Author's Note and Updates

September 2025

The following document is the original version of my Bachelor's Thesis, submitted in April 2025 in fulfillment of the requirements for the BS degree at the Indian Institute of Science.

Subsequent research on this project, aimed at preparing the results for conference submission, has revealed a refinement to one of the underlying assumptions. The analysis in Section 3.2, which discusses the ideal communication cost, is based on an assumption about the optimal tree structure. My continued work has shown that while the algorithms developed in this thesis successfully optimize towards a balanced tree with low communication cost, the initially assumed "ideal" structure is not the true global optimum.

Crucially, this refinement does not invalidate the primary contribution of the thesis: the design and validation of the ILP-Opt algorithms which demonstrate a significant improvement over existing methods. The work stands as a robust analysis of practical optimizations for TreeKEM. The ongoing research builds upon this foundation by correcting the theoretical ideal and further enhancing the algorithms.

- Sourabh Peruri

PS: The curriculum limits Bachelor's thesis (in mathematics) to four pages. Therefore, some details and nuances are glossed over or omitted.

Optimizing TreeKEM: Algorithmic Improvements in the Messaging Layer Security standard

A Thesis submitted for the completion of requirements for the degree of

Bachelor of Science (Research)

by

Peruri Sourabh
Mathematics Undergraduate
Indian Institute of Science

Under the supervision of

Prof. Sanjit Chatterjee
Dept. of Computer Science and Automation, Indian Institute of Science

And

Prof. Vamsi Pritham Pingali
Dept. of Mathematics, Indian Institute of Science

Abstract

The Messaging Layer Security (MLS) protocol is an upcoming internet standard that aims to enable decentralized and asynchronous key establishment in a group messaging service while offering dynamic membership (add and remove) features and end-to-end encryption. At the heart of MLS protocol lies TreeKEM, a data structure that aims to provide a logarithmic scaling of the size of certain handshake messages (called "commits"), thereby making it highly efficient. However, numerous studies indicate that this efficiency does not show up in practical implementations. Furthermore, to make TreeKEM quantum-secure, it must be instantiated with a post-quantum (PQ) Key Encapsulation Mechanism (KEM). However, most PQ-KEMs have huge public key and ciphertext sizes, which poses a practical problem in the migration from classical methods: accommodating the necessary increase in bandwidth, leading to significant interest in its optimization. This thesis aims to study the efficiency of existing TreeKEM implementations, and propose further optimizations, specifically in the context of Kyber KEM, the recently NIST-standardized PQ-KEM.

1 Preliminaries

1.1 Key Encapsulation Mechanisms

The MLS standard [2] relies on the TreeKEM[1] protocol for secure group key management. Here, 'KEM' refers to a Key Encapsulation Mechanism. While private key encryption schemes are efficient, they require secure key distribution. While public key schemes provide a solution to it, they are computationally expensive and need additional infrastructure. Public key based KEMs allow two users to arrive at a shared secret that is used as the *key* for a private key scheme, capturing the best of both worlds. TreeKEM works by pairing a generic KEM with a tree structure.

1.2 Tree Terminology

Notation: root (ρ), ancestor, parent (p_ν), children ($\mathcal{C}_\nu$), leaf (l), depth (d), subtree (T_ν). The *degree* of a node is the number of its children. The *arity* of a tree is the maximum degree that any node can have. An *internal node* is any non-leaf node. A *perfect tree* is a tree where each internal node has the same degree and each leaf is at the same depth. Given a node ν, its *siblings* $\mathcal{S}_\nu$ are the other children of its parent. Its *direct path* $\mathcal{P}_\nu$ is the set of all of its ancestors along with itself and the root. Its *copath* $\mathcal{CP}_\nu$ is the union of the siblings of all the nodes on its direct path.

2 MLS and the TreeKEM protocol

Group messaging protocols need to account for dynamic membership changes. To maintain security, the group key is refreshed whenever the group is modified, requiring coordinated communication between devices to ensure proper key management. TreeKEM leverages a tree structure to efficiently manage group keys, ensuring logarithmic scaling of overhead communication, making it suitable for large groups. A simplified version of the TreeKEM protocol [3] is outlined below. Section 2.1 gives an abstract description while Section 2.2 goes into details.

