0-ART. Asynchronous and Verifiable Group Management for Decentralized Applications

Draft, v.0.1

Yevhen Hrubiian*, Illia Melnyk*, Volodymyr Dubinin*, Oleksandr Kurbatov*, Serhii Volynets*, Roman Perebynos*, Yevhenii Serdiukov*

*Distributed Lab

October 8, 2025

Abstract

The majority of modern e2e private applications face scalability issues that limit their functionality, broader adoption, and overall user experience. Some of them organize private groups as a set of peer-to-peer chats, which leads to an overall quadratic complexity in the size of group communication and a linear time complexity in the number of members for encryption. Others apply more scalable group key establishment constructions (such as ART), but at the same time, they do not support verifiable and concurrent updates. In this paper, we introduce the 0-ART protocol, which aims to address the aforementioned issues by (1) verifiable group operations; (2) a causal tree construction allowing for multiple concurrent updates and efficient member removal; (3) anonymous credentials, making privacy in the group available while keeping operations authentic. We implemented the 0-ART framework and applied it to a decentralized collaborative work application. According to our benchmark, executing the most complex operation (performing the provable AddMember operation in a group of size 2^{20}) takes 1.57 seconds on the user's device and 24Kb of proof size.

Contents

1 Introduction **3**
1.1 Contributions . . . 4

2 Background and Related Work **4**
2.1 Previous Constructions . . . 4

3 Technical Preliminaries **6**
3.1 Notation . . . 6
3.2 Zero-Knowledge Proofs . . . 7
3.3 Key types . . . 8
3.4 ART . . . 8
3.5 Bulletproofs . . . 9

4 0-ART **10**
4.1 Argument system . . . 10
4.1.1 ART operation eligibility . . . 10

4.1.2 ART update proof . . . 10
4.2 Argument system based on Bulletproofs . . . 11
4.3 Causal ART . . . 11

5 Group Management based on the 0-ART Framework 13

6 Anonymous Role Management in groups based on 0-ART 22
6.1 Anonymous Credentials . . . 22
6.1.1 General construction . . . 22
6.1.2 Basic Functions . . . 23

7 Security 23
7.1 Malicious-insider attack . . . 24
7.1.1 Definition . . . 24
7.1.2 Attack scenario . . . 24
7.1.3 Terms of success . . . 24
7.2 0-ART security model . . . 25
7.2.1 Forward Secrecy . . . 25
7.2.2 Post-Compromised Secrecy . . . 26
7.2.3 Operations Verifiability . . . 26

8 Implementation and Benchmarks 27
8.1 Benchmarks . . . 27

9 Conclusions 28

A Circuit logic for proving scalar multiplication 31

B ART Multi-Device 32
B.0.1 Main concept . . . 32
B.0.2 Construction . . . 32

1 Introduction

In systems that presume data exchange, messaging, or the handling of sensitive information, there is an increasing demand for robust encryption mechanisms. It ensures privacy and guarantees that only legitimate users (entities) can access sensitive information. One of the most successful continuous key management protocols is the Double Ratchet algorithm [MP16]. Its compelling combination of the Diffie-Hellman protocol [DH76] and ratchet chains [MP16; KE10] enables the assurance of forward secrecy (FS) and post-compromise secrecy (PCS).

The natural next step was to support FS and PCS properties for end-to-group encryption (e2g), enabling efficient private communication among a large number of users. However, since Double Ratchet was originally designed solely for e2e communication, the naive way to adapt it for group settings was to establish pairwise encryption between all participants. Essentially, this approach can be compared to a large number of peer-to-peer chats between every pair of participants that must operate simultaneously. It is not efficient, leading to quadratic complexity in the group key establishment process and linear encryption complexity.

Therefore, to support larger groups, only a single symmetric ratchet is typically used. This reduces overhead but still does not provide the scalability required for very large groups. Such an approach is employed by Signal [PM16]: for small groups, it uses regular pairwise Double Ratchet, while for larger groups, it relies on the Sender Key protocol, as described in [BCG23]. In this protocol, a user establishes a private peer-to-peer channel with each group member to distribute their sender key, which is then used in a symmetric ratchet. Therefore, every participant holds the sender keys of all group members and can decrypt their messages. This approach preserves forward secrecy, but backward secrecy is only partially achieved: if a sender key is compromised, an adversary can decrypt all messages encrypted under it until the key is refreshed via private channels. In addition to Signal, many other private messengers adopt the same approach with minor modifications to the original Double Ratchet (Olm [Mat19], Megolm [Mat22], and Ibex [PPD23]). However, these adaptations are tailored to their specific solutions and do not address the scalability problem.

Notably, protocols and applications using the Double Ratchet approach do not achieve user anonymity because the algorithm is fundamentally based on known participants. To decrypt a message, the recipient needs to know which specific ratchet to use to derive the correct encryption key, which directly links the message to its sender. Signal has implemented a ”Sealed sender” which can hide the sender’s identity, but this approach provides only partial anonymity at the server level.

In 2018, a new approach to group encryption was proposed: Asynchronous Ratchet Tree (ART) [Coh+18]. It is a cryptographic protocol designed to enhance the efficiency of private group communications in centralized or decentralized settings. ART introduces a mechanism for sequential updates of a ratchet tree structure, providing FS without requiring all participants to be online simultaneously. In addition, the protocol provides PCS, which ensures that an attacker cannot derive future group keys even after a session key compromise. It reduces the complexity of encryption and key establishment but causes another problem, such as the inability to accept concurrent updates and the threat of malicious group members [Xia+25].

A latter protocol, TreeKEM [Bha+18], was adapted to support concurrent updates, but the problem of malicious group members remains unresolved; it is explicitly stated that this issue should be handled at the application layer. The MLS protocol, based on TreeKEM, is not yet widely adopted; however, some messengers are already actively using it. One such example is Wire [Wir25], which currently supports private groups of up to 2,000 users, although it remains a centralized messenger.

