Quarantined-TreeKEM: a Continuous Group Key Agreement for MLS, Secure in Presence of Inactive Users

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

Abdul Rahman Taleb
ANSSI
Paris, France
abdulrahman.taleb@ssi.gouv.fr

ABSTRACT

The recently standardized secure group messaging protocol "Messaging Layer Security" (MLS) is designed to ensure asynchronous communications within large groups, with an almost-optimal communication cost and the same security level as point-to-point secure messaging protocols such as "Signal". In particular, the core subprotocol of MLS, a Continuous Group Key Agreement (CGKA) called TreeKEM, must generate a common group key that respects the fundamental security properties of "post-compromise security" and "forward secrecy" which mitigate the effects of user corruption over time.

Most research on CGKAs has focused on how to improve these two security properties. However, post-compromise security and forward secrecy require the active participation of respectively all compromised users and *all* users within the group. Inactive users – who remain offline for long periods – do not update anymore their encryption keys and therefore represent a vulnerability for the entire group. This issue has already been identified in the MLS standard, but no solution, other than expelling these inactive users after some disconnection time, has been found.

We propose here a CGKA protocol based on TreeKEM and fully compatible with the MLS standard, that implements a "quarantine" mechanism for the inactive users in order to mitigate the risk induced by these users during their inactivity period and before they are removed from the group. That mechanism indeed updates the inactive users' encryption keys on their behalf and secures these keys with a secret sharing scheme. If some of the inactive users eventually reconnect, their quarantine stops and they are able to recover all the messages that were exchanged during their offline period. Our "Quarantined-TreeKEM" protocol thus increases the security of original TreeKEM, with a very limited – and sometimes negative – communication overhead.

KEYWORDS

MLS, TreeKEM, CGKA, Quarantine, Forward Secrecy, Post-Compromise Security

1 INTRODUCTION

While point-to-point secure communication has reached a high degree of maturity with the development of end-to-end secure messaging (SM) protocols that have been thoroughly studied, group communication has suffered until recently from a lack of dedicated research. In practice, secure messaging applications that offer a functionality of group communication rely on *ad-hoc* protocols that are either less secure than their point-to-point counterpart (e.g. the SenderKey protocol, used by WhatsApp [1]) or that are quite inefficient, especially with a communication cost scaling linearly with the number n of group members.

To remedy this situation, the IETF has released in July 2023, after a five-year study, RFC 9420 [9] that standardizes "Messaging Layer Security" (MLS). This state-of-the-art Secure Group Messaging (SGM) protocol is designed to enable secure communication in large groups of users – up to tens of thousands members – with an almost-optimal communication cost.

The core component of MLS is its mechanism of authenticated key exchange between all members of a group, called a "Continuous Group Key Agreement" (CGKA) [5], which needs to be run continuously for security considerations.

The CGKA protocol that has been most thoroughly studied, and that was adopted in the final IETF standard, is TreeKEM [13]. Its architecture, close to the original ART protocol [16], relies on binary trees in order to exchange handshake data between n users with an almost-optimal complexity of $O(\log_2(n))$.

1.1 Security Properties of a CGKA

Among all the security properties that a CGKA must fulfill (cf. Section 2.4.1), two in particular – Post-Compromise Security (PCS) and Forward Secrecy (FS) – require an active participation of respectively all compromised users and all group members, who must update their keying material and the one of the tree's internal nodes above them. These properties are especially hard to ensure in a CGKA, due to the asynchronicity of the protocol and the fact that within a potentially large group, it appears unlikely that all users behave correctly by updating regularly their keying material.

1.1.1 Post-Compromise Security (PCS). This property represents the ability of a protocol to heal from the corruption of a group member, that leaked that user's private state, provided that the adversary remains passive after the end of that compromise.

Most papers aiming to improve the security of CGKA protocols focus on post-compromise security and try to minimize the original number n of rounds (n being the number of group members) necessary to heal a fully-compromised tree. For instance, the CoCoA protocol [3] allows concurrent updates in a single round, with a mechanism of prioritization between them, that permits to reach PCS in only $\lceil \log(n) \rceil + 1$ rounds. Going further, the alternate DeCAF protocol [2] reduces this healing complexity to $\lfloor \log(t) \rfloor + 1$ rounds when only t users among n are compromised.

But the most efficient method to ensure PCS is the "Propose & Commit" paradigm, which has been part of the MLS IETF working draft since version 8 [8]. This protocol allows a full healing of the binary tree in only two rounds, whatever the number of compromised users, yet at the cost of a non-negligible communication overhead (since the binary tree is temporarily destructured).

1.1.2 Forward Secrecy (FS). This fundamental security property states that non-compromised past communication cannot be jeopardized in the future by any user corruption. This property can be ensured at the scale of a session (in the case of a CGKA, by securing past epochs) or of a message, using symmetric ratchet to make the symmetric encryption key evolve after sending each encrypted message. Similarly to PCS, FS *at the scale of an epoch* relies on the fresh randomness brought by the key agreements performed by the CGKA. However, it suffers from the need to update *all* encryption keys in the tree (not only all users' keys but also the ones of all internal nodes).

To the best of our knowledge, the only work improving the original FS of TreeKEM is the RTreeKEM protocol of [4]. It provides a stronger forward secrecy than other CGKAs, by automatically updating – using a non-standard "updatable public-key encryption scheme"[1] – the encryption keys of all internal nodes and leaves that receive or emit any encrypted message.

1.1.3 Dealing with Inactive Users. However, none of these works deals with the issue of user behavior, which is yet at the root of a major security flaw. Indeed, even if a protocol can force online users to regularly update their keys and if the question of updating the internal nodes has already been addressed in various ways (e.g. by blanking entire "direct paths", from users to the root, in the "Propose & Commit" model or by somehow merging several concurrent path updates in [25], [3] or [2]), the case of users remaining offline for long periods is not considered, as it is seen intrinsic to the asynchronicity of the protocol. Since these inactive users no longer update their encryption keys, it only takes one of them to compromise the forward secrecy of the *entire group*. Similarly, a single corrupted inactive user is enough to undermine the whole group's post-compromise security. RFC 9420 identifies this problem but only recommends that users who have been offline for too long be removed from the group.

[1]This scheme is derived from the secretly key-updatable PKE from [20], that is used in a variant of our QTK protocol and is described in Section 2.3.

1.2 Our Contribution

We propose in this paper *Quarantined-TreeKEM (QTK)*, a TreeKEM-based CGKA protocol which mitigates the effects, both on forward secrecy and post-compromise security, of inactive group members who no longer update their keying material and the one of their direct path (i.e. the internal nodes above them).

Instead of passively waiting for that inactive users to be eventually expelled from the group, our protocol temporarily puts them aside, in what we call a "quarantine". We call "ghosts" such quarantined users. The randomly-chosen user who initiates this procedure (cf. Section 3.3 for details on the selection of this "quarantine initiator") for a certain ghost is responsible for blanking the latter's direct path and updating its encryption keys on its behalf, so that future handshake messages delivered by the Delivery Service are not encrypted with an old and potentially compromised encryption key known by this ghost, but with fresh keying material.

The use of a proxy to update another user's encryption keying material on its behalf has been proposed by the Tainted TreeKEM protocol [21] in order to add or remove users without having to blank their direct paths. However, Tainted TreeKEM does not improve TreeKEM's security and instead enhances the CGKA's efficiency by keeping a Ratchet Tree structured at all time.

Moreover, unlike a proxy in Tainted TreeKEM, the quarantine initiator in Quarantined-TreeKEM does not retain the (secret) decryption key belonging to the ghost user. Instead, the secret seed that was used to deterministically generate the ghost's encryption key-pair is split up using a secret sharing scheme and distributed to all group members. The ghost's secret seed and private key are then deleted from the initiator's internal state. In this way, the confidentiality of the ghost's secret key no longer relies on the security of a single user (the quarantine initiator) but on that of several active group members (the number of which depends on the secret sharing parameters).

When a ghost finally reconnects and updates its keying material, its quarantine automatically stops and the users that kept shares related to that ghost send them to it. The former ghost is therefore able to reconstruct the secret seeds corresponding to its quarantine keys and to eventually decrypt the handshake messages that it missed during its offline time and that remained buffered by the Delivery Service. In the few cases where the former ghost does not receive enough shares to reconstruct its quarantine keys – which is highly unlikely in large groups –, it remains able to reconnect to the group but it looses its quarantine history.

This quarantine mechanism strongly strengthens TreeKEM's post-compromise security by enabling this property at the beginning of a ghost's quarantine – after a period of inactivity that is fully controlled by the protocol – instead of after some hypothetical update of that inactive user, that may never happen until its eviction from the group.

Regarding forward secrecy, our protocol does not change the time at which this property is assured for the group[2]. Nevertheless, before forward secrecy is reached, QTK greatly decreases, in

[2]Indeed, forward secrecy needs the update of every group member – including inactive users – after the generation of the challenge group key. As a reconnecting ghost recovers its quarantine history, forward secrecy is assured only when all the ghosts in the group have either been removed from the group or have already recovered their quarantine shares and updated.

comparison with TreeKEM, the chances of an adversary to successfully attack past communication by corrupting inactive users. These chances are captured by the concept of "critical window", issued from [21] and detailed in Section 4.2 and Figure 5, that corresponds to the period of vulnerability of a user. The critical window of a ghost with QTK is much smaller than the one of an inactive user with TreeKEM, as depicted by Figure 6, which enhances the protocol's security.

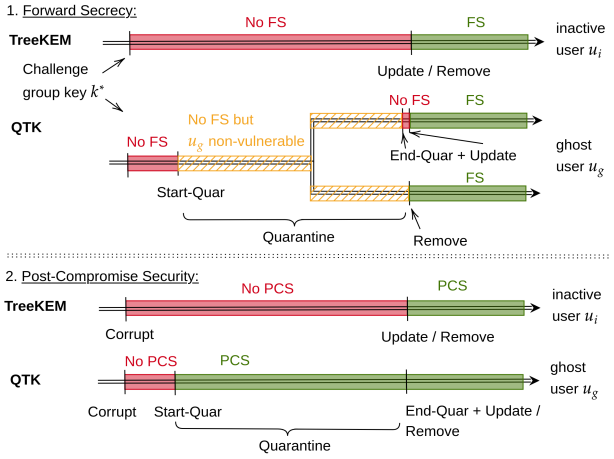

Figure 1: Comparison of the forward secrecy and post-compromise security of TreeKEM and our QTK protocol, with a focus on an inactive user that weakens the security of the entire group. Post-compromise security is achieved earlier with QTK and forward secrecy is enhanced by reducing the period of vulnerability of that inactive user.

As in MLS standard and several recent works [21], [3], [2], the security of QTK is analyzed in this paper by considering a partially active adversary that is able to corrupt any user and leak all of its secret elements except for its private signature key, and consequently cannot impersonate these compromised users. The justification of this adversarial model is detailled in Section 4.1, while the main risks induced by a fully active adversary on our quarantine mechanism and some solutions to overcome them are briefly discussed in Appendix A.1.

1.3 Outline of the Paper

We describe in Section 3 how our QTK protocol works, and in particular, how a quarantine is carried out from start to end and how a secret sharing scheme is used to distribute secret information among the group.

Security is studied in Section 4 in a game-based model inspired from [21]. We show that our protocol is CGKA-secure in this framework and that the main differences with standard TreeKEM are the periods during which users are vulnerable to corruption, through the aforementioned concept of critical window (cf. Section 4.2).

Section 5 details the performances of our protocol in terms of communication cost. After theoretical computations, the communication overhead induced by a ghost's quarantine, for realistic parameters, is given at the end of this section in order to show the feasibility of our QTK CGKA in real-life use cases. Indeed, it appears that the overhead of a quarantine is very limited and is sometimes less costly than the regular updates performed by an active user.

Finally, we present in Appendix B an enhancement to our basic QTK protocol, called "jointly-implemented quarantine", that further increases the security offered by QTK by using several users instead of a single one for each operation of quarantine initialization or update.

Furthermore, we propose as additional content to our study an open-source implementation of our QTK protocol, forked from an official implementation of MLS in Kotlin. This program will be made public at the time of publication of this paper and can be provided to reviewers, on request.

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 ":=".

".||." is used for the concatenation operation. $|\mathcal{S}|$ denotes the cardinality of a set $\mathcal{S}$. $\lfloor\ \rceil$ and $\lceil\ \rceil$ respectively denote the rounding and ceiling values of a decimal number. $\log()$ denotes the logarithm in base 2.

In the context of a secret sharing scheme, $[x]_i$ denotes the specific share i associated to the value x; $[x] = ([x]_i)_{i\in[\![0,n-1]\!]}$ represents the entire collection of n shares associated with x.

In order to dissociate internal nodes from leaves in the Ratchet Tree of a CGKA, we note x_{v_i} the value x associated to an internal node v_i whereas the value x related to a leaf ℓ_i (a.k.a a user u_i) is simply noted x_i.

2.2 Secret Sharing [24]

Let us recall the definition of a secret sharing scheme, issued from [14].

