in the copy of the Ratchet Tree that is part of the CGKA welcome message).
- Contrary to [2], the CGKA upon which IAS relies in our study signs all its handshake messages, and all its group members thus have signature keys that are regularly updated – similarly to TreeKEM. The refreshment of an administrator's signature key during an **update-admin** operation is therefore already carried out by the CGKA (considering that every admin undergoes a *full* update). The only additional signature operation in IAS, compared to the underlying CGKA, is the automatic signature refreshment of the committer.
- A welcome message W in the original IAS is sent together with the associated control message T, which permits to sign only one message but that does not take advantage of the unicast capability of the delivery service. Consequently, for a fair performance comparison with SUMAC, we consider a variation of IAS where the control message T and the welcome message W are signed and sent separately (the former in broadcast, the latter in unicast), which allows to send the large welcome message to a single recipient instead of to the whole group, therefore greatly decreasing the bandwidth of the protocol, at the cost of only an extra signature.

The migration operations of **upgrade-user** and **downgrade-admin** with IAS simply imply to communicate to the whole group the change of membership in the admin group. This implies one signed broadcast commit message, along with the credentials corresponding to the refreshed signature key of the committer. Therefore, the communication cost of the modified IAS protocol, using a black-box CGKA, is the following, with $op \in \{\textbf{add}, \textbf{rem}, \textbf{upd}\}$ (either for admins or standard users) and $mig \in \{\textbf{upgrade}, \textbf{downgrade}\}$:

$$c^{ias}_{op} = c^{cgka}_{op} + g|spk| \tag{46}$$

$$c^{ias}_{mig} = g(|spk| + |\sigma|) \tag{47}$$

DGS. In this protocol, the administrators use a common administration signature key to authenticate the administration messages. This admin key needs regular rotation, especially due to the changes in the admin group membership. A dedicated admin CGKA (called cgka) takes care of updating this unique admin signature key, in addition to the regular user CGKA (called simply cgka). Consequently, an administrative commit (that concerns an administrator) implies to run both CGKAs (with the same operation) and to broadcast the new admin signature public key. By contrast, a regular commit does not refresh the admin signature key-pair – contrary to IAS – and consequently has the same communication cost as the user CGKA. Finally, the migration operations of **upgrade-user** and **downgrade-admin** require to add or remove a group member to or from the admin CGKA cgka, which updates the admin signature key-pair (and implies to transmit, once again, the signature public key to the whole group).

Thus, the communication cost of DGS is as follows, with *adm-op* and *usr-op* denote respectively the **add**, **remove** and **update** operations carried out on administrators and standard users.

$$c^{dgs}_{adm\text{-}op} = c^{\mathrm{cgka}}_{adm\text{-}op} + c^{\mathrm{cgka}}_{adm\text{-}op} + |spk| \tag{48}$$

$$c^{dgs}_{usr\text{-}op} = c^{\mathrm{cgka}}_{usr\text{-}op} \tag{49}$$

$$c^{dgs}_{upgrade} = c^{\mathrm{cgka}}_{add} + |spk| \tag{50}$$

$$c^{dgs}_{downgrade} = c^{\mathrm{cgka}}_{rem} + |spk| \tag{51}$$

D.3.9 Results. The communication cost of these protocols, assessed for specific parameters (group size, classical or post-quantum framework, in the *broadcast-only* setting), is depicted in Table 4 below.

E Algorithms of a TMKA and SUMAC

We provide below the pseudocode description and the algorithms of the group operations in a TMKA (Figures 14 and 15) and in SUMAC (Figures 17 and 18), along with the auxiliary functions used by both these protocols (Figure 16).

The term **send**$(x \longrightarrow y, msg)$ means that sender x transmits to recipient y (which can be itself either a group or a single group member) the message msg.

F Large Size Figures

We put beneath the figures related to a TMKA (Figure 2 from Section 3.1) and to SUMAC (Figure 1 from Section 1 and Figures 3, 4, 5, 6, 7 and 8 from Section 4.3) in larger sizes than in the main body, so as to remain readable even after being printed.

```
add $([u_p,]\ u_j, \mathcal{T}^i)$
    / Add of user $u_j$ by $gm$, possibly proposed by $u_p$
    / $\boxed{u_j}$: Initialization of its state
1 : $\delta_j^{i+1} \leftarrow \mathbf{init}(u_j)$,    $pk_j^{i+1} := \delta_j^{i+1}.\mathsf{pk}$
    / $\boxed{u_p}$ (possibly $u_j$): potential add proposal
2 : if $u_p \neq u_j$ then :
3 :     $\big[(\delta_p^{i+1}, P = (\mathbf{add}(u_j), \sigma_p))$
                $\leftarrow \mathbf{propose}(\delta_p^i, \mathsf{gid}, \mathbf{add}(u_j))\big]$
4 : else :
5 :     $\big[(\delta_j^{i+1}, P = (\mathbf{add}(u_j), pk_j^{i+1}, \sigma_j))$
                $\leftarrow \mathbf{propose}(\delta_j^i, \mathsf{gid}, \mathbf{add}(u_j))\big]$
6 : $\big[\mathbf{send}(u_p \longrightarrow gm, P)\big]$
    / $\boxed{gm}$: commit
7 : $(\gamma^{i+1}, I^{i+1}, (C_{usr}, C_{ind})) \leftarrow \mathbf{commit}(\gamma^i, \mathsf{gid}, P/\mathbf{add}(u_j))$
8 : $\mathbf{send}(gm \longrightarrow \mathcal{G}_u, C_{usr})$
9 : $\mathbf{send}(gm \longrightarrow u_j, C_{ind})$
```

```
remove $([u_p,]\ u_j, \mathcal{T}^i)$
    / Removal of user $u_j$ by $gm$, possibly proposed by $u_p$
    / $\boxed{u_p}$: potential remove proposal
1 : $\big[(\delta_p^{i+1}, P = (\mathbf{rem}(u_j))) \leftarrow \mathbf{propose}(\delta_p^i, \mathsf{gid}, \mathbf{rem}(u_j))\big]$
2 : $\big[\mathbf{send}(u_p \longrightarrow gm, P)\big]$
    / $\boxed{gm}$: commit
3 : $(\gamma^{i+1}, I^{i+1}, C_{usr}) \leftarrow \mathbf{commit}(\gamma^i, \mathsf{gid}, P/\mathbf{rem}(u_j))$
4 : $\mathbf{send}(gm \longrightarrow \mathcal{G}_u, C_{usr})$
```

```
update $(u_j, type \in \{\text{“full”}, \text{“part”}\}, \mathcal{T}^i)$
    / Update of user $u_j$ by $gm$, possibly self-proposed
    / $\boxed{u_j}$: potential self-update proposal
1 : $\big[(\delta_j^{i+1}, P = (\mathbf{upd}(u_j, type), pk_j^{i+1}, \sigma_j))$
                $\leftarrow \mathbf{propose}(\delta_j^i, \mathsf{gid}, \mathbf{upd}(u_j, type))\big]$
2 : $\big[\mathbf{send}(u_j \longrightarrow gm, P)\big]$
    / $\boxed{gm}$: commit
3 : $(\gamma^{i+1}, I^{i+1}, (C_{usr}, C_{ind})) \leftarrow \mathbf{commit}(\gamma^i, \mathsf{gid}, P/\mathbf{upd}(u_j, type))$
4 : $\mathbf{send}(gm \longrightarrow \mathcal{G}_u, C_{usr})$
5 : $\mathbf{send}(gm \longrightarrow u_j, C_{ind})$
```

Figure 14: Pseudo-code description of the group operations in a TMKA, as described in Section 4.1. Optional lines are within brackets. $k^i_{e_{cp\text{-}ch(\ell)}}$ denotes the (symmetric) encryption key of the child node v_{ch} of v_ℓ (with path secret ps_ℓ) in the copath of u_j. Exchanges are depicted in blue. When a fresh encryption public key is needed by the group manager (add or update operations), it is either provided within a self-update proposal or it is requested by the group manager within the commit algorithm (cf. Figure 15), thus generating additional communication.

Table 3: Detailed generic communication cost of a no-proposal commit for the various group operations in SUMAC, when relying on a TreeKEM-based admin CGKA. The formulas from this table correspond to the *group distribution* method from Appendix D.3.1, but are applicable to the *broadcast-only* framework by modifying the values of the distribution coefficients to $g_a = g_u := g$ and by removing the extra signatures in red, within brackets. Moreover, the elements in blue, within brackets, correspond to the additional cost of full-updates, compared to partial ones.

Group Ops	Exchanges	Communication Cost
add-admin	1 : **init** ⇒ **register-keys**(pk, sk)	$g_{as}(\vert pk\vert + \vert spk\vert)$
	2 : **propose** ⇒ **get-keys**(pk)	$g_{as}(\vert pk\vert + 2\vert\sigma\vert)$
	3 : **send**($a_p \to \mathcal{G}_a, P_{adm}$)	$g_a(\vert\sigma\vert)$
	4 : **send**($a_p \to a_j, P_{ind}$)	$g_{ind}(\vert ct^{hpke}(2n_u - 1)\vert + \vert\sigma\vert)$
	5 : **commit** ⇒ **get-keys**(pk, spk)	$g_{as}(\vert pk\vert + \vert spk\vert + 2\vert\sigma\vert)$
	6 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a((h_a + 1)\vert pk\vert + \vert spk\vert + h_a\vert ct^{hpke}(1)\vert + \vert\sigma\vert)$
	7 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(\vert ct^{aead}(1)\vert\ [+\vert spk\vert + \vert\sigma\vert])$
	8 : **send**($a_c \to a_j, C_{ind}$)	$g_{ind}(\vert ct^{hpke}(2n_u - 1)\vert + n_u\vert spk\vert + \vert\mathrm{RT}_{adm}\vert + \vert\sigma\vert)$
	Total	$g_a((h_a + 1)\vert pk\vert + \vert spk\vert + h_a\vert ct^{hpke}(1)\vert + 2\vert\sigma\vert) + g_u(\vert ct^{aead}(1)\vert\ [+\vert spk\vert + \vert\sigma\vert]) + g_{ind}(n_u\vert spk\vert + 2\vert ct^{hpke}(2n_u - 1)\vert + \vert\mathrm{RT}_{adm}\vert + 2\vert\sigma\vert) + g_{as}(3\vert pk\vert + 2\vert spk\vert + 4\vert\sigma\vert)$
remove-admin	4 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a(h_a(\vert pk\vert + \vert ct^{hpke}(1)\vert) + \vert\sigma\vert)$
	5 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
	Total	$g_a(h_a(\vert pk\vert + \vert ct^{hpke}(1)\vert) + \vert\sigma\vert) + g_u(\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
[full-]update-admin	3 : **commit** ⇒ **request-keys**(pk, [spk])	$g_{ind}(\vert pk\vert\ [+\vert spk\vert] + 2\vert\sigma\vert)$
	4 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a((h_a + 1)\vert pk\vert\ [+\vert spk\vert] + h_a\vert ct^{hpke}(1)\vert + \vert\sigma\vert)$
	5 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(\vert ct^{aead}(1)\vert\ [+\vert spk\vert + \vert\sigma\vert])$
	6 : **send**($a_c \to a_j, C_{ind}$)	$g_{ind}(\vert ct^{hpke}(2n_u - 1)\vert + \vert\sigma\vert)$
	Total	$g_a((h_a + 1)\vert pk\vert\ [+\vert spk\vert] + h_a\vert ct^{hpke}(1)\vert + \vert\sigma\vert) + g_u(\vert ct^{aead}(1)\vert\ [+\vert spk\vert + \vert\sigma\vert]) + g_{ind}(\vert pk\vert\ [+\vert spk\vert] + \vert ct^{hpke}(2n_u - 1)\vert + 3\vert\sigma\vert)$
add-user	1 : **init** ⇒ **register-keys**(pk, sk)	$g_{as}(\vert pk\vert + \vert spk\vert)$
	7 : **commit** ⇒ **get-keys**(pk, spk)	$g_{as}(\vert pk\vert + \vert spk\vert + 2\vert\sigma\vert)$
	8 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a(\vert ct^{aead}(h_u)\vert + \vert pk\vert + \vert spk\vert + \vert\sigma\vert)$
	9 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
	10 : **send**($a_c \to u_j, C_{ind}$)	$g_{ind}(\vert ct^{hpke}(1)\vert + n_a\vert spk\vert + \vert\sigma\vert)$
	19 : **for** $\ell \in [1..n_a]\backslash\{c\}$ **do** : **send**($a_\ell \to u_j, C_{ind}(u_j)$)	$g_{ind}(n_a - 1)(\vert ct^{hpke}(h_u + 1)\vert + \vert\sigma\vert)$
	Total	$g_a(\vert ct^{aead}(h_u)\vert + \vert pk\vert + \vert spk\vert + \vert\sigma\vert) + g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert]) + g_{ind}(n_a\vert spk\vert + (n_a - 1)\vert ct^{hpke}(h_u + 1)\vert + \vert ct^{hpke}(1)\vert + n_a\vert\sigma\vert) + g_{as}(2\vert pk\vert + 2\vert spk\vert + 2\vert\sigma\vert)$
remove-user	4 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a(\vert ct^{aead}(h_u)\vert + \vert\sigma\vert)$
	5 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
	Total	$g_a(\vert ct^{aead}(h_u)\vert + \vert\sigma\vert) + g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
[full-]update-user	3 : **commit** ⇒ **request-keys**(pk, [spk])	$g_{ind}(\vert pk\vert\ [+\vert spk\vert] + 2\vert\sigma\vert)$
	4 : **send**($a_c \to \mathcal{G}_a, C_{adm}$)	$g_a(\vert ct^{aead}(h_u + 1)\vert + \vert pk\vert\ [+\vert spk\vert] + \vert\sigma\vert)$
	5 : **send**($a_c \to \mathcal{G}_u, C_{usr}$)	$g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert])$
	6 : **send**($a_c \to u_j, C_{ind}$)	$g_{ind}(\vert ct^{hpke}(1)\vert + \vert\sigma\vert)$
	Total	$g_a(\vert ct^{aead}(h_u + 1)\vert + \vert pk\vert\ [+\vert spk\vert] + \vert\sigma\vert) + g_u(h_u\vert ct^{aead}(1)\vert\ [+\vert\sigma\vert]) + g_{ind}(\vert pk\vert\ [+\vert spk\vert] + \vert ct^{hpke}(1)\vert + 3\vert\sigma\vert)$
upgrade-user	**SUMAC.remove-user**	
	SUMAC.add-admin	
downgrade-admin	**SUMAC.remove-admin**	
	SUMAC.add-user	