2.1 High-level Protocol Overview

In a group, an *epoch* corresponds to a given state of the group and a certain group key. Each time the group is modified, the group key evolves and the epoch changes. In an epoch, messages are exchanged by encrypting with the group key and broadcasting them to every user. TreeKEM uses a generic *propose-and-commit*[2] paradigm, which is explained here:

- **Group Initialization.** User u creates a new group with users $\mathcal{U}$. A welcome message W is sent to all the members with the information necessary to join the group.
- **Proposal.** User u proposes to change a group's state through an action $a \in \mathbb{A} \supseteq \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Update}\}$ and generates a proposal message P with the relevant information which is broadcasted to the whole group.
- **Commit.** User u takes a proposal P, validates it and *commits* it (as explained in Section 2.2) to generate a new group key k. It then broadcasts a commit message C to the entire group.
- **Process.** User u receives a message $m \in \{C, W\}$ and computes a new group key k from it.

In this paradigm, performing any action is a two step process: propose and commit. Any user can commit the broadcasted proposal, making them the *committer.*

2.2 Detailed TreeKEM Operations

TreeKEM uses *ratchet trees*, simply called trees here. Each group is assigned a tree, with the users placed at the leaves. Every user maintains a `leaf-secret` that (deterministically) computes a public key-pair for the KEM. An internal node also *may* contain a `node-secret` , which similarly yields a key-pair. A node is called *filled* if it has a key-pair, and *blank* otherwise. The `root-secret` computes the group key, but it does not have a key-pair. The public keys of all nodes (if they exist) are known to every user. A `leaf-secret` is known only to the leaf's user. A `node-secret` is known only to users in the subtree rooted at that node. The `root-secret` is known to every user. To summarize, the protocol maintains the *tree invariant* [2]:

$$(\text{TI})^3 \qquad u_i \text{ knows } \texttt{node-secret} \text{ of } \nu \iff \nu \in \mathcal{P}_{u_c}$$

The hierarchical structure of a tree paired with TI offers significantly efficient communication. A part of **commit** is to send an encrypted message to all the users in a node's subtree. Instead of sending an encryption for each user, a single encryption under the node's public key suffices since these users know the node's secret key.

Since it is **commit** that is responsible for the modification of the tree structure, we study it in detail. It has three key steps: proposal implementation, group key calculation, and commit message formulation.

[1]While the term 'TreeKEM' itself doesn't appear in the MLS proposed standard, it is implemented via 'ratchet trees', which themselves are heavily inspired by the original TreeKEM protocol [3], leading to an interchangeable use of these terms in literature.

[2]A major simplification done here is to discard the notion of 'local' and 'global' states.

[3]The notion of 'unmerged leaves', explained in MLS, makes the $\iff$ in (TI) an $\Longrightarrow$ but we skip it for clarity.

2.2.1 Proposal implementation

The committer first implements the proposal P, which is one of the following:

Initialize. When a group of N users is created, a blank, perfect tree of the required depth $d = \lceil \log_2(N) \rceil$ (MLS specifies a binary tree) is initialized and the N users placed at the leaves from left to right, which become filled.

Add. To add a user, the committer places them (their public key) in the left-most blank leaf, making it filled. If all leaves are filled, **Expand** algorithm is run before addition.

Expand. A new root is created, with the original tree becoming its left subtree and a blank perfect tree of the same depth becoming its right subtree.

Remove. The committer clears the user from the corresponding filled leaf, thereby blanking it, and then blanks all the nodes along the leaf's direct path (up to the root). If this results in an empty right subtree, **Prune** (reverse of **Expand**) is run and the right subtree discarded.

Update. The committer clears the user's public key from the leaf, places their new public key (broadcasted in their proposal) and then blanks all nodes on the direct path.

These operations are driven by security requirements and adhere to the tree invariant. For example, when a user is removed, all secrets known to them must be discarded, i.e., all nodes in his path must be blanked.

2.2.2 Group key calculation

The committer u_c next calculates a new group key by 'propagating' new secrets to the root. They first randomly choose a new leaf-secret. Then, for every ν on u_c's path[4], node-secret(p_ν) is replaced with $\mathsf{H}($node-secret$(\nu))$, where H is a cryptographic hash function: given $\mathsf{H}(x)$, it is hard to find x. This removes the possibility of backtracking. New key-pairs are then computed for each node (except the root) from their secrets, and then the group key derived as $k = \mathsf{KDF}($root-secret$)$ from a key derivation function (Figure 1). Asynchronosity arises exactly because the committer need not communicate with anyone to establish the key. The commit message can be processed whenever a user comes online.

2.2.3 Commit message formulation

Clearly, the committer must broadcast the list of proposals he implemented, along with all the new public keys on the (filtered) direct path (excluding the root) to enable future communication to these nodes. At this point, the committer's sibling lacks the (new) node-secret of his parent, violating (TI). To restore this, the committer includes this node-secret in the commit message, encrypted under the sibling's public key (which the committer knows). Extending this to all nodes on the direct path, the committer must send distinct encrypted secrets to each node on his copath, which may be blank. To handle such cases succinctly, we introduce a definition.