Definition 2.1 (Threshold Secret Sharing Scheme). A (t, m)-(threshold) secret sharing scheme over a finite set $\mathcal{Z}$ is a pair of efficient algorithms (Distr, Comb) that respectively perform the following tasks:

- **Distributing the secret**: Distr is a probabilistic algorithm that splits up a secret $\alpha \in \mathcal{Z}$, according to parameters t, m (which respectively denote the recovery threshold and the total number of shares to emit), into a collection of m shares $([\alpha]_i)_{i\in[\![0,m-1]\!]}$ such that at least t of them are necessary to reconstruct the shared secret α.

$$\boxed{[\alpha] = ([\alpha]_0, \cdots, [\alpha]_{m-1}) \leftarrow Distr(\alpha, t, m)}$$

- **Reconstructing the secret**: Comb is a deterministic combination algorithm that reconstructs the shared secret α with a subset $([\alpha]_i)_{i\in\mathcal{I}\subseteq[\![0,m-1]\!]}$ of the share collection, of size at least t.

$$\boxed{\alpha := Comb(([\alpha]_i)_{i\in\mathcal{I}}, \mathcal{I})}$$

A secret sharing scheme must abide by the correctness property, which states that for every secret $\alpha \in \mathcal{Z}$, for every possible output $[\alpha]$ of the distributing algorithm Distr(α, t, m) and every subset $\mathcal{I}$

of $[\![0, m-1]\!]$ of size at least t, we have:

$$Comb\big((\llbracket\alpha\rrbracket_i)_{i\in\mathcal{I}}, \mathcal{I}\big) = \alpha$$

Nota: We only consider in this paper:

- **Perfect** secret sharing schemes, for which any collection of $t-1$ shares related to a secret $\alpha \in \mathcal{Z}$ gives strictly no information about that shared secret. Consequently, for any unbounded adversary $\mathcal{A}$ trying to recover α given a subset $(\llbracket\alpha\rrbracket_i)_{i\in\mathcal{I}'\subseteq[\![0,m-1]\!]}$ of size strictly smaller than t, we have:
$$Pr\left[\alpha \leftarrow \mathcal{A}\big((\llbracket\alpha\rrbracket_i)_{i\in\mathcal{I}'}, \mathcal{I}'\big)\right] = \frac{1}{|\mathcal{Z}|}$$
- **Ideal** secret sharing schemes: these are perfect schemes that additionally generate shares belonging to the same set $\mathcal{Z}$ as the shared secret, thus with identical sizes.

2.3 Secretly Key-Updatable Public Key Encryption

Informally, a secretly key-updatable public key encryption scheme (skuPKE) (originally defined in [20]) is a PKE whose public and private keys can be updated by independently generated update elements (Θ, θ). The update element Θ for the public encryption key pk can be publicly disclosed, whereas the update element θ for the private decryption key sk must remain secret.

Definition 2.2 (Secretly Key-Updatable Public Key Encryption [20]). A secretly key-updatable public key encryption scheme (skuPKE) consists of six polynomial-time algorithms:

- **KeyGen** takes as input the security parameter λ and probabilistically outputs a couple of public and private keys (pk, sk).
- **Enc** takes as input a public key $pk \in \mathcal{PK}$ and a plaintext $m \in \mathcal{M}$ and probabilistically yields a ciphertext c.
- **Dec** takes as input a secret key $sk \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$ and deterministically generates a plaintext m. As for a regular PKE, a skuPKE scheme is *correct* if we have: $\forall(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$, $\forall m \in \mathcal{M}, \text{Dec}(sk, \text{Enc}(pk, m)) = m$.
- **UpdGen** takes as input the security parameter λ and probabilistically outputs a couple of public and private update elements (Θ, θ).
- **UpdPk** takes as input a public key $pk \in \mathcal{PK}$ and a public update element Θ and yields an updated public key pk'.
- **UpdSk** takes as input a private key $sk \in \mathcal{SK}$ and a private update element θ and outputs an updated private key sk'.

$(pk, sk) \leftarrow \text{KeyGen}(1^\lambda)$	$(\Theta, \theta) \leftarrow \text{UpdGen}(1^\lambda)$
$c \leftarrow \text{Enc}(pk, m)$	$pk' := \text{UpdPk}(pk, \Theta)$
$m := \text{Dec}(sk, c)$	$sk' := \text{UpdSk}(sk, \theta)$

2.4 TreeKEM CGKA Protocol

We give hereunder a brief description of how TreeKEM – as standardized in RFC 9420 [9] – works as a Continuous Group Key Agreement (CGKA) protocol.

2.4.1 Continuous Group Key Agreement. A CGKA is a sub-protocol of a "secure group messaging" protocol, that aims to securely generate a group key which is common to all group members and evolves over time in order to provide the security properties of forward secrecy and post-compromise security. The definition below is adapted [4] in order to take into account TreeKEM's Propose & Commit paradigm.

Definition 2.3 (Propose & Commit CGKA). A CGKA with the Propose & Commit Paradigm is a tuple of the following algorithms:

- **Initialization**: user u_i creates its initial state γ_i:
$\boxed{\gamma_i \leftarrow \textbf{init}(u_i)}$
- **Group Creation**: user u_i, with state γ_i, creates a new group that must include users from the list $G = (u_i)_{i\in[\![1,n]\!]}$. A message welcome W is sent to all members from G, with the information necessary to join the group: $\boxed{(\gamma_i', W) := \textbf{create}-\textbf{group}(\gamma_i, G)}$
- **Propose**: user u_i proposes a change to the group's state through an action $a \in \mathbb{A}$, with $\mathbb{A} \supseteq$ {Add, Remove, Update} the set of actions authorized by the CGKA. In particular:
  - **add**(u_j): u_i proposes to add user u_j to the group;
  - **remove**(u_j): u_i proposes to remove u_j from the group;
  - **update**: u_i updates its own encryption keying material (the one of its leaf) and generates an updated state γ_i'.

  User u_i then broadcasts a Proposal message P to the entire group: $\boxed{(\gamma_i', P) \leftarrow \textbf{propose}(\gamma_i, a\ [, u_j])}$
- **Commit**: when receiving a set of p proposal messages $\mathbb{P} = \{P_i\}_{i\in[\![1,p]\!]}$, user u_i validates them and updates its own encryption keying material and the one of its direct path, generating a new group key k. It then updates its state into γ_i' to take into account that changes, and broadcasts a Commit message C as well as (potentially) a Welcome message for the new group members:
$\boxed{(\gamma_i', k, C\ [, W]) \leftarrow \textbf{commit}(\gamma_i, \mathbb{P})}$
- **Process**: user u_i processes a Commit message C or a Welcome Message W it has received from a committer, updates accordingly its own state and computes the new group key k resulting from these changes:
$\boxed{(\gamma_i', k) := \textbf{process}(\gamma_i, m \in \{C, W\})}$

A CGKA must fulfill the following properties, stated informally below and evaluated in the security game of Section 4.1.2.

- **Correctness**: every user in the group must compute the same group key.
- **Privacy**: a group key is indistinguishable from a random value for an adversary who has access to the transcript of handshake messages exchanged within the group until the generation of that group key.
- **Forward secrecy** and **post-compromise security**, as described in Section 1.1.

2.4.2 Ratchet Tree. In order to optimize the communication cost between group members, TreeKEM implements an architecture based on a binary tree called "Ratchet Tree", where users are at the leaves and the group key is elaborated at the root. Similarly to

TreeKEM, we consider in this paper a descending full binary tree, where the two nodes beneath another node are called its "children" and the one above is its "parent".

We explain beneath some tree notions that are used in TreeKEM to perform dynamic tree operations such as updates.

Node's State. Each node of this Ratchet Tree, except for the root, is associated with a local state with public and private components.

- The public state ${}^{p}\gamma$ comprises, among other elements
  - for an internal node v: its public encryption key pk_v;
  - for a user (leaf) u_i: its public encryption and signature keys pk_i and spk_i, with the related credentials. It also includes the signature, under the user's private signature key, of the other fields of that public state.
- The private state ${}^{s}\gamma$ contains:
  - the group key and all the group secrets derived from it;
  - the private encryption keys of that node and of its filtered direct path, as well as the temporary secret elements (leaf secret, path secrets) associated with that keys.

As we see in Section 4.1.1, the private state of a leaf does not comprise the user's private signature key. This one must indeed be separated from the other private elements of that user and stored in a secure enclave. This compartmentalization is of importance in case of user corruption.

Blank Nodes. Deleted nodes from TreeKEM's Ratchet Tree are not removed – since the latter must remain a **full** binary tree, with two children for each internal node – but their state is deleted instead. Such empty nodes are called "blank" and do not take part in TreeKEM's processes until they are filled again.

Resolution of a Node. The resolution of a node v from a binary tree is a set of nodes defined as follows:

- if v is a non-blank node, then $Res(v) = \{v\}$;
- if v is a blank leaf, then $Res(v) = \emptyset$;
- if v is a blank internal node, then $Res(v) = \cup_{v' \in Children(v)} Res(v')$.

(Filtered) Direct Path and Copath of a Leaf. A user u_i's direct path is composed of all the ancestors of the leaf associated with that user, up to the root. Its filtered direct path, written $\mathcal{P}_i$, is its direct path whose nodes that have a child with an empty resolution are removed. A user's copath, $\mathcal{CP}_i$, contains the siblings of the direct path's nodes.

2.4.3 *Updates with TreeKEM.*

The update of the encryption keying material is implemented differently in TreeKEM whether it belongs to a user (i.e. a leaf) or an internal node.

Indeed, as stated in Definition 2.3, all tree operations are performed in two rounds with the "Propose & Commit" paradigm from TreeKEM:

- a first one where any user is free to submit *proposals* (adding new users, removing current group members, updating its own keying material...);
- a second one where the valid proposals are grouped together and implemented within a *commit* by a single user, called "committer".

Update of the Committer's Filtered Direct Path. During a "commit" process, as shown by Figure 2, the committer randomly draws a secret seed called "leaf secret"; this one is derived, with a key derivation function, into a "node secret" that serves as a seed to deterministically generate a fresh encryption key-pair.

In parallel, the leaf secret is derived into another secret ps_{v_1}, called a "path secret", that is associated with this leaf's parent v_1. This path secret ps_{v_1} is itself derived into a node secret to deterministically generate an encryption key-pair for the benefit of that leaf's parent v_1. It is then derived once again into a new path secret ps_{v_2}, related to another node v_2, higher in the leaf's filtered direct path, and so on, up to the tree root.

The group key k is then computed by deriving the root's path secret ps_{root}.

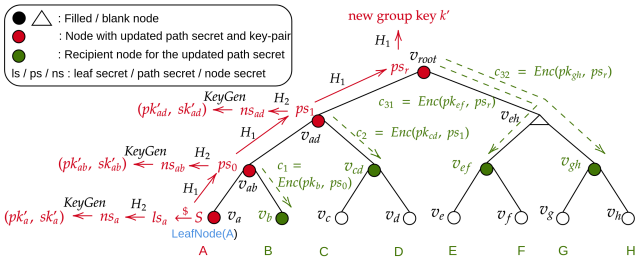

Figure 2: Update, with TreeKEM, of a user's filtered direct path (here user A). This process updates the encryption key-pairs of that user and of all its ancestors; it also generates a new group key.

Broadcast of a Commit Message to the Ratchet Tree. After updating its encryption key-pair, its filtered direct path and the group key, the committer u_c must transmit to the other group members the information they need to compute the new group key. To do so, the committer generates a commit message C that is broadcasted to the whole group (through a central server that simply plays a role of an untrusted Delivery Service).

This commit message C consists of:

- the list of proposals that the commit implements ($\mathbb{P}$);
- the updated (signed) public local state ${}^{p}\gamma'_c$ of the committer;
- the new public encryption keys $(pk'_{v_p})_{v_p \in \mathcal{P}_c}$ from the committer's filtered direct path;
- the path secrets of the nodes $v_p \in \mathcal{P}_c$ from the committer's filtered direct path, encrypted under the public keys of the nodes v_r belonging to the resolution $\mathcal{R}_{v_p}$ of v_p's child on the committer's copath.

$$C(u_c) = \mathbb{P} \,||\, {}^{p}\gamma'_c \,||\, \left(pk'_{v_p}\right)_{v_p \in \mathcal{P}_c} \,||\, \left(\mathrm{Enc}(pk_{v_r}, ps_{v_p})\right)_{v_p \in \mathcal{P}_c}^{v_r \in \mathcal{R}_{v_p}}$$

2.4.4 *Tree Evolution and Epochs.*

The evolution of the group over time is represented by the notion of "epoch". Each epoch corresponds to a given state of the user group, with a certain group key. Each time this group state is modified by a commit, the group key evolves and the epoch is incremented of one unit.

We now describe our QTK protocol, with its associated mechanism of "quarantine" applied on inactive users.

3 QTK PROTOCOL

Definition 3.1 (Quarantine TreeKEM (QTK)). The Quarantine TreeKEM protocol is a TreeKEM-based CGKA, associated with a (t, m)-perfect secret sharing scheme[3], that implements a mechanism of quarantine for inactive users – called "ghost users" – within the group.

This quarantine process updates the ghosts' keying material on their behalf and uses the secret sharing scheme to collectively secure the secret information related to these updates.

In this paper, we describe QTK with processes from the TreeKEM protocol standardized by IETF [9]. However, our protocol remains compatible with most – if not all – TreeKEM-derived CGKAs that are proposed in the literature ([3], [2], [4], [25], [21]…).