Although ART-based applications usually do not provide anonymity, this property can be achieved. In this case, the system derives a single key for all participants, allowing them to be indistinguishable, and

simultaneously determines their membership in this group.

Group-based privacy can be relevant not only for messengers but for a much wider range of applications. Collaborative workspaces that encrypt changes to documents, for example, can adapt to this approach. Keyhive [Swi25] proposed the BeeKEM scheme [Ink20] for easy merging of concurrent updates. Another project is NextGraph [Nex25], which is developing a platform for creating decentralized Web 3.0 applications with collaborative editing support. Interestingly, NextGraph suggests that collaborative workspaces do not require Perfect FS (PFS) because it negatively impacts new group members. Nevertheless, PFS is an essential system feature because the group must be resistant to attacks that recover previous keys. Additionally, the system must effectively handle key updates. However, NextGraph allows you to maintain anonymity within the group.

1.1 Contributions

We present **0-ART (zero-ART)**, the anonymous group key establishment protocol that extends the ART with verifiable operations.

- The protocol enables trustless group communication in decentralized settings, where every group member proves the correctness of tree updates, making the disruption of ART state practically infeasible. This approach enhances the scalability and security of ART without relying on a trusted party's assistance
- A new update merge scheme is proposed, inspired by [Wei19], which allows concurrent updates to be processed without losing current members' access to the group
- Additionally, we offer a new zk-based anonymous credentials scheme that enables anonymous operation within the group, while maintaining provable permissions

Structure. In Section 2, we provide the related protocols and compare their properties with the 0-ART approach. Section 3 presents the technical preliminaries and constructions employed in the protocol. Section 4 consists of the protocol description. The details and algorithms for group operations are defined in Section 5. Section 6 describes an extension for verifiable role management. We analyze the security of the presented approach in Section 7. Section 8 includes some specific details related to the practical implementation of 0-ART and benchmarks for group operations. Section 9 includes conclusions and future research ideas. We leave circuit logic for scalar multiplication in Appendix A and multi-device ART construction in Appendix B.

2 Background and Related Work

2.1 Previous Constructions

The Asynchronous Ratchet Tree [Coh+18] is based on two protocols: (1) a tree-based group DH protocol, which unites multiple users into a single system; (2) a KeyUpdate scheme, which enables secure key changes without requiring a complete regeneration of all system keys. Given the impossibility of simultaneous updates, this protocol was rejected as a candidate for the key establishment standard.

Later, the TreeKEM protocol was proposed in [BBR18]. Maintaining a tree-based architecture, TreeKEM substitutes Diffie–Hellman key agreement with a key-encapsulation mechanism. This change improves group key update efficiency by requiring the sender to perform only a logarithmic number of encryptions

with respect to the group size, and a single decryption at each recipient. TreeKEM was later chosen as the core scheme for the MLS [Bar+23] standard. This made it possible to improve the TreeKEM protocol by adapting it for practical implementations.

Today, MLS adoption remains slow for several reasons. One major factor is its architectural complexity and the need for careful update management. A propose-commit scheme typically handles this. MLS also does not mitigate risks from malicious insiders or guarantee operational integrity.

We additionally note that tree-based algorithms face a clear challenge when removing members: a revoked user may retain a large portion of the tree's key material, which must be deleted. This requires a special update procedure that restores the tree, allowing all remaining members to recompute the group secret.

Thus, we identify three apparent weaknesses discussed above: concurrent updates, participant removal, and the verifiability of the correctness of group operations. Below, we describe partial solutions that address these issues.

Merging updates. To address the issue of handling simultaneous updates in ART, we draw on prior work related to TreeKEM. The BeeKEM project [Ink20] proposes an alternative to TreeKEM, which enables merging simultaneous updates as a conflict resolution mechanism. In turn, they were inspired by the work of Matthew Weidner, who proposed an alternative scheme called Causal TreeKEM [Wei19]. While TreeKEM requires a total order imposed by a central server, Causal TreeKEM only requires a causal order. In turn, we took this concept to create a merging algorithm in ART, thereby enabling the resolution of conflicts between simultaneous updates while preserving all new secrets.

Tainted Remove. The dependency of each descendant on its children prevents ART from maintaining empty internal nodes. This makes it impossible to perform node removal operations in the tree by clearing the path, as in TreeKEM/MLS, for example. Our idea of filling the deleted leaf with some random secret is not new, and a similar proposal was made in Tainted TreeKEM [Kle+21]. This work raised the issue of possible linear amounts of secret key encryption in TreeKEM. Thus, artificial tree restoration allows this to be done only logarithmically. However, this scheme has another vulnerability. The user who created the contamination is aware of these nodes and, therefore, the alternative path. This problem is relevant for both protocols in Tainted settings, ART and TreeKEM, and will be described below in Section 4.

Proving of ART operations. In most cases, only the issue of proving the correctness of operations was raised, but no clear solutions were proposed for the ART protocol. However, a similar scheme was proposed in [DDF21] for TreeKEM. In that work, the authors employ the zero-knowledge proof systems ZKBoo and ZKBoo++ to construct proofs of branch updates in the tree. The main drawback of this approach is the large proof size, which is inefficient for processing a large number of updates.

See Table 1 for a comparison of 0-ART to the other continuous key management protocols.