Definition 1 (Resolution of a node [2]). *The resolution* $\mathsf{Res}(\nu)$ *of a node* ν *is defined as follows:*

- *If* ν *is a filled node, then* $\mathsf{Res}(\nu) = \{\nu\}$
- *If* ν *is a blank leaf,* $\mathsf{Res}(\nu) = \emptyset$
- *If* ν *is a blank internal node, then*

$$\mathsf{Res}(\nu) = \cup_{\mu \in \mathcal{C}(\nu)} \mathsf{Res}(\mu)$$

Intuitively, $\mathsf{Res}(\nu)$ is the least set of filled nodes that 'cover' all the users in T_ν. So for every node $\nu \in \mathcal{P}_{u_c} \backslash u_c$, node-secret$(\nu)$ has to be encrypted and sent to all the users in $\mathsf{Res}(\mu)$ for $\mu \in \mathcal{C}_\nu \backslash \mathcal{P}_c$. This forms the final part of the commit message. (Figure 1). Any user receiving this message can decrypt exactly one of these secrets, using which the group key can be calculated as in Section 2.2.2.

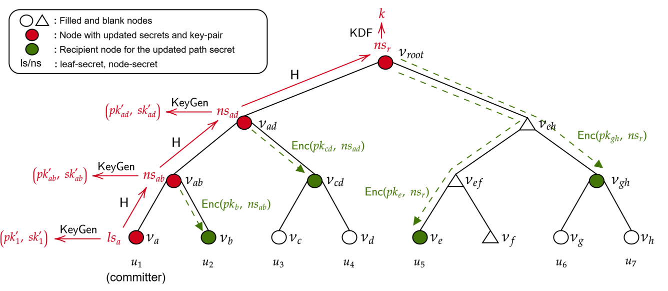

Figure 1: Commit operation in a 7-member group. Copath nodes require secrets to restore TI. Newly generated public keys and associated ciphertexts form the commit message.

This reveals the influence of filled and blank nodes on the commit size. We now study this more formally.

3 Influence of the Tree Structure

Optimizing bandwidth of MLS reduces to optimizing the commit message size which in turn, reduces to optimizing the tree structure. We first recall from [4] a measure of efficiency for tree structures and then suggest an extension.

3.1 The Communication Cost Metric

Definition 2 (Path cost). *The path cost* $v_p(u)$ *of a user is the number of nodes in its (filtered) direct path, excluding the root. The path cost of a tree* v_p^T *is the sum of path costs of all its users. Here,* $[\![\cdot]\!]$ *equals 1 if true and 0 otherwise, while* $\mathsf{users}(\nu)$ *denotes the number of users present in* T_ν.

$$v_p(u) = \sum_{\nu \in \mathcal{P}_u \backslash \rho} [\![\mathsf{users}(p_\nu) \neq \mathsf{users}(\nu)]\!] \qquad (1)$$

[4] The MLS standard defines a *filtered* direct path, which is the correct path to use here. The simplification is made for a clearer understanding.

Definition 3 (Copath cost)**.** *The copath cost $v_c(u)$ of a user is the number of nodes in the resolution of its copath. The copath cost of a tree v_c^T is the sum of copath costs of all its users.*

$$v_c(u) = \sum_{\nu \in \mathcal{CP}_u} |\mathsf{Res}(\nu)| \tag{2}$$

Excluding the structure-independent proposal list, the commit message size for user u simply becomes:

$$C(u) = v_p(u)|\mathsf{pk}| + v_c(u)|\mathsf{ct}|$$

where $|\mathsf{pk}|$ and $|\mathsf{ct}|$, i.e, public key and ciphertext sizes, depend only on the KEM parameters. Letting their ratio be the fixed global parameter τ, we define:

Definition 4. *The communication cost of a tree $C(T)$ is the sum of the sizes of commit messages:*

$$C(T) = \sum_{u \in \mathcal{U}} C(u) = |\mathsf{ct}|(v_p^T \tau + v_c^T) \tag{3}$$

Our goal is to optimize the *core* communication cost:

$$\kappa^T = v_p^T \tau + v_c^T \tag{4}$$

Also, we have the following simplified expressions:

$$v_p^T = \sum_{\nu \in T \setminus \rho} \mathsf{users}(\nu) [\![\mathsf{users}(p_\nu) \neq \mathsf{users}(\nu)]\!] \tag{5}$$

$$v_c^T = \sum_{\nu \in T \setminus \rho} |\mathsf{Res}(\nu)| \left(\mathsf{users}(p_\nu) - \mathsf{users}(\nu)\right) \tag{6}$$