3.1 Message Delivery Mode

We detail two variants of our QTK protocol, that depend on the ability of the Central Server's Delivery Service to perform fine-grained message-delivery:

- **broadcast-only setting**: all handshake messages are broadcasted to the entire group;
- **server-aided setting**: the regular TreeKEM messages (proposals, commits...) are broadcasted, but two types of messages specific to our protocol ("Share Distribution Message" and "Share Recovery Message", cf. below) are only sent to the adequate recipients.

The server-aided setting, already studied in the CGKA literature [17], [18], [6], permits to greatly improve the communication cost, especially in large groups, but it is not as generalizable as a broadcast-only protocol – such as the standardized MLS – where no assumption is made on the Central Server's capacities.

3.2 QTK Public States

In TreeKEM, each user keeps an updated view of the whole Ratchet Tree, and in particular, of all other group members, through their public states ${}^{p}\gamma$ (a.k.a "leaf nodes").

In our QTK protocol, this public state includes two additional fields necessary to conduct a quarantine:

- The first one (e_{pk}) corresponds to the epoch of last update of the user's encryption key-pair, that the committer of every epoch checks when creating its commit, in order to detect inactive users that are to be quarantined (cf. Section 3.3).
- The second one (e_{quar}) is the epoch corresponding to the start of the quarantine. This field allows committers to check whether a ghost reaches the maximum quarantine duration δ_{quar} that is parametrized at the application level. For active users, this field remains empty.

3.3 Start of a Quarantine

3.3.1 Initialization Process. At each commit, the unique committer[4] checks that the encryption keys of *all* other active users in the tree have not exceeded a maximum age defined by the parameter δ_{inact}. If, at a given epoch e^i, certain users have keys that are too old ($e^i - e_{pk} \geq \delta_{inact}$), they are declared "ghost users" and the committer is responsible for quarantining them.

We note $\mathcal{G}^i$ the set of ghost users at epoch e^i, and $\mathcal{N}_{\mathcal{G}}{}^i \subseteq \mathcal{G}^i$ the subset of ghost users starting their quarantine at epoch e^i.

The quarantine initialization process, at epoch e^i, consists of the following steps:

(1) The committer u_c for that epoch, as "quarantine initiator", updates the ghost list $\mathcal{G}^{i+1}$ for the epoch e^{i+1} that will follow the commit, by adding new ghosts and removing ghosts that are reconnecting or that have reached the limit of their quarantine duration and must be removed from the group at epoch e^{i+1}.

(2) The committer blanks the direct paths of the new ghosts so that they are (functionally) directly linked to the tree root.

(3) For each of the new ghosts, the committer randomly draws a seed from a seed space $\mathcal{S}$, that is used to deterministically generate a fresh encryption key-pair:

$$\forall u_g \in \mathcal{N}_{\mathcal{G}}{}^{i+1},\ s_g^{i+1} \xleftarrow{\$} \mathcal{S}$$
$$(pk_g^{i+1}, sk_g^{i+1}) := \mathrm{KeyGen}(1^\lambda; s_g^{i+1})$$

(4) With a (t, m)-perfect secret sharing scheme, the committer splits up the seeds into m shares (the number of shares being defined by the share distribution method), with a threshold $t < m$ whose choice is a trade-off between security and availability of the protocol[5]:

$$\forall u_g \in \mathcal{N}_{\mathcal{G}}{}^{i+1},\ [s_g^{i+1}] \leftarrow Distr(s_g^{i+1}, t, m)$$
$$\text{with } [s_g^{i+1}] := \{[s_g^{i+1}]_0, \cdots, [s_g^{i+1}]_{m-1}\}.$$

(5) The committer records in its private state ${}^{s}\gamma_c$ the first share $[s_g^{i+1}]_0$ of each new ghost and distributes the remaining $m-1$ shares in the tree, according to a share distribution process detailed below.

(6) The committer includes in its pending commit message the new ghosts' public keys, along with its own new key and the new ones from its direct path $\mathcal{P}_c^{i+1}$.

(7) The committer deletes from its private state the secret key sk_g^{i+1}, seed s_g^{i+1} and shares $([s_g^{i+1}]_i)_{i \in [\![1, m-1]\!]}$ of each ghost.

3.3.2 Share Distribution in the Ratchet Tree. The distribution within the tree of the shares previously emitted depends on the message delivery mode (cf. Section 3.1) and on the share distribution method. Figure 3 compares the two share distribution methods, detailed below, for an unbalanced Ratchet Tree.

In the *broadcast-only setting*, the default share distribution method is adapted to the architecture of a binary tree, and therefore optimizes the communication cost of this exchange. We however propose an alternate method, called "horizontal share distribution" that must be used when the conditions are not conducive to that

[3] Cf. Section 2.2 for additional details on that primitive.

[4] TreeKEM selects the committer for a given epoch as the first group member trying to exchange content data after a proposal has been issued by another user and has not yet been taken into account in a commit.

[5] Indeed, with a high threshold, the secret sharing scheme needs most of the shares in order to reconstruct the secret. It is therefore more secure than with a low threshold; however the probability to be unable to legitimately recover that secret increases, at the expense of the scheme's availability. We underline that in even in case of failure of the secret sharing reconstruction, the ghost remains able to reconnect to the group.

default method (when the number of users is too low to generate a number of shares greater than or equal to the secret sharing threshold). In the *server-aided setting*, on the other hand, only the horizontal share distribution method can be implemented.

Default Method: Shares with Path Secrets. By default, shares are joined to the path secrets created by the committer's path update, which implies that they are sent to the same recipients as these ones[6]. Consequently, the same share is distributed to all users[7] beneath each node belonging to the resolution of the nodes in the committer's copath.

Consequently, the number of users who keep a same share strongly varies, according to their relative positions in the tree with respect to the committer. Indeed, the committer and its sibling – if any – are the only keepers of the first two shares $[s_g^{i+1}]_0$ and $[s_g^{i+1}]_1$ associated with a new ghost u_g, whereas on the other end of the tree, the whole opposite subtree at the root of the tree (filled with up to $\frac{n}{2}$ users for a full binary tree) is given the same share $[s_g^{i+1}]_{m-1}$.

When the structure of the Ratchet Tree differs from a full, blank-node-free, binary tree, the number of nodes in the committer's filtered direct path may vary from 1 to $n-1$ (depending on the tree balance and on the committer's location in this tree). As this value also represents the number of path secrets, and therefore the number of shares to transmit, in a worst-case scenario where this path only comprises one node[8], the default share distribution method only generates two shares: one corresponding to that single node in the committer's filtered direct path, and one for the committer itself. Clearly, this number of shares is too low to be acceptable, especially in order to implement a recovery threshold for the secret sharing scheme.

Consequently, an alternative share distribution method must be used when the number of shares emitted with the default method falls below a minimum value $m_{min} > 2$, defined at the application level.

Alternate Method: the Horizontal Share Distribution. In this case, the committer no longer tries to include the shares in the encrypted path secrets. Instead, shares are encrypted with the public keys of the internal nodes belonging to a same (horizontal) level $\mathcal{L}$, that is chosen so that its number of nodes is greater than or equal to the minimum value m_{min}. In the end, a share is received by all active users under the same node from that level. The committer (and all users depending on the same node from level $\mathcal{L}$) is assigned the first share $[s_g]_0$ and the other shares are then attributed from left to right, from that starting point.

With a broadcast-only protocol, these encrypted shares are joined to the commit message, which saves the cost of additional signatures. In the server-aided paradigm, each share is sent in a separate "Share Distribution Message", which comprises:

- the encrypted share;

[6]The only exception is if the only recipient of an encrypted path secret is the ghost user itself. In this case, the path secret is sent anyway, but without any share attached, which decreases by one the total number of emitted shares.

[7]Active group members and ghost users as well, except for the ghost associated with the shares.

[8]This scenario may happen even with a large number of users and even for a left-balanced binary tree, if we have a group of $2^x + 1$ users and if the committer happens to be the single leaf of the right root's subtree.

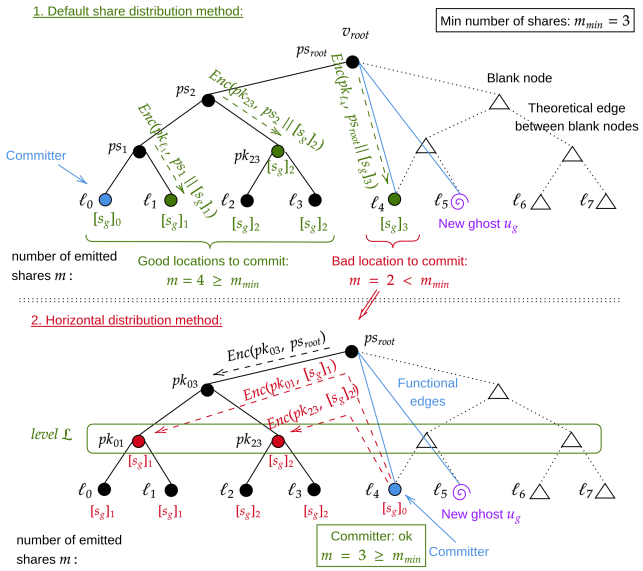

Figure 3: Compared share distribution methods, with the default method (top) and the horizontal share distribution (bottom). The latter must be used when the number m of emitted shares falls under the minimum allowed value m_{min}.

- the index of the internal node under whose key the share is encrypted;
- the sender's signature.

3.3.3 Commit Message. A commit message with a quarantine initialization, in the broadcast-only setting, therefore comprises the following parts (in blue: additional elements compared to a classical commit message from TreeKEM):

- The committer's new signed public state ${}^p\gamma_c^{i+1}$, which includes the committer's updated public encryption key pk_c^{i+1}.
- The updated public encryption keys of the committer's filtered direct path $v_p \in \mathcal{P}_c^{i+1}$ and of the new ghosts $u_g \in \mathcal{N_G}^{i+1}$:
$$(pk_{v_p}^{i+1})_{v_p \in \mathcal{P}_c^{i+1}} \,||\, (pk_g^{i+1})_{u_g \in \mathcal{N_G}^{i+1}}$$
- The leaf indices of the new ghosts whose shares are sent – in the same order – in this commit message: $(\ell_g)_{u_g \in \mathcal{N_G}^{i+1}}$.
- *With the default share distribution method*: For each node $v_r \in \mathcal{R}_{v_p}^{i+1}$ from the resolution of the committer's copath, the encryption, under v_r's public key, of:
  - the adequate path secret $ps_{v_p}^{i+1}$ (which is the seed of node $v_p \in \mathcal{P}_c^{i+1}$, the closest ancestor of v_r on the committer's direct path);
  - a share – dedicated to v_r – for the secret seed s_g^{i+1} of each of the $\nu = \left|\mathcal{N_G}^{i+1}\right|$ new ghosts $u_g \in \mathcal{N_G}^{i+1}$:

$$\left(Enc\left(pk_{v_r}^{i}, ps_{v_p}^{i+1} \,||\, [s_1^{i+1}]_{v_r} \,||\, \cdots \,||\, [s_\nu^{i+1}]_{v_r}\right)\right)_{v_r \in \mathcal{R}_{v_p}^{i+1}}^{v_p \in \mathcal{P}_c^{i+1}}$$

- *With the horizontal share distribution method*: The encryptions of path secrets and shares are dissociated. Consequently, for each node v_r from the resolution of the committer's copath and each node v_ℓ of the tree level $\mathcal{L}^{i+1}$ chosen for the horizontal distribution, we have:
  - the encryption, under v_r's public key, of the corresponding path secret $ps_{v_p}^{i+1}$;
  - the encryption, under v_ℓ's public key, of one share associated with *each* new ghost $u_g \in \mathcal{N_G}^{i+1}$.

$$\left(\left(Enc\left(pk_{v_r}^i, ps_{v_p}^{i+1}\right)\right)_{v_r \in \mathcal{R}_{v_p}^{i+1}}^{v_p \in \mathcal{P}_c^{i+1}}, \left(Enc\left(pk_{v_\ell}^i, [s_1^{i+1}]_{v_\ell} \,||\, \cdots \,||\, [s_\nu^{i+1}]_{v_\ell}\right)\right)_{v_\ell \in \mathcal{L}^{i+1}}\right)$$

3.3.4 Shareholder Rank. In order to avoid redundancy at the stage of share recovery, at the end of the ghost's quarantine, we implement a process to prioritize shareholders that keep the same share, through the concept of "shareholder rank".

An active user who receives shares related to one or several ghosts' quarantine(s) is called a "shareholder". A group of shareholders that have received the same share is called a "shareholder family" with respect to that share. As every user has a complete view of the Ratchet Tree, including the location of every other group member, a shareholder is able to determine – with both share distribution methods – its shareholder family related to the share it has received, and its own position within this family.

The shareholder rank corresponds to a shareholder's location in its shareholder family, starting from left to right.

3.3.5 Shareholder Share Recording. A shareholder u_s records in its private state ${}^s\gamma_s$ information about the share(s) it has received, as a list of tuples of the following form:

- the ghost's leaf index: ℓ_g;
- the ghost's share received: $[s_g^{i+1}]_{ind}$;
- its shareholder rank related to this share: rk;
- the index associated with the share: ind;
- the creation epoch of this share: e^{i+1}.

All these fields, except the ghost's leaf index and the share itself, are locally computed by the shareholder, thanks to its complete view of the Ratchet Tree, with no need of extra communication.

