The Art of Bonsai: How Well-Shaped Trees Improve the Communication Cost of MLS

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

Jérôme Plût
ANSSI
Paris, France
jerome.plut@ssi.gouv.fr

ABSTRACT

Messaging Layer Security (MLS) is a Secure Group Messaging protocol that uses for its handshake a binary tree – called a Ratchet Tree – in order to reach a logarithmic communication cost w.r.t. the number of group members. This Ratchet Tree represents users as its leaves; therefore any change in the group membership results in adding or removing a leaf associated with that user. MLS consequently implements what we call a *tree evolution mechanism*, consisting in a *user add* algorithm – determining where to insert a new leaf – and a *tree expansion* process – stating how to increase the size of the tree when no space is available for a new user.

The tree evolution mechanism currently used by MLS is designed so that it naturally left-balances the Ratchet Tree. However, such a Ratchet Tree structure is often quite inefficient in terms of communication cost. Furthermore, one may wonder whether the binary tree used in that Ratchet Tree has a degree optimized for the features of a handshake in MLS – called a *commit*.

Therefore, we study in this paper how to improve the communication cost of a commit in MLS by considering both the tree evolution mechanism and the tree degree used for the Ratchet Tree. To do so, we determine the tree structure that optimizes its communication cost, and we propose optimized algorithms for both the user add and tree expansion processes, that allow to remain close to that optimal structure and thus to have a communication cost as close to optimal as possible.

We also determine the Ratchet Tree degree that is best suited to a given set of parameters induced by the encryption scheme used by MLS. This study shows that when using classical (i.e. pre-quantum) ciphersuites, a binary tree is indeed the most appropriate Ratchet Tree; nevertheless, when it comes to post-quantum algorithms, it generally becomes more interesting to use instead a ternary tree.

Our improvements do not change TreeKEM protocol and are easy to implement. With parameter sets corresponding to practical ciphersuites, they reduce TreeKEM's communication cost by 5 to 10%. In particular, the 10% gain appears in the Post-Quantum setting – when both an optimized tree evolution mechanism and a ternary tree are necessary –, which is precisely the context where any optimization of the protocol's communication cost is welcome, due to the important bandwidth of PQ encrypted communication.

KEYWORDS

MLS, TreeKEM, CGKA, Key Tree, Binary Tree, Ternary Tree

1 INTRODUCTION

With the development of Secure Messaging protocols that enable end-to-end encrypted communication between two users, the need of a dedicated Secure Group Messaging protocol has arisen in the last few years. Indeed, the group conversation functionality offered in practice by (point-to-point) Secure Messaging applications relies on *ad hoc* constructions that either lower the protocol's security level – such as the *SenderKey* protocol, used e.g. by WhatsApp [1] – or that are quite inefficient – like the *Pairwise protocol* of Signal application –, with a communication cost scaling linearly with the number n of group members.

Messaging Layer Security (MLS) [7] is an IETF standard, released in July 2023, for a Secure Group Messaging protocol that is designed to keep the security properties of point-to-point Secure Messaging protocols, among which Forward Secrecy (FS) and Post-Compromise Security (PCS) (cf. Appendix A.2.1), with a communication cost growing logarithmically with the number of users: $O(\log_2(n))$.

For this purpose, MLS uses as core component a group key exchange mechanism – conceptualized by [4] as a *Continuous Group Key Agreement* (CGKA) – named TreeKEM, that provides the group with a common group key that securely evolves over time in order to maintain the aforementioned forward secrecy and post-compromise security, despite the changes in the group membership.

1.1 Optimizing Group Communication with a Key Tree

1.1.1 TreeKEM's Ratchet Tree. In order to reach the aimed communication cost of $O(\log_2(n))$, TreeKEM relies on a binary tree called a Ratchet Tree, whose leaves represent the group members and where all nodes are associated with encryption key-pairs that are hierarchically organized: each key-pair is deterministically generated from a secret seed that is directly derived – through a key derivation process – from one of the seeds associated with the two

children nodes located beneath. A central concept in TreeKEM, called the *tree invariant*, is that any user (leaf) in the tree knows all the secret elements (secret seeds and secret encryption keys) from the nodes above them, and only them. Consequently, the secret seed at the root of the tree is the only secret element known by all users; the group key is directly derived from that common value.

The idea behind a key tree such as TreeKEM's Ratchet Tree is that the rotation of the group key, initiated after the arrival or departure of a group member or the update of a user's keying material in a process named a *commit*, impacts the seeds and keys of all the nodes located above that updater, up to the root, in what is called its *direct path*. Consequently, the public and private information[1] that needs to be transmitted to the group is proportional to the number of nodes in the updater's direct path. In a binary tree, the average length of a user's direct path is in $O(\log_2(n))$, which corresponds to the expected logarithmic communication cost of that CGKA.

1.1.2 Influence of the Tree Structure. In practice, the lower bound of $\log_2(n)$ is rarely reached. This ideal value indeed corresponds to the cost of a *perfect* binary tree, i.e. a tree whose nodes all have exactly two children and where the leaves are all located at the same level.

Not only is this ideal tree impossible when the number of users is not a power of two, but for a given number n of users, the tree structure, i.e. the unordered arrangement in space of the leaves, also has a significant influence on the communication cost. The structure of a CGKA's key tree is determined by the way the protocol adapts this tree to the natural evolution of the group membership, with user departures at arbitrary locations and user arrivals at locations specified by the protocol. A CGKA's *tree evolution mechanism* therefore consists of the following algorithms[2]:

- a *user add* algorithm, which specifies where to add a new user among all the available locations;
- a *tree expansion* algorithm that increases the size of the tree when a new user has to be added to the group and there is no free space available for it, and that decreases the tree size whenever it is possible.

In this regard, TreeKEM adds new users at the leftmost available location in the tree. When needed, it increases the Ratchet Tree to the right, by creating a new root above the current one and by adding the extra leaves in a new subtree attached to the right of that new root. This mechanism left-balances TreeKEM's Ratchet Tree, by grouping most leaves on the left side of the tree.

However, we show below that a left-balanced structure does not minimize the communication cost of a CGKA, and consequently, that the tree evolution mechanism used by TreeKEM is not optimal in that matter.

1.1.3 Influence of the Tree Degree. The degree, or arity, of a tree is the maximum number of children that any node in the tree is allowed to have. The aforementioned lower bound of $\log_2(n)$ stems from the fact that TreeKEM uses a binary tree as its Ratchet Tree. This optimal bound varies according to the tree degree: the higher the degree, the shorter the average direct path of a user becomes, which decreases the number of updated public keys that have to be broadcasted to the entire group.

However, this advantage comes at a cost. For instance, in CGKAs using Diffie-Hellman (DH) trees (cf. Section 1.3), a tree of degree m implies that any internal node's secret must be computed with a key agreement relying on the m-party generalized Group Diffie-Hellman problem, which is more costly – computationally speaking as well as regarding the bandwidth consumption – than a standard 2-party DH key agreement.

In the case of TreeKEM, increasing the tree degree also increases the number of recipients to whom the encrypted secret seeds associated with the nodes of the updater's direct path must be transmitted. This number of recipients corresponds to the number of nodes in the updater's copath, i.e. the number of sibling nodes of this user and of its ancestors (cf. Section 2.1).

The optimal tree degree for TreeKEM is therefore the one that offers the best trade-off between the number of nodes in a user's direct path and the number of nodes in its copath, for all users in the tree. However, to the best of our knowledge, no study has been carried out to determine precisely the optimal degree for TreeKEM.

1.2 Our Contributions and Outline of this Paper

We study in this paper how to optimize the communication cost of TreeKEM by considering the two factors of tree structure and tree degree detailed above.

We focus in Section 3 on the optimization of a binary tree, as already used by TreeKEM for its Ratchet Tree. To do so, we start by defining in Section 3.1 a precise communication cost metric, adapted to the specificities of TreeKEM and, in particular, that takes into account the two diverging notions of direct path length and copath length. We then use that metric to determine in Section 3.2 the optimal structure of a full[3] tree of an arbitrary degree $m \geq 2$, i.e. the one that minimizes the communication cost of a commit in a group of n users. In Section 3.4 we propose an optimized tree evolution mechanism, based on simple, yet effective, *user add* and *tree expansion* algorithms that are applicable to trees of any degree m. With an implementation in Python, we experimentally show that the resulting optimized communication cost only exceeds the optimal one by around 0.5%, whereas TreeKEM has an average cost more than 5% above that ideal cost.

In Section 4, we study the influence of the tree degree on the communication cost of TreeKEM, with respect to our metric, in the most general case of *non-full* trees that corresponds to the feature of real Ratchet Trees. We prove that binary and ternary (non-full) trees are the most efficient Ratchet Trees in the most common use cases[4]. We also compare, theoretically and experimentally, these two degrees and determine a bound for the ciphersuite parameter τ, that separates their respectives areas of optimality and underlines that for most post-quantum ciphersuites, it is more relevant to use a ternary tree than a binary one.

[1] The public elements mentioned here are the updated public keys and the private ones are the secret seeds related to that public keys.

[2] The case of a user removal is not considered in a tree evolution mechanism, since the protocol does not control the departure of users from the group and thus cannot choose which leaves to remove from the Ratchet Tree.

[3] As detailed later, a *full* tree of degree m is a tree where any node has either zero or m children.

[4] All standard encryption schemes that we have considered are indeed adapted to these two tree degrees, except for the HPKE ciphersuite based on the post-quantum KEM *ClassicMcEliece*.

Our experimental results show that in that context, the combination of using our optimized tree evolution algorithms and replacing the current binary tree with a ternary one, brings an average gain of around 10% compared to the current implementation of TreeKEM, at the expense of only few additional computations demanded by the optimized algorithms. This gain is highly appreciable since it occurs in the post-quantum framework, which has an important bandwidth and thus for which any improvement is welcome.

1.3 Background on Key Trees

The idea to use key graphs in order to decrease the communication cost of a group key exchange goes back long before MLS: the seminal concurrent works of [20] and [35] introduce that concept – called *Logical Key Hierarchy* – in the late 1990s, in the context of a centralized key distribution system, where a key server manages the creation and distribution of all keys in the graph.

In the decentralized framework, where peers mutually interact without any central authority in order to generate a common group key, [6], followed by the ELK protocol [30], proposes a new tree-based architecture called *One-Way Function Tree* (OFT) in which any node key is no longer randomly generated by a central entity, but can be deterministically computed, with the use of symmetric primitives such as pseudo-random functions, from the keys belonging to that node's children.

1.3.1 Diffie-Hellman Trees. The seminal works of [29] (*NAGKA*) and [23] (*TGDH*) pave the way to Diffie-Hellman trees by combining the concepts of children-dependent key graph and (2-party) DH key agreement. A number of subsequent papers continue in this vein, [24], [37], [16], [14] among others. However, ART [15] is the first fully asynchronous DH-tree-based group key agreement protocol, that no longer requires all users to be online for every group operation and additionally provides the expected security property of post-compromise security. This protocol has been chosen as MLS's CGKA in its initial IETF RFC draft [9].

1.3.2 TreeKEM Protocol. TreeKEM [11] is a versatile CGKA that uses a binary tree in which the cryptographic primitives – and in particular, a public-key encryption scheme (PKE) – are used as black boxes. Consequently, it is not linked to any particular cryptographic assumption and offers the crypto-agility needed to overcome the failure of any used primitive. This protocol constitutes MLS's CGKA since version two of its RFC draft [10], until the current IETF standard [7].

1.3.3 Tree Degree Variations. Within the large body of literature devoted to tree-based group key agreement, only few papers seem to have questioned the use of binary trees for this type of protocol, notably because degree 2 is best suited for the numerous protocols relying on the (2-party) Diffie-Hellman problem. The early work of [35] studies the best key graph in the framework of centralized Logical Key Hierarchy, and reaches the conclusion that a rooted-tree of degree 4 is the best architecture for their *Secure Group*. Their conclusion is however inadapted to our present study, since their analysis is based on the computational cost of the protocol and not on the communication one.

The main alternate propositions to binary trees are protocols using the Bilinear Diffie-Hellman problem [26], [22], after a pairing-based three-party DH variant has been proposed by [21]. For its part, [34] uses, inside a ternary tree, the *GDH.2* protocol of [33] based on the generalized Group Diffie-Hellman problem, and shows that this architecture is more efficient in terms of communication cost than a binary Diffie-Hellman tree. [25] extends the latter work by allowing a number of group members different than a power of three, using non-full ternary trees. No analysis is however carried out regarding the most efficient structure of that tree.

Regarding TreeKEM, the authors [11] specify that their protocol works with trees of any degre; however it has been used by MLS with binary trees only.

2 PRELIMINARIES

2.1 Trees

A tree is defined in graph theory as a connected acyclic graph, i.e. a set of nodes (a.k.a. vertices) that are all connected to each other by exactly one edge. In a rooted tree, one of the vertices is designated as the tree *root*; when the tree edges are directed towards the root, the tree is called in-tree. Otherwise, it is an out-tree.

2.1.1 Trees Used in a CGKA. We consider in this paper rooted out-trees T – simply called *trees* in the following pages –, where the root is located on top of the tree. Each node descends from a parent node above it, and has under it zero or several children, in which cases this node is called respectively a leaf and an internal node.