3.2 Ideal communication cost

While [4] analyzed the ideal cost for full trees of arity m, we extend it by analyzing arbitrary trees with N users. Since v_p is unaffected by the tree's internal nodes as seen from Equation 5, for a given tree, v_c would be the lowest if all the internal nodes are filled, hence making $|\mathsf{Res}(\nu)| = 1$ for all nodes. Further, [4] proves that the optimal cost is achieved when the path cost of each user differs by at most 1, which they call *depth-balance*. Letting $d = \lceil \log_m(N) \rceil$, the result translates to saying that each node at depth $(d-1)$ (just above the leaves) has at least one user in a depth-balanced tree.

So, for a tree with arity m and N users, the ideal tree structure has depth d with all internal nodes filled. Let $x_1, ..., x_m$ denote the number of nodes at depth $(d-1)$ with $1, ..., m$ children respectively. Then the constraints and the costs are given as follows:

$$\sum_{i=1}^{m} x_i = m^{d-1} \qquad \sum_{i=1}^{m} i \cdot x_i = N \tag{7}$$

$$v_p = N(d-1) + (N - x_1) = N \cdot d - x_1 \tag{8}$$

$$v_c = mdN - N(m+d) + \sum_{i=1}^{m} i^2 \cdot x_i \tag{9}$$

At depth d, only the nodes with a single user make the boolean condition in Equation 1 false, making the path cost for that depth $N - x_1$.

Optimizing the core cost κ^T now leaves us with an Integer Linear Program (ILP) with the following minimization function on the integral variables $x_1, x_2, ...x_m$:

$$-(\tau - 1)x_1 + \sum_{i=2}^{m} i^2 x_i \tag{10}$$

The ideal cost κ_{ideal}^T can be calculated after solving this ILP. Note that x_1 will be maximized in this setting since it has a negative coefficient ($\tau > 1$). Looking at the trends in the ideal costs, $v_p = \mathcal{O}(N \log_m N)$ and $v_c = \mathcal{O}(mN \log_m N)$. Hence the average communication cost per user, at least ideally, has a very efficient logarithmic scaling. Also, increasing arity reduces the path cost but increases the copath cost. Depending on τ, an optimal arity is determined.

4 Practical Implementation

To model the efficiency of the tree structure, we abstract away all the details about key pairs, hash functions, and secrets. Only the states of the internal nodes matter. So, the commit algorithm is simply modelled to effect the proposals and fill the nodes on its filtered direct path. Of the algorithms mentioned in Section 2.2.1, **remove**, **update** and **commit** are completely user-dictated, and have no scope for optimization. **Prune** is invoked to reduce the tree size. It doesn't impact the bandwidth cost.

So the optimization reduces to the remaining algorithms. In **initialize**, there is freedom of choosing the user placement. In **add**, we can dictate the empty leaf location. In **expand**, new leaf locations can be dictated.

4.1 The MLS implementation

In the MLS standard [2], every algorithm is left balanced:

- **Initialize.** The N users fill the leaves from left to right.
- **Add.** The added user is assigned the left-most empty leaf.
- **Expand.** As explained in Section 2.2.1

Even theoretically, there is justified inefficiency. For eg, with initialize, of the M nodes at depth $d-1$, many will have m users while many others have zero, which is far from the ideal. Same is the case with the expand algorithm.

4.2 The Bonsai implementation

In [4], more optimized algorithms are proposed (with **initialize** same as MLS). To the best of our knowledge, these are the most optimal existing algorithms. This implementation will be called 'Bonsai' here.

- **Add.** The user location is determined by starting from the root and picking the lightest (in terms of $\mathsf{users}(\cdot)$) subtree at each step, giving the tree a more balanced structure.
- **Expand.** The tree expands in the bottom direction by adding m new leaves to each old leaf, making it an internal node, and displacing the user in the old leaf to one of the m new leaves.

The paper proves that MLS-**Expand** is the least optimal and their Bonsai-**Expand**, the most. It can be justified heuristically: bottom expansion leads to all nodes at the old leaf depth still having a single user, leading to a maximal value for x_1. Even the user add minimizes the square term in the v_c, leading to a better solution.

4.3 Our ILP-Optimized implementation

Building on the Bonsai approach, we propose a further optimized implementation with ILP-driven decisions.