Every time a ghost quarantine expires (either with a successful reconnection or with a removal from the group – after reaching the maximum quarantine duration δ_{quar}), shareholders delete from their share recording all the data associated to this ghost.

A shareholder considers that a ghost successfully completed its quarantine recovery when it receives the *second* Update proposal[9] from that ghost after its reconnection. On the other side, a ghost removal is notified by a Committer[10] in a formal Remove operation inluded in the commit message.

[9]The reconnecting ghost updates a first time when going back online, and a second time after receiving all its shares and recovering the associated quarantine keys.
[10]Committers not only check the seniority of all users' encryption keys, they also verify that ghosts do not exceed the maximum quarantine duration δ_{quar}, thanks to the "quarantine start epoch" field in the public state of the latter.

3.4 Course of a Quarantine

During its quarantine, a ghost remains part of the group, and as such, receives all handshake and application messages that are exchanged within the group – except for its own shares.

Quarantine Key Update. When preparing a commit, a committer checks the age of the ghosts' quarantine keys thanks to the "quarantine start epoch" field of the ghosts' public state (cf. Section 3.2). When a quarantine key gets older than a limit given by a parameter called $\delta_{quar-upd}$, a process of "quarantine key update" is initiated.

This process is similar to the initialization of a quarantine, except that the committer in charge of the update may not be the one who started the quarantine. As in a quarantine initialization, the committer (called an updater) draws a random secret seed on behalf of the ghost, generates an encryption key-pair, splits up the seed into shares that are distributed to the online users. Once again, these new shareholders may not be the same as the one from the quarantine start. Consequently, depending on their activity in the group, active users may record zero, one or several shares associated to the quarantine of a given ghost.

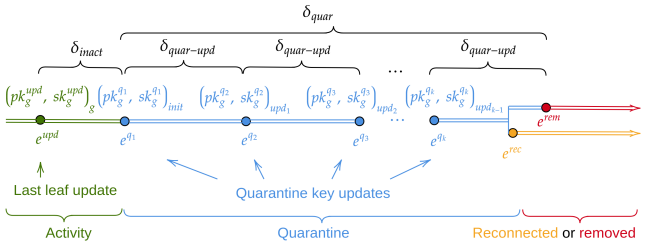

Figure 4: Timeline of a quarantine with QTK protocol. $(pk_g^i, sk_g^i)_x$ denotes a ghost's encryption key-pair of generation i, updated by user x. δ_{inact} is the duration, after the last key update, before initiating a quarantine; δ_{quar} is the length of this quarantine before removing the ghost, and $\delta_{quar-upd}$ is the period of quarantine key update.

3.5 End of a Quarantine

When a ghost user u_g finally reconnects at epoch e^{rec}, the end-of-quarantine process is automatically activated.

(1) The former ghost u_g – which, at this stage, does not know yet that it was quarantined – asks the Delivery Service of the central server to provide it with the messages that were buffered during its offline period. Instead, the server notifies it its "quarantined" status.

(2) u_g refreshes its keying material into $(pk_g^{rec+1}, sk_g^{rec+1})$. Following this, it transmits to the group a "Quarantine End" proposal, which is an "Update" proposal – used to refresh a user's encryption key-pair – that additionally indicates that its sender has come back online and needs to carry out a reconnection process.

(3) Upon receiving the "Quarantine End" proposal, each active user within the group verifies whether it possesses, in its private state, one or several quarantine shares linked to the former ghost and checks its associated shareholder rank(s).

If such shares exist and if the associated shareholder rank is $rk = 1$, the user, called a primary shareholder, encrypts them – along with their associated indices – under u_g's new encryption key and dispatches the ciphertext in a "Share Recovery Message".

(4) In parallel, the committer for the epoch e^{rec+1}, that has received the "Quarantine End" proposal, includes in its commit message the encryption, under the former ghost's fresh encryption key, of the new group key induced by the current commit. The idea is that even if the former ghost has troubles recovering its quarantine history (because of missing shares that prevent it from reconstructing the quarantine keys), it remains able to join the group from now on.

(5) The Delivery Service forwards to the former ghost all pending messages that were buffered by the server, much like when an active user reconnects.

(6) After receiving a sufficient number of Share Recovery Messages sent by the online users at epoch e^{rec+1}, the former ghost reconstructs the initial quarantine seed s_g^{q1} that was split up at the beginning of its quarantine and recovers the related quarantine encryption key-pair. If needed, the former ghost proceeds similarly with intermediate quarantine encryption keys, whose seeds are also reconstructed with shares sent at epoch e^{rec+1}:

$$\forall i \in [\![1, k]\!],\ s_g^{q_i} := Comb\big(([s_g^{q_i}]_j)_{j\in\mathcal{I}'}, \mathcal{I}'\big)$$
$$(pk_g^{q_i}, sk_g^{q_i}) := \mathrm{KeyGen}(1^\lambda; s_g^{q_i})$$

With its quarantine key-pair(s), the former ghost can now decrypt all the handshake and content messages that were exchanged during its quarantine period.

(7) If the number of shares received at the previous stage is not enough to reconstruct one of its quarantine seeds[11], the former ghost sends to the group a "Share Resend" proposal, identifying the missing shares with their indices and creation epochs.
When receiving this proposal, secondary shareholders (with shareholder rank $rk = 2$) for the missing shares send the appropriate Share Recovery Messages, either in broadcast or straightly to the former ghost if the Delivery Service allows it.
If the new batch of Share Recovery Messages is still not enough to reconstruct the associated seed, this process is iterated until the seed is reconstructed or the number of Share Resend proposals reaches a maximum value n_{resend}^{max} that is part of the parameters set.
If, despite these attempts, some quarantine seeds cannot be reconstructed, the content related to the period they cover is considered lost for the former ghost. The parameter n_{resend}^{max} therefore represents a necessary tradeoff between communication cost and availability of the CGKA.

[11]This may happen if too many of the primary shareholders are unresponsive (e.g. because they are themselves quarantined or even removed from the group) at the time of the ghost's reconnection.

4 SECURITY OF QTK PROTOCOL

4.1 Security Model

We use the security model from [21], which considers a game-based "CGKA security" with a partially active and fully adaptive adversary. The concept of "safe predicate" is used to rule out, in the associated security game, trivial attacks such as the compromise of a group key for a given epoch by corrupting one of the group members at that same epoch.

4.1.1 Adversarial Model. In that model, the adversary has full control over the Delivery Service from the server: therefore it can arbitrarily block messages and change their delivery order. Furthermore, the adversary is able to corrupt any group member at any time for a limited period of time defined by the predicates **start-corrupt** and **end-corrupt**. In this case, the private state of the corrupted users is leaked.

However, [21] restricts the adversary's ability to impersonate group members, even in case of corruption. The Authentication Service provided by the server is consequently assumed secure and the corruption of a group member neither leaks its private signature key nor gives the adversary a signature oracle.

This partially-active adversarial model corresponds to the one used by the MLS standard. TreeKEM uses a mechanism of "confirmation tag" from [7] to mitigate the effect of an active adversary by forcing it to access both a user's signature key or oracle and the current group key in order to impersonate that user. However, this security mechanism is of no use against a full user's compromise, where the adversary accesses both the victim's signature key or oracle and its private state. [12] explicitly states that vulnerability of MLS and consequently advises additional security measures to protect a user's signature key:

- compartmentalization between the signature key and other secret elements, and protection of this key by a secure enclave in the user's device;
- rotation of the signature key, with credential revocation.

4.1.2 CGKA Security Game. We state below the definition of CGKA security, issued from [21] and adapted to include the Propose & Commit paradigm of TreeKEM and the quarantine from QTK.

Definition 4.1 (Asynchronous CGKA Security). The security for CGKA is modelled using a game between a challenger C and an adversary $\mathcal{A}$. At the beginning of the game, the challenger creates a group G with identities $(u_1, \cdots, u_n)$. The adversary $\mathcal{A}$ can then make a sequence of the queries enumerated below, in any arbitrary order[12]. At a high level, **propose**$(\cdot,$ **add**$, \cdot)$ and **propose**$(\cdot,$ **remove**$, \cdot)$ allow the adversary to control the structure of the group, whereas the query **process** allows it to control the message scheduling. Moreover,the **start-upd-quarantine** and **end-quarantine** queries allow the adversary to arbitrarily quarantine any group member, while choosing the timing of this quarantine as well as the quarantine initiator[13].

[12]Except for some natural constraints on the queries order, such as ending a corruption or a quarantine after the start of the process, which are explicitly indicated.

[13]These queries give the adversary more capabilities in the security game than in the regular execution of QTK, where the starting and ending epochs of a quarantine are determined by the inactive user's behavior.

(1) **propose**$(u_i, a, [u_j])$: user u_i proposes to implement action $a \in \mathbb{A} \supseteq \{\textbf{add}, \textbf{remove}, \textbf{update}\}$ regarding user u_j.
(2) **commit**$(u_c, \mathbb{P})$: user u_c implements a list of proposals $\mathbb{P}$ that it received after the previous **commit** query, and updates accordingly its state γ_c and its filtered direct path.
(3) **start-upd-quarantine**(u_c, u_g): user u_c initiates a quarantine for user u_g or updates u_g's quarantine keys if the latter is already quarantined. This query necessarily precedes a **commit**$(u_c, \cdot)$ query associated with user u_c, where the latter distributes the secret shares for u_g's quarantine keys.
(4) **end-quarantine**(u_c, u_g): the quarantine of the ghost user u_g ends. This query necessarily follows a **start-upd-quarantine**$(\cdot, u_g, \cdot)$ request for that user and precedes the queries below, during which u_g updates its state γ_g and recovers the shares of all its quarantine seeds:
   - a **propose**$(u_g, \textbf{update})$ query where the ghost's keying material is refreshed before the ghost receives the shares of its quarantine keys;
   - a **commit**$(u_c, \cdot)$ query performed by user u_c.
(5) **process**(q, u_i): if the query q belongs to one of the previous categories, this action forwards the Welcome (W) or Commit (C) message to user u_i which immediately processes it.
(6) **start-corrupt**(u_i): from now on the private state ${}^s\gamma_i$ of u_i is leaked to the adversary.
(7) **end-corrupt**(u_i): ends the leakage of user u_i's private state. This query necessarily follows a **start-corrupt** request for that user.
(8) **challenge**(q^*): the adversary $\mathcal{A}$ picks a query q^* corresponding to an action $a^* \in \{$ **create-group**, **commit** $\}$. Let k_0 denote the group key that is sampled during this operation and k_1 be a fresh random key. The challenger tosses a coin b and – if the safe predicate below is satisfied – the key k_b is given to the adversary (if the predicate is not satisfied the adversary gets nothing).

At the end of the game, the adversary outputs a bit $\hat{b}$ and wins if $\hat{b} = b$. We call a CGKA scheme (Q, ϵ, t)-CGKA-secure if for any adversary $\mathcal{A}$ making at most Q queries of the form **propose**, **commit**, **start-upd-quarantine** and **end-quarantine** and running in time t, it holds:

$$\mathrm{Adv}^{\mathrm{CGKA}}(\mathcal{A}) := \left| \Pr[1 \leftarrow \mathcal{A} \mid b = 0] - \Pr[1 \leftarrow \mathcal{A} \mid b = 1] \right| \leq \epsilon$$

Nota: As the group only evolves, in the security game, by queries made by the adversary, we designate time points by the queries associated with them. No adequation can be made between epochs and queries in the Propose & Commit paradigm, since some of the latter induce a change of epoch (**commit**, **start-quarantine**) and the others do not.

4.2 Safe Predicate

The safe predicate defines the trivial situations where a challenge group key cannot be protected from an adversary, in particular when the adversary corrupts a user that possesses that challenge group key. Therefore, these situations must be identified and excluded from the security game that defines the CGKA security.

4.2.1 Proper Critical Window. The first component of the safe predicate is the "(proper) critical window" of a user u_i, in the view of a user u^* at a time point represented by a query q^*. This concept from [21], that we adapt below to fit a Propose & Commit CGKA, defines the period during which a group key k^* issued by u^* at time q^* can possibly be leaked by u_i if the latter is corrupted at that time.

Definition 4.2 (Proper Critical Window). Let Π be a Propose & Commit CGKA protocol as defined in Definition 2.3 and G^* the set of users after processing a query q^* corresponding to an action $a^* \in \{$**create-group**, **commit**$\}$ of a user $u^* \in G^*$, that ends up in generating a new group key k^*.

Let $\mathcal{S}_i^* = \{(pk_i^*, sk_i^*), \{(pk_v^*, sk_v^*)\}_{v \in \mathcal{P}(u_i)}\}$ be the set of encryption key-pairs associated with a user $u_i \in G^*$ – possibly u^* itself – and the nodes in u_i's filtered direct path, according to the view of u^* at time q^*.

The proper critical window of u_i in the view of u^* at time q^* is the period of time between two bounds $q^- < q^*$ and $q^+ > q^*$ s.t.:

- q^- is the query that starts to set $\mathcal{S}_i^*$ in u_i's state before q^*, in the view of u^*. More precisely, this is the *earliest* **commit** query, processed by u^* and setting:
  - either (pk_i^*, sk_i^*) into u_i's state, through an **update** proposal;
  - or part of $\{(pk_v^*, sk_v^*)\}_{v \in \mathcal{P}(u_i)}$ into u_i's filtered direct path.
- q^+ is the query that completely invalidates $\mathcal{S}_i^*$ in u_i's state after q^*, in the view of u^*. In other terms, it corresponds to the *latest* **commit** processed by u^*, that refreshes the remaining parts of u_i's state with keys belonging to $\mathcal{S}_i^*$.