Definition 2.1 (Tree). A (rooted) tree T is recursively defined as either a leaf ℓ or a finite ordered set of $b \in \mathbb{N}^*$ subtrees – called its branches – linked together by a root: $T = \ell \mid (B_i)_{i \in [\![1,b]\!]}$.

We exclude from our study the special case where nodes are allowed to have a single child. To do so, we define below the concept of *non-linear trees*, where this unwanted structure is excluded. We adopt this restriction in the tree structure because the Ratchet Tree used in TreeKEM-based CGKAs does not allow such an architecture: indeed, internal nodes in that Ratchet Tree are used to detain intermediate encryption keys common to a subgroup of leaves. When a node has only one child, the encryption key of that node and the one of its child are related to the same subgroup of leaves beneath them and thus appear redundant.

Definition 2.2 (Non-Linear Tree). A tree is said to be non-linear if none of its nodes has a single child: $T^{nl} = \ell \mid (B_i)_{i \in [\![1,2 \leq b]\!]}$.

2.1.2 Tree features. We define below the basic features characterizing trees.

Node and Tree Degree. The number of children belonging to a node v is referred to as that node's degree, noted $deg(v)$. The maximum number of children that any node can have, i.e. the maximum node degree in the tree, is called the tree degree, or tree arity and is noted $m = deg(T)$.

Full and Perfect Trees. A full tree of degree $m \geq 2$ is a tree where each node can only have 0 or m children[5].

[5] A full tree is therefore a particular case of a non-linear tree from Definition 2.2.

A perfect m-ary tree is a full tree whose leaves are all located at the same depth (cf. below).

Non-Planar Tree. A non-planar tree is a tree that remains unchanged by the permutation of its branches: $T = \ell \mid \{B_i\}_{i \in [\![1,b]\!]}$.

The Ratchet Tree used in CGKAs is a planar tree, where the order of the leaves corresponds to the users' indices. However, as horizontally permuting the Ratchet Tree's internal nodes and leaves (i.e. the subtrees B_i in the recursive view) has no impact on that tree's communication cost, we only consider in this paper non-planar trees.

Node and Tree Heights. The height $h(v)$ of a node v is defined as the length (i.e. the number of nodes) in the longest downward path from this node to a leaf descending from that node. A leaf consequently has a height zero.

The height of a tree T, noted $h(T)$ or simply h, corresponds to the height of its root. It is thus the length of the longest path between the root and one of the leaves in the tree.

Node Depth. The depth $d(v)$ of a node v is the length of its path up to the tree root. Consequently, the root itself has a depth zero, whereas the lowest leaves in the tree have a depth equal to the tree height. Node height and depth are linked by the following relation: $\forall v \in T,\ h(v) = h(T) - d(v)$.

Weighted Tree and Weight Function. A weighted tree is a tree where each leaf is associated with a coefficient $c_\ell \in \mathbb{R}$ labeled as its weight. It is a generalization of a non-weighted tree, the later having all leaves associated with a weight $c_\ell = 1$.

The weight function is recursively defined as follows:

$$w(T) = \begin{cases} c_\ell & \text{if } T = \ell \\ \sum_{i=1}^{b} w(B_i) & \text{if } T = (B_i)_{i \in [\![1,b]\!]} \end{cases} \quad (1)$$

Nota: The weight of a non-weighted tree represents the number of leaves of that tree. Similarly, the weight of an internal node in a non-weighted tree corresponds to the number of leaves in the subtree rooted at that node.

2.2 TreeKEM's Ratchet Tree

A detailed description of how TreeKEM – as standardized in RFC 9420 [7] – works as a CGKA protocol, is given in Appendix A.2. We focus hereunder on the Ratchet Tree used by TreeKEM.

2.2.1 Ratchet Tree. As stated above, the key tree used by TreeKEM to perform its group key agreement – called the Ratchet Tree– is a full binary rooted tree where users are represented by the leaves and the group key is computed at the root. Each node of this Ratchet Tree, except for the root, is associated with a local state γ that notably includes an encryption key-pair. This one is issued from a seed called a *path secret*, that is itself derived from the one of the node's children. Details on a node's state are given in Appendix A.2.

Formal and Logical Tree Representations. For practical reasons, MLS standard represents a Ratchet Tree as a perfect binary tree, for all numbers of users n and whatever their relative locations in the tree. Consequently, the 2^h leaves in such a tree of height h are not all associated with a group member – except when $n = 2^h$ – and some of them therefore have an empty state; these leaves are called *blank*. Moreover, as TreeKEM's Ratchet Tree is non-linear, an internal node cannot have a single child. Consequently, if at least one of an internal node's children and all this child's descendants are blank, then that internal node itself is blanked.

Since their state is empty, blank nodes do not take part in TreeKEM's processes until they are filled again. A perfect tree with blank nodes can thus be represented, logically speaking, by a single non-perfect tree where blank nodes are removed and where leaves that previously had blank ancestors are attached higher in the tree[6] (cf. Figure 1). The latter representation is called *logical representation* whereas the architecture specified in MLS standard is named *formal representation* and corresponds to the practical implementation of that tree.

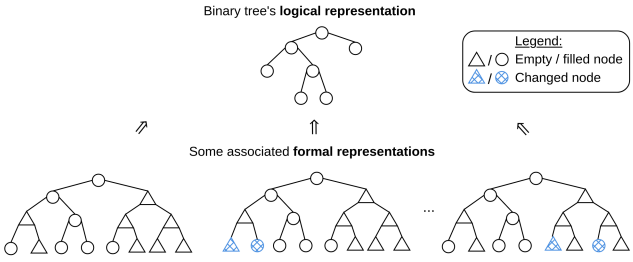


Figure 1: Correspondence between a tree's *logical* representation and the various *formal* representations associated with it (only some of which are depicted in this instance).

The use of blank nodes with the formal representation of TreeKEM's Ratchet Tree requires to adapt the concepts of direct path and copath of a node (cf. below). This is done *via* the notion of *resolution*, detailed beneath:

Resolution of a Node (from [4]). The resolution of a node v from a 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 Node. A node's direct path, noted $\mathcal{P}_v$, is composed of all the ancestors of that node, up to the root. A node's copath $C\mathcal{P}_v$ contains the sibling(s) of that node and the ones of its ancestors (i.e. of the nodes belonging to its direct path).

In a formal tree representation, a node's direct path may include blank nodes that must not be taken into account. Consequently, MLS standard [7] defines the notion of a node's *filtered direct path*, which is that node's direct path from which all nodes that have a child with an empty resolution are removed. The filtered direct path is thus the transcription, in a formal tree representation, of the concept of direct path used in logical representation.

Nota: In the remainder of this document, we generically use the term *direct path* to refer both to the standard direct path in logical tree representation and to the filtered direct path in formal representation, as they are equivalent.

[6]The converse is not true: as the formal representation is more precise regarding the leaf arrangement than the logical one, a single logical topology may correspond to several formal representations that are logically equivalent.

2.2.2 Ratchet Tree Evolution: the Commits. The update of TreeKEM's Ratchet Tree, in order to follow the evolution of the group membership as well as to periodically refresh users' keying material, is realized *via* a *commit* that groups together and validates proposals for change sent by any group member.

As shown in the detailed description of a commit, in Appendix A.2, a commit refreshes the committer's direct path by updating the path secret of all nodes in its direct path. These refreshed path secrets must then be encrypted and sent to all users that are located beneath these nodes. Figure 13 in Appendix A.2 depicts this update process.

Broadcast of a commit message to the Ratchet Tree. All information related to the commit is grouped within a single *commit message* C that is broadcasted to the group, and that 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 direct path;
- the path secrets ps_{v_p} of the nodes $v_p \in \mathcal{P}_c$ from the committer's direct path, each encrypted, with the HPKE ciphersuite at use, under the public key of v_p's child v_r on the committer's copath[7].

$$C(u_c) = \mathbb{P} \,||\, {}^{p}\gamma'_c \,||\, \left(pk'_{v_p}\right)_{v_p \in \mathcal{P}_c} \,||\, \left(\mathrm{Enc}(pk_{v_r}, ps_{v_p})\right)_{v_p \in \mathcal{P}_c}^{v_r \in \mathcal{R}_{v_p}} \quad (2)$$

Consequently, decreasing the amount of data exchanged during the evolution of the Ratchet Tree comes to minimizing the size of the commit message. That size partly depends on the number of path secrets that need to be encrypted and on the number of different ciphertexts sent for each path secret, which themselves depend on the structure of the Ratchet Tree. The next part of our paper is therefore devoted to determining, with a specific metric, the tree-dependent factors for the size of the commit message, that are relevant for our study.

Encryption Process. MLS states that the encryption of private messages is carried out through the *HPKE* paradigm [8]: a Key Encapsulation Mechanism (KEM) generates a shared secret that is used as key and nonce for an AEAD[8]-based symmetric encryption of the plaintext.

The ciphertext size $|ct_m|$ generated by an HPKE scheme thus depends on the plaintext size $|m|$, on the size of the AEAD tag and on the KEM's features. In this paper, we focus on the encryption of the path secrets that must be transmitted within a commit. For a KEM $\mathcal{K}$, an AEAD scheme $\mathcal{E}$ yielding a tag of size $|tag|$ and a given path secret size[9] $|ps|$, we define the ratio τ as:

$$\tau^{hpke}_{\mathcal{K},\mathcal{E},|ps|} = \frac{\left|pk^{hpke}\right|}{\left|ct^{hpke}_{|ps|}\right|} = \frac{\left|pk^{\mathcal{K}}\right|}{\left|ct^{\mathcal{K}}\right| + \left|tag^{\mathcal{E}}\right| + |ps|} \quad (3)$$

The values of τ for the (classical) ciphersuites advised by MLS standard, as well as for the post-quantum ciphersuites that are most likely to be used, are detailed in Table 1 in Appendix C.

3 OPTIMIZING A TREE STRUCTURE

We study in this part the influence of a Ratchet Tree's structure, i.e. the disposition of its leaves, on its communication cost. With a metric defined in Section 3.1 that enables us to determine the communication cost associated with a tree, we show in Section 3.2 that a full tree *of any given degree* has an optimal structure, yielding a minimal communication cost, that we capture with a property we call *depth-balance*. We show in Section 3.3 that the tree evolution mechanism currently used in TreeKEM is not optimal regarding this cost function, and we propose in Section 3.4 optimized tree evolution algorithms to improve the efficiency of that CGKA, that we experimentally compare to TreeKEM's.

3.1 Communication Cost Metric

3.1.1 General Expression. Given the content of the commit message and the fact that it is broadcasted, the communication cost of a TreeKEM-based CGKA must take into account two factors:

- the number of public keys associated with the ancestor nodes of the committer, called the committer's *parent cost* and noted v_p;
- the number of encrypted *path secrets* that are transmitted to the nodes of the committer's copath, hereunder named its *sibling cost* and noted v_c.

Definition 3.1 (Parent Cost and Sibling Cost). The parent cost v^{ℓ}_p of a leaf is the number of nodes in that leaf's direct path. The sibling cost v^{ℓ}_c of that leaf is the number of nodes in its copath.

The parent cost v_p and the sibling cost v_c of a tree are the sums of respectively the parent costs and the sibling costs of all the leaves of that tree:

$$v_p(T) = \sum_{\ell_i \in T} v^{\ell_i}_p \quad \text{and} \quad v_c(T) = \sum_{\ell_i \in T} v^{\ell_i}_c \quad (4)$$

We recursively write the parent cost and the sibling cost of a m-ary tree T as follows, with $w(T)$ denoting its weight.

	$T = \ell$	$T = (B_i)_{i \in [\![1,b \le m]\!]}$
$v_p(T)$	0	$\sum_{i=1}^{b} \left(v_p(B_i) + w(B_i)\right)$
$v_c(T)$	0	$\sum_{i=1}^{b} \left(v_c(B_i) + (b-1) \cdot w(B_i)\right)$

Definition 3.2 (Communication Cost of a CGKA's Ratchet Tree). Let us consider a Ratchet Tree T used in a TreeKEM-based CGKA, with a set of n users $U = (u_i)_{i \in [\![1,n]\!]}$ associated with leaves $(\ell_i)_{i \in [\![1,n]\!]}$.

Let v_p and v_c be the parent and sibling costs of that tree, and $|pk|$ and $|ct|$ the sizes of respectively the public keys and ciphertexts yielded by the HPKE scheme used by the CGKA, and τ be their ratio, as defined in Section 2.2.2.

We define the communication cost c_T of a tree T as the sum of the commit message sizes $|C_i|$ when all users in the tree act as committers:

[7] MLS standard blanks all the nodes in the direct path of a leaving member. Consequently, until these internal nodes are one-by-one filled again by subsequent path updates initiated by the neighbors of the removed user, we may have in the tree several leaves joined together by a blank node, which corresponds, in logical representation, to attaching these leaves higher in the tree, and sometimes even directly to the root. In this case, we may need to encrypt several times the same path secret. However, as this structure is only temporary, we do not take it into account in our study and we consider that a path secret is only encrypted as many times as the node has children in the committer's copath (once in the case of a binary tree).