Table 4: Compared communication costs per user of the group operations – carried out separately – with DGS-TreeKEM, IAS-TreeKEM, TreeKEM and SUMAC, both for the classical and post-quantum frameworks, in the *broadcast-only* setting. The rightmost columns display the relative performances of SUMAC with respect to IAS-TreeKEM – which stands as the most efficient A-CGKA – and TreeKEM.

Group Operation	Setting	Number of std users	Number of admins	Communication Cost per User (Byte) DGS-TK	IAS-TK	TreeKEM	SUMAC	Gain (%) of SUMAC w.r.t. IAS-TK	TreeKEM
add-admin	Classical	16	4	1,176	834	800	582	30.1	27.2
		256	16	1,681	1,188	1,156	740	37.7	36.0
		65,536	256	2,817	1,952	1,920	1,120	42.6	41.7
	PQ	16	4	25,241	18,212	18,178	7,637	58.1	58.0
		256	16	37,874	27,033	27,001	11,227	58.5	58.4
		65,536	256	65,348	45,521	45,489	20,210	55.6	55.6
remove-admin	Classical	16	4	805	571	538	269	52.9	50.0
		256	16	1,349	931	899	450	51.7	50.0
		65,536	256	2,496	1,696	1,664	832	50.9	50.0
	PQ	16	4	17,151	12,247	12,214	4,939	59.7	59.6
		256	16	30,367	21,021	20,989	9,378	55.4	55.3
		65,536	256	58,097	39,505	39,473	18,624	52.9	52.8
part / full update-admin	Classical	16	4	891 / 965	614 / 651	581 / 618	370 / 406	39.8 / 37.6	36.4 / 34.2
		256	16	1,414 / 1,479	964 / 997	932 / 964	543 / 576	43.7 / 42.3	41.7 / 40.3
		65,536	256	2,560 / 2,624	1,728 / 1,760	1,696 / 1,728	928 / 960	46.3 / 45.5	45.3 / 44.5
	PQ	16	4	19,887 / 19,961	13,615 / 13,652	13,582 / 13,618	6,520 / 6,557	52.1 / 52.0	52.0 / 51.9
		256	16	32,762 / 32,827	22,219 / 22,251	22,186 / 22,219	10,704 / 10,737	51.8 / 51.7	51.8 / 51.7
		65,536	256	60,465 / 60,529	40,689 / 40,721	40,657 / 40,689	19,936 / 19,968	51.0 / 51.0	51.0 / 50.9
add-user	Classical	16	4	800	834	800	399	52.1	50.1
		256	16	1,156	1,188	1,156	528	55.6	54.3
		65,536	256	1,920	1,952	1,920	883	54.8	54.0
	PQ	16	4	18,178	18,212	18,178	2,239	87.7	87.7
		256	16	27,001	27,033	27,001	1,961	92.7	92.7
		65,536	256	45,489	45,521	45,489	2,301	94.9	94.9
remove-user	Classical	16	4	538	571	538	252	55.9	53.1
		256	16	899	931	899	434	53.4	51.8
		65,536	256	1,664	1,696	1,664	816	51.9	51.0
	PQ	16	4	12,214	12,247	12,214	319	97.4	97.4
		256	16	20,989	21,021	20,989	562	97.3	97.3
		65,536	256	39,473	39,505	39,473	1,072	97.3	97.3
part / full update-user	Classical	16	4	581 / 618	614 / 651	581 / 618	322 / 358	47.7 / 45.0	44.6 / 42.0
		256	16	932 / 964	964 / 997	932 / 964	483 / 516	49.9 / 48.3	48.2 / 46.5
		65,536	256	1,696 / 1,728	1,728 / 1,760	1,696 / 1,728	864 / 896	50.0 / 49.1	49.1 / 48.1
	PQ	16	4	13,582 / 13,618	13,615 / 13,652	13,582 / 13,618	1,838 / 1,874	86.5 / 86.3	86.5 / 86.2
		256	16	22,186 / 22,219	22,219 / 22,251	22,186 / 22,219	1,800 / 1,833	91.9 / 91.8	91.9 / 91.8
		65,536	256	40,657 / 40,689	40,689 / 40,721	40,657 / 40,689	2,288 / 2,320	94.4 / 94.3	94.4 / 94.3
upgrade-user	Classical	16	4	376	67	-	834	-1,141.7	-
		256	16	525	64	-	1,173	-1,726.3	-
		65,536	256	897	64	-	1,936	-2,925.0	-
	PQ	16	4	7,062	67	-	7,956	-11,739.3	-
		256	16	10,873	64	-	11,789	-18,253.0	-
		65,536	256	19,859	64	-	21,282	-33,152.3	-
downgrade-admin	Classical	16	4	267	67	-	668	-894.0	-
		256	16	450	64	-	977	-1,421.5	-
		65,536	256	832	64	-	1,715	-2,579.7	-
	PQ	16	4	4,938	67	-	7,178	-10,582.1	-
		256	16	9,378	64	-	11,339	-17,553.0	-
		65,536	256	18,624	64	-	20,926	-32,595.9	-

```
**init**$(id)$
    / Initialization of the group manager's state
1 :  if $id = gm$ then :
2 :      $(spk^i_{gm}, ssk^i_{gm}) \leftarrow \mathbf{SIG.key\text{-}gen}()$
3 :      $\mathbf{register\text{-}keys}(spk^i_{gm})$    / Upload to the Authent. Service
4 :      $\gamma^i := (ssk^i_{gm}, spk^i_{gm})$
5 :      return $\gamma^i$
    / Initialization of a user's state
6 :  elseif $id = u_j$ then :
7 :      $(spk^i_j, ssk^i_j) \leftarrow \mathbf{SIG.key\text{-}gen}()$
8 :      $(pk^i_j, sk^i_j) \leftarrow \mathbf{PKE.key\text{-}gen}()$
9 :      $\mathbf{register\text{-}keys}(spk^i_j, pk^i_j)$    / Upload to the Authent. Service
10 :     $\delta^i_j := ((ssk^i_j, sk^i_j), (spk^i_j, pk^i_j))$
11 :     return $\delta^i_j$
```

```
**create-group**$(\gamma^i, (u_k)^{n_u}_{k=1})$
    / Creation of a member group $\mathcal{G}$
1 :  $\mathcal{G} := ((u_k)^{n_u}_{k=1}, gid)$
2 :  $\overrightarrow{op} := (\mathbf{add}(u_k))_{u_k \in \mathcal{G}}$
3 :  $(\gamma^{i+1}, I^{i+1}, T) \leftarrow \mathbf{commit}(\gamma^i, \mathrm{gid}, \varnothing, \overrightarrow{op})$
4 :  return $\gamma^{i+1}, \mathcal{G}, I^{i+1}, T$
```

```
**process**$(\delta^i_k, \mathrm{gid}, C)$
    / Authenticating the message
1 :  $(\widehat{C_{usr}} = (u_j, CT), \sigma_{usr}, [\widehat{C_{ind}}, \sigma_{ind}]) := \mathbf{parse}(C)$
2 :  if $\perp := \mathbf{SIG.verif}(spk_{gm}, \sigma_{usr/ind})$ then :
3 :      return $\perp$
    / If $u_k$ is the new/updated user $u_j$
4 :  if $\widehat{C_{ind}} \neq \varnothing$ then :
5 :      if $u_k \neq u_j$ then :
6 :          return $\perp$
7 :      else :
8 :          $ls^{i+1}_j := \mathbf{HPKE.dec}(sk^{i+1}_j, \widehat{C_{ind}})$
9 :          $len := h(\mathcal{T})$
10 :         $(I^{i+1}, \mathcal{P}(u_k, \mathcal{T}^{i+1})) := \mathbf{derive\text{-}path}(ls^{i+1}_j, len)$
    / If $u_k$ is not user $u_j$
11 :  else :
12 :     $v_\ell := \mathbf{find\text{-}common\text{-}ancestor}(u_j, u_k)$
13 :     $ps_\ell := \mathbf{USKE.dec}(k_{e_{cp\text{-}ch(\ell)}}, ct_\ell \in CT)$
14 :     $len := h(\mathcal{T}) - h(v_\ell)$
15 :     $(I^{i+1}, \mathcal{P}(u_k, \mathcal{T}^{i+1},)) := \mathbf{derive\text{-}path}(ps_\ell, len)$
16 :  $\delta^{i+1}_k := \delta^i_k, \quad \delta^{i+1}_k.PATH := \mathcal{P}(u_k, \mathcal{T}^{i+1})$
17 :  return $\delta^{i+1}_k, I^{i+1}$
```

```
propose($\delta_p^i$, gid, $op(u_j)$)
    / Case by default
    $\widehat{P} := (op(u_j)), \quad \delta_p^{i+1} := \delta_p^i$
    / Self proposal of update
    if $op$ = upd and $u_p = u_j$ then :
        $(pk_j^{i+1}, sk_j^{i+1}) \leftarrow$ PKE.key-gen()
        $\delta_p^{i+1}.\mathrm{pk} := pk_j^{i+1}, \quad \delta_p^{i+1}.\mathrm{sk} := sk_j^{i+1}$
        $\widehat{P} := (op(u_j), \delta_p^{i+1}.\mathrm{pk})$
    / Full update (refreshment of the signature key)
        if upd.type = full then :
            $(spk_j^{i+1}, ssk_j^{i+1}) \leftarrow$ SIG.key-gen()
            $\delta_p^{i+1}.\mathrm{spk} := spk_j^{i+1}, \quad \delta_p^{i+1}.\mathrm{ssk} := ssk_j^{i+1}$
            $\widehat{P} := (op(u_j), \delta_p^{i+1}.\mathrm{pk}, \delta_p^{i+1}.\mathrm{spk})$
    / Self proposal of add
    elseif $op$ = add and $u_p = u_j$ then :
        $\delta_p^{i+1} \leftarrow \mathrm{init}(u_p)$
        $\widehat{P} := (op(u_j), \delta_p^{i+1}.\mathrm{pk}, \delta_p^{i+1}.\mathrm{spk})$
    $P := (\widehat{P}, \sigma \leftarrow \mathrm{SIG.sign}(\delta_p^{i+1}.\mathrm{ssk}, \widehat{P}))$
    return $\delta_p^{i+1}, P$
```

```
commit($\gamma^i$, gid, $\overrightarrow{P}, \overrightarrow{op}$)
    / Processing and validating the pending operations
    $(op_i)_i :=$ validate-ops($\overrightarrow{P}, \overrightarrow{op}$)
    for $op(u_j) \in (op_i)_i$ do :
    / Add or update operation
        if $op(u_j) \in \{$add, upd$\}$ then :
            if $op(u_j)$ = add and $pk_j^{i+1} = spk_j^{i+1} = \varnothing$ then :
                $(pk_j^{i+1}, spk_j^{i+1}) \leftarrow$ get-keys($u_j$, (pk, spk))
            elseif $op(u_j)$ = full-upd and $pk_j^{i+1} = spk_j^{i+1} = \varnothing$ then :
                $(pk_j^{i+1}, spk_j^{i+1}) \leftarrow$ request-keys($u_j$, (pk, spk))
            elseif $op(u_j)$ = part-upd and $pk_j^{i+1} = \varnothing$ then :
                $pk_j^{i+1} \leftarrow$ request-keys($u_j$, pk)
            $ls_j^{i+1} \xleftarrow{\$} \mathcal{K}$
            $\mathcal{P}(u_j, \mathcal{T}^{i+1}) \leftarrow$ path-update($u_j, h(\mathcal{T}^{i+1}), ls_j^{i+1}$)
            $\widehat{C_{ind}} :=$ HPKE.enc($pk_j^{i+1}, ls_j^{i+1}$)
            $C_{ind} := (\widehat{C_{ind}}, \sigma_{ind} \leftarrow \mathrm{SIG.sign}(\gamma^{i+1}.\mathrm{ssk}, \widehat{C_{ind}}))$
    / Remove operation
        else :
            delete($u_j$)
            $\mathcal{P}(u_j, \mathcal{T}^{i+1}) \leftarrow$ path-update($u_j, h(\mathcal{T}^{i+1})$)
            $C_{ind} := \varnothing$
    / Update of the group manager's state
    $\gamma^{i+1} := \gamma^i, \quad \gamma^{i+1}.\mathrm{tree} := \mathcal{T}^{i+1}$
    / Computation of the group key
    $I^{i+1} :=$ derive($\mathcal{T}^{i+1}$.root, "path")
    $CT \leftarrow (\mathrm{USKE.enc}(k^i_{e_{cp\text{-}ch(\ell)}}, ps_\ell^{i+1}))_{v_\ell \in \mathcal{P}(u_j, \mathcal{T}^{i+1})}$
    $\widehat{C_{usr}} := (u_j, CT)$
    $C_{usr} := (\widehat{C_{usr}}, \sigma_{usr} \leftarrow \mathrm{SIG.sign}(\gamma^{i+1}.\mathrm{ssk}, \widehat{C_{usr}}))$
    return $\gamma^{i+1}, I^{i+1}, C := (C_{usr}, C_{ind})$
```