Protocol	Group Update Complexity	Verifiability	Concurrent Update
Double Ratchet [MP16]	$O(n)$	Not verifiable	No such problem exists
ART [Coh+18]	$O(\log(n))$	Not verifiable	Not supported
MLS TreeKEM [BBR18; Bar+23]	$O(\log(n))$ for sender $O(1)$ for receivers	Not verifiable	Supported but with strict control
0-ART (this work)	$O(\log(n))$	Verifiable by ZKP	Supported

Table 1: Comparison table of existing approaches with ours 0-ART. The "Group Update Complexity" column indicates the cost of updating the group's secret key. In the case of the Double Ratchet, that cost corresponds to advancing the ratchet state in each of the n individual communication channels. The column "Verifiability" indicates whether the correctness of operations can be verified. The "Concurrent Update" column indicates whether an algorithm can handle concurrent updates. In the case of the Double Ratchet, this problem does not exist because communication with each user in the group occurs over a separate channel.

3 Technical Preliminaries

3.1 Notation

In this section, we briefly introduce the primary notation we are going to use.

We define $\langle G \rangle = \mathbb{G}$ as a cyclic (abstract) group of order q written additively, with generator G. A key pair is given by a tuple (d, Q), where $d \in \mathbb{Z}_q$ is a secret key and $Q = [d]G \in \mathbb{G}$ is a public key.

One-way functions. We use several one-way functions denoted as follows:

(1) $\iota : \mathbb{G} \to \mathbb{Z}_q$ – a *point-to-scalar* function.

(2) $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ – a hash function with an n-bit output.

(3) $\mathcal{H}_{\text{KDF}} : \{0,1\}^* \to \mathbb{Z}_q$ – a key derivation function.

DH scheme. We will denote by dh a shared secret, calculated as:

$$\mathsf{dh} = \mathsf{DH}(Q_1, Q_2) = [d_1]Q_2 = [d_2]Q_1 = [d_1 \cdot d_2]G$$

X3DH. Let $\mathsf{X3DH}(k_A, d_A, Q_B, Q_{B,\mathsf{spk}})$ be a function that establishes a shared secret key between two parties who mutually authenticate each other based on public keys. To compute a shared secret with user B, user A uses their own secret key d_A and ephemeral private key k_A, and the public key Q_B and the public signed prekey $Q_{B,\mathsf{spk}}$[1] of user B. The result of the protocol, the shared secret ek, is calculated as

$$\begin{aligned}
\mathsf{dh}_1 &= \mathsf{DH}(d_A, Q_{B,\mathsf{spk}}),\\
\mathsf{dh}_2 &= \mathsf{DH}(k_A, Q_B),\\
\mathsf{dh}_3 &= \mathsf{DH}(k_A, Q_{B,\mathsf{spk}}),\\
\mathsf{ek} &= \mathcal{H}_{\mathsf{KDF}}(\mathsf{dh}_1 || \mathsf{dh}_2 || \mathsf{dh}_3).
\end{aligned}$$

Infra components. We also define special ancillary services: Directory Service and Service Provider. Directory Service (or **DS**) is a special service that stores identifiers and public keys of users who have agreed

[1] A signed prekey a temporary public key, signed by the user's identity Key, that provides a secure and verifiable way to initiate a key exchange

to disclose their identity public keys. This allows group owners and admins to invite specific users to their groups. Alternatively, users could establish trust in a fully decentralized manner without a dedicated **DS** by exchanging their public keys directly. The Service Provider (**SP**) acts merely as a networking layer that guaranties message delivery.

3.2 Zero-Knowledge Proofs

During our further discussion, we operate over binary relations $\mathcal{R}$, where each pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ consists of a public statement $\mathbb{x}$ and a witness $\mathbb{w}$. The prover wants to convince the verifier of the knowledge of the witness $\mathbb{w}$ corresponding to the public (valid) $\mathbb{x}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$. One important example of such a relation that we further rely on is the **Schnorr Proof of Knowledge**:

$$\mathcal{R}_{\mathsf{PoK}} = \left\{ \begin{matrix} \mathbb{x} = (Q) \in \mathbb{G} \\ \mathbb{w} = (\alpha) \in \mathbb{Z}_q \end{matrix} \;\middle|\; Q = [\alpha]G \right\}.$$

This relation reads as follows: given the group element $Q \in \mathbb{G}$, the prover convinces the verifier of the knowledge of scalar $\alpha \in \mathbb{Z}_q$ such that $Q = [\alpha]G$. Essentially, this proves that the verifier knows the secret key α corresponding to the public key Q.

To further prove and verify such relations, we need to define the *non-interactive argument of knowledge* (NARK). We provide the following definition.

Definition 3.1 (NARK)**.** The **non-interactive argument of knowledge** (NARK) is the protocol $\Pi_{\mathsf{NARK}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ where:

- $\mathsf{Setup}(1^\lambda, \mathcal{R}) \to (\mathsf{vp}, \mathsf{pp})$ — the setup algorithm takes the security parameter $\lambda \in \mathbb{N}$ and relation $\mathcal{R}$ (commonly arithmetized according to the given proving scheme), and outputs verification and proving parameters vp, pp, respectively.
- $\mathsf{Prove}(\mathsf{pp}, \mathbb{x}, \mathbb{w}) \to \pi$ — the proving algorithm takes prover parameters pp, public statement $\mathbb{x}$, and a witness $\mathbb{w}$, and outputs the proof π.
- $\mathsf{Verify}(\mathsf{vp}, \mathbb{x}, \pi) \to \{0, 1\}$ — the verification procedure requires verifier parameters vp, public statement $\mathbb{x}$, and proof π, and outputs 1 iff π is a valid proof.

In particular, we require Π_{NARK} to be **complete** and **knowledge sound**. If the scheme is **zero-knowledge**, we call NARK a **zk-NARK**. In turn, when $|\pi| = O_\lambda(\log |\mathtt{C}|)$ and $T = O_\lambda(|\mathbb{x}|, \log |\mathtt{C}|)$ (where $|\mathtt{C}|$ is the effective size of the circuit that represents the relation), we call this NARK a **SNARK** (**succinct** NARK).