[8] Authenticated Encryption with Associated Data

[9] The size of a path secret is defined by MLS standard as the output of the Extract stage of the Key Derivation Function used within the selected ciphersuite. This value depends on the desired security level.

$$c_T = \sum_{\ell_i \in T} |C_i| = \nu_p\,|pk| + \nu_c\,|ct| + \iota = \kappa_T \cdot |ct| + \iota$$
$$\text{with } \iota = \sum_{\ell_i \in T} \left(\left|\mathbb{P}_{\ell_i}\right| + \left|{}^{p}\gamma_{\ell_i} \setminus \{pk_{\ell_i}\}\right| \right)$$
$$\text{and } \boxed{\kappa_T = \nu_p \tau + \nu_c} \tag{5}$$

Nota 1: As the Ratchet Tree's root does not have any public key attached to it, the number of public keys associated with a committer's direct path are $\nu_p^{\ell_c} - 1$. However, as a commit is always associated with a key update of the committer itself, we chose to include in the communication cost the committer's new public key pk_{ℓ_c}, which must therefore be removed from its public state ${}^{p}\gamma_{\ell_c}$ when the latter is comprised in the constant term ι.

Nota 2: For simplicity reasons, our analysis displayed in the following pages focuses on the term κ_T – often simply noted κ – that we call *normalized communication cost*, which does not take into consideration the ι and $|ct|$ factors that are independent of the tree structure.

3.1.2 Communication Cost of a Full m-ary Tree. In a full tree of a degree $m \geq 2$ (noted T_m^f), each node except the root has, by definition, exactly one parent and $m-1$ siblings. Consequently, for each leaf ℓ_i in the tree, a direct path of $\nu_p^{\ell_i}$ nodes yields a copath of $\nu_c^{\ell_i} = (m-1)\nu_p^{\ell_i}$ nodes. If we sum the encryption and parent costs for all users in the tree, we come to the following equation:

$$\boxed{T_m^f : \quad \nu_c = (m-1)\nu_p \quad \Rightarrow \quad \kappa = \nu_p(\tau + m - 1)} \tag{6}$$

Nota: In the particular case of a a full binary tree (T_2^f), the number of nodes in a leaf's copath is the same as the one of its direct path and the Ratchet Tree's communication cost is simplified into:

$$\boxed{T_2^f : \quad \kappa = \nu_p(\tau + 1)} \tag{7}$$

The equations above show that optimizing the communication cost of a CGKA using an encryption scheme with a parameter τ and whose Ratchet Tree has a given degree m, only comes to minimizing the sum ν_p of all direct paths lengths in the tree. This can be precisely reached by enhancing the tree balance according to a concept called *depth-balance*.

3.2 Optimal Structure of a Full m-ary Tree

3.2.1 Depth-Balance of a Full Tree. It is clear that the way a tree is balanced directly affects the length of some of – if not all – the leaves' direct paths, and therefore, the parent cost ν_p. The sibling cost ν_c is also affected by the balance of the tree.

As an instance, Figure 2 compares the values of ν_p for a totally unbalanced binary tree (left), which has a maximized height $h_{unbal} = n - 1$, and for a perfect binary tree that minimizes its height ($h = \log_2(n)$). This instance underlines the need to sum the sibling cost $\nu_p^{\ell_i}$ of all leaves: indeed, in an unbalanced tree, the leaves that have the lowest depth (i.e. that are the closest to the root) have a reduced direct path.

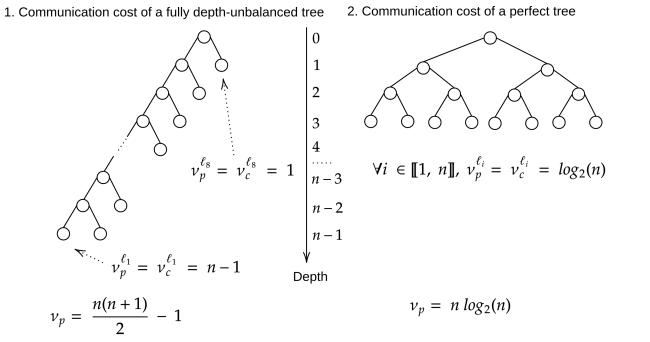


Figure 2: Comparison of the parent costs of a full binary tree bearing n leaves, with unbalanced (left) and balanced (right) structures.

We define below the concept of depth-balance, in the general case of a full m-ary tree. We then prove in Theorem 3.4 that a depth-balanced full m-ary tree has an optimal communication cost, and we determine that optimal cost.

Definition 3.3 (Depth-Balance of a Full Tree). Let us consider a full m-ary tree T, with an arbitrary degree $m \geq 2$. We say that this tree is depth-balanced if the difference of depth between its higher and lower leaves is lesser than or equal to one: $d_{max}(T) - d_{min}(T) \leq 1$.

Nota: A perfect m-ary tree is a particular case of a depth-balanced tree, that occurs when the number of leaves is precisely $n = m^k$ ($k \in \mathbb{N}$). It can be shown easily that with such a number of users, the only way for the tree to be depth-balanced is to be perfect.

3.2.2 Lower Bound of a Full Tree's Communication Cost. We prove hereunder that the lower bound of a full tree's communication cost is achieved when that tree is depth-balanced (Theorem 3.4) and we determine its communication cost in that case (Lemma 3.5).

Theorem 3.4. *Let T_m^f be a full m-ary tree with a given number of leaves n. Then T_m^f has an optimal (normalized) communication cost κ if and only if it is depth-balanced.*

Proof Sketch. Our proof, which is detailed – for lack of space – in Appendix B.1, is organized as follows :

- **Step 1**: We define a one-step bottom-up balancing process (depicted in Figure 14) that moves upwards a group of $m-1$ leaves from the bottom level of the tree. We show that, when applied on *any depth-unbalanced* full m-ary tree, this balancing process automatically decreases that tree's communication cost.
- **Step 2**: We then prove that in a *random depth-balanced* tree, any leaf movement – including the aforementioned balancing process – either leaves that tree unchanged[10] or increases its communication cost. These two steps prove that a depth-balanced structure has an optimal communication cost.

[10]Indeed, as it is detailed below, the resulting tree only undergoes a horizontal leaf reordering, which is not considered a modification as our communication cost study deals with non-planar trees that have no node ordering.

- **Step 3**: We finally prove, with a demonstration by contradiction, that the optimal cost of a tree is only achieved with a depth-balanced structure.

Lemma 3.5 (Optimal Communication Cost of a Full m-ary Tree). *The optimal (i.e. minimal) communication cost of a full m-ary tree T_m^f, of height $h = \lceil \log_m(n) \rceil$ and bearing n leaves, is:*

$$\kappa^{optim}(T_m^f) = \left(hn + \frac{n - m^h}{m-1}\right)(\tau + m - 1) \tag{8}$$

Nota: In the case of a full binary tree (T_2^f), this optimal communication cost becomes:

$$\kappa^{optim}(T_2^f) = \left(n(h+1) - 2^h\right)(\tau + 1) \tag{9}$$

Proof. According to Theorem 3.4, the optimal communication cost of a full m-ary tree corresponds to the cost of that tree with a depth-balanced structure (T_m^{db}). In that case, by definition, the n leaves of T_m are dispatched in two depths. Let x be the number of bottom leaves, at depth $d_{max} = h = \lceil \log_m(n) \rceil$, and y the number of top leaves at depth $d_{min} = h - 1$. We note that the particular case of a perfect tree can be easily deduced by stating $x = n$ and $y = 0$.

As we have $x + y = n$ and $x + my = m^h$, we can deduce the values of x and y as:

$$x = \frac{m(n - m^{h-1})}{m-1} \qquad y = \frac{m^h - n}{m-1} \tag{10}$$

Consequently, we have:

$$v_p^{optim}(T_m^f) = v_p(T_m^{f,db}) = xh + y(h-1) = hn + \frac{n - m^h}{m-1} \tag{11}$$

$$v_c^{optim}(T_m^f) = v_c(T_m^{f,db}) = (m-1)v_p(T_m^{f,db}) \tag{12}$$

$$\kappa^{optim}(T_m^f) = \kappa(T_m^{f,db}) = \left(hn + \frac{n - m^h}{m-1}\right)(\tau + m - 1) \tag{13}$$

□

3.3 Efficiency of TreeKEM's Binary Ratchet Tree

As stated above, TreeKEM's tree evolution mechanism adds a new user to the group by filling the leftmost blank leaf in the tree – in formal tree representation – and increases the tree size by adding a new root on top of the current one and attaching to the right of the latter a blank subtree whose leaves are then filled, from left to right, by the additional user(s). The tree is reduced when its entire right subtree is blank, in which case that subtree and the current tree root are removed.

This process tends to left-balance the Ratchet Tree, which appears, in several cases, not to be an effective manner to reach the depth-balance that optimizes the communication cost. This process may even worsen the tree's structure, as illustrated as instance by Figure 3.

3.3.1 Efficiency of TreeKEM's Tree Expansion. As the tree expansion mechanism is always applied on a perfect tree (in logical representation), it can be precisely studied without the need of a simulation. And it turns out that expanding a Ratchet Tree to the right is particularly inefficient in terms of communication cost – in fact, it is the most inefficient technique in this context.

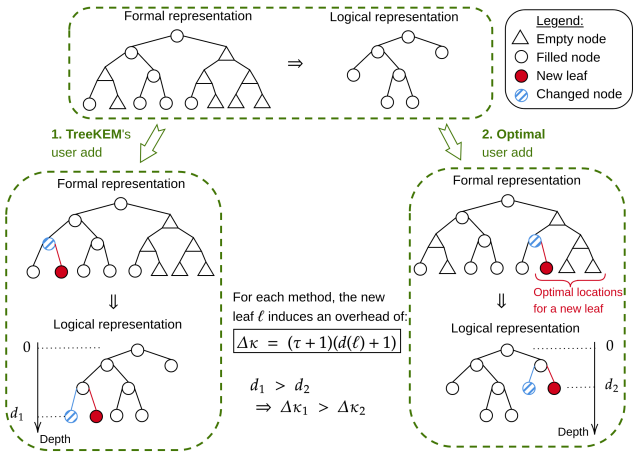


Figure 3: Communication cost of a user add in a full binary tree, with TreeKEM's left-balanced method (left) and the optimal one (right). Because this cost only depends on the depth at which the new user is inserted, left-balancing the Ratchet Tree often leads to costly tree structures.

Indeed, the perfect tree that needs to be expanded is depth-balanced and therefore yields an optimal communication cost. Keeping that balance implies adding any additional leaf at a depth whose difference from that of the other leaves is less than or equal to one. Intuitively, this means adding the leaf as close as possible to the bottom of the tree.

However, TreeKEM's mechanism does precisely the opposite. Adding a new user in a new right subtree attached to the root, whatever this user's location in that subtree, corresponds in logical representation to adding a leaf just below the root, at depth $d = 1$. As the previous leaves are all at depth $d = \log_2(n)$ (with n the original number of users), adding a user this way induces a communication overhead of $\Delta\kappa_{expand}^{TK} = (\tau+1)(n+1)$. This value must be compared to the optimal overhead – when the expanded tree remains depth-balanced – of $\Delta\kappa_{expand}^{optim} = (\tau+1)(\log_2(n)+2)$.

3.3.2 Efficiency of TreeKEM's User Add. Contrary to the tree expansion process, it is not possible to generically evaluate the efficiency of TreeKEM's user add process since it strongly depends on the initial structure of the tree in which a new leaf must be added. This efficiency is therefore analyzed experimentally in the remainder of this paper.

Despite its suboptimal communication cost, the tree evolution mechanism adopted by TreeKEM presents the advantage that the choice of a new user's location is deterministic and needs only few computations. Therefore, the committer that adds the new user has no need to broadcast that location to the entire group. Instead, any member – that has a full view of the Ratchet Tree – locally updates its own copy of that tree.

3.4 Our Improved Tree Evolution Algorithms

We present here two algorithms designed to optimize the operations of tree expansion and user add in a full binary tree such as the one

used with TreeKEM, while keeping the simplicity and deterministic aspect of TreeKEM's algorithms. Both algorithms are generalizable to trees of any degree $m \geq 2$.

3.4.1 Optimal Tree Expansion. As we have previously seen, the best way to expand a perfect tree of initial height h consists in adding the additional leaves at a depth as close as possible to the one of the original leaves. Since the tree is perfect, the new leaves must be added at the bottom of the tree, one depth over the original leaves. This ensures that the resulting tree remains depth-balanced, and that its communication cost consequently stays optimized.

In formal tree representation, this method is carried out by moving all the 2^h current leaves down one level (to depth h+1), replacing them at depth h by internal nodes, and by attaching a blank sibling to each of them. The bottom layer of the Ratchet Tree, at depth $h+1$, thus becomes an alternation of 2^h filled and 2^h empty leaves (cf. Figure 4) that can be used afterwards to welcome new users.

This *bottom expansion* method involves modifying the leaf indices of the existing users, which are multiplied by a factor two: $\forall i \in [\![0, 2^h - 1]\!], \ell'_i = 2\ell_i$.

Even if, with TreeKEM, users keep the same leaf index as long as they belong to the group, the standard clearly specifies that the correlation between a user ID and a leaf index depends on the epoch[11]. Consequently, nothing prevents users from periodically changing their leaf index, as long as all group members are aware of these changes and update their local view of the Ratchet Tree accordingly. In the case of our bottom tree expansion, all group members receiving a commit that comprises a user add, whereas the Ratchet Tree is full, know that they must expand the latter by multiplying all users' indices by two.