Figure 15: Algorithms of a TMKA, as defined in Definition 3.1, with the associated auxiliary functions.

```
path-update$(u_j, len[, ls_j^{i+1}])$
      / If the leaf secret is provided (add, update)
1 :   if $ls_j^{i+1} \neq \varnothing$ then :
2 :     $ps_1^{i+1}$ := derive$(ls_j^{i+1}, \text{“path”})$
      / If the leaf secret is null (remove)
3 :   else :
4 :     $ps_1^{i+1} \xleftarrow{\$} \mathcal{K}$
      / Derivation of the upper path secrets
5 :   $(I^{i+1}, \mathcal{P}(u_j, \mathcal{T}^{i+1}))$ := derive-path$(ps_1^{i+1}, len)$
6 :   return $(I^{i+1}, \mathcal{P}(u_j, \mathcal{T}^{i+1}))$
```

```
derive-path$(sec, len)$
      / Path derivation from the initial leaf/path secret
1 :   $ps_0 := sec$
2 :   for $\ell \in [\![1, len]\!]$ do :
3 :     $ps_\ell^{i+1}$ := derive$(ps_{\ell-1}^{i+1}, \text{“path”})$
      / Computation of the new group key
4 :   $I^{i+1}$ := derive$(ps_{len}^{i+1}, \text{“path”})$
5 :   return $(I^{i+1}, \mathcal{P}(u_j, \mathcal{T}^{i+1}) = (ps_\ell)_{\ell=1}^{len})$
```

```
derive$(\mathcal{S}, \text{“label”})$
      / Derivation of a set $\mathcal{S}$ of nodes with hash function H
1 :   $\mathcal{S}' := \varnothing$
2 :   for $s \in \mathcal{S}$ do :
3 :     $s' := \mathsf{H}(s, \text{“label”})$
4 :     $\mathcal{S}' := \mathcal{S}' \cup \{s'\}$
5 :   return $\mathcal{S}'$
```

```
regen$(\mathcal{S} = (s_i)_{i=1}^m, \mathcal{R} = (r_i)_{i=1}^m)$
      / Regeneration of a set $\mathcal{S}$ of nodes with hash function G
1 :   $\mathcal{S}' := \varnothing$
2 :   for $i \in [1..m]$ do :
3 :     $s_i' := \mathsf{G}(s_i, r_i)$
4 :   $\mathcal{S}' := (s_i')_{i=1}^m$
5 :   return $\mathcal{S}'$
```

```
request-keys$(role, u_j, (keyType_\ell \in \{pk, spk\})_\ell)$
1 :   / Actions of the committer (com)
2 :   if $role$ = com then :
3 :     $\widehat{Req}$ := (encode$(keyType_\ell))_\ell$
4 :     $Req := (\widehat{Req}, \sigma \leftarrow$ SIG.sign$(ssk_c^{i+1}, \widehat{Req}))$
5 :     send(committer $\longrightarrow u_j, Req$)
      / Actions of the recipient $u_j$
6 :   elseif $role = u_j$ then :
7 :     for $keyType \in (keyType_\ell)_\ell$ do :
8 :       if $keyType = pk$ then :
9 :         $(pk_j^{i+1}, sk_j^{i+1}) \leftarrow$ PKE.key-gen()
10 :        $key := pk_j^{i+1}$
11 :      elseif $keyType = spk$ then :
12 :        $(spk_j^{i+1}, ssk_j^{i+1}) \leftarrow$ SIG.key-gen()
13 :        $key := spk_j^{i+1}$
14 :    $\widehat{Keys} := (key_\ell)_\ell$
15 :    $Keys := (\widehat{Keys}, \sigma \leftarrow$ SIG.sign$(ssk_j^{i+1}, \widehat{Keys}))$
16 :    send($u_j \longrightarrow$ committer, $Keys$)
```

Figure 16: Auxiliary algorithms potentially used in a Group Key Agreement protocol. Exchanges are depicted in blue.

add-admin $((a_p, a_c), a_j)$

```
    / Add admin $a_j$ by $a_c$, after proposal by $a_p$
    / $\boxed{a_j}$: Initialization of its state
1 : $\gamma_j^i \leftarrow \mathbf{init}(a_j),\ pk_j^i := \gamma_j^i.\mathrm{pk},\ spk_j^i := \gamma_j^i.\mathrm{spk}$
    / $\boxed{a_p}$: compulsory add proposal
2 : $(\gamma_p^{i+1}, P = (P_{adm}, P_{ind}))$
        $\leftarrow \mathbf{propose}(\gamma_p^i, \mathrm{gid}, \mathbf{add}(a_j))$
3 : $\mathbf{send}(a_p \longrightarrow \mathcal{G}_a, P_{adm})$
4 : $\mathbf{send}(a_p \longrightarrow a_j, P_{ind})$
    / $\boxed{a_c}$: commit
5 : $(\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), (C_{adm}, C_{usr}, C_{ind}))$
        $\leftarrow \mathbf{commit}(\gamma_c^i, \mathrm{gid}, P_{adm})$
6 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_a, C_{adm})$
7 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_u, C_{usr})$
8 : $\mathbf{send}(a_c \longrightarrow a_j, C_{ind})$
    / $\boxed{a_j}$: processing $P_{ind}$
9 : $\overline{\gamma_j}^{i+1} \leftarrow \mathbf{process}(\gamma_j^i, P_{ind})$
    / $\boxed{\text{all but } a_c}$: processing the commit message
10 : for $k \in [\![1, n_a]\!] \setminus \{c, j\}$ do :
11 :     $(\gamma_k^{i+1}, (I_g^{i+1}, I_{u_k}^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
12 : for $k \in [\![1, n_u]\!]$ do :
13 :     $(\delta_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}), \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
14 : $(\gamma_j^{i+1}, (I_g^{i+1}, I_{u_j}^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_j^i, (C_{adm}, C_{ind}))$
```

remove-admin $([m_p,]\ a_c, a_j)$

```
    / Remove admin $a_j$ by $a_c$, poss. proposed by $m_p$
    / $\boxed{m_p}$: potential remove proposal
1 : $\big[(st_p^{i+1}, P = (\mathbf{rem}(u_j)))$
        $\leftarrow \mathbf{propose}(st_p^i, \mathrm{gid}, \mathbf{rem}(a_j))\big]$
2 : $\big[\mathbf{send}(m_p \longrightarrow \mathcal{G}_a, P)\big]$
    / $\boxed{a_c}$: commit
3 : $\gamma_c^{i+1}, (I_g^{i+1}, I_a^{i+1}), (C_{adm}, C_{usr})$
        $\leftarrow \mathbf{commit}(\gamma_c^i, \mathrm{gid}, P/\mathbf{rem}(a_j))$
4 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_a, C_{adm})$
5 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_u, C_{usr})$
    / $\boxed{\text{all but } a_c}$: process
6 : for $k \in [\![1, n_a]\!] \setminus \{c\}$ do :
7 :     $(\gamma_k^{i+1}, (I_g^{i+1}, I_a^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
8 : for $k \in [\![1, n_u]\!]$ do :
9 :     $(\delta_k^{i+1}, I_g^{i+1}, \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
```