As a concrete example, Schnorr Proof of Knowledge relation $\mathcal{R}_{\mathsf{PoK}}$ is typically proven using Σ-protocols compiled to the non-interactive protocol using the Fiat-Shamir heuristic. This gives a zk-NARK for $\mathcal{R}_{\mathsf{PoK}}$. In what follows, we employ *Bulletproofs* as the primary zk-NARK.

Remark 3.2 (On notation). In what follows, we commonly denote by $\Pi_{\mathcal{R}}.\mathsf{Prove}(\cdot)$ or $\Pi_{\mathcal{R}}.\mathsf{Verify}(\cdot)$ calling a NARK over relation $\mathcal{R}$. Moreover, instead of writing $\Pi_{\mathcal{R}_\bullet}$, we simply write $\Pi_\bullet$. Additionally, we commonly omit writing pp and vp in $\mathsf{Prove}(\cdot)$ and $\mathsf{Verify}(\cdot)$ procedures explicitly for clarity.

3.3 Key types

Note, that for each private key k, the public key is calculated as $Q = [k]P$. Our framework scheme uses the key pairs defined in Table 3.3.

Key pair	Definition
$(d_A, Q_{\mathsf{ID},A})$	The identity key pair
$(k_{\mathsf{suk}}, Q_{\mathsf{suk}})$	The ephemeral **s**et**u**p **k**ey pair (temporary keys generated by an admin or group owner to perform a key agreement protocol with new group members)
$(k_{\mathsf{inv},A}, Q_{\mathsf{inv},A})$	The **inv**itation key pair, generated by the user A if A wants to be added to the group
$(k_{\mathsf{inv}',P}, Q_{\mathsf{inv}',P})$	The **inv**itation key pair generated by an admin or a group owner to establish a shared secret with an unregistered in **DS** user P
$(\lambda_{A,i}, Q_{A,i})$	The node key pair representing a particular position in the group tree. The root key pair corresponds to the group key

Table 2: Key pairs used in the protocol

Some operations require the use of both the **previous** and **new** secret keys / public keys, among others. We will differ them with as λ (old) and $\lambda^{'}$ (new) values.

3.4 ART

Let $\mathcal{T}$ be an instance of ART of the height ℓ, where each node contains a key pair $(\lambda_{A,i}, Q_{A,i})$. Here, A denotes the user who possesses the key pair of this node, and i denotes the level of the node in the tree. In our notation, level 1 corresponds to the leaf nodes, and levels increase toward the root (max level equals ℓ).

Each leaf node is occupied by some group member A with a key pair $(\lambda_{A,1}, Q_{A,1})$. For simplicity, the leaf key pair may be denoted as (λ_A, Q_A).

For each internal node, the key pair $(\lambda_{A,i}, Q_{A,i})$ is computed as follows:

$$\forall i \in [\ell - 1] : \lambda_{A,i+1} \leftarrow \iota([\lambda_{A,i}]Q_{B,i}), \quad \forall i \in [\ell] : Q_{A,i} \leftarrow [\lambda_{A,i}]G$$

where $(\lambda_{A,i}, Q_{A,i})$ and $(\lambda_{B,i}, Q_{B,i})$ are key pairs of sibling nodes at level i. As a result, both users A and B, who hold the key pairs of these sibling nodes, derive the same key pair at level $i+1$ and are considered to possess the new internal node.

We define a **co-path** $\vec{Q}_B \in \mathbb{G}^{\ell-1}$ as the array of public values in the tree needed for a particular (λ_A, Q_A) to compute the root value. We define the **leaf path**(or **user's branch**) as the list of predecessors of A's leaf. It consists of two parts: (1) private $\vec{\lambda}_A = (\lambda_{A,i})_{i\in[\ell]} \in \mathbb{Z}_q^\ell$ and; (2) public $\vec{Q}_A = (Q_{A,i})_{i\in[\ell]} \in \mathbb{G}^\ell$. If the user wants to update their λ_A with λ'_A, they need to update the corresponding leaf path. The update operation and the resulting leaf path are shown in Figure 1.

The tree key tk for the tree instance $\mathcal{T}$ is defined as $(\lambda_{A,\ell}, Q_{A,\ell})$. The tree $\mathcal{T}$ is periodically updated, starting from the epoch $\xi = 0$. To distinguish between different versions (epochs) of the tree, we denote by $\mathcal{T}_\xi$ the tree at epoch ξ. The tree key for epoch ξ is denoted as tk_ξ. When we use $\mathcal{T}_\xi$ notation over the paper for the particular tree instance, we mean only public node values $(Q_{i,j})_\xi$ of the tree.

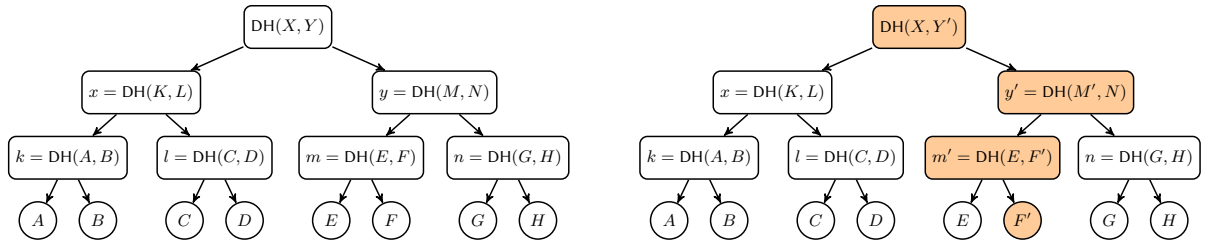


Figure 1: An illustration of the ART structure (a) and its update (b). When user F refreshes their leaf secret, the path from that leaf to the root is recomputed, where each intermediate node is derived through a Diffie–Hellman operation.