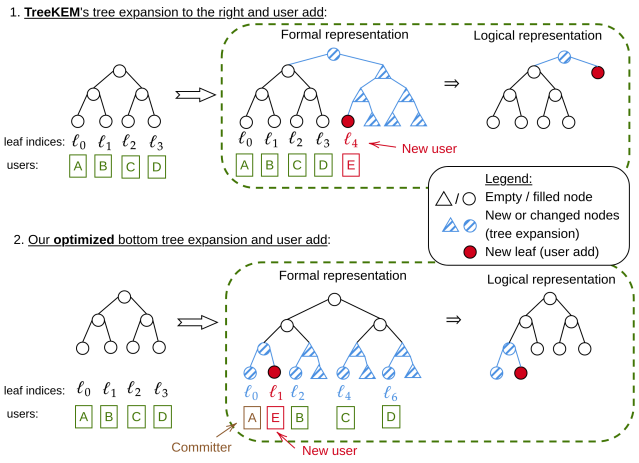


Figure 4: Ratchet Tree expansion with TreeKEM's right-expansion (top) and our optimized bottom method (bottom) that keeps the tree's depth-balance but modifies the leaf indices of the existing users.

The main drawback of our method comes from the *unmerged leaves* process, carried out by TreeKEM in order to maintain the CGKA's forward secrecy at the arrival of a new user. This process indeed blanks the whole direct path of the arriving user, so that none of these nodes contains secret elements related to previous epochs. Because all nodes above the new user are blank, the latter is logically attached straightly to the root. Later, as the newcomer's neighbors update their own path, that user is attached further down the tree.

Since TreeKEM's tree expansion to the right puts the new user in a separate new subtree that is already blank, no path blanking is performed on the original tree (that now constitutes the current left subtree). With our bottom tree expansion, nevertheless, we need to blank the direct path of the new user's location in the original tree, comprising h nodes, which affects in consequence the structure of half of the 2^h original leaves.

Fortunately, the effect of adding a single user can be totally offset if the protocol adds that newcomer (after expanding downward the Ratchet Tree) as the committer's sibling. Because the commit automatically updates the committer's direct path, which is identical to its sibling's, the nodes that have been blanked by the user add are filled again right after.

3.4.2 Almost-Optimal User Add. The optimal location where a new user should be added in a full binary tree is at depth $d_{min} + 1$, attached to the top leaf (or one of the top leaves) at original depth d_{min} in the tree.

The reason for it is simple: as the future sibling of the new leaf moves down a level, from d_{ℓ_s} to $d'_{\ell_s} = d_{\ell_s} + 1$, its parent cost is incremented by one: $\Delta v_p^{\ell_s} = 1$. Moreover, the new leaf ℓ_n induces an additional parent cost of $v_p^{\ell_n} = d_{\ell_s} + 1$. The tree's parent cost therefore varies as follows: $\Delta v_p = \Delta v_p^{\ell_s} + v_p^{\ell_n} = d_{\ell_s} + 2$ and

$$\boxed{\Delta\kappa_{userAdd} = \Delta v_p(\tau + 1) = (d_{\ell_s} + 2)(\tau + 1)} \quad (14)$$

Consequently, mitigating the overhead associated with a user add comes to minimizing the original depth d_{ℓ_s} of the leaf to which the new leaf will be attached. In other terms, the higher the new leaf is located in the tree, the better it is for the tree's communication cost.

Difficulty Finding the Top Leaves of a Ratchet Tree. A naive approach to determine the top leaf/leaves in the Ratchet Tree consists in recording, in each user's state, the depths of all leaves in the tree. This can be seen as an array of all the tree leaves, where each leaf index is associated with its depth. However, the computations needed to keep this array updated, after each user add and removal, and sorted by depth can become quickly costly. Indeed, adding or removing a leaf has an influence on the depth of all the leaves in the subtree rooted at that leaf's sibling. In a worst-case scenario, this number can reach 2^{h-1} in a perfect binary tree of height h.

Our Lightest Child Algorithm. Considering the difficulty to keep an up-to-date record of the leaf depths in a large tree, we have designed a simple yet effective algorithm that computes, when a user add is requested, a *close-to-optimal* insertion location for that new leaf. This algorithm is called the *Lightest Child Algorithm*.

The idea behind it is that the lower the weight of a tree (i.e. the fewer leaves it has), the smaller the average depth of its leaves is. Consequently, comparing sibling subtrees of the Ratchet Tree– i.e. subtrees rooted at the same depth – by considering their respective

[11] Indeed, due to the changes in the group membership, the protocol may assign the same leaf index to different users, at different time points.

weights determines which one of them have in average the highest leaves: this one is the lightest subtree. Consequently, it is likely that the highest leaf in the tree, that we are looking for, is located precisely in that lightest subtree.

We proceed recursively by selecting the lightest subtree and comparing the weights of that subtree's subtrees, and so on, until the subtree we are working on comprises two nodes (one of which is necessarily blank, unless the Ratchet Tree is totally filled and needs an expansion). In total, in a Ratchet Tree with 2^x leaves, this operation must be carried out $x-1$ times, logarithmically in relation to the number of group members.

For efficiency considerations, instead of recursively determining the lightest subtree of each lightest subtree in the Ratchet Tree, and proceeding this way at every user add[12], we determine at the beginning of the group evolution a Weighted Ratchet Tree. This tree, depicted in Figure 5, is a copy of the Ratchet Tree in its formal representation, in which each node is associated with its weight (cf. Section 2.1.2)[13].

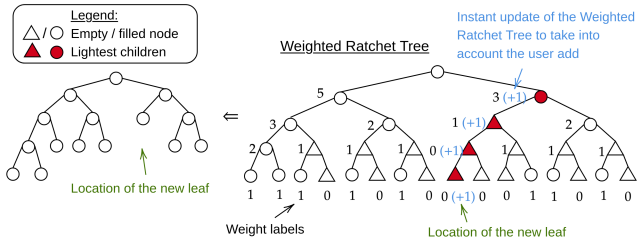


Figure 5: User add in a Ratchet Tree with our Lightest Child algorithm, relying on the associated Weighted Ratchet Tree. Determining a location for a new user requires reading only a logarithmic number of nodes in that tree.

Thanks to this Weighted Ratchet Tree, recursively comparing the lightest subtrees of the Ratchet Tree comes to selecting, from the root, the *lightest child* of each previously selected node, i.e. the node with the minimal associated weight[14]. With this method, building (once) the Weighted Ratchet Tree has a computational cost in $O(n)$ but afterwards, each use of that Weighted Ratchet Tree has a computational complexity of only $O(\log_2(n))$ (since it comes to reading the direct path of a leaf in a perfect binary tree).

The Weighted Ratchet Tree evolves with the group membership and must be updated at each user add or removal, by increasing or decreasing by one the weight of each node in the added/removed leaf. For efficiency considerations, the update of the Weighted Ratchet Tree during a user add is carried out at the same time as the determination of the lighest child (blue increment in Figure 5).

Figure 20 details the pseudocode for the Bottom Tree Expansion and Lightest Child algorithms (including the construction of a Weighted Ratchet Tree), extended to the generic case of a m-ary tree.

[12]Such basic approach implies to read all n users, then $\frac{n}{2}$ users of the lightest subtree, and so on during the $\log_2(n)$ stages of the process, which induces a computational cost in $O(n)$.

[13]This Weighted Ratchet Tree can also be seen as a slight modification of the Ratchet Tree, where a new *weight* field is added at the public state of the nodes.

[14]In case several children have an identical weight, the leftmost child is selected.

Since the optimal location for a new leaf strongly depends, once again, on the tree structure at the time of that user add, the efficiency of our algorithm can only be assessed through experimental results. These ones, detailed below, show that the Lightest Child algorithm, paired with our Bottom Tree Expansion, gives the tree a communication cost very close to the optimal one, with an average overhead around 0.5 %.

3.5 Experimental Results

In order to assess in practice the efficiency of our optimizations, we have modeled in a Python program the random evolution of a full binary Ratchet Tree and we have compared the communication costs of such tree with three different tree evolution mechanisms: TreeKEM's, our optimized one and what we call the *wild evolution*.

More specifically, the Ratchet Tree is initialized as a tree whose number of leaves is the minimum power of the tree degree that contains the desired number of users: $n_\ell = m^{\lceil \log_m(n_{users}) \rceil}$. The arrangement of the n_{users} filled leaves and $n_\ell - n_{users}$ blank leaves is then randomized. The evolution pattern of that initial tree is modeled by a *random walk*: at each iteration, the tree undergoes, with an equal probability, either a user add or a user removal. As in real life, the removed user is selected randomly among all (filled) leaves. Conversely, the location where the new user is added to the tree and the way the tree is expanded, if needed, are determined by the aforementioned three methods:

- With TreeKEM's tree evolution mechanism, as specified in Section 3.3, users are added on the leftmost possible location in the tree and the Ratchet Tree is expanded to the right.
- Our optimized method uses both our Lightest Child user add algorithm and our Bottom Tree Expansion.
- The wild evolution randomly inserts new users among blank leaves and expands the Ratchet Tree to the right, similarly to TreeKEM's tree expansion.

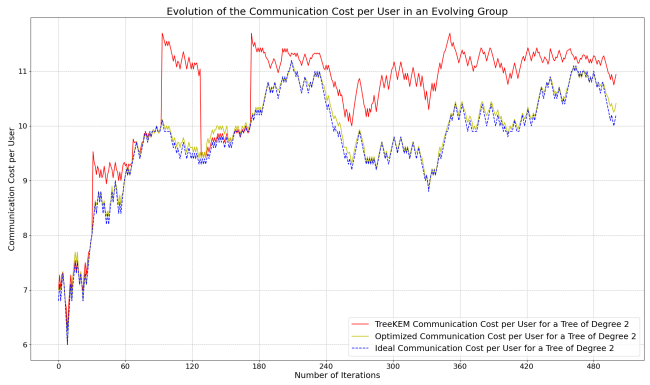


Figure 6: Instance of the compared communication costs of a randomly evolving Ratchet Tree, with TreeKEM's tree evolution mechanism and our optimized algorithms.

Our experimental results displayed in Figure 6 show that TreeKEM actually behaves quite honorably, with an average overhead, w.r.t.

the optimal cost for full binary trees, of 5.6%[15]. For comparison, the *wild* evolution has, in the same conditions, an average overhead of 1.9% whereas our optimized method reaches an average overhead of only 0.3% w.r.t. that optimal cost.

We nevertheless underline that when the initial tree is already unbalanced, the efficiency of TreeKEM is impacted – the protocol has, in that case, a communication overhead of 10% compared to the optimal cost – whereas the efficiency of our optimized protocol remains unchanged, with an overhead of less than 1%.

4 TREE DEGREE SELECTION

4.1 General Considerations

The degree of a Ratchet Tree has an obvious influence on the communication cost of a commit. Indeed, given a number n of users, the higher the tree degree, the shorter and wider this tree is. Consequently, the parent cost v_p decreases and the sibling cost v_c increases along with the tree degree.

4.1.1 Tau and the Tree Degree. The optimal degree for a Ratchet Tree mainly depends on the value τ related to the ciphersuite used by the CGKA. Indeed, this parameter represents the ratio of public key size over the ciphertext size associated with an encryption scheme. The higher τ is, the more interesting it becomes to increase the tree degree so that the number of broadcasted public keys decreases, even at the cost of a higher number of ciphertexts.

Table 1 in Appendix C details the values of τ for the encryption schemes that are expected to be used with MLS, both in the classical and in the post-quantum frameworks. It emerges from this table that all classical ciphersuites are associated with a ratio $\tau \in [0.4, 0.6]$, whereas most promising PQ ciphersuites – including the newly strandardized ML-KEM – have a higher ratio $\tau \in [0.9, 1.0]$. The next parts are devoted to determining, with theoretical bounds tightened by experimental simulations, the optimal tree degree in our parameter range $\boxed{\tau \in [0.4, 1.0]}$.

4.1.2 Determination of the Optimal Degree in the General Case. Based on the communication cost $\kappa_T(\tau)$ of a tree, we define the cost function of a set $\mathcal{S}$ of trees as a function[16] $\kappa_{\mathcal{S}}$ of the parameter τ:

$$\kappa_{\mathcal{S}}(\tau) = \inf \{\kappa_T(\tau),\ T \in \mathcal{S}\} \tag{15}$$

For a set $\mathcal{S}$ of trees bearing n leaves, let us define:

- the points $v(T) = (v_p(T), v_c(T)) \in \mathbb{R}^2$ and
- the set of points $v(\mathcal{S}) = \{v(T),\ T \in \mathcal{S}\}$.

As a tree communication cost $\kappa_T(\tau) = v_p\tau + v_c$ is an affine function of τ, minimizing this cost over the set $\mathcal{S}$ (i.e. minimizing the cost function $\kappa_{\mathcal{S}}$) comes to determining the *lower-left* part of the convex hull H of the set $v(\mathcal{S})$.