Nota: This definition also applies to CGKAs outside the Propose & Commit paradigm, by disregarding **update** proposals that only refresh a leaf's keying material. In this case, key rotation is only realized by **commits** that update both users' keying material and the one of their filtered direct path.

TreeKEM Critical Window. Figure 5 illustrates the notion of critical window with a generic TreeKEM-based protocol. This window is wrapped around the group key creation time and is bounded, in the general case[14], by:

- a lower bound q^-: u_i's last update before q^*.
- an upper bound q^+: its first last update after q^*.

QTK Critical Windows. For the QTK protocol, we consider separately the cases where the query q^* occurs before or during a ghost's quarantine. We also distinguish four types of users related to that ghost, with different (proper) critical windows that are detailed in Figure 8 in Appendix A:

- the **ghost user** itself;
- the **quarantine updater**: this is the committer who last updated the ghost's quarantine keys before the challenge query q^*. If q^* occurs before any quarantine update, the quarantine updater is the quarantine initiator;

[14] Specificities of CGKAs may slightly modify these bounds, like in Tainted TreeKEM [21] where the critical window is not bounded by the update times but by the confirmation or rejection time of these updates.

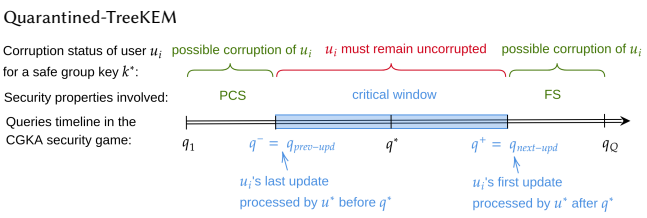

Figure 5: Critical window, for a TreeKEM-based CGKA, of a user u_i in the view of a user u^* issuing a challenge group key k^* at time q^*.

- the ghost's **shareholders**: users who detain a share of the ghost's quarantine encryption key used at time q^*;
- the other **active users**. These group members are not shareholders because they were not part of the group yet when the ghost's quarantine key was last updated before q^*.

Figure 6 compares the critical windows of QTK and TreeKEM. The significant reduction, in QTK, of an inactive user's *proper* critical window (in red) casts the light on the security improvement brought by QTK protocol. Indeed, even if, with our protocol, a single inactive user may still prevent the group from reaching forward secrecy and post-compromise security, the period during which a corruption of that inactive user jeopardizes a group key – which is precisely what its critical window refers to – is greatly reduced. The only other way for the adversary to leak the group key is to corrupt a sufficient number of shareholders associated with that ghost; this risk is evaluated by the concept of *shared* critical window (cf. below).

Case 1: q^* precedes the ghost's quarantine. In that case, there are no changes from the standard TreeKEM protocol. Active users have a regular critical window around the challenge query q^*, as in Figure 5.

A ghost's critical window extends from its last key update before q^* to the one following this query, which occurs at its reconnection time at the end of its quarantine; in this case, this long critical window is similar to the one of an inactive user with TreeKEM. This issue can be partially solved, both in TreeKEM and in QTK, by forcing the messaging application of the inactive user to locally delete, after some time, the secret elements stored in its local state[15].

Case 2: q^* occurs within the ghost's quarantine. The critical windows of the users related to the ghost u_g are defined as follows:

- **Ghost user**: the only critical window of a ghost starts at its reconnection after a quarantine (query q^{ghost}_{rec}), when it recovers all the shares associated with its encryption key used at time q^*. This window closes at the former ghost's following update ($q^{ghost}_{upd-after-rec}$), which overwrites the sensitive former ghost's local state. As the share recovery may last for several epochs, depending on the activity status of the shareholders during this reconnection stage, the window size may vary between one and several epochs. However, if a ghost never reconnects until its reaches its quarantine maximum period δ_{quar} and is removed from the group, the aforementioned critical window does not exist.

[15]This operation does not require the inactive user to log in, but its messaging application must at least run in the background, which cannot be imposed to all inactive users.

- **Shareholders**: beside their proper critical window that steams from their status of active user, these users have a "shared critical window" defined and detailed below.
- **Quarantine updater**: its critical window only lasts during the preparation of the commit that is joined with the quarantine key update (from time $q^{updater}_{quar-upd}$ to $q^{updater}_{commit-after-quar-upd}$). Once the commit message is sent, the quarantine updater deletes all sensitive information from its state.
- **Active users**: non-shareholder active users at time q^* have a critical window similar to any TreeKEM-based CGKA protocol, centered around q^* and bounded by the **prop-update** queries preceding and following that time point. These active users may include the quarantine updater related to q^* as well as most of the shareholders[16].

Nota: To improve the security of QTK regarding the quarantine updater, we present in Appendix B an enhancement called "jointly-implemented quarantine", where the ghost's quarantine keys are commonly generated by several users that only have a partial knowledge of these sensitive data. Consequently, the proper window of a single quarantine updater is replaced by several shared windows, much more secure.

4.2.2 Shared Critical Window. In the context of QTK protocol, which is associated to a secret sharing scheme which protects the secret information by splitting it into shares, the notion of proper critical window appears insufficient to implement the safe predicate.

We therefore define hereunder the concept of "shared critical window", that represents the period during which a user (shareholder) possesses *shares* of a secret information that can compromise the group key k^*. Hence, the security cannot be evaluated anymore with the safety of a single user; instead we must determine whether a sufficient number of shareholders associated with the same secret have remained uncorrupted as long as they held these shares.

Definition 4.3 (Shared Critical Window). Let Π be the QTK protocol associated with a (t, m)-perfect secret sharing scheme, G^* the set of users after processing a query q^* corresponding to an action $a^* \in \{$**create-group**, **commit**$\}$ of a user $u^* \in G^*$, that ends up in generating a new group key k^*.

The shared critical window of a shareholder $u_s \in G^*$ (possibly u^* itself) related to a ghost user u_g in the view of u^* at time q^*, is the period of time during which u_s holds a share of a quarantine secret key that leads to the challenge group key k^*.

Consequently, the corruption of at least t shareholders (from different shareholder families w.r.t that share collection) related to a ghost u_g results in the compromise of the group key k^* generated by u^* at time q^*.

In the case of our QTK protocol, a **shareholder** of a ghost u_g has a shared critical window that:

- starts at the commit associated with u_g's last quarantine key update before q^*: $q^{updater}_{commit-after-quar-upd}$;

[16]Indeed, some shareholders associated with a ghost, and even its quarantine initiator or updater(s), may have become inactive at the time of a subsequent challenge request.

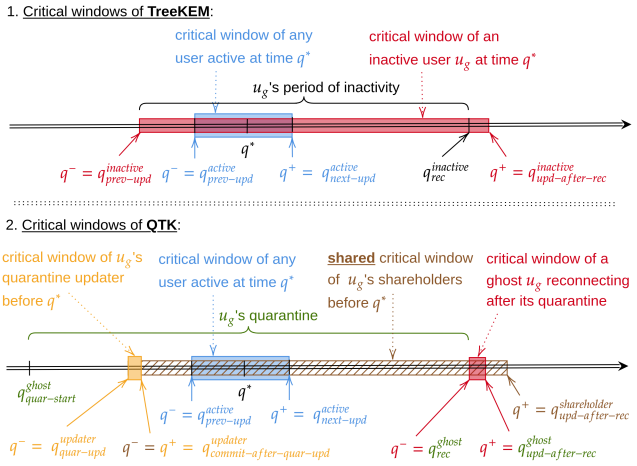

Figure 6: Compared critical windows, between TreeKEM and QTK, for the different types of users in a group, in the view of user u^* issuing a group key k^* at time q^*. The crosshatched brown box represents a *shared* critical window (cf. Definition 4.3), more secure than a *proper* one.

- ends at the shareholder's update following u_g's quarantine end: $q^{shareholder}_{upd-after-rec}$.

4.2.3 Safe Group Key. We adapt for the QTK protocol the safe predicate concept from [21], that states the conditions needed for a group key to be safe, by including in these conditions the shared security implied by the above-mentioned shared critical window.

Definition 4.4 (Safe Predicate with a Shared Critical Window). Let Π be a QTK protocol associated with a (t, m)-perfect secret sharing scheme. Let k^* be a group key generated in an action $a^* \in$ {**create-group**, **commit**} at time $q^* \in [\![q_1, q_Q]\!]$ and let G^* be the set of users ending up in the group after processing query q^*, as viewed by the generating user u^*.

Moreover, let us consider an arbitrary number of ghost users ($u_g \in \mathcal{G}^*$) quarantined at time q^*, with their associated shareholders.

Then the challenge group key k^* is considered safe if the following two statements are fulfilled:

- No user from the group (including u^* itself) has been corrupted in its *proper critical window* at time q^* in the view of u^*;
- For each ghost u_{g_i} quarantined at time q^*, strictly less than t of its shareholders from different shareholder families, i.e. with different shares, have been corrupted in their *shared critical windows* at time q^* in the view of u^*.

4.3 CGKA Security Proof for QTK

4.3.1 Overview. The security proof of our protocol relies on the one from Tainted TreeKEM in [21]. This work defines the safe predicate and the challenge graph (cf. below) associated with their protocol and proves in a lemma – similar to our Lemma 4.5 beneath – that the respect of the safe predicate implies no leakage of any secret element from that challenge graph. Finally, it uses the concept of "Generalized Selective Decryption" (GSD), adapted from [19], to turn the selective CGKA security of Tainted TreeKEM into an adaptive security in the Random Oracle Model.

As stated in [21], the part of that proof which uses GSD can be generalized to other CGKAs, such as TreeKEM or QTK, since the demonstration does not depend on the structure of the protocol's CGKA graph but only on its maximum number of nodes.

Consequently, our security proof for QTK consists in determining the safe predicate (already done in Section 4.2) and the challenge graph corresponding to our protocol, and proving in Lemma 4.5 that no secret information from that challenge graph can lead to the leakage of the challenge group key. These stages are sufficient to prove the CGKA security of our protocol.

4.3.2 CGKA and Challenge Graphs. We must firstly define the **CGKA graph**, adapted from [21], which represents the evolution of the CGKA's Ratchet Tree throughout the security experiment. This graph is therefore the juxtaposition of different generations of nodes from the Ratchet Tree, partially superposed when some nodes remain unchanged from one epoch – i.e. one query in the security experiment – to another. The edges of the CGKA graph are either the derivations of the nodes' secret seeds or the public-key encryption of these seeds.

The challenge graph related to a challenge group key k^*, also issued from [21], is the sub-graph of the CGKA graph composed of all nodes – internal or leaves – that contain secret information permitting to recover that challenge group key.

With a standard CGKA such as TreeKEM, the challenge graph for a group in a consistent state[17] (at the time of the challenge query q^*) simply consists in the Ratchet Tree at time q^*. When the group view is in inconsistent state, the challenge graph is the Ratchet Tree at time q^* *in the view of committer* u^*_c which generated the challenge group key (cf. [21]).

The challenge graph for QTK, detailed in Figure 7, differs from that of TreeKEM by:

- the addition of two nodes, corresponding to the quarantine updater and to the reconnecting ghost;
- the particular case of the "challenge ghost node", which corresponds to the last updated ghost's quarantine keys before q^*. This node cannot be corrupted directly by the adversary in the CGKA security experiment. Instead, its corruption occurs if a sufficient number of its associated shareholders are themselves corrupted.

To illustrate the latter point, we define another sub-graph of the CGKA graph called the "Quarantine Graph", related to a challenge group key k^*. That graph comprises all the nodes from the CGKA graph that record a share related to the challenge ghost node.

Lemma 4.5. *For any safe challenge group key in QTK, i.e. for which the safe predicate is respected, it holds that none of the seeds and secret keys in the challenge graph are leaked to the adversary via user corruptions.*

Proof. We proceed with a proof by contradiction, by showing – with a separate analysis for each user type – that the leakage of the challenge key, resulting from the leakage of a secret element from

[17] Consistency means that all users in the group have the same view of the Ratchet Tree at a precise moment.

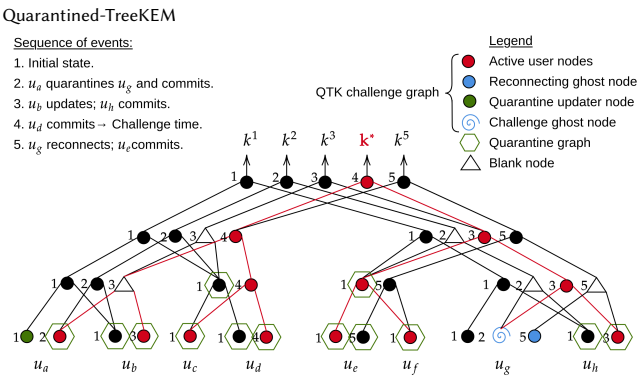

Figure 7: CGKA, challenge and quarantine graphs for QTK protocol.

its challenge graph, implies the violation of the safe predicate for the concerned user.