3.5 Bulletproofs

For proving relations in ART, we use Bulletproofs [Bun+18], a non-interactive zero-knowledge proof protocol. In the heart of the protocol lies an inner-product argument that allows us to prove statements such as range proofs or general arithmetic circuit satisfiability. The main advantages of Bulletproofs are an absence of a trusted setup and security against eavesdropping that relies on the discrete-logarithm assumption without any other auxiliary structures like bilinear pairings.

Bulletproofs arithmetization slightly differs from the classic R1CS. There are two types of variables in Bulletproofs constraint system: low-level and high-level. Typically, high-level variables are provided with Pedersen commitments $\mathbf{V} \in \mathbb{G}^m$ as the private witness inputs $\mathbf{v} \in \mathbb{F}_p^m$ to the circuit, while low-level variables $\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O \in \mathbb{F}_p^n$ are the intermediate witness values of computation. We will define circuit as a set of multiplication constraints operating with low-level variables and set of linear constraints which links low-level variables between each other and high-level variables as well.

Multiplication constraints are defined with one vector equation:

$$\mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O$$

Linear constraints are defined via:

$$\mathbf{W}_L \cdot \mathbf{a}_L + \mathbf{W}_R \cdot \mathbf{a}_R + \mathbf{W}_O \cdot \mathbf{a}_O = \mathbf{W}_V \cdot \mathbf{v} + \mathbf{c}$$

Thus, we can build an argument system for the following arithmetic circuit satisfiability relation:

$$\mathcal{R}_{\mathsf{sat}} = \left\{ \begin{array}{l} (G, B) \in \mathbb{G}^2, \mathbf{V} \in \mathbb{G}^m, \\ (\mathbf{W}_L, \mathbf{W}_R, \mathbf{W}_O) \in (\mathbb{F}_p^{q \times n})^3, \\ \mathbf{W}_V \in \mathbb{F}_p^{q \times m}, \mathbf{c} \in \mathbb{F}_p^q; \\ (\mathbf{a}_L, \mathbf{a}_R, \mathbf{a}_O) \in (\mathbb{F}_p^q)^3, \\ (\mathbf{v}, \mathbf{r}) \in (\mathbb{F}_p^m)^2 \end{array} \middle| \begin{array}{l} \forall i \in [m] : V_i = [v_i]G + [r_i]B, \\ \mathbf{a}_L \circ \mathbf{a}_R = \mathbf{a}_O, \\ \mathbf{W}_L \cdot \mathbf{a}_L + \mathbf{W}_R \cdot \mathbf{a}_R + \mathbf{W}_O \cdot \mathbf{a}_O = \\ \mathbf{W}_V \cdot \mathbf{v} + \mathbf{c} \end{array} \right\}.$$

Constructed protocol Π_{sat} for a given relation $\mathcal{R}_{\mathsf{sat}}$ has perfect completeness, statistical zero-knowledge, and computational soundness under the discrete logarithm assumption.

4 0-ART

This section introduces 0-ART, a framework that enhances and formalizes the practical use of ART by proving the correctness of all group operations. We present 0-ART's core designs in two parts.

First, we present the argument system and the framework for proofs and relations required for all group operations (described in Section 5). Second, we present a new merge scheme, named Causal ART, which enables the resolution of conflicts arising from simultaneous updates in the tree.

4.1 Argument system

Here, we build the argument system for tree update operations. Our goal is to develop an efficient zero-knowledge argument system capable of proving any tree update operation and update eligibility.

4.1.1 ART operation eligibility

Some group operations in our protocol change group membership, thus the administrator(entity that changes group membership) must prove their eligibility for that operation. Firstly, note that in our protocol the *group owner* always possesses the leftmost ART leaf, which could be easily verified by checking proof of knowledge of the leftmost leaf's secret. In the other case an administrator A must generate a proof π_{el} of the following general relation:

$$\mathcal{R}_{\mathsf{el}} = \{(\bot, A) | A \in \mathcal{A}\}$$

where $\mathcal{A}$ – subset of administrators of a group. In Section 6 we present a variant of an argument system for the relation $\mathcal{R}_{\mathsf{el}}$ based on *anonymous verifiable credentials*.

4.1.2 ART update proof

A corresponding proof must accompany each tree update. Suppose we have an ART instance $\mathcal{T}$ of height ℓ. The user controls λ_A and wants to update it with λ'_A. They can do it by providing a proof π_{upd} for the following relation:

$$\mathcal{R}_{\text{upd}} = \left\{ \begin{array}{l} (\vec{Q}'_A, \vec{Q}_B, Q_{A,1}) \in \mathbb{G}^\ell \times \mathbb{G}^{\ell-1} \times \mathbb{G}; \\ (\vec{\lambda}'_A, \lambda_{A,1}) \in \mathbb{Z}_q^\ell \times \mathbb{Z}_q \end{array} \middle| \begin{array}{l} Q_{A,1} = [\lambda_{A,1}]G, \\ \lambda'_{A,i+1} = \iota([\lambda'_{A,i}]Q_{B,i})|_{i\in[\ell-1]}, \\ Q'_{A,i} = [\lambda'_{A,i}]G|_{i\in[\ell]} \end{array} \right\}$$

We can divide $\mathcal{R}_{\text{upd}}$ relation into per-level sub relations – for j-th($j \in [\ell - 1]$) level of ART user A proves the following:

$$\mathcal{R}^{(j+1)}_{\mathsf{ART}} = \left\{ \begin{array}{l} (Q'_{A,j}, Q_{B,j}) \in \mathbb{G}^2; \\ (\lambda'_{A,j}, \lambda'_{A,j+1}) \in \mathbb{Z}_q^2 \end{array} \middle| \begin{array}{l} \lambda'_{A,j+1} = \iota([\lambda'_{A,j}]Q_{B,j}), \\ Q'_{A,j+1} = [\lambda'_{A,j+1}]G \end{array} \right\}$$