To do so, we define the set $v(\mathcal{S})^+$ as the Minkowski sum [17] of $v(\mathcal{S})$ and the upper-right quadrant of $\mathbb{R}^2$. The convex hull H' of $v(\mathcal{S})^+$ corresponds precisely to the lower-left quadrant of the convex hull of $v(\mathcal{S})$, that we are interested in for our cost optimization.

[15]The main factors influencing the efficiency are the initial number of users for the random walk and the method used to generate the initial trees. Our average results are computed over a range of initial numbers of users from 10 to 120, with 100 repetitions for each initial number of users.

[16]The function $\kappa_{\mathcal{S}}$ may also be seen as the Fenchel transform [12, 3.3] of the set $\mathcal{S}$ of trees.

Figure 7 depicts the aforementioned convex hull H' (shaded area). In that graph, any affine line with a negative slope represents all trees of n leaves, with identical communication cost κ_T and parameter τ (the slope of that line being $-\tau$). When considering several parallel lines (thus with an identical τ), the leftmost line has a lower communication cost that the other ones; this is the reason why we focus on the lower-left part of the dual convex hull of a set of trees, that groups together the optimal trees of that set for different values of τ. More precisely, if a vertex $v(T_i)$ of H' lies at the intersection of the segments with slopes $-\tau_i$ and $-\tau_{i+1}$, then for any $\tau \in [\tau_i, \tau_{i+1}]$ and any $T' \in \mathcal{S}$, $\kappa_{T'}(\tau) \geq \kappa_{T_i}(\tau)$. In other terms, the tree T_i related to the vertex $v(T_i)$ is optimal – among its set $\mathcal{S}$ – for any $\tau \in [\tau_i, \tau_{i+1}]$.

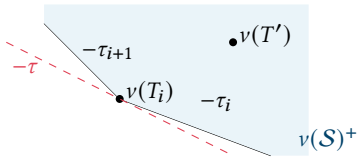


Figure 7: Optimal trees in a set $\mathcal{S}$, viewed as vertices of the lower-left part of the convex hull of $v(\mathcal{S})^+$.

Complexity Analysis. The complexity of computing the function $\kappa_{\mathcal{S}}$ over a finite set $\mathcal{S}$ of N trees is thus that of computing the convex hull of a planar set of N points, which is $O(N \log(N))$ [19].

On the other hand, the cardinality of the set of all trees with exactly n leaves is given by the OEIS sequence A000669 [32][18, I.45]. This cardinality is $\Omega(\rho^n)$, where $\rho \geq 3.5$ [18, VII.5]. As a consequence, it appears impossible in practice to enumerate and study all such trees beyond small values of n. We provide an enumeration of all optimal trees up to $n = 25$ in attached artifact best_trees.toml.

While an extensive study is out of computational reach, we can focus on specific families of trees that seem to be of interest in practical ranges of τ intuited by our exhaustive study of small trees stated above. As a first step, we provide below an analysis on full trees.

4.1.3 Optimal Degree of a Full Tree. As we have previously proved that a m-ary tree minimizes its communication cost when it is depth-balanced, determining the optimal degree of full trees comes to analyzing the set of depth-balanced full trees of various degrees. When the number of leaves in that trees is fixed, that comparison is easily done with the above convex hull method[17]. This basic study leads to Figure 8, which shows that:

- The exact number of leaves of the tree has a significant influence on the optimal degree: namely, perfect trees (depth-balanced trees whose number of leaves is a power of m) are optimal for large intervals of the parameter τ. This factor tends to be influential on small trees but becomes asymptotically negligeable.
- The optimal degree of a full tree increases along with the parameter τ.

However, when its degree exceeds two, a CGKA's Ratchet Tree never stays full. Indeed, a m-ary tree can be full only when the

[17]We can also use Equation (6) in Section 3.2.2, that gives the communication cost of a depth-balanced full tree of degree m, to compare the area of optimality of trees with consecutive degrees.

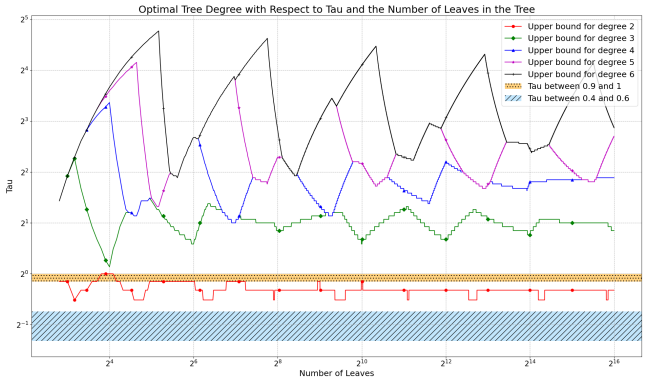


Figure 8: Areas of optimal tree degree for a *full* tree, according to the values of the encryption scheme's parameter τ and the number of users in the tree (in a logarithmic scale).

number of leaves is $n_\ell = m + k(m-1)$, $k \in \mathbb{N}$ and when these leaves are equally distributed within that tree. As we do not control when and where users leave the group (i.e. when and where leaves are removed from the tree), our analysis on the optimal tree degree must extend to the case of the numerous non-full trees.

4.2 Degree Bounds for Practical Values of τ

4.2.1 Collapsed Trees. In order to study the inner structure of a tree while abstracting away the effect of its leaves, we want to make use of weighted trees. In this intent, we extend our cost functions v_p and v_c so that, if T is a single leaf of weight w, then $v_p(T) = v_c(T) = w$. For larger trees, the recursive definitions from Definition 3.1 are unchanged.

Let T' be the weighted tree obtained by collapsing an inner branch (subtree) B of a tree T to a single leaf of weight $w(B)$. Then we see from the recursive definition of v_p that $v_p(T') = v_p(T)$. The same results holds for v_c, and also after multiple successive collapses.

This allows replacing the study of a tree T, with an arbitrarily large number of leaves n, by that of a collapsed tree T' with a very small number of leaves, each one of these representing a whole branch of the original tree. In particular, re-arranging the leaves of the collapsed tree T' corresponds to re-arranging the branches of the original tree T, and the impact of such a transformation on the costs v_p and v_c is identical for both trees.

4.2.2 Trees with Three Leaves. We now describe the configuration space for the optimal trees with two or three branches at their root, depending on the parameter τ as well as on the relative sizes of the branches. For this we start by studying the weighted trees with three leaves and – without loss of generality – a normalized total weight of one. There are only two such trees: a binary T_2 and a ternary T_3 (cf. Figure 9) defined as: $T_2(\ell_1, \ell_2, \ell_3) = \{\ell_1, \{\ell_2, \ell_3\}\}$ and $T_3(\ell_1, \ell_2, \ell_3) = \{\ell_1, \ell_2, \ell_3\}$, where nodes between brackets are direct siblings and $\widetilde{w}_1 + \widetilde{w}_2 + \widetilde{w}_3 = 1$ are the respective normalized weights of ℓ_1, ℓ_2 and ℓ_3. We have:

$$v_p(T_2) = v_c(T_2) = \widetilde{w}_1 + 2(\widetilde{w}_2 + \widetilde{w}_3) \tag{16}$$
$$v_p(T_3) = \widetilde{w}_1 + \widetilde{w}_2 + \widetilde{w}_3 \tag{17}$$
$$v_c(T_3) = 2(\widetilde{w}_1 + \widetilde{w}_2 + \widetilde{w}_3) \tag{18}$$

This permits to determine for which value of τ these trees are optimal:

$$\kappa_{T_3}(\tau) - \kappa_{T_2}(\tau) = \widetilde{w}_1 - \tau(1 - \widetilde{w}_1)$$
$$\boxed{\kappa_{T_3}(\tau) \leq \kappa_{T_2}(\tau) \Leftrightarrow \tau \geq \frac{\widetilde{w}_1}{1 - \widetilde{w}_1}} \tag{19}$$

On the other hand, both trees T_3 and T_2 have symmetries. Namely, using the symmetry of the tree T_3, one may assume that $\widetilde{w}_1$ is the largest of the three weights, and in particular that $\widetilde{w}_1 \geq \frac{1}{3}$. In the same way, for any tree with two branches and at least four leaves – represented by T_2 –, we can select which branch is collapsed into the weighted leaf ℓ_1 and which branches into the leaves $\{\ell_2, \ell_3\}$. Since the cost of such a tree increases with $\widetilde{w}_1$, the optimal tree is reached when the ℓ_1 branch is the lightest branch from the root of the tree, which implies that $\widetilde{w}_1 \leq \frac{1}{2}$. Putting all these results together, we can represent the set of optimal weighted trees with three leaves as in Figure 10a.

In particular, we deduce the following:

(1) For $\tau \leq \frac{1}{2}$, the optimal trees are binary;
(2) for $\tau \geq 1$ and $n \geq 3$, the optimal trees with n leaves are *at least* ternary.

For $\tau \in [\frac{1}{2}, 1]$, the optimal tree configurations actually have a mixture of binary and ternary nodes; as it will be further studied in Section 4.3.

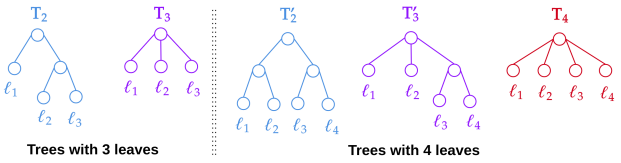


Figure 9: Small trees with three or four branches.

4.2.3 Trees with Four Leaves. Of the five trees with four leaves, the only one which cannot be decomposed using the previously defined families T_2 and T_3, is the flat tree $T_4(\ell_1, \ell_2, \ell_3, \ell_4) = \{\ell_1, \ell_2, \ell_3, \ell_4\}$.

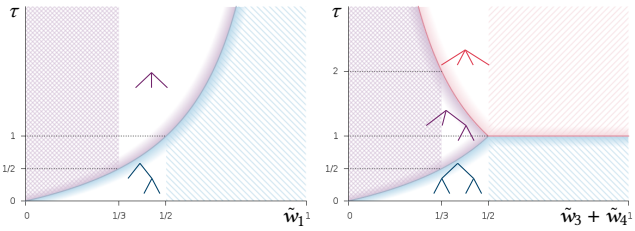
We can compare this tree with $T'_3(\ell_1, \ell_2, \ell_3, \ell_4) = \{\ell_1, \ell_2, \{\ell_3, \ell_4\}\}$ and with $T'_2(\ell_1, \ell_2, \ell_3, \ell_4) = \{\{\ell_1, \ell_2\}, \{\ell_3, \ell_4\}\}$ (cf. Figure 9). Using the same methods as previously, we find that:

(1) $\kappa_{T_4}(\tau) \leq \kappa_{T'_3}(\tau)$ for $\tau \geq \frac{\widetilde{w}_1 + \widetilde{w}_2}{\widetilde{w}_3 + \widetilde{w}_4}$;
(2) $\kappa_{T_4}(\tau) \leq \kappa_{T'_2}(\tau)$ for $\tau \geq 1$.

Moreover, by using the symmetries for T_4 or T'_2, one may always ensure that $\widetilde{w}_3 + \widetilde{w}_4 \geq \frac{1}{2}$ in both cases. Furthermore, let T be a large enough tree with a ternary root. We can always collapse that tree in a way such that the leaves ℓ_3 and ℓ_4 are the two heaviest ones; this ensures that $\widetilde{w}_3 + \widetilde{w}_4 \geq \frac{1}{3}$. We thus obtain the configuration space shown in Figure 10b and the following conclusions:

(1) For $\tau \leq 1$, the optimal trees with n leaves are *at most* ternary.

(2) for $\tau \geq 2$ and $n \geq 4$, the optimal trees with n leaves are *at least* quaternary.



(a) Trees with three leaves **(b) Trees with four leaves**

Figure 10: Optimal tree configurations with a small number of leaves. The crossed-out areas mark the configurations which can be eliminated by symmetries.

The previous parts have bounded the area of optimiality, w.r.t. τ, of binary and ternary trees. When considering an arbitrary tree degree m, determining a generic lower bound for this degree area of opimality, seems to be a hard topic. The following part however gives an upper bound for such a degree.

4.2.4 Upper Bound of an Arbitrary Tree Degree. When one wants to implement a tree in pratice, the dimensioning parameter corresponds to the tree degree. Upper bounding such a degree is thus mandatory and we want to do it while keeping opportunities for optimal communication cost. We give below an upper bound for the degree m_{optim} of an optimal tree, with respect to τ.