add-user $([m_p,]\ a_c, u_j)$

```
    / Add user $u_j$ by $a_c$, possibly proposed by $m_p$
    / $\boxed{u_j}$: Initialization of its state
1 : $\delta_j^i \leftarrow \mathbf{init}(u_j),\quad pk_j^i := \delta_j^i.\mathrm{pk}$
    / $\boxed{m_p}$ (possibly $u_j$): potential add proposal
2 : if $m_p \neq u_j$ then :
3 :     $\big[(st_p^{i+1}, P = (\mathbf{add}(u_j), \sigma_p)) \leftarrow \mathbf{propose}(st_p^i, \mathrm{gid}, \mathbf{add}(u_j))\big]$
4 : else :
5 :     $\big[(\delta_j^{i+1}, P = (\mathbf{add}(u_j), pk_j^{i+1}, \sigma_j)) \leftarrow \mathbf{propose}(\delta_j^i, \mathrm{gid}, \mathbf{add}(u_j))\big]$
6 : $\big[\mathbf{send}(m_p \longrightarrow \mathcal{G}_a, P)\big]$
    / Stage 1
    / $\boxed{a_c}$: commit
7 : $(\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), (C_{adm}, C_{usr}, C_{ind})) \leftarrow \mathbf{commit}(\gamma_c^i, \mathrm{gid}, P/\mathbf{add}(u_j))$
8 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_a, C_{adm})$
9 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_u, C_{usr})$
10 : $\mathbf{send}(a_c \longrightarrow u_j, C_{ind})$
    / $\boxed{\text{all but } a_c}$: process
11 : for $k \in [\![1, n_a]\!] \setminus \{c\}$ do :
12 :     $(\gamma_k^{i+1}, (I_g^{i+1}, I_{u_k}^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
13 : for $k \in [\![1, n_u]\!]$ do :
14 :     $(\delta_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}), \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
15 : / Stage 2
    / $\boxed{a_{\ell \neq c}}$: subsequent commits by other admins
16 : for $\ell \in [\![1, n_a]\!] \setminus \{c\}$ do :
17 :     $(\gamma_\ell^{i+x}, (I_g^{i+x}, I_{u_\ell}^{i+x}), (C_{ind}(u_j), \dots)) \leftarrow \mathbf{commit}(\gamma_\ell^{i+x-1}, \mathrm{gid}, \dots)$
18 :     / We consider here only the message sent to $u_j$
19 :     $\mathbf{send}(a_\ell \longrightarrow u_j, C_{ind}(u_j))$
        / $\boxed{u_j}$: process
20 :     $(\delta_j^{i+x}, (I_g^{i+x}, (I_{u_\ell}^{i+x})_{\ell=1}^{n_a}), \top) \leftarrow \mathbf{process}(\delta_j^{i+x-1}, (C_{ind}(u_j), \dots))$
```

remove-user $([m_p,]\ a_c, u_j)$

```
    / Remove user $u_j$ by $a_c$, possibly proposed by $m_p$
    / $\boxed{m_p}$: potential remove proposal
1 : $\big[(st_p^{i+1}, P = (\mathbf{rem}(u_j))) \leftarrow \mathbf{propose}(st_p^i, \mathrm{gid}, \mathbf{rem}(u_j))\big]$
2 : $\big[\mathbf{send}(m_p \longrightarrow \mathcal{G}_a, P)\big]$
    / $\boxed{a_c}$: commit
3 : $(\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), (C_{adm}, C_{usr})) \leftarrow \mathbf{commit}(\gamma_c^i, \mathrm{gid}, P/\mathbf{rem}(u_j))$
4 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_a, C_{adm})$
5 : $\mathbf{send}(a_c \longrightarrow \mathcal{G}_u, C_{usr})$
    / $\boxed{\text{all but } a_c}$: process
6 : for $k \in [\![1, n_a]\!] \setminus \{c\}$ do :
7 :     $(\gamma_k^{i+1}, (I_g^{i+1}, I_{u_k}^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
8 : for $k \in [\![1, n_u]\!]$ do :
9 :     $(\delta_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}), \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
```

```
**update-admin** $(a_c, a_j)$
    / Update admin $a_j$ by $a_c$, possibly self-proposed
    / $\boxed{a_j}$: potential self-update proposal
1:  $\big[(\gamma_j^{i+1}, P = (\mathbf{upd}(a_j, type), \gamma_j^{i+1}.\mathsf{pk}, \sigma_j))$
        $\leftarrow \mathbf{propose}(\gamma_j^i, \mathsf{gid}, \mathbf{upd}(a_j, type))\big]$
2:  $\big[\mathbf{send}(a_j \rightarrow \mathcal{G}_a, P)\big]$
    / $\boxed{a_c}$: commit
3:  $(\gamma_c^{i+1}, I_g^{i+1}, (C_{adm}, C_{usr}, C_{ind}))$
        $\leftarrow \mathbf{commit}(\gamma_c^i, \mathsf{gid}, P/\mathbf{upd}(a_j, type))$
4:  $\mathbf{send}(a_c \rightarrow \mathcal{G}_a, C_{adm})$
5:  $\mathbf{send}(a_c \rightarrow \mathcal{G}_u, C_{usr})$
6:  $\mathbf{send}(a_c \rightarrow a_j, C_{ind})$
    / $\boxed{\text{all but } a_c}$: process
7:  for $k \in [\![1, n_a]\!] \setminus \{c\}$ do :
8:      $(\gamma_k^{i+1}, (I_g^{i+1}, [I_{u_j}^{i+1}]), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
9:  for $k \in [\![1, n_u]\!]$ do :
10:     $(\delta_k^{i+1}, (I_g^{i+1}, I_{u_j}^{i+1}), \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
```

```
**update-user** $(a_c, u_j)$
    / Update user $u_j$ by $a_c$, possibly self-proposed
    / $\boxed{u_j}$: potential self-update proposal
1:  $\big[(\delta_j^{i+1}, P = (\mathbf{upd}(u_j, type), \delta_j^{i+1}.\mathsf{pk}, \sigma_j))$
        $\leftarrow \mathbf{propose}(\delta_j^i, \mathsf{gid}, \mathbf{upd}(u_j, type))\big]$
2:  $\big[\mathbf{send}(u_j \rightarrow \mathcal{G}_a, P)\big]$
    / $\boxed{a_c}$: commit
3:  $(\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), (C_{adm}, C_{usr}, C_{ind}))$
        $\leftarrow \mathbf{commit}(\gamma_c^i, \mathsf{gid}, P/\mathbf{upd}(u_j, type))$
4:  $\mathbf{send}(a_c \rightarrow \mathcal{G}_a, C_{adm})$
5:  $\mathbf{send}(a_c \rightarrow \mathcal{G}_u, C_{usr})$
6:  $\mathbf{send}(a_c \rightarrow u_j, C_{ind})$
    / $\boxed{\text{all but } a_c}$: process
7:  for $k \in [\![1, n_a]\!] \setminus \{c\}$ do :
8:      $(\gamma_k^{i+1}, (I_g^{i+1}, I_{u_k}^{i+1}), \top) \leftarrow \mathbf{process}(\gamma_k^i, C_{adm})$
9:  for $k \in [\![1, n_u]\!]$ do :
10:     $(\delta_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}), \top) \leftarrow \mathbf{process}(\delta_k^i, C_{usr})$
```

```
**downgrade-admin** $((a_{c_1}, a_{c_2}), a_j)$
    / Downgrade of admin $a_j$ into user $u_j$ by $a_{c_1}$ and $a_{c_2}$
1:  $((\gamma_\ell^{i+1})_{\ell=1}^{n_a}, (\delta_\ell^{i+1})_{\ell=1}^{n_u}, I_a^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}, I_g^{i+1})$
        $\leftarrow \mathbf{remove\text{-}admin}(a_{c_1}, u_j)$
2:  $((\gamma_\ell^{i+2})_{\ell=1}^{n_a}, (\delta_\ell^{i+2})_{\ell=1}^{n_u}, I_a^{i+2}, (I_{u_\ell}^{i+2})_{\ell=1}^{n_a}, I_g^{i+2})$
        $\leftarrow \mathbf{add\text{-}user}(a_{c_2}, a_j)$
```

```
**upgrade-user** $((a_{c_1}, a_{c_2}), u_j)$
    / Upgrade of user $u_j$ into admin $a_j$ by $a_{c_1}$ and $a_{c_2}$
1:  $((\gamma_\ell^{i+1})_{\ell=1}^{n_a}, (\delta_\ell^{i+1})_{\ell=1}^{n_u}, I_a^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a}, I_g^{i+1})$
        $\leftarrow \mathbf{remove\text{-}user}(a_{c_1}, u_j)$
2:  $((\gamma_\ell^{i+2})_{\ell=1}^{n_a}, (\delta_\ell^{i+2})_{\ell=1}^{n_u}, I_a^{i+2}, (I_{u_\ell}^{i+2})_{\ell=1}^{n_a}, I_g^{i+2})$
        $\leftarrow \mathbf{add\text{-}admin}(a_{c_2}, a_j)$
```