Special case when $j = 1$:

$$\mathcal{R}^{(1)}_{\mathsf{ART}} = \left\{ \begin{array}{l} (Q_{A,1}, Q'_{A,1}) \in \mathbb{G}^2; \\ (\lambda_{A,1}, \lambda'_{A,1}) \in \mathbb{Z}_q^2 \end{array} \middle| \begin{array}{l} Q_{A,1} = [\lambda_{A,1}]G, \\ Q'_{A,1} = [\lambda'_{A,1}]G \end{array} \right\}$$

A proof of update correctness π_{upd} is composed of $\mathcal{R}^{(j)}_{\mathsf{ART}}$ proofs : $\pi_{\mathsf{upd}} = \cup_{j=1}^{\ell} \pi^{(j)}_{\mathsf{ART}}$, where $\pi^{(j)}_{\mathsf{ART}}$ is a proof for the relation $\mathcal{R}^{(j)}_{\mathsf{ART}}$.

4.2 Argument system based on Bulletproofs

Here we present an argument system for $\mathcal{R}_{\text{upd}}$ based on Bulletproofs over some cyclic group $\mathbb{G}_2$ with generators of unknown discrete logarithm $\{P, H\}$. We start with committing to each element of $\vec{\lambda}'_A$:

$$\begin{aligned}\mathsf{Com}(\vec{\lambda}'_A, \vec{r}) =& (\mathsf{Com}(\lambda'_{A,1}, r_1), \mathsf{Com}(\lambda'_{A,2}, r_2), \ldots, \mathsf{Com}(\lambda'_{A,\ell}, r_\ell)) \\ =& ([\lambda'_{A,1}]P + [r_1]H, [\lambda'_{A,2}]P + [r_2]H, \ldots, [\lambda'_{A,\ell}]P + [r_\ell]H) \in \mathbb{G}_2^\ell\end{aligned}$$

So we can rewrite the relation using a committed witness:

$$\mathcal{R}^{\mathsf{bp}}_{\text{upd}} = \left\{ \begin{array}{l} (\vec{Q}'_A, \vec{Q}_B, Q_{A,1}, \mathsf{Com}(\vec{\lambda}'_A, \vec{r})) \in \\ \mathbb{G}^\ell \times \mathbb{G}^{\ell-1} \times \mathbb{G} \times \mathbb{G}_2^\ell; \\ (\vec{\lambda}'_A, \vec{r}, \lambda_{A,1}) \in \mathbb{Z}_q^\ell \times \mathbb{Z}_q^\ell \times \mathbb{Z}_q \end{array} \middle| \begin{array}{l} Q_{A,1} = [\lambda_{A,1}]G, \\ \lambda'_{A,i+1} = \iota([\lambda'_{A,i}]Q_{B,i})|_{i\in[\ell-1]}, \\ Q'_{A,i} = [\lambda'_{A,i}]G|_{i\in[\ell]}, \\ \mathsf{Com}(\lambda'_{A,i}, r_i) = [\lambda'_{A,i}]P + [r_i]H|_{i\in[\ell]} \end{array} \right\}$$

We can build an argument system for $\mathcal{R}^{\mathsf{bp}}_{\text{upd}}$ based entirely on bulletproofs by diving into per-level subrelations, which could be proven separately in parallel:

$$\mathcal{R}^{(i)}_{\mathsf{ART}} = \left\{ \begin{array}{l} (Q'_{A,i}, Q_{B,i}) \in \mathbb{G}^2, \\ (\mathsf{Com}(\lambda'_{A,i}, r_i), \\ \mathsf{Com}(\lambda'_{A,i+1}, r_{i+1})) \in \mathbb{G}_2^2; \\ (\lambda'_{A,i}, \lambda'_{A,i+1}, r_i, r_{i+1}) \in \mathbb{Z}_q^4 \end{array} \middle| \begin{array}{l} Q'_{A,i} = [\lambda'_{A,i}]G, \\ \lambda'_{A,i+1} = \iota([\lambda'_{A,i}]Q_{B,i}), \\ \mathsf{Com}(\lambda'_{A,i}, r_i) = [\lambda'_{A,i}]P + [r_i]H, \\ \mathsf{Com}(\lambda'_{A,i+1}, r_{i+1}) = [\lambda'_{A,i+1}]P + [r_{i+1}]H \end{array} \right\}$$

To prove the scalar multiplication result, we utilize an optimized technique [Bon+25] that avoids the use of expensive point addition and multiplication. We leave the circuit logic for scalar multiplication in Appendix A. Another idea is to use cross-sigma proof construction, as discussed in [Hru25], but it's another research subject.

4.3 Causal ART

Causal ART is an extension of the classic Asynchronous Ratchet Tree protocol, inspired by the Causal TreeKEM protocol [Wei19; Swi25], which aims to support concurrent updates to the ratchet tree. This enables the eventually consistent merging of multiple concurrent, same-epoch trees into a single one.

Let group members A and B occupy $\mathcal{T}$ leaf nodes with key pairs (λ_A, Q_A), (λ_B, Q_B). Consider the parties perform concurrent tree update operations on the same epoch, broadcasting their updated branches $\vec{Q}'_A$ and $\vec{Q}'_B$. For the ART tree, this means that some nodes become conflicting, having two possible interpretations. The causal ART approach proposes to apply the addition operation to public node keys (Figure 2).

Consider conflicting key pairs $(\lambda'_{A,t}, Q'_{A,t}) \in \mathcal{T}_A, (\lambda'_{B,t}, Q'_{B,t}) \in \mathcal{T}_B$ where $\mathcal{T}_A$ and $\mathcal{T}_B$ are versions of the updated tree. Merging is done by simple addition of updated public keys in conflicting nodes $Q_{M,t} = Q'_{A,t} + Q'_{B,t} = [\lambda'_{A,t} + \lambda'_{B,t}]G$. The merging procedure extended to any number of concurrent ART updates is described in Algorithm 4.3.