For any integer b and weights $\widetilde{w}_1 + \cdots + \widetilde{w}_b = 1$, let us compare the two small trees:

$$T_b = \{\ell_1, \ldots, \ell_b\} \quad \text{and } T'_{b-1} = \{\{\ell_1, \ell_2\}, \ell_3, \ldots, \ell_b\}. \tag{20}$$

T_b and T'_{b-1} have the same number of leaves but not the same degree. Let $x = \widetilde{w}_1 + \widetilde{w}_2$ be the joint weight of ℓ_1 and ℓ_2. We have:

$$v_p(T'_{b-1}) = v_p(T_b) + x \quad \text{and } v_c(T'_{b-1}) = v_c(T_b) - (1-x) \tag{21}$$

$$\kappa_{T'_{b-1}}(\tau) < \kappa_{T_b}(\tau) \quad \Leftrightarrow \quad x(\tau+1) < 1 \tag{22}$$

Using the symmetry between the branches, we may always choose ℓ_1, ℓ_2 as the two heaviest leaves, so that $x \geq \frac{2}{b}$. Therefore:

$$\kappa_{T'_{b-1}}(\tau) < \kappa_{T_b}(\tau) \quad \Leftrightarrow b > 2(\tau+1) \tag{23}$$

This means that it remains interesting to decrease the degree of any node from a tree, from b to $b-1$, as long as $b > 2(\tau+1)$. Consequently, the optimal tree degree m_{optim} is upper-bounded as follows:

$$\boxed{m_{optim} \leq 2(\tau+1)} \tag{24}$$

4.3 Experimental Comparison of non-Full Trees

The study above shows that when $\tau \in [0.5, 1]$, the optimal tree degree is two, three or even four (for the upper bound $\tau = 1$). However, it does not permit to theoretically select one of these degrees because the optimal value not only depend on τ and on the number of leaves in the tree, but also on that tree's structure, i.e. on the relative disposition of the nodes, that cannot be considered in a general case.

Consequently, we have carried out simulations using the same program as in Section 3.5, during which we have compared the average communication cost per user[18], both with TreeKEM's methodology and with our optimized method, of:

- ternary and quaternary trees, for $\tau \geq 1$;
- binary and ternary trees, for $\tau \in [0.5, 1]$.

4.3.1 Ternary vs Quaternary Trees. The simulation shows that when $\tau = 1$, the communication cost per user of a 4-ary tree is in average 4.4% higher than its ternary counterpart. In fact the ternary tree remains more efficient than the quaternary one as long as $\tau \leq 1.72$, which corresponds to the use case of most classical or PQ HPKE ciphersuites (more details are ploted in Figure 17).

4.3.2 Binary vs Ternary Trees. The results of the experimental comparison of binary and ternary trees are depicted in Figure 11. This figure shows a limit value of tau, $\tau^{2-3} \in [0.71, 0.78]$, under which a binary tree has a better communication cost than a non-full ternary one, and over which a non-full ternary tree is better suited[19]. This limit appears quite similar to the one between *full* binary and ternary trees (cf. Figure 18), which also varies in the interval $[0.7, 0.8]$, with an asymptotic limit (regarding the number of leaves in the tree) of 0.73.

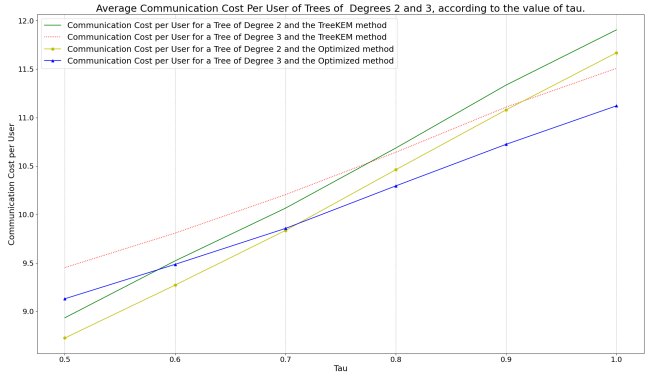


Figure 11: Compared communication costs per user of binary and ternary trees evolving with TreeKEM's and our optimized methods. It shows an optimality bound between these two degrees at $\tau \in [0.7, 0.8]$.

In addition, the experimental comparison reveals that this limit value of tau depends on the tree evolution mechanism that is being used: this limit is indeed higher with TreeKEM's tree evolution mechanism (in which case $\tau_{tk}^{2-3} \approx 0.78$) than with our optimized method (with which $\tau_{optim}^{2-3} \approx 0.71$). This difference may be due to the fact that the structure of full binary trees is easier to balance, even with a non-optimized method such as TreeKEM's, than the one of a non-full ternary tree. Consequently, at the limit between the areas of optimality of that two degrees, a binary tree evolving

[18] These average results are computed over 120 simulations evolving with random walks whose initial numbers of users vary from 10 to 120.

[19] As non-linear trees are not considered here, binary trees are always full.

with TreeKEM's method takes precedence on the ternary tree because it tends to be naturally better balanced.

Figure 12 sums up the tree degree adapted to various classical and PQ ciphersuites used for HPKE encryption. It underlines that in the post-quantum framework, and especially with the standardization of ML-KEM, ternary trees should be adopted as ratchet trees for MLS's CGKA.

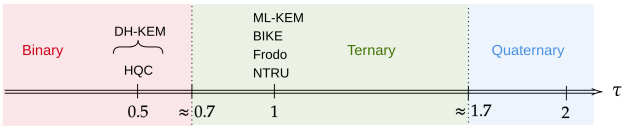


Figure 12: Optimal degree of a CGKA's Ratchet Tree, according to the HPKE ciphersuite used.

REFERENCES

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

[2] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. 2022. *HQC.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[3] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. 2022. *Classic McEliece.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions.

[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] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. 2022. *BIKE.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[6] David M. Balenson, David McGrew, and Dr. Alan T. Sherman. 1999. *Key Management for Large Dynamic Groups: One-Way Function Trees and Amortized Initialization.* Internet-Draft draft-balenson-groupkeymgmt-oft-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-balenson-groupkeymgmt-oft/00/ Work in Progress.

[7] 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

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

[9] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2018. *The Messaging Layer Security (MLS) Protocol.* Internet-Draft draft-ietf-mls-protocol-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/00/ Work in Progress.

[10] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2018. *The Messaging Layer Security (MLS) Protocol.* Internet-Draft draft-ietf-mls-protocol-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/02/ Work in Progress.

[11] 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

[12] Jonathan M Borwein, Adrian S Lewis, et al. 2006. *Convex Analysis and Nonlinear Optimization.* Canadian Mathematical Society/Société mathématique du Canada.

[13] 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.

[14] Timo Brecher, Emmanuel Bresson, and Mark Manulis. 2009. Fully Robust Tree-Diffie-Hellman Group Key Exchange. In *CANS 09 (LNCS, Vol. 5888)*, Juan A. Garay, Atsuko Miyaji, and Akira Otsuka (Eds.). Springer, Heidelberg, 478–497.

[15] 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

[16] Yvo Desmedt, Tanja Lange, and Mike Burmester. 2007. Scalable Authenticated Tree Based Group Key Exchange for Ad-Hoc Groups. In *FC 2007 (LNCS, Vol. 4886)*, Sven Dietrich and Rachna Dhamija (Eds.). Springer, Heidelberg, 104–118.

[17] Günter Ewald. 1996. *Combinatorial Convexity and Algebraic Geometry.* Springer Graduate Texts in Mathematics 168.

[18] Philippe Flajolet and Robert Sedgewick. 2009. *Analytic combinatorics.* cambridge University press.

[19] Ronald L. Graham. 1972. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Lett.* 1 (1972), 132–133.

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

[21] Antoine Joux. 2004. A One Round Protocol for Tripartite Diffie-Hellman. *Journal of Cryptology* 17, 4 (Sept. 2004), 263–276. https://doi.org/10.1007/s00145-004-0312-y

[22] Ho-Hee Kim and Soon-Ja Kim. 2012. A Ternary Tree-based Authenticated Group Key Agreement For Dynamic Peer Group. https://api.semanticscholar.org/CorpusID:60089781

[23] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2000. Simple and Fault-Tolerant Key Agreement For Dynamic Collaborative Groups. In *ACM CCS 2000*, Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati (Eds.). ACM Press, 235–244. https://doi.org/10.1145/352600.352638

[24] Yongdae Kim, Adrian Perrig, and Gene Tsudik. 2004. Tree-based group key agreement. *ACM Trans. Inf. Syst. Secur.* 7, 1 (feb 2004), 60–96. https://doi.org/10.1145/984334.984337

[25] Abhimanyu Kumar and Sachin Tripathi. 2013. Ternary Tree based Group Key Agreement Protocol Over Elliptic Curve for Dynamic Group. *International Journal of Computer Applications* 86 (12 2013). https://doi.org/10.5120/14997-3072

[26] Sangwon Lee, Yongdae Kim, Kwangjo Kim, and Dae-Hyun Ryu. 2003. An Efficient Tree-Based Group Key Agreement Using Bilinear Map. In *ACNS 03 (LNCS, Vol. 2846)*, Jianying Zhou, Moti Yung, and Yongfei Han (Eds.). Springer, Heidelberg, 357–371. https://doi.org/10.1007/978-3-540-45203-4_28

[27] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. 2020. *FrodoKEM.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[28] National Institute of Standards and Technology. 2023. Module-Lattice-Based Key-Encapsulation Mechanism Standard. FIPS 203 ipd. https://doi.org/10.6028/NIST.FIPS.203.ipd

[29] Adrian Perrig. 1999. Efficient Collaborative Key Management Protocols for Secure Autonomous Group Communication. In *Proceedings of the International Workshop on Cryptographic Techniques and E-Commerce (CrypTEC).* 192–202. /publications/papers/secmcast.pdf

[30] Adrian Perrig, Dawn Xiaodong Song, and J. D. Tygar. 2001. ELK, A New Protocol for Efficient Large-Group Key Distribution. In *2001 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, 247–262. https://doi.org/10.1109/SECPRI.2001.924302

[31] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, Damien Stehlé, and Jintai Ding. 2022. *CRYSTALS-KYBER.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[32] N. J. A. Sloane and J. Riordan. [n. d.]. Sequence A000669 in the On-Line Encyclopedia of Integer Sequences (n.d.). https://oeis.org/A000669. Accessed 2024-04-26.

[33] Michael Steiner, Gene Tsudik, and Michael Waidner. 1996. Diffie-Hellman Key Distribution Extended to Group Communication. In *ACM CCS 96*, Li Gong and Jacques Stern (Eds.). ACM Press, 31–37. https://doi.org/10.1145/238168.238182

[34] Sachin Tripathi and G. P. Biswas. 2009. Design of efficient ternary-tree based group key agreement protocol for dynamic groups *(COMSNETS'09).* IEEE Press, 30–35.

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

[36] Zhenfei Zhang, Cong Chen, Jeffrey Hoffstein, William Whyte, John M. Schanck, Andreas Hulsing, Joost Rijneveld, Peter Schwabe, and Oussama Danba. 2019. *NTRUEncrypt.* Technical Report. National Institute of Standards and Technology. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions.

[37] Shanyu Zheng, David Manz, and Jim Alves-Foss. 2007. A communication–computation efficient group key algorithm for large and dynamic groups. *Computer Networks* 51, 1 (2007), 69–93. https://doi.org/10.1016/j.comnet.2006.03.008

A OMITTED PRELIMINARIES

A.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. Without any base explicitly indicated, $\log()$ denotes the logarithm in base 2.

A.2 TreeKEM CGKA protocol

We give hereunder a brief description of how TreeKEM – as standardized in RFC 9420 [7] – works as a Continuous Group Key Agreement (CGKA) protocol.

A.2.1 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, periodically and following changes in the group membership.

Definition A.1 (Propose & Commit CGKA (adapted from [4])). 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 $\mathbb{A} \supseteq \{\text{Add, Remove, Update}\}$ 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 $\mathbb{P} = \{P_i \in \mathbb{A}\}_{i\in[\![1,p]\!]}$, 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:
  $\boxed{(\gamma_i', k) := \textbf{process}(\gamma_i, m \in \{C, W\})}$

Nota: The state γ_i of any user u_i is composed of a public part ${}^p\gamma_i$, known by every group member (which includes a complete view of the Ratchet Tree, cf. Section 2.2.1), and a private part ${}^s\gamma_i$, that this user keeps secret and that is needed to recover the group keys generated by other members.

A CGKA must fulfill the following properties, stated informally below. These properties are captured by the CGKA security game (in a game-based security model) such as the one described in figure 1 of [4][20].

- **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**: the corruption of any user at some epoch does not leak any secret element (neither the group key nor the secret seeds and keys) from previous epochs.
- **Post-Compromise Security**: following the corruption of any user, the tree's secret elements become secret again after the update of all the corrupted users (provided that the adversary stays passive during these 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.

A.2.2 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 A.1, 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 13, 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.

[20] This security model is not recalled in this paper, as it is outside the scope of our study.

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}.

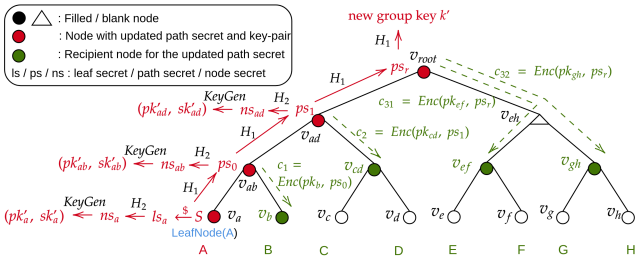


Figure 13: Update, with TreeKEM, of a user's filtered direct path (here user A). This process updates the encryption key-pairs of that user and of all its ancestors; it also generates a new group key.

A.2.3 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.

B OMITTED PROOFS

B.1 Proof of Theorem 3.4

Step 1. We consider a random full m-ary tree T that is not depth-balanced, i.e. whose highest leaves have a depth difference with the bottom leaves greater than one: $d_{min}(T) < d_{max}(T) - 1$. As this tree is full, each node has either 0 or m children; in particular, leaves are all grouped in clusters of m siblings.