The case of active users from the challenge graph and internal nodes ancestors of these active users is similar to that of TreeKEM: as the critical window of these users for the challenge group key k^* is centered on the challenge query q^* and bounded by these users' key updates – which refresh the leaves and blank their paths –, a secret key or seed that may lead to the challenge group key is recorded in these users' private states only during their critical window. Consequently, a leakage of such sensitive elements can be generated only by corrupting one of these users during its critical window, which is possible only by contradicting the safe predicate.

Similarly, a ghost user only records secret elements associated with the challenge group key at its reconnection time, which also corresponds to its critical window. This is also the case for the quarantine updater, which has knowledge of the ghost's secret key associated with the challenge group key only when creating the commit in which the shares are sent, which also corresponds to its critical window.

A specific feature of QTK is the case of the share-holding nodes. Regarding the shareholders (i.e. the share-holding leaves), their shared critical window extends from the moment they receive the shares to their (leaf) update following the ghost reconnection – where they delete these shares. A leakage of the challenge group key would imply to corrupt a sufficient number of them (with respect to the secret sharing recovery threshold) during their shared critical window, which is also a violation of the safe predicate.

As a leaf update is always associated with the blanking of the leaf's direct path, the shareholders' update that constitutes the upper bound of their critical window deletes in their internal state the private keys and seeds of the share-holding internal nodes above them. Consequently, the sensitive elements in these internal nodes cannot be reached by the adversary unless by corrupting one of their descendant shareholders during its critical window, which also contradicts the safe predicate.

Theorem 4.6 (QTK CGKA Security). *If the encryption scheme in QTK is $(\tilde{\epsilon}, t)$-IND-CPA-secure and if the cryptographic hash functions used to derive the seeds are modelled as random oracles, then this CGKA is (Q, ϵ, t)-CGKA-secure, with Q the number of queries in the security game, n the number of users in the group and $\epsilon = \tilde{\epsilon} \cdot 8(nQ)^2 + negl$.*

Proof. The proof of this theorem is straightly issued from Theorem 3 and Theorem 4 from [21], which are used to prove the CGKA security of Tainted TreeKEM but are also applicable to other CGKAs such as TreeKEM and QTK. An overview of the proof for Tainted TreeKEM [21] is detailed in Appendix A.3.

The security bound $\tilde{\epsilon}$ of QTK originates from the equation of Theorem 3 which gives $\epsilon = 2N^2\tilde{\epsilon} + \frac{mN}{2^{\ell-1}} = 2N^2\tilde{\epsilon} + negl$, with N the number of nodes in the CGKA graph, m the number of oracle queries to the random oracle and ℓ the length of the secret seeds in the CGKA graph.

For Q queries in the CGKA security experiment, the CGKA graph has a size bounded as $N < 2nQ$ (since the Ratchet Tree for n users has at most $2n-1$ nodes and the CGKA in the worst case is the juxtaposition of Q separate Ratchet Trees). This upper bound determines the security factor $\epsilon = \tilde{\epsilon} \cdot 8(nQ)^2 + negl$ given above.

5 PERFORMANCES

We study here the communication cost *per user* induced by a single ghost's quarantine, in the broadcast-only and the server-aided settings. This cost is measured as the size of the exchanged messages, counted once between the sender and the Delivery Service, and either $n-1$ times (in case of a broadcasted message) or once (in case of a message sent individually to its recipient) between the Delivery Service and the other users, the total being finally divided by the number n of users.

5.1 Broadcast-Only Regular Quarantine

5.1.1 Initialization and Updates. A ghost u_g quarantined for a period $t_{quar} \leq \delta_{quar}$ uses a number of quarantine encryption keys defined as: $n_{qkey} := \left\lceil \frac{t_{quar}}{\delta_{quar-upd}} \right\rceil$.

We recall that each quarantine initialization or update is associated with a commit. The *additional information*, related to the quarantine, in a commit message sent at epoch e^i, is:

- the ghost's leaf index ℓ_g;
- the ghost's fresh quarantine public key pk_g^{i+1};
- the encrypted shares for the quarantine secret seed s_g^{i+1}.

We consider here the most-likely case where an *ideal* secret sharing scheme (cf. Section 2.2) is used in QTK. In this case, each one of the m shares has a size equal to the seed from which they originate. Therefore, $\forall j \in [\![0, m-1]\!], \left|[s_g^{i+1}]_j\right| = \left|s_g^{i+1}\right| = |s|$.

The number of shares to distribute depends on the number n of users, on the tree structure and on the share distribution method. Furthermore, the size of an encrypted share differs according to the share distribution method:

- With the **default share distribution method**, each share is joined to a path secret already encrypted by HPKE [10]. Consequently, it is not necessary neither to encapsulate once again a symmetric key *via* a KEM, nor to provide another authenticity tag (related to the AEAD encryption schem), and the encryption cost of the share is thus linear with its size: $\forall i \in [\![0, m-1]\!], \left|Enc^{hpke}([s]_i)\right| = |s|$.
- With the **horizontal share distribution method**, all shares are encrypted separately. Therefore, the communication cost of the HPKE encryption of a single share is:

$\forall i \in [\![0, m-1]\!], \left|Enc^{hpke}([s]_i)\right| = |ct| + |s| + |tag|$, with ct the ciphertext output by the KEM used within the HPKE ciphersuite and tag the authenticity tag yielded by the AEAD encryption scheme.

Consequently, we have an initialization and update cost bounded as follows:

$$cc^{bdct}_{init-upd} \in \big[n_{qkey}(\,|pk| + |int| + m\,|s|\,),\; n_{qkey}(\,|pk| + |int| + m(|ct| + |s| + |tag| + |int|))\big]$$

5.1.2 Quarantine End. As stated in Section 3.5, the reconnecting ghost broadcasts a "Quarantine End" proposal and – in the worst case – a number ρ of "Share Resend" proposals. In return, its associated shareholders forward it the shares of its quarantine keys.

Quarantine End Proposal. This message is based on an Update proposal and has a size equal to the latter: $|quar - end| = |upd| = |sig| + |pk| + |spk| + |cred|$ with "spk" and "cred" the former ghost's public signature key and associated credentials.

"Share Resend" Queries. These messages only comprise the creation epochs and the indices of the $n_{missing}$ missing shares at that time, encrypted under the current group key. Let us note $|int|$ the size of an integer used to represent an epoch or a leaf index and let us consider a i^{th} "Share Resend" query ($i \in [\![1, n^{max}_{resend}]\!]$) with $n_{missing_i}$ missing shares:

$$|resend_i| = |sig| + 2n_{missing_i}\,|int| \leq |sig| + 2n_{missing_0}\,|int| \leq |sig| + 2n_{qkey}t\,|int|$$

Consequently, the communication cost of ρ "Share Resend" messages is bounded by: $cc_{resend} \leq \rho(\,|sig| + 2n_{qkey}t\,|int|\,)$.

Share Recovery Messages. In response to its Quarantine End proposal or its i^{th} Resend query, the reconnecting ghost receives from the shareholders a number $n_{shmsg_i} \leq m$ of initial Share Recovery Messages for *each* of its n_{qkey} quarantine keys.

This number depends on the number of active users within the group at epoch e^{rec}. If $n_{shmsg_0} \in [\![t, m]\!]$, the former ghost does not need to receive additional shares ($\rho = 0$). On the contrary, if $n_{shmsg_0} < t$, a number $\rho \in [\![1, n^{max}_{resend}]\!]$ of "Share Resend" queries appears necessary.

The best case appears when a number t of shareholders send all the n_{qkey} generations of shares to the reconnecting ghost. The worst case, on the other hand, occurs when m shareholders send a Share Recovery Message with only one share inside, which implies in total $n_{qkey}.m$ distinct messages.

Reconnection Communication Cost. Consequently, the communication cost induced by the a quarantine end is bounded as follows:

$$cc^{bdct}_{end} \in \big[\,|upd| + t(\,|sig| + |ct| + |tag| + n_{qkey}(|s| + 2\,|int|)),\; |upd| + n^{max}_{resend}(\,|sig| + 2n_{qkey}t\,|int|\,) + mn_{qkey}(\,|sig| + |ct| + |s| + |tag| + 2\,|int|\,)\big]$$

5.2 Server-Aided Regular Quarantine

5.2.1 Initialization and Updates. As only the horizontal share distribution method is used in this paradigm, the communication cost of a quarantine initialization and updates is:

$$cc^{s-a}_{init-upd} = n_{qkey}\Big(\,|pk| + |int| + \big(1 + \tfrac{m-1}{n}\big)\big(|sig| + |ct| + |s| + |tag| + |int|\,\big)\Big)$$

5.2.2 Quarantine End. Similarly to the broadcast-only setting, the range of values depends on the shareholders' responsiveness to the reconnecting ghost's Quarantine End proposal and on the way these shareholders group the shares inside the Share Recovery Messages.

$$cc^{s-a}_{end} \in \big[\,|upd| + \tfrac{2}{n}t(\,|sig| + |ct| + |tag| + |int| + n_{qkey}(|s| + 2\,|int|)),\quad |upd| + n^{max}_{resend}(\,|sig| + 2n_{qkey}t\,|int|\,) + \tfrac{2}{n}mn_{qkey}(\,|sig| + |ct| + |s| + |tag| + 3\,|int|\,)\big]$$

5.3 Practical Efficiency

To give a broad idea of the communication cost induced by a ghost's quarantine, Table 1 details its communication cost under several settings. The factors influencing this communication cost can be sorted as follows:

- the **algorithms** used to ensure the HPKE and digital signature functionalities;
- the features of the **user group** (number of users and structure of the Ratchet Tree), issued from the group history;
- the **quarantine parameters** (maximum duration of a quarantine, frequency of quarantine key update, secret sharing scheme recovery threshold...);
- some **encoding settings** (integer encoding).

In our instance, we consider two main paradigms that influence the choice of algorithms: the **classical framework**, where our protocol only implements pre-quantum encryption and signature algorithms, and the **post-quantum framework** which uses post-quantum encryption but keeps a classical signature primitive.

5.3.1 Parameter Choice.

Encryption and Signature Primitives. In the classical framework, the encryption is carried out by the HPKE paradigm [10], with an ECDH-KEM such as X25519 [11] to ensure the key transport functionality and a symmetric encryption scheme like AES-256 for the data encryption. In the post-quantum framework, the Data Encapsulation Mechanism (DEM) remains unchanged but the classical KEM is replaced by a post-quantum one. We choose to instantiate Crystals Kyber [15], standardized by the NIST as ML-KEM, as our post-quantum KEM.

In both frameworks, ECDSA [22] with an elliptic curve on a 256-bit prime field is selected as the digital signature algorithm[18].

Quarantine Parameters. Given a key renewal period δ_{upd} for active users, we study the case of short, medium and long quarantines of respective durations $t_{quar_s} = 7.\delta_{upd}$, $t_{quar_m} = 14.\delta_{upd}$, $t_{quar_\ell} = 28.\delta_{upd}$, with a quarantine key renewal period of $\delta_{quar-upd} = 2.\delta_{upd}$.

[18] We choose not to adopt post-quantum signatures in our PQ framework, as we consider that it remains difficult to instantly forge a classical signature, even for a quantum adversary.

The original parameter δ_{upd} is itself likely to vary greatly depending on the settings of the applications using the CGKA protocol. However, if we take for instance a daily key renewal, our settings correspond to a quarantine key refreshment every couple of days and quarantines that last one, two and four weeks, which seems consistent with realistic use cases of these quarantines.

As for the secret sharing parameters, the number of emitted shares is computed as $m = \lceil \log(n) \rceil + 1$, which corresponds to the number of shares needed by the default share distribution within a well structured binary tree, knowing that even with an horizontal share distribution, the number of emitted shares is roughly the same. The recovery threshold is chosen as $t = \lceil \frac{m}{2} \rceil$, in order to have a good trade-off between security and efficiency.

Group Parameters. We consider groups of various sizes, up to $2^{16} = 65,536$ users, as the MLS protocol specifications indicate the need to scale up to this order of magnitude. We point out that the number of users has various – and potentially opposite – consequences on a quarantine communication cost *in the broadcast-only setting*:

- Regarding the **initialization and update cost**, the larger the group, the higher the number of shares that have to be distributed within the group. However, as this number grows logarithmically with the number of users (for a perfect binary tree), the consequences on the communication cost are quite negligible.
  On the other side, large groups ensure that the *default* share distribution method, far more efficient than the *horizontal* one, can be implemented. With that default method, the communication cost reaches the "best case" of the communication cost range provided in Table 1 (i.e. the smallest value within that range).
- The **quarantine end cost** is more subtle: a high number of users makes it unlikely that additional Share Recovery Messages need to be requested.
  However, it is also unlikely that shareholders keep several shares for the ghost with the same shareholder rank[19]. Consequently, it increases the number of separate Share Recovery Messages, which – especially due to the particularly important post-quantum encryption cost – impacts all the more the communication cost.
  The second factor being prominent over the first one, large groups tend to have a quarantine end communication cost close to the worst case (i.e. the highest value) of the range displayed in Table 1.

We compare this communication cost with the one of a user remained active at the same period and who updates its keying material (by sending Update proposals of size $|upd|$) with a renewal period of δ_{upd}.