Note that the ART invariant is fulfilled: each member could compute every private key along its full path to the root node, including members performing conflicting updates, as each node's secret component could be computed given parallel updates from other members; therefore, their sum forming the node's secret could also be easily evaluated. As users may retain their old private keys for a short period to ensure system integrity, they can compute all possible tree update options. Note that if two users send updates simultaneously, but only one is received by the group, the other update will be discarded if it arrives with a significant delay.

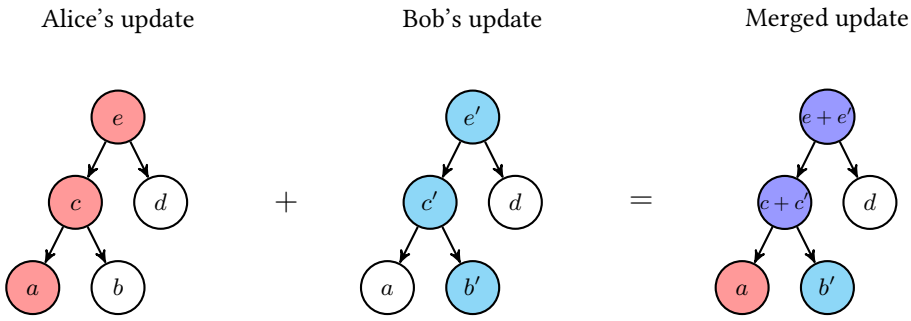


Figure 2: When simultaneous updates occur, the versions can be merged into a single tree, where each conflicting path node is computed as the sum of the corresponding nodes from each version.

Algorithm 1 Merging of conflicting trees' instances

Input: $\mathcal{T}_{\xi,k\in\mathcal{S}}$ // different tree instances for the same epoch ξ
Execute:
$\mathcal{T}_{\xi,f} \leftarrow O$
for $i \in [\ell]$ **do**
  for $j \in [2^{\ell-i-1}]$ **do**
    $t \leftarrow []$
    for $k \in \mathcal{S}$ **do**
      if $\mathcal{T}_{\xi,k}[i][j] \neq$ null **then**
        t.append($\mathcal{T}_{\xi,k}[i][j]$) // collecting node values for the same position from different $\mathcal{T}_{\xi,k}$
      end if
    end for
    if $t \neq []$ **then**
      t.removeDup() // leave only unique values in array
      $\mathcal{T}_{\xi,f}[i][j] = \sum_{s=0}^{|t|-1} t[s]$ // initialize the final $\mathcal{T}$ instance with the sum of unique nodes
    end if
  end for
end for
return $\mathcal{T}_{\xi,f}$

Theorem 4.1 (Merged tree consistency). *Let $\mathcal{P}$ be a set of all group members and $\mathcal{T}_\xi$ be a current group ART. If users A and B with corresponding tree branches $(\vec{\lambda}_A, \vec{Q}_A), (\vec{\lambda}_B, \vec{Q}_B)$ concurrently update the tree to the epoch $\xi + 1$, resulting in two different branch updates: $(\vec{\lambda}'_A, \vec{Q}'_A)$ and $(\vec{\lambda}'_B, \vec{Q}'_B)$, then each user $U \in \mathcal{P}$ can derive the secret root key of $\mathcal{T}_{\xi+1}$ after merging conflicting tree instances.*

Proof. Let $|\vec{Q}_A| = \ell_A, |\vec{Q}_B| = \ell_B$ and c – be an intersection point of vectors $\vec{Q}_A, \vec{Q}_B$: $\forall i \in [c] : Q_{A,\ell_A-c+i} = Q_{B,\ell_B-c+i}$, so the users branches intersect at c nodes. Updating branches have a conflict in each node in this intersection as A and B want to overwrite them with their own public keys. Therefore, during the merge, each party sums the public keys in conflicting nodes: $\forall i \in [c] : Q_{M,i} \leftarrow Q'_{A,\ell_A-c+i} + Q'_{B,\ell_B-c+i}$. Let's show that A and B could compute each secret $\lambda_{M,i}$ corresponding to each

public key $Q_{M,i}$. A computes merged secrets:

$$\lambda_{M_A,i} \leftarrow \iota([\lambda'_{A,\ell_A-c+i-1}]Q_{B,\ell_B-c+i-1}) + \iota([\lambda_{A,\ell_A-c+i-1}]Q'_{B,\ell_B-c+i-1})$$

with corresponding merged public keys:

$$\begin{aligned} Q_{M_A,i} &\leftarrow [\iota([\lambda'_{A,\ell_A-c+i-1}]Q_{B,\ell_B-c+i-1})]G + [\iota([\lambda_{A,\ell_A-c+i-1}]Q'_{B,\ell_B-c+i-1})]G \\ &= Q'_{A,\ell_A-c+i} + Q'_{B,\ell_B-c+i} \end{aligned}$$

B instead computes their merged secrets:

$$\lambda_{M_B,i} \leftarrow \iota([\lambda'_{B,\ell_B-c+i-1}]Q_{A,\ell_A-c+i-1}) + \iota([\lambda_{B,\ell_B-c+i-1}]Q'_{A,\ell_A-c+i-1})$$

with corresponding merged public keys:

$$\begin{aligned} Q_{M_B,i} &\leftarrow [\iota([\lambda'_{B,\ell_B-c+i-1}]Q_{A,\ell_A-c+i-1})]G + [\iota([\lambda_{B,\ell_B-c+i-1}]Q'_{A,\ell_A-c+i-1})]G \\ &= Q'_{B,\ell_B-c+i} + Q'_{A,\ell_A-c+i} \end{aligned}$$