We define a one-step bottom-up balancing process which moves $m-1$ of a group of leaves belonging to the bottom of the tree (with an initial depth $d_i = d_{max} = h_T$) higher in the tree, into a destination depth $d_f < d_{max}$. As a single child is not allowed in a full tree, the remaining leaf ℓ_s from that altered leaf cluster moves up a level and replaces its parent node.

The $m-1$ moving leaves $(\ell_i)_{i\in[\![1,m-1]\!]}$ become attached to another leaf ℓ'_s initially located in a leaf cluster at depth $d_f - 1$. That leaf moves down a level into depth d_f, is grouped with the $m-1$ moving leaves in order to form a new cluster, and a parent node is created at the former location of ℓ'_s.

That process consequently modifies the depths of $m+1$ leaves as follows:

- for the $m-1$ moving leaves: from d_i to d_f;
- for the initial sibling leaf ℓ_s: from d_i to $d_i - 1$;
- for the final sibling leaf ℓ'_s: from $d_f - 1$ to d_f.

As the depth of a node is, by definition, the length of this node's direct path up to the tree root, the depth of a leaf corresponds to its parent cost: $\forall \ell \in T, d_\ell = v_p^\ell$. Consequently, we have:

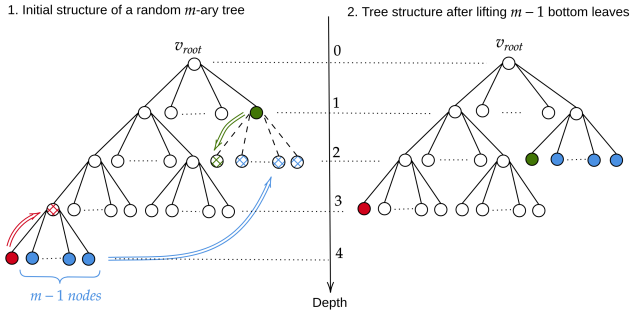


Figure 14: One-Step Bottom-Up Depth-Balancing process applied on a random, non-depth-balanced full m-ary tree. This process, which moves upwards a group a $m-1$ leaves from the bottom of the tree, decreases that tree's communication cost. Leaves depicted with a crosshatched pattern represent the future locations of the moving leaves.

- for the $m-1$ moving leaves $(\ell_i)_{i\in[\![1,m-1]\!]}$: $\Delta v_p^{\ell_i} = d_f - d_i$;
- for the initial sibling leaf ℓ_s: $\Delta v_p^{\ell_s} = (d_i - 1) - d_i = -1$;
- for the final sibling leaf ℓ'_s: $\Delta v_p^{\ell'_s} = d_f - (d_f - 1) = 1$.

When considering all the leaves in the tree, the parent cost of that tree is modified as follows:

$$\begin{aligned} &\Delta v_p = (m-1)(d_f - d_i) \\ \Rightarrow\ &\boxed{\Delta\kappa = (\tau + m - 1)(m-1)(d_f - d_i)} \end{aligned} \tag{25}$$

As $d_f < d_i$, $\Delta\kappa < 0$, which corresponds to an enhancement of the tree's communication cost regardless of its initial structure (as long as it was depth-unbalanced, so that it is possible to move upwards the bottom leaves).

Step 2. Let us now consider a random depth-balanced full m-ary tree. By definition, we have $d_{min}(T) \geq d_{max}(T) - 1$. As only empty or m-large leaf clusters are allowed in a full tree, any change on the tree structure necessarily implies a simultaneous move of $m-1$ leaves, similarly as in the aforementioned balancing process.

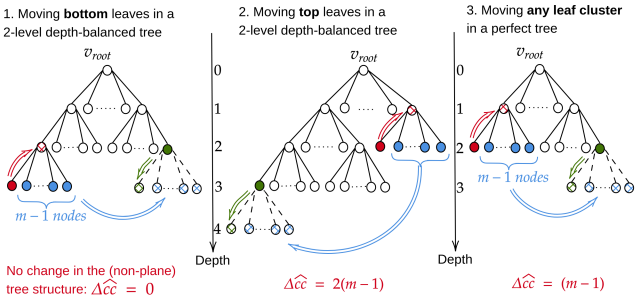


Figure 15: Moving leaves in a depth-balanced tree. These changes either leave the tree structure unaltered (left) or increase the tree's communication cost (middle and right). Leaves depicted with a crosshatched pattern represent the future locations of the moving leaves.

A depth-balanced tree has at most two levels of leaves, which implies to consider three cases (cf. Figure 15):

- If the depth-balanced tree has exactly two levels of leaves (case of a non-perfect tree: $d_{min} = d_{max} - 1$) and $m - 1$ bottom leaves are moved upwards, the only possible destination depth for them is : $d_f = d_{min} + 1 = d_{max} = d_i$. The moving leaves stay at the same depth of the tree. Because a Ratchet Tree is non-planar, such a horizontal migration is not considered a change in the tree structure.
- If, in a 2-level depth-balanced tree, $m - 1$ top leaves are moved, their only possible destination depth is: $d_f = d_{max} + 1 = d_{min} + 2 = d_i + 2$. We can deduce from step 1 that the tree's communication cost is increased as follows: $\Delta\kappa = (m-1)(d_f - d_i) = 2(m-1) > 0$.
- Similarly, in a perfect tree (where all leaves are at the same depth $d_{min} = d_{max}$), moving any group of $m - 1$ leaves induces a communication increase of: $\Delta\kappa = (m-1)(d_f - d_i) = m - 1 > 0$.

We have shown above that *any* full m-ary tree can have its communication cost decreased as long as it is not depth-balanced. When this is the case, no further enhancement can be made. This proves that a depth-balanced structure yields an optimal communication cost. The converse is proven in the following step.

Step 3. Conversely, it is straightforward to show, with a demonstration by contradiction, that the depth-balance property is the only way to provide the optimal communication cost. To do so, we assume a depth-unbalanced tree structure with an optimal communication cost. According to the step 1 of that proof, the one-step bottom-up balancing process can be carried out on the tree, resulting on a decrease in the tree's communication cost. This contradicts the initial assumption according to which that depth-unbalanced tree has an optimal cost, and concludes that an optimal cost is only associated with a depth-balanced tree structure.

C TAU VALUE OF HPKE CIPHERSUITES

Let us consider the values of τ corresponding to the encryption schemes that are expected to be used with MLS. The standard for this protocol [7] specifies that the encryption is performed according to the HPKE paradigm (cf. [8]), using a Key Encapsulation Mechanism (KEM) to exchange a symmetric key and an Authenticated Encryption with Associated Data (AEAD) scheme to symmetrically encrypt the plaintext. The AEAD schemes recommended by that standard are AES-GCM and Chacha20-Poly1305, that both yield a 16-byte-long authentication tag.

Regarding the KEMs:

- In the classical – i.e. pre-quantum – framework, MLS standard recommends KEMs based on Diffie-Hellman on elliptic curves (DHKEMs), such as DHKEM-X25519, DHKEM-X448, DHKEM-P256 or DHKEM-P521. In a DHKEM, one of the public elements exchanged is modeled as the KEM's public key, whereas the other one is considered as its ciphertext. The Key Derivation Functions (KDFs) used within that KEMs are HKDF with hash functions of various fingerprint sizes. Let us note that these KDFs determine the size of the path secret to be encrypted by MLS (cf. [7]), and therefore have some influence on the value of τ.

Table 1: Tau ratio and optimal tree degree for the main classical and post-quantum HPKE ciphersuites. This table underlines the strong difference between the classical framework, where τ varies between 0.4 and 0.6 and the ciphersuites are adapted to binary trees, and the post-quantum one, where most schemes have τ around 0.9-1.0, which fits ternary trees.

KEM Type		$\lvert ps \rvert$ (bytes)	$\lvert pk^{hpke} \rvert$ (bytes)	$\lvert ct^{hpke} \rvert$ (bytes)	τ	Optim. Tree
Classical HPKE Ciphersuites						
DH-KEM [8]	X25519	32	32	80	0.4	Bin
	P256	32	65	113	0.6	
	P384	48	97	161	0.6	
	X448	64	56	136	0.4	
	P521	64	133	213	0.6	
Post-Quantum HPKE Ciphersuites						
HQC [2]	128	32	2,249	4,561	0.5	Bin
	192	32	4,522	9,106	0.5	
	256	32	7,245	14,549	0.5	
ML-KEM [31]	512	32	800	848	1.0	Tern
	768	32	1,184	1,168	1.0	
	1024	32	1,568	1,648	1.0	
BIKE [5]	Level 1	32	1,541	1,653	1.0	Tern
	Level 3	32	3,083	3,195	1.0	
	Level 5	32	5,122	5,234	1.0	
Frodo [27]	640	32	9,616	9,800	1.0	Tern
	976	32	15,632	51,840	1.0	
	1344	32	21,520	21,744	1.0	
NTRU [36]	hps2048509	32	699	747	0.9	Tern
	hps2048677	32	931	979	1.0	
	hps4096821	32	1,230	1,278	1.0	
	hrss701	32	1,138	1,186	1.0	
Classic McEliece [3]	348864	32	261,120	176	1,813	> Tern
	460896	32	524,160	236	2,569	
	6688128	32	1,044,992	288	4,082	
	6960119	32	1,047,319	274	4,328	
	8192128	32	1,357,824	288	5,304	
PQ hybrid HPKE Ciphersuites						
ML-KEM & DH-KEM	512 & X25519	32	832	848	1.0	Tern
	768 & X25519	32	1216	1168	1.0	
	1024 & X25519	32	1600	1648	1.0	

- When it comes to post-quantum (PQ) algorithms, that are not specified in the MLS standard, we have considered the most promising KEMs arising from the NIST's PQ competition. This process has led to the standardization of Crystals Kyber [13] [31] – under the name *ML-KEM* [28] – while three other schemes remain studied in the fourth round of the competition as alternatives to Kyber: BIKE [5], HQC [2] and Classic McEliece [3]. We have also looked at some other KEMs from the PQC competition: FrodoKEM [27] and NTRU [36], as well as an hybrid KEM composed of

both ML-KEM and DH-KEM-X25519. In this setting, the size of the path secret is of 32 bytes, whatever the PQ KEM considered.

Table 1 underlines that classical HPKE ciphersuites are suited to binary Ratchet Trees currently used by MLS. However, most PQ schemes, and in particular the NIST's PQ standard ML-KEM are more adapted to ternary trees due to their τ value around 1.

D ADDITIONAL EXPERIMENTAL RESULTS

This appendix includes some experimental results that did not fit in the body text.

D.1 Efficiency Analysis of a Binary Tree

Firstly, in complement to the efficiency analysis displayed in Section 3, Figure 16 shows an instance of the communication cost of a binary Ratchet Tree evolving freely (i.e. with the *wild* tree evolution mechanism), compared to TreeKEM's method. It underlines the results from Section 3.5, which state that this free evolution is more efficient than TreeKEM's tree evolution.

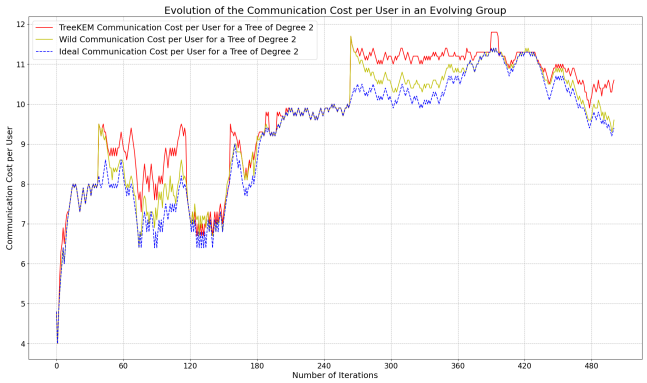


Figure 16: Compared communication costs of a randomly evolving Ratchet Tree, between TreeKEM's methodology and the *wild evolution*.

D.2 Comparison between Binary and Ternary Trees

Figure 17 hereunder depicts the bound of τ under which a non-full ternary tree is in average more efficient than a quaternary one. This experimentally determined bound is located at $\tau \approx 1.72$.

We additionally provide in Figure 18 the bound of optimality between binary and ternary *full* trees, determined from Equation (8). Despite the fact that in real use-cases, full trees with a degree $m \geq 3$ are not used, this limit value of τ appears very to the one determined experimentally between binary and *non-full* ternary trees (cf. Section 4.3.2).

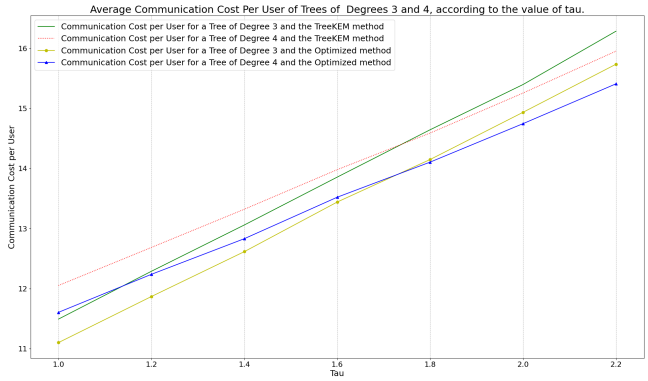


Figure 17: Compared communication costs per user of ternary and quaternary trees evolving with TreeKEM's and our optimized methods. It shows an optimality bound between these two degrees at $\tau \approx 1.72$.