Figure 17: Pseudo-code description of the group operations in SUMAC, carried out in the separate-operations paradigm. In this figure, the actions are performed by one or several committing administrator(s) a_c (or a_{c_1} and a_{c_2}) upon a standard user u_j or another administrator a_j. The reader is invited to refer to Section 3.2 and Section 4.1 for the operations carried out by a CGKA and a TMKA, seen as black-boxes, as well as to Figure 16 for the subroutines of derivation and regeneration of a set of nodes.

```
$\mathbf{init}(m_j \in \{a_j, u_j\})$
    / Initialization of a group member's state
1 :  $(pk_j^i, sk_j^i) \leftarrow \mathbf{PKE.key\text{-}gen}()$
2 :  $(spk_j^i, ssk_j^i) \leftarrow \mathbf{SIG.key\text{-}gen}()$
3 :  $\mathbf{register\text{-}keys}(pk_j^i, spk_j^i)$    / Upload to the Authent. Service
4 :  $st_j^i \in \{\gamma_j^i, \delta_j^i\} := ((sk_j^i, ssk_j^i), (pk_j^i, spk_j^i))$
5 :  $\mathbf{return}\ st_j^i$
```

```
$\mathbf{create\text{-}group}(\gamma_1^i, (a_k)_{k=1}^{n_a}, (u_k)_{k=1}^{n_u})$
    / Creation of a member group $\mathcal{G}$
1 :  $\mathcal{G}_a := (\mathsf{gid}, (a_k)_{k=1}^{n_a})$
2 :  $\mathcal{G}_u := (\mathsf{gid}, (u_k)_{k=1}^{n_u})$
3 :  $\mathcal{G} = \mathcal{G}_a \cup \mathcal{G}_u$
    / Adding firstly administrators, then standard users
4 :  $\overrightarrow{op} := ((\mathbf{add}(a_k))_{a_k \in \mathcal{G}_a}, (\mathbf{add}(u_k))_{u_k \in \mathcal{G}_u})$
5 :  $(\gamma_1^{i+1}, I^{i+1}, C) \leftarrow \mathbf{commit}(\gamma_1^i, \mathsf{gid}, \varnothing, \overrightarrow{op})$
6 :  $\mathbf{return}\ \gamma_1^{i+1}, \mathcal{G}, I^{i+1}, C$
```

```
$\mathbf{propose}(st_p^i \in \{\gamma_p^i, \delta_p^i\}, \mathsf{gid}, op(m_j))$
    / $\boxed{m_p}$: default proposal by $m_p$ for $m_j$
1 :  $\widehat{P_{adm}} := op(m_j), \quad \widehat{P_{ind}} = \varnothing, \quad st_p^{i+1} := st_p^i$
    / $\boxed{m_j}$: self proposal of $\mathbf{upd\text{-}usr}$ or $\mathbf{upd\text{-}adm}$
2 :  $\mathbf{if}\ op = \mathbf{upd}\ \mathbf{and}\ m_p = m_j\ \mathbf{then}$ :
3 :      $(pk_j^{i+1}, sk_j^{i+1}) \leftarrow \mathbf{PKE.key\text{-}gen}()$
4 :      $st_p^{i+1}.\mathsf{pk} := pk_j^{i+1}, \quad st_p^{i+1}.\mathsf{sk} := sk_j^{i+1}$
5 :      $\widehat{P_{adm}} := (\mathbf{upd}(u_j), st_p^{i+1}.\mathsf{pk})$
    / Full update (refreshment of the signature key)
6 :      $\mathbf{if}\ \mathbf{upd}.\mathsf{type} = \mathsf{full}\ \mathbf{then}$ :
7 :          $(spk_j^{i+1}, ssk_j^{i+1}) \leftarrow \mathbf{SIG.key\text{-}gen}()$
8 :          $st_p^{i+1}.\mathsf{spk} := spk_j^{i+1}, \quad st_p^{i+1}.\mathsf{ssk} := ssk_j^{i+1}$
9 :          $\widehat{P_{adm}} := (\widehat{P_{adm}}, st_p^{i+1}.\mathsf{spk})$
    / $\boxed{u_j}$: self proposal of $\mathbf{add\text{-}user}$
10 : $\mathbf{elseif}\ op = \mathbf{add\text{-}usr}\ \mathbf{and}\ m_p = u_j\ \mathbf{then}$ :
11 :     $\delta_j^{i+1} \leftarrow \mathbf{init}(u_j)$
12 :     $\widehat{P_{adm}} := (\mathbf{add\text{-}usr}(u_j), \delta_j^{i+1}.\mathsf{pk}, \delta_j^{i+1}.\mathsf{spk})$
    / $\boxed{a_p}$: proposal of $\mathbf{add\text{-}admin}$ for $a_j$
13 : $\mathbf{elseif}\ op = \mathbf{add\text{-}adm}\ \mathbf{then}$ :
14 :     $(pk_j^{i+1}) \leftarrow \mathbf{request\text{-}keys}(a_j, \mathsf{pk})$
15 :     $\widehat{P_{adm}} := (\mathbf{add\text{-}adm}(a_j))$
        / Creation of a temporary user tree $\overline{\mathcal{T}_{u_j}}^{i+1}$ for $a_j$
16 :     $\overline{\mathcal{T}_{u_j}}^{i+1} := \mathbf{derive}(\mathcal{T}_{u_p}^i)$
17 :     $\widehat{P_{ind}} \leftarrow \mathbf{HPKE.enc}(\gamma_j^{i+1}.\mathsf{pk}, \overline{\mathcal{T}_{u_j}}^{i+1})$
18 :     $P_{ind} := (\widehat{P_{ind}}, \sigma \leftarrow \mathbf{SIG.sign}(\gamma_p^{i+1}.\mathsf{ssk}, \widehat{P_{ind}}))$
19 : $P_{adm} := (\widehat{P_{adm}}, \sigma \leftarrow \mathbf{SIG.sign}(st_p^{i+1}.\mathsf{ssk}, \widehat{P_{adm}}))$
20 : $\mathbf{return}\ st_p^{i+1}, P := (P_{adm}, P_{ind})$
```

```
$\textbf{commit}(\gamma_c^i, \text{gid}, \overrightarrow{P}, \overrightarrow{op})$
    / Validating the pending operations
    $(op_i)_i := \textbf{validate-ops}(\overrightarrow{P}, \overrightarrow{op})$
    / Case of new users still attached to the root of tree $\mathcal{T}_{u_c}^i$
    $(u_\ell)_\ell := \textbf{rooted-users}(\gamma_c^i.\text{tree})$
    for $u_\ell \in (u_\ell)_\ell$ do :
        if $\textbf{rem-usr}(u_\ell) \notin (op_i)_i$ then
            $ls_\ell^{i+1} \xleftarrow{\$} \mathcal{K}$
            $\mathcal{P}_x(u_\ell, \mathcal{T}_{u_c}^{i+1}) := ls_\ell^{i+1} \cup \mathcal{P}(u_\ell, \mathcal{T}_{u_c}^i)$
            $C_{ind}(\ell) \leftarrow \textbf{HPKE.enc}(\delta_\ell^i.\text{pk}, \mathcal{P}_x(u_\ell, \mathcal{T}_{u_c}^{i+1}))$
    / Carrying out the pending operations
    for $op(m_j) \in (op_i)_i$ do :
        / Getting public keys of $m_j$ if necessary
        if $op(m_j) = \textbf{add}$ and $pk_j^{i+1} = spk_j^{i+1} = \varnothing$ then :
            $(pk_j^{i+1}, spk_j^{i+1}) \leftarrow \textbf{get-keys}(m_j, (\text{pk}, \text{spk}))$
        elseif $op(m_j) = \textbf{full-upd}$ and $pk_j^{i+1} = spk_j^{i+1} = \varnothing$ then :
            $(pk_j^{i+1}, spk_j^{i+1}) \leftarrow \textbf{request-keys}(m_j, (\text{pk}, \text{spk}))$
        elseif $op(m_j) = \textbf{part-upd}$ and $pk_j^{i+1} = \varnothing$ then :
            $pk_j^{i+1} \leftarrow \textbf{request-keys}(m_j, \text{pk})$
        $st_j^{i+1}.\text{pk} := pk_j^{i+1}$,    $[st_j^{i+1}.\text{spk} := spk_j^{i+1}]$
        / Distinction between admin and user commits
        if $m_j = a_j$ then :
            $(\gamma_c^{i+1}, (I_g^{i+1}, I_a^{i+1}), C) \leftarrow \textbf{adm-commit}(\gamma_c^i, \text{gid}, op(a_j), \gamma_j^{i+1})$
        elseif $m_j = u_j$ then :
            $(\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), C) \leftarrow \textbf{usr-commit}(\gamma_c^i, \text{gid}, op(u_j), \delta_j^{i+1})$
    return $\gamma_c^{i+1}, (I_g^{i+1}, I_a^{i+1}/I_{u_c}^{i+1}), C$
```

```
$\textbf{adm-commit}(\gamma_c^i, \text{gid}, op(a_j), \gamma_j^{i+1})$
    / Administrator commit by $a_c$ upon $a_j$
    / Commit for $op(a_j)$ in the admin CGKA
    / $C_{adm}$ issued by the CGKA comprises $pk_j^{i+1}$ and $spk_j^{i+1}$
    if $op \in \{\textbf{rem-adm}, \textbf{upd-adm}\}$ then :
        $(\gamma_c^{i+1}, I_{cgka}^{i+1}, C_{adm}) \leftarrow \textbf{CGKA.commit}(\gamma_c^i, \text{gid}, op(a_j))$
        $\widehat{C_{ind}} := \varnothing$
    elseif $op = \textbf{add-adm}$ then :
        $(\gamma_c^{i+1}, I_{cgka}^{i+1}, (C_{adm}, \widehat{C_{ind}})) \leftarrow \textbf{CGKA.commit}(\gamma_c^i, \text{gid}, op(a_j))$
        $\widehat{C_{ind}} := (\widehat{C_{ind}}, (spk_\ell)_{\ell=1}^{n_u})$    / Std users' signature keys
    / Add or update: creation by $a_c$ of the regeneration set
    if $op \in \{\textbf{add-adm}, \textbf{upd-adm}\}$ then :
        $\mathcal{R} = (r_i)_{i=1}^{2n_u-1} := \textbf{derive}(\mathcal{T}_{u_c}^i, \text{“regen”})$
        $\widehat{C'_{ind}} \leftarrow \textbf{HPKE.enc}(\gamma_j^{i+1}.\text{pk}, \mathcal{R})$
        $\widehat{C_{ind}} := (\widehat{C_{ind}}, \widehat{C'_{ind}})$
        $C_{ind} \leftarrow (\widehat{C_{ind}}, \textbf{SIG.sign}(\gamma_c^{i+1}.\text{ssk}, \widehat{C_{ind}}))$
    / Computation of the new admin and group keys
    $I_a^{i+1} := \textbf{derive}(I_{cgka}^{i+1}, \text{“path”})$
    $I_g^{i+1} := \textbf{derive}(I_{cgka}^{i+1}, \text{“key”})$
    $\widehat{C_{usr}} \leftarrow \textbf{AEAD.enc}(I_{u_c}^i, I_g^{i+1})$
    $C_{usr} \leftarrow (\widehat{C_{usr}}, \textbf{SIG.sign}(\gamma_c^{i+1}.\text{ssk}, \widehat{C_{usr}}))$
    return $\gamma_c^{i+1}, (I_g^{i+1}, I_a^{i+1}), C = (C_{adm}, C_{usr}, C_{ind})$
```

```
$\textbf{usr-commit}(\gamma_c^i, \text{gid}, op(u_j), \delta_j^{i+1})$
    / User commit by $a_c$ upon $u_j$
    / Commit for $op(u_j)$ in $a_c$'s TMKA
    $(\gamma_c^{i+1}, I_{u_c}^{i+1}, (C_{usr}, \widehat{C_{ind}})) \leftarrow \textbf{TMKA.commit}(\gamma_c^i, \text{gid}, op(u_j))$
    if $op = \textbf{add-usr}$ then
        $\widehat{C_{ind}} := (\widehat{C_{ind}}, (spk_\ell)_{\ell=1}^{n_a})$    / administrators' signature keys
    $C_{ind} \leftarrow (\widehat{C_{ind}}, \textbf{SIG.sign}(\gamma_c^{i+1}.\text{ssk}, \widehat{C_{ind}}))$
    / Creation by $a_c$ of the regeneration set
    if $op = \textbf{add-usr}$ or $\textbf{rem-usr}$ then :
        $\mathcal{R} = (r_i)_{i=1}^{h_u} := \textbf{derive}(\mathcal{P}(u_j, \mathcal{T}_{u_c}^{i+1}))$
    elseif $op = \textbf{upd-usr}$ then :
        $\mathcal{R} = (r_i)_{i=1}^{h_u+1} := \textbf{derive}(\mathcal{P}_x(u_j, \mathcal{T}_{u_c}^{i+1}))$
    / Creation of the admin message
    $\widehat{C_{adm}} \leftarrow \textbf{AEAD.enc}(I_a^i, \mathcal{R})$
    if $op \in \{\textbf{add-usr}, \textbf{full-upd-usr}\}$ then :
        $\widehat{C_{adm}} := (\widehat{C_{adm}}, pk_j^{i+1}, spk_j^{i+1})$
    elseif $op = \textbf{part-upd-usr}$ then :
        $\widehat{C_{adm}} := (\widehat{C_{adm}}, pk_j^{i+1})$
    $C_{adm} \leftarrow (\widehat{C_{adm}}, \textbf{SIG.sign}(\gamma_c^{i+1}.\text{ssk}, \widehat{C_{adm}}))$
    / Computation of the new group key
    $I_g^{i+1} := \textbf{derive}(\mathcal{R}.\text{root} = r_1, \text{“key”})$
    return $\gamma_c^{i+1}, (I_g^{i+1}, I_{u_c}^{i+1}), C = (C_{adm}, C_{usr}, C_{ind})$
```

```
**process**($st_k^i \in \{\gamma_k^i, \delta_k^i\}$, gid, $msg \in \{C, P_{ind}\}, \gamma_t^i$)
    / **Processing a message emitted by admin $a_e$**
    $I_g^{i+1} = \varnothing, \quad \forall \ell \in [\![1, n_a]\!], I_{u_\ell}^{i+1} = \varnothing$
    / Authenticating the message
    $(\widehat{msg}, \sigma) :=$ **parse**($msg$)
    if **SIG.verif**($\gamma_t^i$.spk, $\sigma, \widehat{msg}$) $= \perp$ then :
        return $\perp$
    / Different processing cases
    if $\widehat{msg} = \widehat{P_{ind}}$ and $m_k = a_j$ then :
        $\overline{\gamma_k}^{i+1} \leftarrow$ **prop-process**($\gamma_k^i$, gid, $\widehat{P_{ind}}, \gamma_t^i$)
    elseif $\widehat{msg} = \widehat{C}$ and $m_k = a_k$ then :
        $(\gamma_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a})) \leftarrow$ **adm-com-process**($\gamma_k^i$, gid, $\widehat{C}, \gamma_t^i$)
    elseif $\widehat{msg} = \widehat{C}$ and $m_k = u_k$ then :
        $(\delta_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a})) \leftarrow$ **usr-com-process**($\delta_k^i$, gid, $\widehat{C}, \gamma_t^i$)
    return $\overline{\gamma_k}^{i+1} / st_k^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a})$
```

```
**adm-com-process**($\gamma_k^i$, gid, $\widehat{C}, \gamma_c^i$)
    / **Processing by $a_k$ of a commit message $\widehat{C}$ sent by $a_c$**
    $I_a^{i+1} := \varnothing$
    $(\widehat{C_{adm}}, [\widehat{C_{ind}}]) :=$ **parse**($\widehat{C}$)
    / Admin operations
    if $op(a_j) \in$ {**add-adm**, **upd-adm**, **rem-adm**} then :
        $I_{cgka}^{i+1} \leftarrow$ CGKA.process($\widehat{C_{adm}}$)
        $I_a^{i+1} :=$ **derive**($I_{cgka}^{i+1}$, "path")
        $I_g^{i+1} :=$ **derive**($I_{cgka}^{i+1}$, "key")
        / Case where $a_k = a_j$
        if $a_k = a_j$ and $op(a_j) \in$ {**add-adm**, **upd-adm**} then :
            $\mathcal{R} :=$ **HPKE.dec**($\gamma_k^i$.sk, $\widehat{C_{ind}}$)
            $\mathcal{T}_{u_k}^{i+1} :=$ **regen**($\mathcal{T}_{u_k}^i, \mathcal{R}$)
            $I_{u_k}^{i+1} :=$ **derive**($\mathcal{T}_{u_k}^{i+1}$.root, "key")
        elseif $a_k = a_j$ and $op(a_j) =$ **rem-adm** then :
            **delete**($\gamma_k^i$)
    / User operations
    elseif $op(u_j) \in$ {**add-usr**, **rem-usr**, **upd-usr**} then :
        $\mathcal{R} :=$ **AEAD.dec**($\gamma_k^i$.sk, $\widehat{C_{adm}}$)
        if $op(u_j) =$ **upd-usr** then :
            $\mathcal{P}_x(u_j, \mathcal{T}_{u_k}^{i+1}) :=$ **regen**($\mathcal{P}_x(u_j, \mathcal{T}_{u_k}^i), \mathcal{R}$)
        else :
            $\mathcal{P}(u_j, \mathcal{T}_{u_k}^{i+1}) :=$ **regen**($\mathcal{P}(u_j, \mathcal{T}_{u_k}^i), \mathcal{R}$)
        $I_g^{i+1} :=$ **derive**($\mathcal{R}$.root, "key")
        $I_{u_k}^{i+1} :=$ **derive**($\mathcal{P}(u_j, \mathcal{T}_{u_k}^{i+1})$.root, "key")
    return $\gamma_k^{i+1}, (I_g^{i+1}, I_{u_k}^{i+1}, I_a^{i+1})$
```

```
**prop-process**($\gamma_j^i$, gid, $\widehat{P_{ind}}, \gamma_p^i$)
    / **New admin $a_j$ processing proposal $P_{ind}$ from $a_p$**
    / Recovering an initial temporary user tree $\overline{\mathcal{T}_{u_j}}^{i+1}$
    $\overline{\mathcal{T}_{u_j}}^{i+1} :=$ **HPKE.dec**($\gamma_j^{i+1}$.sk, $\widehat{P_{ind}}$)
    / Creating a temporay new state $\overline{\gamma_j}^{i+1}$
    $\overline{\gamma_j}^{i+1} := \gamma_j^i, \quad \gamma_j^{i+1}$.tree $:= \overline{\mathcal{T}_{u_j}}^{i+1}$
    return $\overline{\gamma_j}^{i+1}$
```

```
**usr-com-process**($\delta_k^i$, gid, $\widehat{C}, \gamma_c^i$)
    / **Processing by $u_k$ of a commit message $\widehat{C}$ sent by $a_c$**
    $(\widehat{C_{usr}}, [\widehat{C_{ind}}]) :=$ **parse**($\widehat{C}$)
    / Admin operations
    if $op(a_j) \in$ {**add-adm**, **upd-adm**, **rem-adm**} then :
        $I_g^{i+1} :=$ **AEAD.dec**($I_{u_c}^i, \widehat{C_{usr}}$)
    if $op(a_j) \in$ {**add-adm**, **upd-adm**} then :
        $\mathcal{R} :=$ **derive**($\mathcal{P}_x(u_k, \mathcal{T}_{u_c}^i)$, "regen")
        $I_{u_j}^{i+1} :=$ **derive**($\mathcal{P}(u_k, \mathcal{T}_{u_j}^{i+1})$.root, "key")
    / User operations for $u_k \neq u_j$
    elseif $u_k \neq u_j$ and $op(u_j) \in$ {**add-usr**, **rem-usr**, **upd-usr**} then :
        $\mathcal{P}_x(u_k, \mathcal{T}_{u_c}^{i+1}), I_{u_c}^{i+1} :=$ TMKA.process($\widehat{C_{usr}}$)
        $v_m :=$ **common-ancestor**($u_j, u_k$)
        $\mathcal{R} :=$ **derive**($\mathcal{P}_x(v_m, \mathcal{T}_{u_c}^i)$, "regen")
        $I_g^{i+1} :=$ **derive**($\mathcal{R}$.root, "key")
        for $\ell \in [\![1, n_a]\!] \setminus \{c\}$ do :
            $\mathcal{P}(u_k, \mathcal{T}_{u_\ell}^{i+1}) :=$ **regen**($\mathcal{P}(u_k, \mathcal{T}_{u_\ell}^i), \mathcal{R}$)
            $I_{u_\ell}^{i+1} :=$ **derive**($\mathcal{P}(u_k, \mathcal{T}_{u_\ell}^{i+1})$.root, "key")
    / User operations for $u_k = u_j$
    elseif $u_k = u_j$ then :
        / Subsequent recovery of new user $u_j$'s secrets in other trees
        if $op(u_j) =$ **add-usr** and $\widehat{C_{ind}} \neq \varnothing$ then :
            $\mathcal{P}_x(u_k, \mathcal{T}_{u_c}^{i+1}) :=$ **HPKE.dec**($\delta_k^i$.sk, $\widehat{C_{ind}}$)
            $I_{u_c}^{i+1} :=$ **derive**($\mathcal{P}(u_k, \mathcal{T}_{u_c}^{i+1})$.root, "key")
        elseif $op(u_j) =$ **upd-usr** then :
            $\mathcal{P}_x(u_k, \mathcal{T}_{u_c}^{i+1}), I_{u_c}^{i+1} :=$ TMKA.process($\widehat{C_{usr}}$)
            $\mathcal{R} :=$ **derive**($\mathcal{P}_x(v_m, \mathcal{T}_{u_c}^{i+1})$, "regen")
            $I_g^{i+1} :=$ **derive**($\mathcal{R}$.root, "key")
            for $\ell \in [\![1, n_a]\!] \setminus \{c\}$ do :
                $\mathcal{P}_x(u_k, \mathcal{T}_{u_\ell}^{i+1}) :=$ **regen**($\mathcal{P}_x(u_k, \mathcal{T}_{u_\ell}^i), \mathcal{R}$)
                $I_{u_\ell}^{i+1} :=$ **derive**($\mathcal{P}_x(u_k, \mathcal{T}_{u_\ell}^{i+1})$.root, "key")
        elseif $op(u_j) =$ **rem-usr** then :
            **delete**($\delta_k^i$)
    return $\gamma_j^{i+1}, (I_g^{i+1}, (I_{u_\ell}^{i+1})_{\ell=1}^{n_a})$
```