Table 1 and Table 2 detail the practical overhead *per user* of a quarantine, respectively in the classical and post-quantum framework, with the above-chosen parameters.

[19]Indeed, the bigger the group, the more unlikely it is that a given user keeps the same relative position (i.e. shareholder rank) w.r.t. different quarantine updaters chosen at random among all users.

Table 1: Practical communication cost per user of a ghost's quarantine in the classical framework. The "broadcast-only average" column represents the most-likely case.

Quar. Length	Group Size	Quarantine Communication Cost per User (kB)		
		Broadcast-Only	Server-Aided	Active
		Best **Average** Worst	Best Worst	User
$7.\delta_{upd}$	8	1.65 – **2.45** – 4.39	1.37 – 2.11	1.67
	128	2.68 – **6.57** – 8.17	1.01 – 1.50	
	65,536	4.85 – **13.24** – 16.64	0.96 – 1.58	
$14.\delta_{upd}$	8	2.46 – **3.86** – 7.36	2.16 – 3.38	3.33
	128	4.09 – **11.17** – 13.97	1,58 – 2.30	
	65,536	7.56 – **22.84** – 28.79	1.50 – 2.45	
$28.\delta_{upd}$	8	4.35 – **7.15** – 14.29	3.99 – 6.32	6.66
	128	7.38 – **21.91** – 27.51	2.92 – 4.16	
	65,536	13.87 – **45.26** – 57.16	2.76 – 4.46	

As expected, that PQ cost largely exceeds the one in the classical framework (cf. Table 1). However, even in the worst case, the overhead of around 500 kB does not sound unrealistic given the important communication cost that a CGKA already has.

Table 2: Practical communication cost per user, in kilobytes, induced by a ghost's quarantine in the post-quantum framework.

Quar. Length	Group Size	Quarantine Communication Cost per User (kB)		
		Broadcast-Only	Server-Aided	Active
		Best **Average** Worst	Best Worst	User
$7.\delta_{upd}$	8	10.6 – **28.3** – 43.9	13.7 – 17.9	9.7
	128	13.7 – **46.1** – 81.5	11.3 – 12.2	
	65,536	20.1 – **90.8** – 166.0	10.9 – 11.6	
$14.\delta_{upd}$	8	14.8 – **45.8** – 75.7	22.3 – 30.1	19.5
	128	18.6 – **79.5** – 141.5	18.7 – 20.2	
	65,536	26.3 – **157.7** – 289.3	18.1 – 19.1	
$28.\delta_{upd}$	8	24.8 – **86.7** – 149.8	42.4 – 58.7	38.9
	128	29.9 – **157.5** – 281.3	35.9 – 38.9	
	65,536	40.7 – **313.9** – 577.1	34.8 – 36.5	

REFERENCES

[1] 2023. *WhatsApp Encryption Overview.* Technical White Paper. WhatsApp Inc. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf

[2] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. 2022. DeCAF: Decentralizable Continuous Group Key Agreement with Fast Healing. Cryptology ePrint Archive, Report 2022/559. https://eprint.iacr.org/2022/559.

[3] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. 2022. CoCoA: Concurrent Continuous Group Key Agreement. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, 815–844. https://doi.org/10.1007/978-3-031-07085-3_28

[4] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2020. Security Analysis and Improvements for the IETF MLS Standard for Group Messaging. In *CRYPTO 2020, Part I (LNCS, Vol. 12170)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Heidelberg, 248–277. https://doi.org/10.1007/978-3-030-56784-2_9

[5] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2021. Modular Design of Secure Group Messaging Protocols and the Security of MLS. Cryptology ePrint Archive, Report 2021/1083. https://eprint.iacr.org/2021/1083.

[6] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. 2022. Server-Aided Continuous Group Key Agreement. In *ACM CCS 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM Press, 69–82. https://doi.org/10.1145/3548606.3560632

[7] Joël Alwen, Daniel Jost, and Marta Mularczyk. 2022. On the Insider Security of MLS. In *CRYPTO 2022, Part II (LNCS, Vol. 13508)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, Heidelberg, 34–68. https://doi.org/10.1007/978-3-031-15979-4_2

[8] 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.

[9] 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

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

[11] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC 2006 (LNCS, Vol. 3958)*, Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin (Eds.). Springer, Heidelberg, 207–228. https://doi.org/10.1007/11745853_14

[12] 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.

[13] 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

[14] Dan Boneh and Victor Shoup. 2023. *A Graduate Course in Applied Cryptography.* http://toc.cryptobook.us/

[15] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, and Damien Stehlé. 2017. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634. https://eprint.iacr.org/2017/634.

[16] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1802–1819. https://doi.org/10.1145/3243734.3243747

[17] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. 2021. MLS Group Messaging: How Zero-Knowledge Can Secure Updates. In *ESORICS 2021, Part II (LNCS, Vol. 12973)*, Elisa Bertino, Haya Shulman, and Michael Waidner (Eds.). Springer, Heidelberg, 587–607. https://doi.org/10.1007/978-3-030-88428-4_29

[18] 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*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 1441–1462. https://doi.org/10.1145/3460120.3484817

[19] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. 2017. Be Adaptive, Avoid Overcommitting. In *CRYPTO 2017, Part I (LNCS, Vol. 10401)*, Jonathan Katz and Hovav Shacham (Eds.). Springer, Heidelberg, 133–163. https://doi.org/10.1007/978-3-319-63688-7_5

[20] Daniel Jost, Ueli Maurer, and Marta Mularczyk. 2019. Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging. In *EUROCRYPT 2019, Part I (LNCS, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 159–188. https://doi.org/10.1007/978-3-030-17653-2_6

[21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. 2021. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 268–284. https://doi.org/10.1109/SP40001.2021.00035

[22] National Institute of Standards and Technology. 2023. Digital Signature Standard. FIPS 186-5. https://doi.org/10.6028

[23] Saurabh Panjwani. 2007. Tackling Adaptive Corruptions in Multicast Encryption Protocols. In *TCC 2007 (LNCS, Vol. 4392)*, Salil P. Vadhan (Ed.). Springer, Heidelberg, 21–40. https://doi.org/10.1007/978-3-540-70936-7_2

[24] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. http://dblp.uni-trier.de/db/journals/cacm/cacm22.html#Shamir79

[25] Matthew Weidner. 2019. Group messaging for secure asynchronous collaboration. (2019). https://mattweidner.com/acs-dissertation.pdf

A ADDITIONAL SECURITY CONSIDERATIONS

A.1 Considerations on a Fully Active Adversary

As aforementioned, and accordingly to the security model of MLS standard and other works – among which [21] –, the formal security analysis of this paper deals with a partially active adversary unable to impersonate any user, even in case of corruption. In that framework, all user messages are thus considered legitimate.

We may wonder what would be the main security issues with QTK, in case of a fully active attacker, that would not occur with TreeKEM. To do so, we consider separately the cases of a shareholder or a quarantine initiator/updater impersonation, that only impact the availability of the protocol but not its security, and the impersonation of a ghost, that lowers QTK's security[20].

A.1.1 Shareholder Impersonation. The only action that the protocol requires shareholders to carry out is to send to a reconnecting ghost the shares that correspond to its quarantine keys. An impersonated shareholder would therefore either voluntarily retain the shares that it was supposed to transmit, send invalid shares or even send wrong shares with bad indices[21], in order to additionally collide with legitimate shares sent by other shareholders. However, all these attacks only impact the ability of the former ghost to reconstruct its quarantine keys, and, as a consequence, the availability of the protocol – which can never be ensured against an active adversary, for any CGKA and notably TreeKEM, especially when the Delivery Service is controlled by this adversary.

A.1.2 Quarantine Initiator/Updater Impersonation. In this case, the adversary can also impact the availability of the protocol by sending invalid shares to all shareholders. The major flaw here is that the unavailability issue only appears at the end of the quarantine, when the ghost reconnects.

The adversary can also arbitrarily quarantine any user, even an active one. If it is coupled with the distribution of invalid shares, it eventually comes to temporarily expelling a user from the group, since that user will not be able to recover its message history when reconnecting. Nevertheless, such an attack does not exceed the capacity of an active adversary in TreeKEM, who can also arbitrarily

[20] The impersonation of the last type of user, the non-shareholding active user, has no particular effect on QTK that would no occur on TreeKEM.

[21] The share indices allow the reconstructing algorithm of the secret sharing scheme to select a valid set of shares. Each "Share Recovery Message" therefore comprises one or several shares, along with their associated indices (cf. Section 3.5).

remove any user[22]. Furthermore, it is easy to trace a malicious quarantine initiator that performs this type of attack, as any user in the group:

- is able to check whether the quarantine is justified (which is the case if the newly quarantined ghost has encryption keys whose seniority exceed the maximum authorized limit δ_{inact});
- knows which committer has initiated that – potentially fallacious – quarantine.

A.1.3 Ghost Impersonation. The impersonation of a ghost strongly impacts QTK's forward secrecy. Indeed, the adversary may require a reconnection of that ghost on its behalf and recover this way the ghost's quarantine keys, and thus, all the content history of the group since the ghost's last key update. We nevertheless underline that even in that worst-case scenario, the forward secrecy yielded by QTK only falls back to the one offered by the original TreeKEM and never falls below.

Albeit such a fully-active adversary stands beyond our security model, we recommend, in order to prevent this case, the use of a multi-factor authentication that would require an active action of the "human" user, either on the same device as the one using the Secure Group Messaging application – such as the "One Tap" version of the Google 2-step process or a biometric authentication – or an out-of-band authentication on a separate device. We leave the formal security analysis associated with such authentication processes, in the framework of QTK as well as TreeKEM, as an open problem for future work.

A.2 Detailed Critical Windows of QTK

In addition to Figure 6 from Section 4.2.1, Figure 8 hereunder details the critical windows induced by our QTK protocol by considering separately the different types of users in a group.

A.3 Security Proof for Tainted TreeKEM [21]

The security proof of Tainted TreeKEM in the ROM relies on the concept of Generalized Selective Decryption (GSD), from [23], that [21] has adapted to the framework of public-key encryption. This notion states the indistinguishability of the secret key associated with a node in a graph. We can straightforwardly deduce the CGKA-security of a protocol from the GSD property, by considering the GSD graph as the CGKA graph (cf. Section 4.3.2) and by focusing on the indistinguishability of the secret element of the roots of the CGKA-graph[23].

Definition A.1 (Generalized Selective Decryption (GSD), adapted from [23] by [21]). Let (KeyGen, Enc, Dec) be a public-key encryption scheme with secret key space $\mathcal{K}$ and message space $\mathcal{M}$ such that $\mathcal{K} \subseteq \mathcal{M}$. The GSD game (for public-key encryption schemes) is a two-party game between a challenger C and an adversary $\mathcal{A}$. On input an integer N , for each $v \in [\![1, N]\!]$ the challenger C picks

[22]With TreeKEM, any user can remove any other one without justification. Even if the application level restricts this right to some administrators within the group, impersonating these administrators gives the adversary a full control over the group composition.

[23]The roots of a CGKA-graph are the root of each generation of Ratchet Tree that composes that graph.

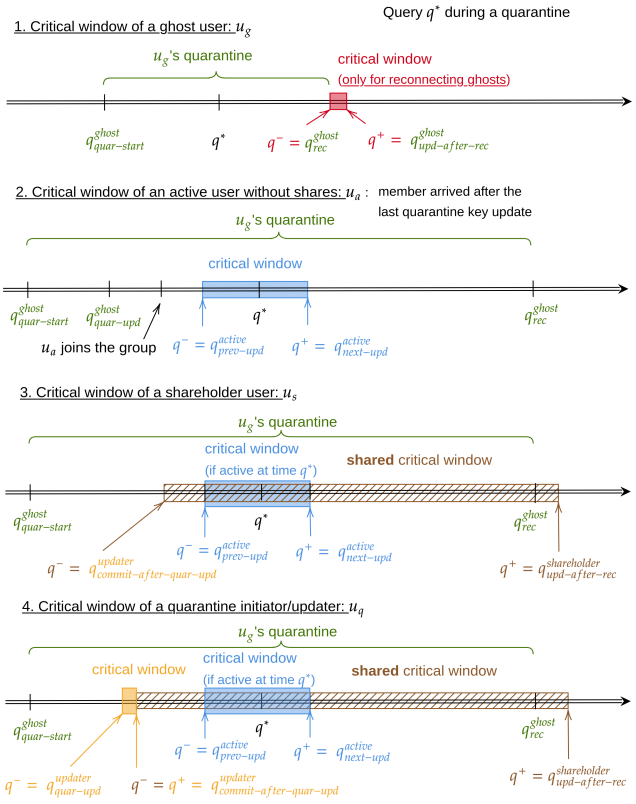

Figure 8: Critical windows, for our QTK protocol, of various types of users in the view of a user u^* issuing a group key k^* at time q^*. Brown crosshatched boxes represent "shared critical windows", specific to QTK protocol.

a key-pair $(pk_v, sk_v) \leftarrow \text{KeyGen}(r)$ (where r is a random seed) and initializes the key graph $G = (V, E) := ([\![1, N]\!], \emptyset)$ and the set of corrupted users $Cor = \emptyset$. $\mathcal{A}$ can adaptively do the following queries:

- (**encrypt**, u, v): On input two nodes u and v, C returns an encryption $c = Enc(pk_u, sk_v)$ of sk_v under pk_u along with pk_u and adds the directed edge (u, v) to E. Each pair (u, v) can only be queried at most once.
- (**corrupt**, v): On input a node v, C returns sk_v and adds v to Cor.
- (**challenge**, v), single access: On input a challenge node v, C samples $b \leftarrow\$ 0, 1$ uniformly at random and returns sk_v if $b = 0$, otherwise it returns a new secret key generated by KeyGen using a new independent uniformly random seed.