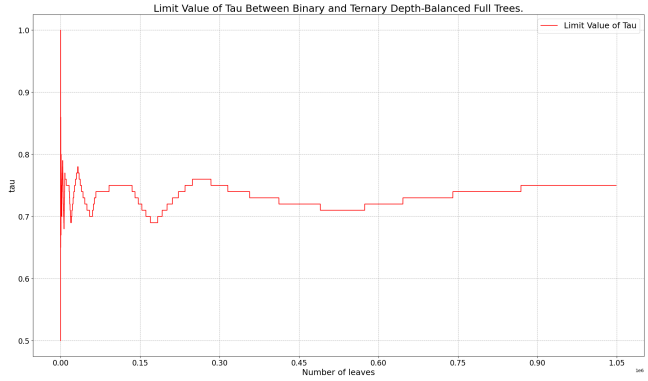


Figure 18: Limit value of tau under which a full binary tree has a better efficiency than a full ternary one.

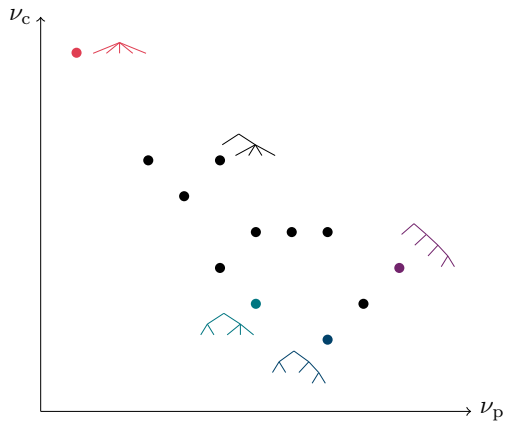
E DETAILS ON THE USE OF CONVEX HULL TO DETERMINE OPTIMAL TREE DEGREES

We detail hereunder in Figure 19 the steps that lead to the computation of the convex hull of the dual of the cost function $\kappa_{\mathcal{S}}$ over a set $\mathcal{S}$ of trees.

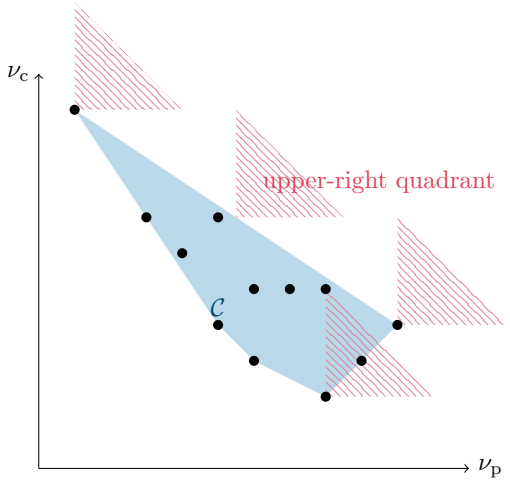
F OPTIMIZATION ALGORITHMS

Figure 20 details the pseudocode for our optimized algorithm for a tree evolution mechanism:

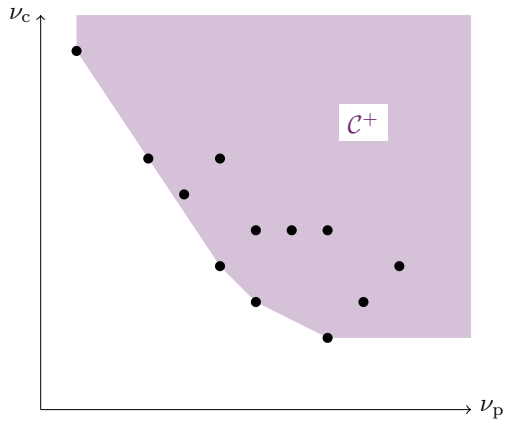
- Lightest Child algorithm, for a user add, with the algorithms to build and update the associated Weight Tree.
- Bottom Tree Expansion.



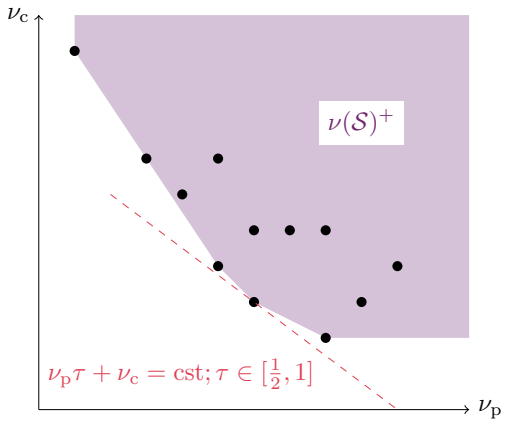
(a) Geometric costs $\nu(\mathcal{S}) = \{(\nu_p(T), \nu_c(T)),\ T \in \mathcal{S}\}$ associated with the set $\mathcal{S}$ of all 12 trees with 5 leaves.



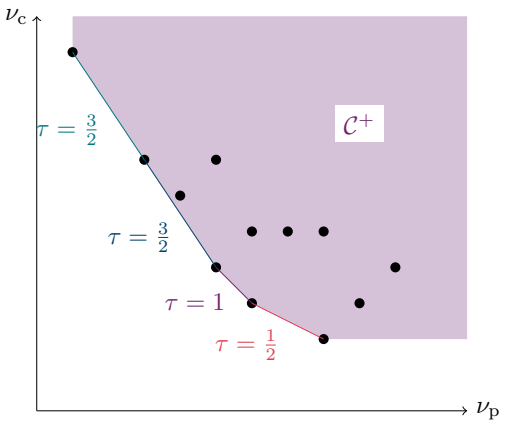
(b) The convex hull $\mathcal{C}$ of $\nu(\mathcal{S})$, and the upper-right quadrant $\mathcal{Q}$, before computing the Minkowski sum.



(c) The Minkowski sum $\mathcal{C}^+ = \mathcal{C} + \mathcal{Q}$; it is also the lower-left convex hull of $\nu(\mathcal{S})$.

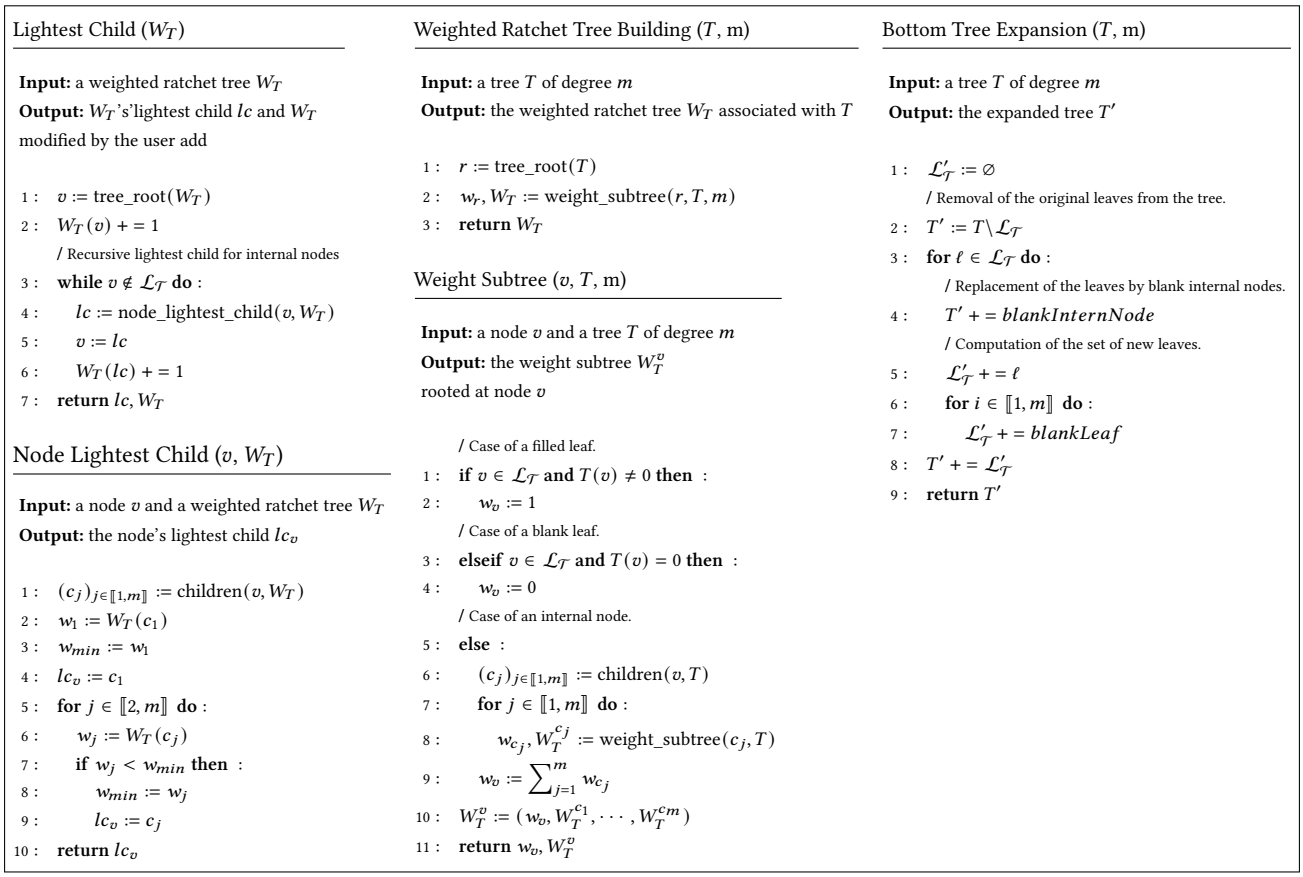


(d) The optimal tree for a given parameter τ (opposite of the slope) is a point on the lower-left convex hull $\mathcal{C}^+$.



(e) The edges of this polygon also correspond to the limiting values of τ between different optimal trees.

Figure 19: Determination of the convex hull of the dual of the cost function $\kappa_{\mathcal{S}}$ over the set $\mathcal{S}$ of trees with five leaves.



Lightest Child (W_T)

Input: a weighted ratchet tree W_T
Output: W_T's'lightest child lc and W_T modified by the user add

$v :=$ tree_root(W_T)
$W_T(v) \mathrel{+}= 1$
/ Recursive lightest child for internal nodes
while $v \notin \mathcal{L}_\mathcal{T}$ **do** :
  $lc :=$ node_lightest_child(v, W_T)
  $v := lc$
  $W_T(lc) \mathrel{+}= 1$
return lc, W_T

Node Lightest Child (v, W_T)

Input: a node v and a weighted ratchet tree W_T
Output: the node's lightest child lc_v

$(c_j)_{j\in[\![1,m]\!]} :=$ children(v, W_T)
$w_1 := W_T(c_1)$
$w_{min} := w_1$
$lc_v := c_1$
for $j \in [\![2, m]\!]$ **do** :
  $w_j := W_T(c_j)$
  if $w_j < w_{min}$ **then** :
    $w_{min} := w_j$
    $lc_v := c_j$
return lc_v

Weighted Ratchet Tree Building (T, m)

Input: a tree T of degree m
Output: the weighted ratchet tree W_T associated with T

$r :=$ tree_root(T)
$w_r, W_T :=$ weight_subtree(r, T, m)
return W_T

Weight Subtree (v, T, m)

Input: a node v and a tree T of degree m
Output: the weight subtree W_T^v rooted at node v

/ Case of a filled leaf.
if $v \in \mathcal{L}_\mathcal{T}$ **and** $T(v) \neq 0$ **then** :
  $w_v := 1$
/ Case of a blank leaf.
elseif $v \in \mathcal{L}_\mathcal{T}$ **and** $T(v) = 0$ **then** :
  $w_v := 0$
/ Case of an internal node.
else :
  $(c_j)_{j\in[\![1,m]\!]} :=$ children(v, T)
  for $j \in [\![1, m]\!]$ **do** :
    $w_{c_j}, W_T^{c_j} :=$ weight_subtree(c_j, T)
  $w_v := \sum_{j=1}^{m} w_{c_j}$
$W_T^v := (w_v, W_T^{c_1}, \cdots, W_T^{cm})$
return w_v, W_T^v

Bottom Tree Expansion (T, m)

Input: a tree T of degree m
Output: the expanded tree T'

$\mathcal{L}'_\mathcal{T} := \emptyset$
/ Removal of the original leaves from the tree.
$T' := T \setminus \mathcal{L}_\mathcal{T}$
for $\ell \in \mathcal{L}_\mathcal{T}$ **do** :
  / Replacement of the leaves by blank internal nodes.
  $T' \mathrel{+}= blankInternNode$
  / Computation of the set of new leaves.
  $\mathcal{L}'_\mathcal{T} \mathrel{+}= \ell$
  for $i \in [\![1, m]\!]$ **do** :
    $\mathcal{L}'_\mathcal{T} \mathrel{+}= blankLeaf$
$T' \mathrel{+}= \mathcal{L}'_\mathcal{T}$
return T'

Figure 20: Pseudocode descriptions of the *Lightest Child* algorithm, for an optimized User Add, and of the *Bottom Tree Expansion* algorithm.