Figure 18: Algorithms of SUMAC, as defined in Definition 3.3. The auxiliary functions are detailed in Figure 16.

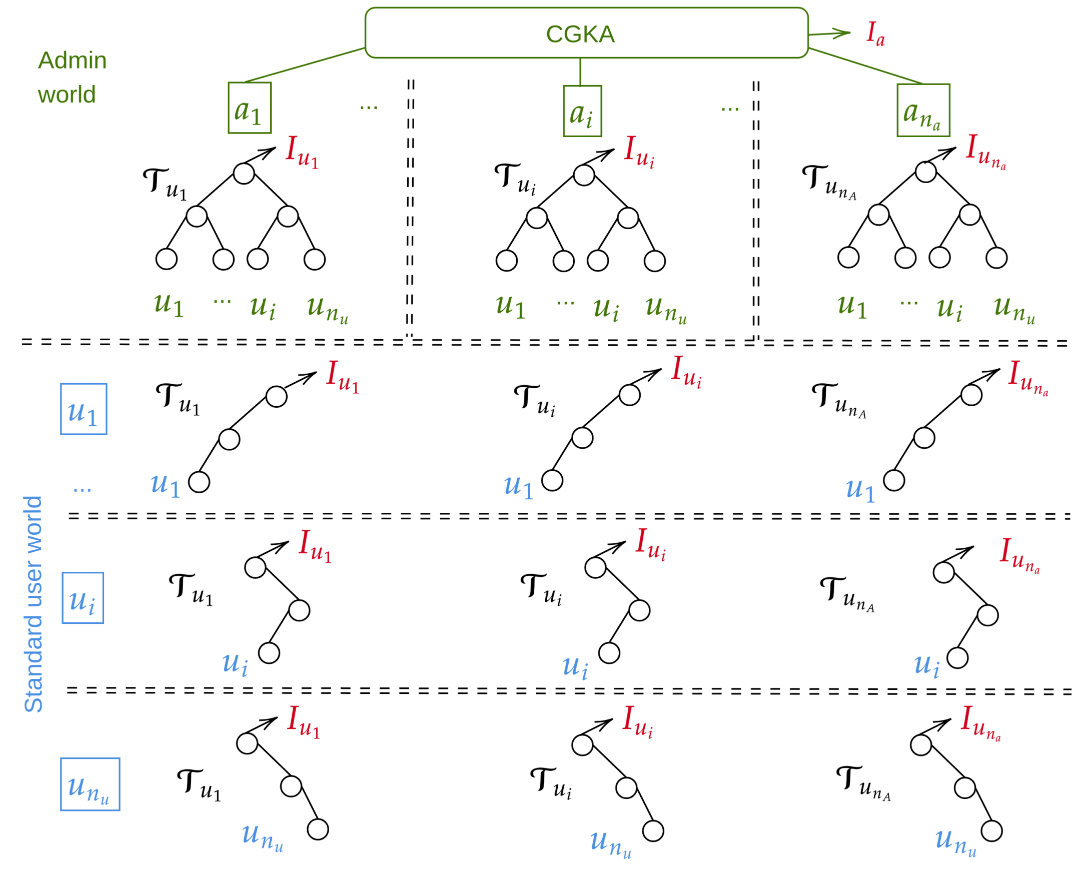

Figure 19: High-level design of SUMAC, composed of a CGKA between the administrators and an instance of a TMKA between each admin and all the standard users. Dashed lines indicate the partitioning between the admin world (green), in which every admin knows the admin key I_a and the entire TMKA tree it manages, and user world (blue), where every standard user knows (only) its direct path and the common key I_{u_i} in *all* the user trees.