In the context of GSD we denote the challenge graph as the graph induced by all nodes from which the challenge node v is reachable. We require that none of the nodes in the challenge graph are in Cor, that G is acyclic and that the challenge node v is a sink. Note that $\mathcal{A}$ does not receive the public key of the challenge node, since it is a sink.

Finally, $\mathcal{A}$ outputs a bit b_0 and it wins the game if $b_0 = b$. We call the encryption scheme $G(\epsilon, t)$-adaptive GSD-secure if for any adversary $\mathcal{A}$ running in time t it holds:

$$Adv_{GSD}(\mathcal{A}) := |\Pr[1 \leftarrow \mathcal{A}|b = 0] - \Pr[1 \leftarrow \mathcal{A}|b = 1]| < \epsilon$$

Theorem 3 from [21] beneath is a general result proving that a GSD graph where edges are constructed by encrypting, with an IND-CPA secure public-key encryption scheme, secret seeds belonging to some nodes, and where public-private key-pairs are derived from that same seeds *via* a random oracle, is GSD-secure. However, in such a graph, all the seeds are random and independent. [21] consequently adapts that result to their Tainted TreeKEM protocol by considering that the seeds are derived the one from another through a random oracle H_1, different from the one H_2 used to generate the key-pairs. This leads to Theorem 4 detailed hereunder, where the GSD graph corresponds to the CGKA graph of their protocol.

Theorem A.2 (Theorem 3 from [21]). *For any public-key encryption scheme* $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$ *and hash function* H*, let the encryption scheme* $\Pi' = (\text{KeyGen}', \text{Enc}', \text{Dec}')$ *be defined as follows:*

(1) KeyGen' *simply picks a random seed* s *as secret key and runs* $\text{KeyGen}(H(s))$ *to obtain the corresponding public key,*
(2) Enc' *is identical to* Enc *and*
(3) Dec'*, given the secret key* s*, extracts the secret key from* $\text{KeyGen}(H(s))$ *and uses* Dec *to decrypt the ciphertext.*

If Π *is* $(\tilde{\epsilon}, t)$*-IND-CPA secure and* H *is modelled as a random oracle, then* Π' *is* (ϵ, t)*-adaptive GSD-secure, where* $\epsilon = 2N^2 \cdot \tilde{\epsilon} + \frac{mN}{2^{\ell-1}}$*, with* N *the number of nodes,* m *the number of oracle queries to* H*,* ℓ *the seed length.*

Theorem A.3 (Theorem 4 from [21]). *If the encryption scheme in TTKEM is* $(\tilde{\epsilon}, t)$*-IND-CPA secure and* H_1*,* H_2 *are modelled as random oracles, then TTKEM is* (Q, ϵ, t)*-CGKA-secure, where* $\epsilon = \tilde{\epsilon} \cdot 8(nQ)^2 +$ negl.

We underline that, as explicitly stated by [21], the above Theorem 4 is applicable to TreeKEM or other similar CGKAs – such as our QTK protocol –, with a potentially different security bound that only depends on the number of nodes in the CGKA graph (but not on its structure).

B JOINTLY-IMPLEMENTED QUARANTINE

We present here an improvement of the original QTK CGKA, called *jointly-implemented quarantine*, that upgrades the security of QTK by replacing the initial *proper* critical window of a ghost's quarantine updater by ℓ more secure *shared* critical windows related to ℓ updaters. This method, which uses a secretly key-updatable PKE (cf. Section 2.3) to generate the ghost's quarantine encryption key-pair, has a communication cost increased by a factor ℓ for the initialization and each update of the quarantine. However, this communication overhead is mitigated by the fact that the higher security provided by this tweak permits to space out the intermediate updates of the quarantine keys.

B.1 Overview

As recalled in Figure 9, a quarantine updater[24] has – beside the classical critical window centered around q^* that every active user has in any CGKA protocol – a critical window that corresponds to the generation of the ghost's quarantine key-pair by this updater. Therefore, until this window closes when the updater deletes the secret key and seed (after sharing that seed), any corruption of this particular user compromises the newly generated ghost's encryption key, which impacts both forward secrecy and post-compromise security.

A ℓ-jointly-implemented quarantine reduces this vulnerability by having ℓ several users (generally two) generate in common the ghost's quarantine keys, in a way such that none of these users knows the secret keys or seeds.

B.1.1 Initialization and Update. The initialization or the update of a ℓ-jointly-implemented quarantine is effective after ℓ epochs of preparation, using a secretly key-updatable PKE instead of a regular PKE. We describe below the process of a quarantine initialization, knowing that a quarantine key update is processed similarly.

(1) At epoch e^i, the committer $u_c^i = u_{init_0}$ that decides to quarantine a new ghost u_g proceeds to a regular quarantine initialization as described in Section 3.3, except that the quarantine does not start at the following epoch but ℓ epochs after (at epoch $e^{i+\ell}$).
   - u_{init_0} generates a *temporary* fresh encryption key-pair $(\widehat{pk}_g^i, \widehat{sk}_g^i)$ for the new ghost $u_g \in \mathcal{N}_{\mathcal{G}}{}^{i+\ell}$:
   $$\hat{s}_g^i \xleftarrow{\$} \mathcal{S}$$
   $$(\widehat{pk}_g^i, \widehat{sk}_g^i) := \text{KeyGen}(1^\lambda; \hat{s}_g^i)$$
   - It distributes to all group members the temporary public key $\widehat{pk}_g^i$.
   - It also sends to the group the shares issued from the secret seeds at the origin of the ghost's key-pair:
   $$[\hat{s}_g^i] \leftarrow \text{Distr}(\hat{s}_g^i, t, m)$$
   - Right after, it deletes from its local state the ghost's quarantine secret seed and private key.

(2) At epochs e^{i+1} to $e^{i+\ell-1}$, the committers $u_c^{i+j} = u_{init_j}$ ($j \in [\![1, \ell-1]\!]$) – which must be different from u_{init_0} and from each other – continue the process of quarantine initialization:
   - u_{init_j} generates fresh update elements from a randomly drawn seed.
   $$\hat{s}_g^{i+j} \xleftarrow{\$} \mathcal{S}$$
   $$(\Theta_g^{i+j}, \theta_g^{i+j}) := \text{UpdGen}(1^\lambda; \hat{s}_g^{i+j})$$
   - It updates, with the public update element Θ_g^{i+j}, the ghost's temporary public key sent by the previous initiator $u_{init_{j-1}}$:
   $$\widehat{pk}_g^{i+j} := \text{UpdPk}(\widehat{pk}_g^{i+j-1}, \Theta_g^{i+j})$$
   - It distributes to all group members the new temporary quarantine public key $\widehat{pk}_g^{i+j}$.
   - It shares within the group the seed $\hat{s}_g^{i+j}$ used to generate the update elements $(\Theta_g^{i+j}, \theta_g^{i+j})$:
   $$[\hat{s}_g^{i+j}] \leftarrow \text{Distr}(\hat{s}_g^{i+j}, t, m)$$

[24] The term "quarantine updater" also comprises the "quarantine initiator".

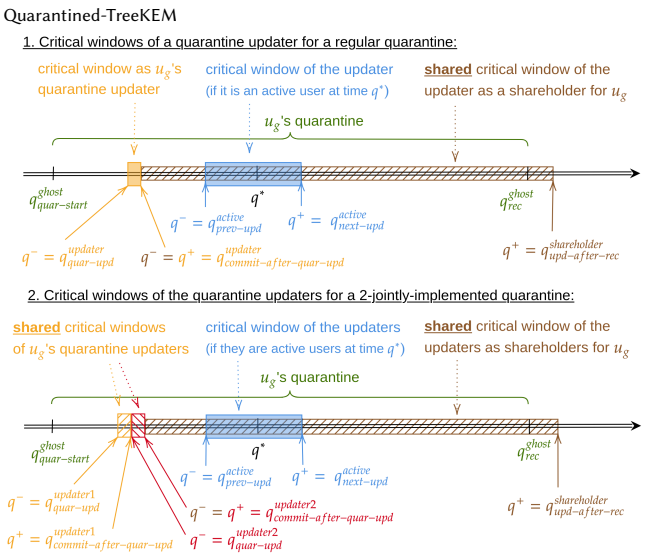

Figure 9: Compared critical windows for the initialization of a regular quarantine and a 2-jointly-implemented quarantine. In the latter case, the single updater's proper critical window is replaced by the *shared* critical windows of the two updaters u_{init_0} and u_{init_1}.

- It deletes from its local state the seed $\hat{s}_g^{i+j}$ and the private update element θ_g^{i+j}.

(3) The quarantine is then effective at epoch $e^{i+\ell}$, with a ghost's quarantine encryption key-pair corresponding to the temporary key-pair of epoch $e^{i+\ell-1}$:

$$(pk_g^{i+\ell}, sk_g^{i+\ell}) := (\widehat{pk}_g^{i+\ell-1}, \widehat{sk}_g^{i+\ell-1})$$

Nota: If a ghost comes back online before epoch $e^{i+\ell}$, its quarantine initialization process instantly aborts and it can immediately recover its offline history as with TreeKEM.

B.1.2 Quarantine End. At the end of the quarantine, the reconnecting ghost recovers from the shareholders a sufficient number of shares associated to its quarantine:

- shares of the seed $\hat{s}_g^i$ that was used to generate the first temporary key-pair $(\widehat{pk}_g^i, \widehat{sk}_g^i)$.
- shares of the seeds $(\hat{s}_g^{i+j})_{j\in[\![1,\ell-1]\!]}$ associated with all the update elements $(\Theta_g^{i+j}, \theta_g^{i+j})_{j\in[\![1,\ell-1]\!]}$.

It reconstructs the secret seeds $(s_g^{i+j})_{j\in[\![0,\ell-1]\!]}$ associated with these share collections. Then, it recomputes the initial temporary private key $\widehat{sk}_g^i$ from the secret seed $\hat{s}_g^i$ and updates it $\ell-1$ times with the reconstructed secret update elements $\theta_g^{i+1}, \cdots, \theta_g^{i+\ell-1}$, in order to get the final private key $sk_g^{i+\ell}$:

$$\begin{aligned}
&\hat{s}_g^i := \text{Comb}([\hat{s}_g^i])\\
&(\widehat{pk}_g^i, \widehat{sk}_g^i) := \text{KeyGen}(1^\lambda; \hat{s}_g^i)\\
&\forall j \in [\![1, \ell-1]\!] :\\
&\quad \hat{s}_g^{i+j} := \text{Comb}([\hat{s}_g^{i+j}])\\
&\quad (\Theta_g^{i+j}, \theta_g^{i+j}) := \text{UpdGen}(1^\lambda; \hat{s}_g^{i+j})\\
&\quad \widehat{sk}_g^{i+j} := \text{UpdSk}(\widehat{sk}_g^{i+j-1}, \theta_g^{i+j})\\
&sk_g^{i+\ell} := \widehat{sk}_g^{i+\ell-1}
\end{aligned}$$

B.2 Security

As none of the quarantine updaters has access to the ghost's quarantine secret key $sk_g^{i+\ell}$ but only to intermediate elements (indeed, u_{init_0} only knows the first temporary private key $\widehat{sk}_g^i$ and $(u_{init_j})_{j\in[\![1,\ell-1]\!]}$ know nothing but their associated secret update element θ_g^{i+j}), the corruption of all but one of them does not give the adversary any clue to recover the quarantine private key. The only way for the adversary to recover this private key is to corrupt each one of these updaters u_{init_j} precisely during their critical window at epoch e^{i+j}.

We consequently consider that these updaters do not have anymore a proper critical window related to the quarantine initialization or update, but a shared critical window with a *full recovery threshold*[25].

B.3 Performances

In this jointly-implemented quarantine variant, the communication costs of the quarantine initialization, of each update and of the "Share Recovery Messages" in the reconnection process scale almost linearly with the number of co-initiators and co-updaters involved.

Consequently, the communication cost of a ℓ-jointly-implemented quarantine is increased as follows, compared to a regular broadcast-only[26] quarantine:

$$\begin{aligned}
cc^{\ell-joint}_{init-upd} &= \ell.cc_{init-upd}\\
cc^{\ell-joint}_{end} \in \big[\ & |upd| + \ell.t.\big(|sig| + |ct| + n_{qkey}.(|s| + 2.|int|)\big),\\
& |upd| + n^{max}_{resend}.\big(|sig| + 2.\ell.n_{qkey}.t.|int|\big)\\
& + \ell.m.n_{qkey}.\big(|sig| + |ct| + |s| + 2.|int|\big)\big]
\end{aligned}$$

However, as stated above, this overhead is mitigated by the possibility to decrease the quarantine key update frequency, due to the higher security brought by the jointly-implemented quarantine variant.

[25] Which means that all these users without exception must be corrupted during their critical window so that the adversary recovers the ghost's quarantine key.
[26] The improvement is similar with the server-aided variant.