- **Initialize.** An ILP solver finds the most optimal solution $\mathbf{x}^*$, then assigns the users in descending order, i.e., the first x_m nodes at depth $(d-1)$ get assigned m users, and so on.
- **Add.** We scan through the list of nodes at depth $(d-1)$ and pick the node that increases the ideal cost the least, depending upon the parameters τ, m.
- **Expand.** Same as Bonsai-**Expand**.

Clearly, at every step, these algorithms make close-to-optimal choices. We call this method ILP-Opt.

5 Simulating TreeKEM

5.1 Methodology

We developed a custom simulation framework implementing all three methods—MLS [2], Bonsai [4], and our proposed ILP-Opt—and evaluated them using Kyber mKEM Level I parameters [1] whose τ = 30.667 and |ct| = 48B. Each simulation runs for 1000 iterations with equal action probabilities (add, remove, update), and trees are defined by method, arity, and initial user count. At each step, trees undergo synchronized random actions and commits, with communication costs computed per user using Equations (3)–(6) and compared against ideal costs from Equations (8), (9).

5.2 Results

- Bonsai outperforms MLS for binary trees, re-establishing the results from [4].
- Increasing arity beyond binary is advantageous, but arity above 16 with ILP-Opt shows no major improvement over binary Bonsai and MLS.
- The ILP-Opt **Initialize** and **Add** algorithms yield better path costs across all simulations, but as N, m increase, copath costs dominate, producing mixed overall results.
- Theoretical optimal arity of 32, as suggested in [5], performs worse in practice than binary MLS or Bonsai.
- Path costs remain near ideal values while copath costs fluctuate significantly, failing to reach theoretical lower bounds due to the sparse filled-node population, with $v_c = \mathcal{O}(N^2)$ at initialization.
- MLS performs better when copath costs dominate, suggesting a change in approach for optimizing copath cost.
- Increasing the commit frequency, previously unexplored, substantially reduces the communication cost.

Tree	**Ideal**	**Simulation**
2-ary MLS	15.18	17.69
2-ary Bonsai	15.18	17.03
2-ary ILP-Opt	15.18	17.02
4-ary ILP-Opt	8.07	12.56
8-ary ILP-Opt	6.43	14.50

Table 1: Average per-user communication costs (KB) for each tree in a run with 1000 initial users. As seen, 4-ary ILP-Opt performs the best.

6 Concluding Remarks

Our optimized implementation matches Bonsai's gains on binary trees and further leverages ILP-driven decisions to achieve the lowest observed costs when arity is increased beyond 2, demonstrating its ability to approach ideal efficiency. While theoretical minimum occurs at 32, the practical costs see an increase, highlighting the increasing dominance of the less-optimized copath costs. We also highlight the necessity of more frequent commits since they serve to fill the internal nodes, periodically bringing its costs down further.

Follow-up work will focus on refining core protocol parameters and structures: (1) determine and formally propose the optimal arity for MLS based on our cost metrics, (2) implement and validate existing theoretical efficiency analyses with our simulation environment, (3) optimize copath costs for high-user-count and high-arity scenarios, and (4) balance path versus copath cost trade-offs to achieve optimal performance across varying group sizes.

Looking further ahead, we plan to (1) develop optimized Kyber mKEM parameter sets for NIST Levels III and V and study their optimality, (2) explore dynamic-arity strategies, and (3) analyze and minimize computational costs – potentially via *server-aided* protocols while preserving security.

References

[1] Joël Alwen, Matthew Campagna, Dominik Hartmann, Shuichi Katsumata, Eike Kiltz, Jake Massimo, Marta Mularczyk, Guillermo Pascual-Perez, Thomas Prest, and Peter Schwabe. How multi-recipient kems can help the deployment of post-quantum cryptography. URL: `https://csrc.nist.gov/csrc/media/Events/2024/fifth-pqc-standardization-conference/documents/papers/how-multi-recipient-kems.pdf`.

[2] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023. URL: `https://www.rfc-editor.org/info/rfc9420`, `doi:10.17487/RFC9420`.

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

[4] Céline Chevalier, Guirec Lebrun, Ange Martinelli, and Jérôme Plût. The art of bonsai: How well-shaped trees improve the communication cost of MLS. Cryptology ePrint Archive, Paper 2024/746, 2024. URL: `https://eprint.iacr.org/2024/746`.

[5] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. Scalable ciphertext compression techniques for post-quantum KEMs and their applications. Cryptology ePrint Archive, Paper 2020/1107, 2020. URL: `https://eprint.iacr.org/2020/1107`.