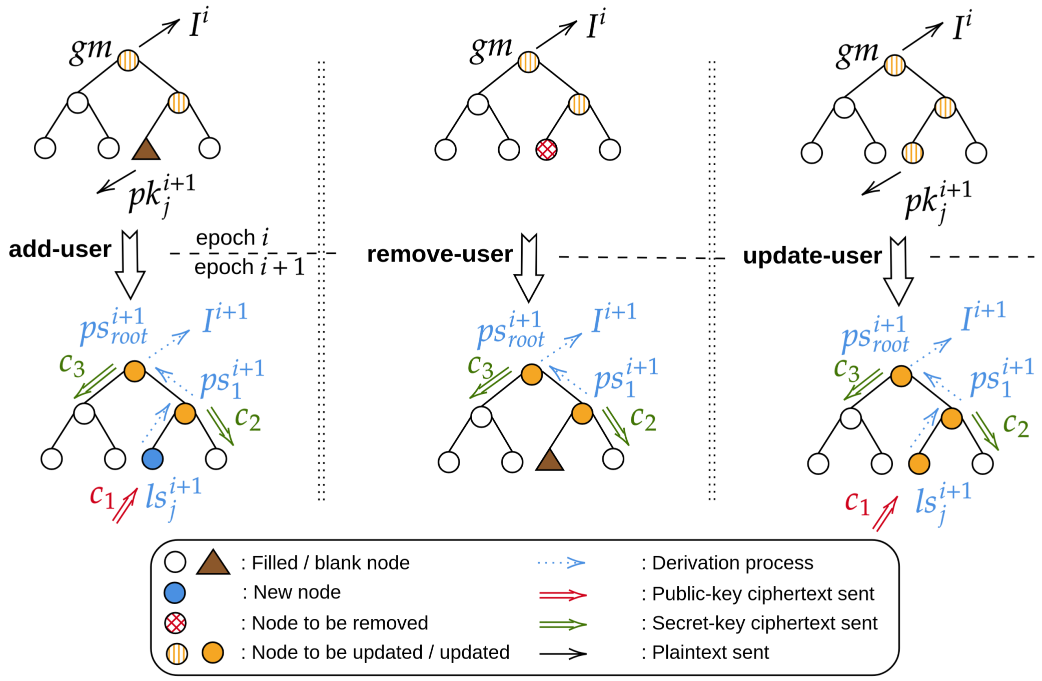

Figure 20: Overview of the group operations in a TMKA, as detailed in Section 3.1. All changes in the tree structure related to these operations (update of the direct path of the concerned user, encryption of the refreshed path secrets, generation of the new common secret) are similar to those of a TreeKEM-like CGKA [8] [4].

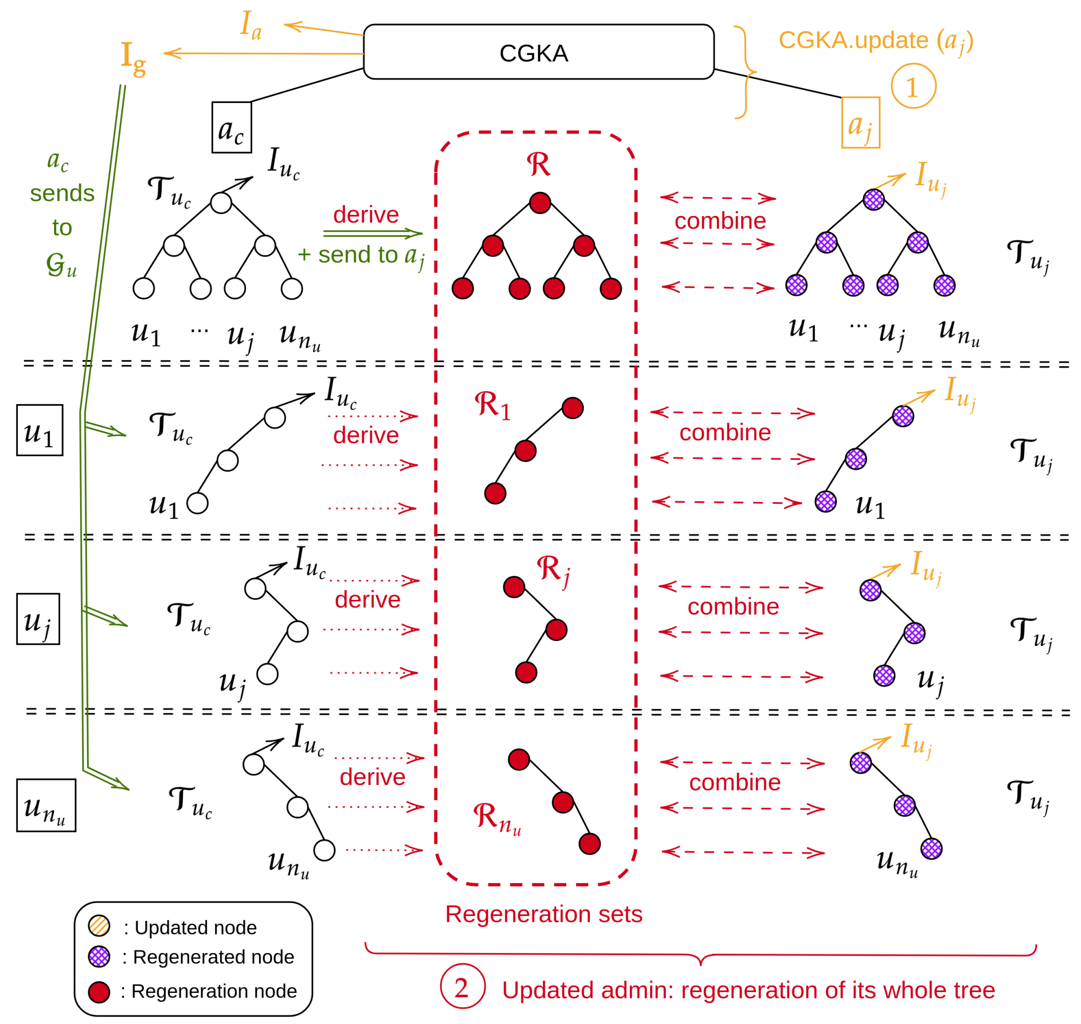

Figure 21: Description of the update-admin operation in SUMAC.

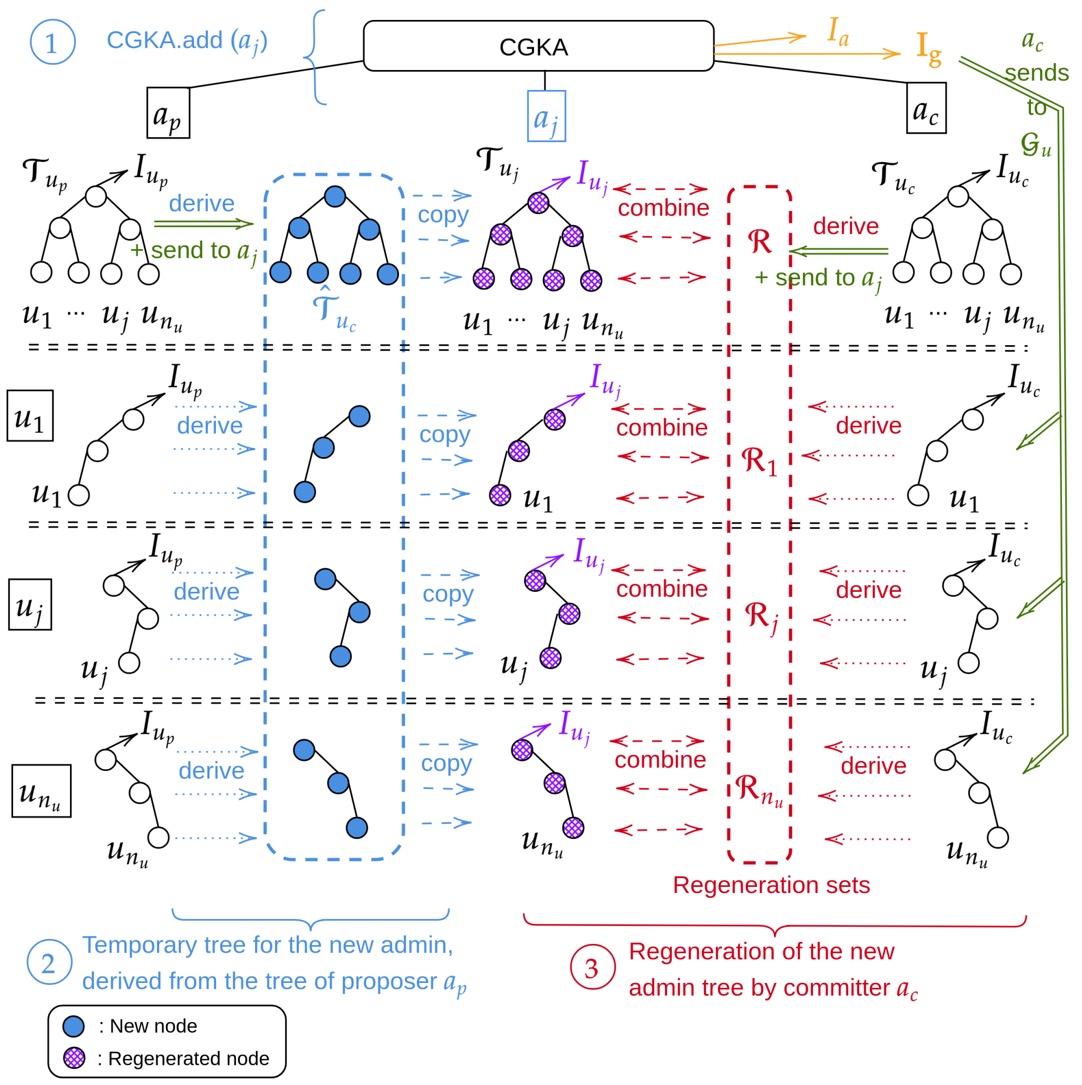

Figure 22: Description of the add-admin operation in SUMAC.

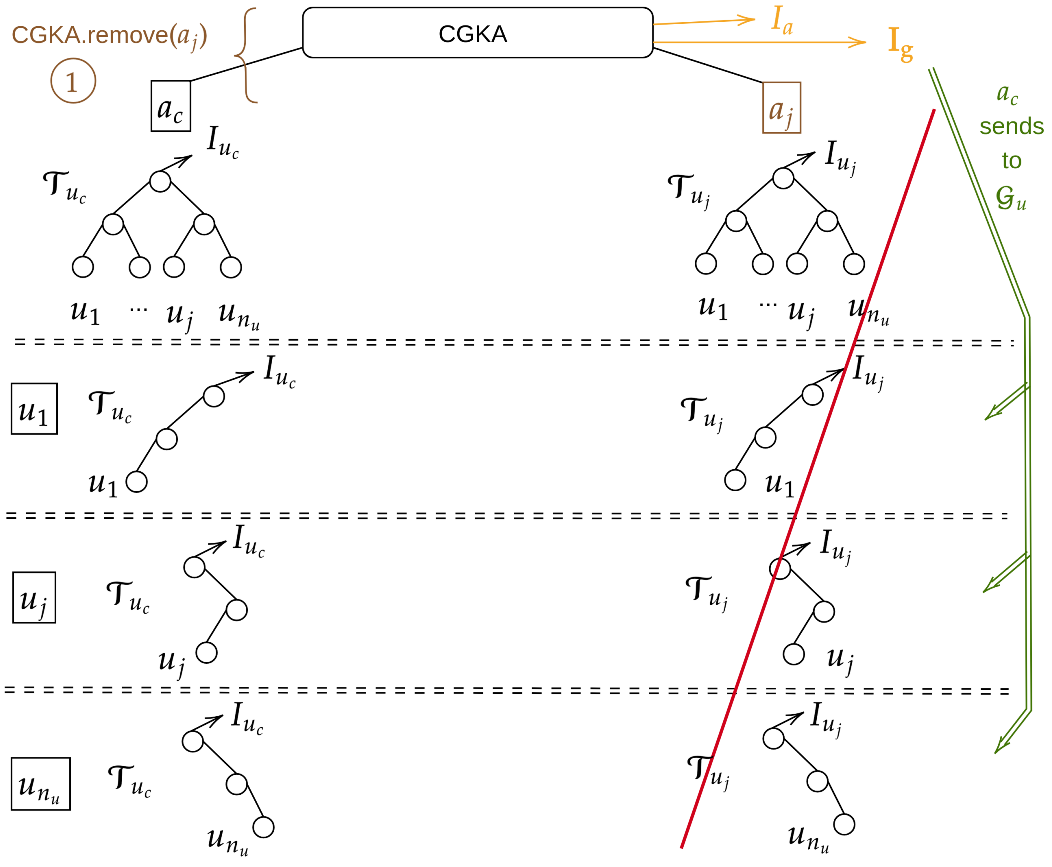

Figure 23: Description of the remove-admin operation in SUMAC.

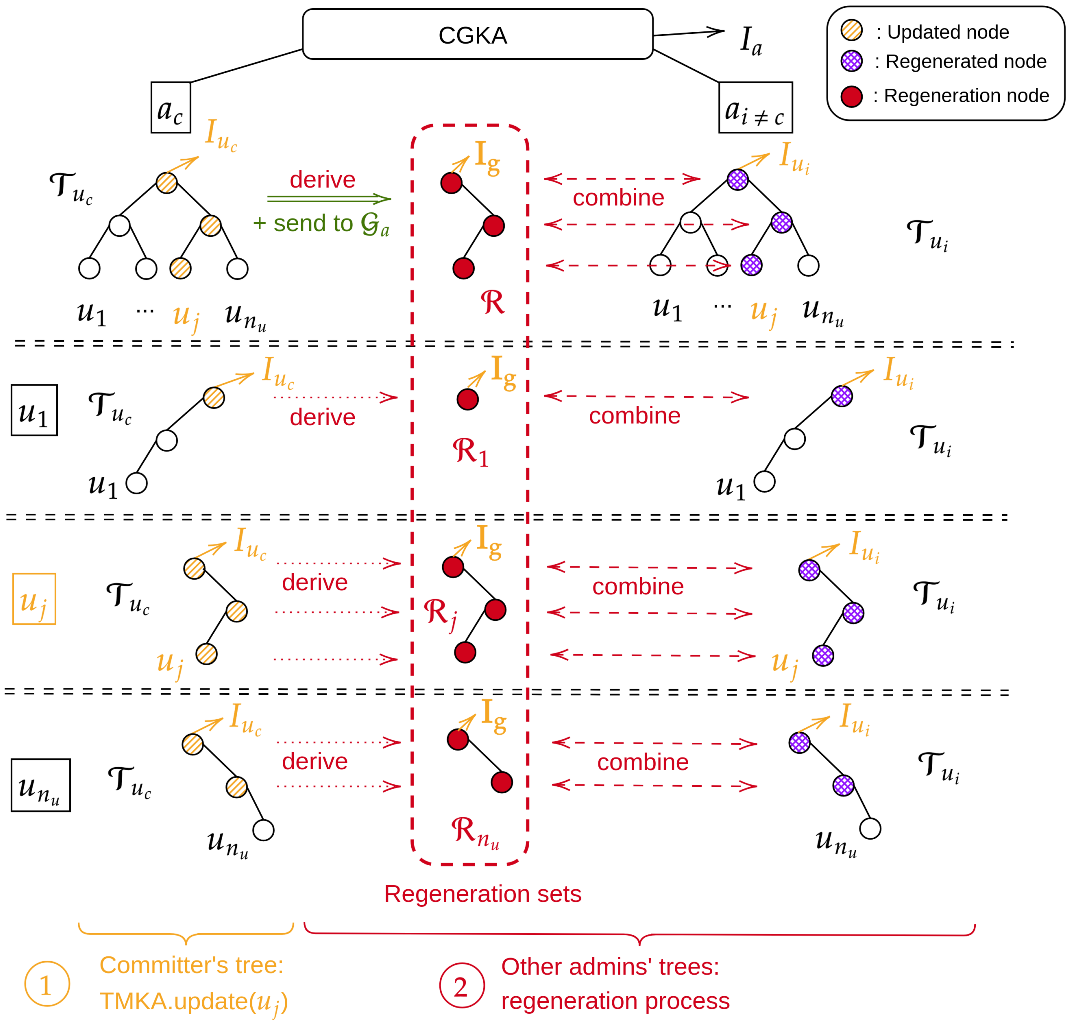

Figure 24: Description of the update-user operation in SUMAC.

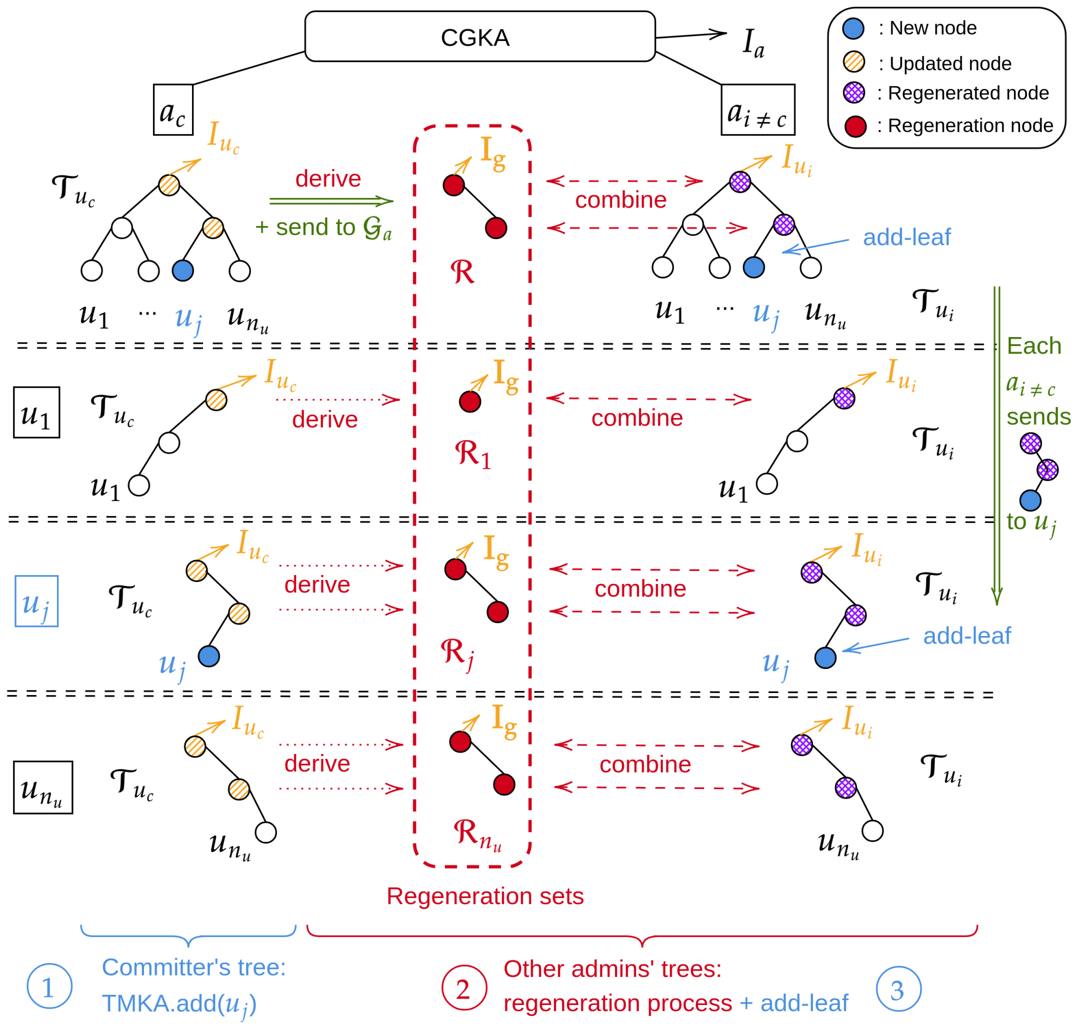

Figure 25: Description of the add-user operation in SUMAC.

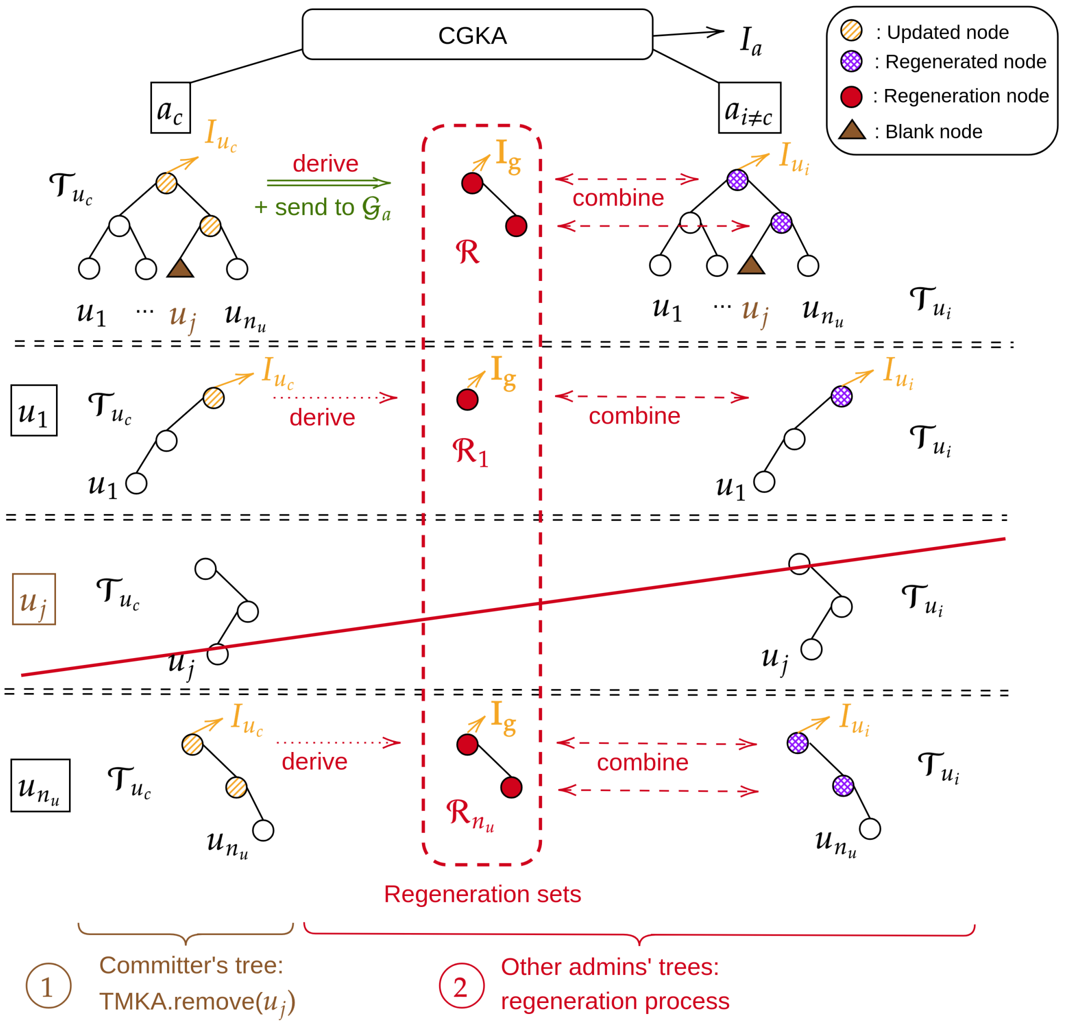

Figure 26: Description of the remove-user operation in SUMAC.