Thus, we see that $\forall i \in [c] : \lambda_{M,i} = \lambda_{M_B,i} = \lambda_{M_A,i}$ as they are the sums of the equal Diffie-Hellman shared secrets and $\forall i \in [c] : Q_{M,i} = Q_{M_B,i} = Q_{M_A,i}$. Finally, the tree key of the merged trees is $tk_{\xi+1} = \lambda_{M_A,c} = \lambda_{M_B,c}$. Furthermore, these secrets $\lambda_{M,i}$ correspond to merged public keys $Q_{M,i}$ that make the merged state $\mathcal{T}_{\xi+1}$ consistent among all group participants (all participants compute the same ART root key). □

Corollary 4.2 (Merging the same branch). *If $c = \ell_A = \ell_B$ (A and B merely update the same branch), the merged tree remains consistent even though the merged leaf secret $\lambda_{M,1} = \lambda'_{A,1} + \lambda'_{B,1}$ is unknown to each party until they collude with one another.*

This implication becomes very useful when we later define *remove confirmation* procedure.

5 Group Management based on the 0-ART Framework

Let $\mathcal{P}$ be a set of all group members, $\mathcal{C} \in \mathcal{P}$ be the group creator, and $\mathcal{A} \subseteq \mathcal{P}$ be a set of administrators. Additionally, we introduce $\mathcal{V}, \mathcal{U} \subseteq \mathcal{I}$ – the subsets of registered and unregistered users in DS which belong to the set of invitees $\mathcal{I}$. To ensure full interaction between group members, we define the main operations with corresponding sub-phases:

(1) InitGroup. $\mathcal{C}$ creates a group based on public keys corresponding to identified members $\mathcal{V}' \subseteq \mathcal{V}$ and self-generated keypairs for unidentified members $\mathcal{U}' \subseteq \mathcal{U}$:

(1.1) $\mathsf{Init} : (d_\mathcal{C}, Q_{\mathsf{ID},\mathcal{C}}), (Q_{\mathsf{ID},i}, Q_{\mathsf{inv},i})|_{i\in\mathcal{V}'\subseteq\mathcal{V}}, (k_{\mathsf{inv}',i}, Q_{\mathsf{inv}',i})|_{i\in\mathcal{U}'\subseteq\mathcal{U}} \to (\mathcal{T}_0, \sigma_\mathcal{T}), (\mathsf{inv}_i, \sigma_i)|_{i\in\mathcal{V}'\cup\mathcal{U}'}$.
Having the list of invitees' key pairs $(Q_{\mathsf{ID},i}, Q_{\mathsf{inv},i})|_{i\in\mathcal{V}'\subseteq\mathcal{V}}(k_{\mathsf{inv}',i}, Q_{\mathsf{inv}',i})|_{i\in\mathcal{U}'\subseteq\mathcal{U}}$ and their own identity key pair $(d_\mathcal{C}, Q_{\mathsf{ID},\mathcal{C}})$, the group creator $\mathcal{C}$ forms invites $(\mathsf{inv}_i, \sigma_i,)|_{i\in\mathcal{V},\mathcal{U}}$ for all group members and the group ART $\mathcal{T}_{\xi=0}$ with the corresponding authenticity proof $\sigma_\mathcal{T}$. See Algorithm 2.

(1.2) $\mathsf{AcceptInvite} : (\mathsf{inv}_A, \sigma_A), (d_A, k_{\mathsf{inv},A}) \to \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A$. After receiving the signed invite $(\mathsf{inv}_A, \sigma_A)$ the user $A \in \mathcal{I}$ can derive the ART tree $\mathcal{T}_\xi$, stage stk_ξ key along with their leaf keypair (λ_A, Q_A). See Algorithm 3.

(1.3) $\mathsf{KeyUpdate} : \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, \lambda'_A, Q'_A \to (\vec{Q}'_A, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$. The user A, who controls the leaf (λ_A, Q_A) in the tree $\mathcal{T}_\xi$ could update their leaf keypair (λ'_A, Q'_A) computing secret nodes through the path to the root. The function results in the updated tree state $\mathcal{T}_{\xi+1}$, stage key $stk_{\xi+1}$, updated tree branch $\vec{Q}'_A$ and a corresponding proof π_{upd} of tree update correctness. See Algorithm 4.

(2) KeyUpdate. A group member A updates their key pair in the tree:

(2.1) $\mathsf{KeyUpdate} : \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, \lambda'_A, Q'_A \to (\vec{Q}'_A, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$. The user A updates their keypair in the group ART, in a way similar to (1.3). See Algorithm 4.

(3) AddMember. The eligible user $A \in \mathcal{A} \cup \{\mathcal{C}\}$ invites a new member $P \in \mathcal{I}$ to the group:

(3.1) $\mathsf{AddMember} : \mathcal{T}_\xi, stk_\xi, (\lambda_A, Q_A), (d_A, Q_{\mathsf{ID},A}), (Q_{\mathsf{ID},P}, Q_{\mathsf{inv},P})$ **or** $k_{\mathsf{inv}',P} \to (\vec{Q}'_P, \pi)$, $(\mathcal{T}_{\xi+1}, stk_{\xi+1}), (\mathsf{inv}_P, \sigma_P^{\mathsf{inv}})$. Eligible entity A invites the new user P to the group using their's identity keypair $(d_A, Q_{\mathsf{ID},A})$, invitee's identity key $Q_{\mathsf{ID},P}$ and invitational key $Q_{\mathsf{inv},P}$ (if the invitee hasn't been registered in DS, the administrator generates $k_{\mathsf{inv}',P}$ and computes corresponding public keys themselves). See Algorithm 5.

(3.2) $\mathsf{AcceptInvite} : (\mathsf{inv}_A, \sigma_A), (d_A, k_{\mathsf{inv},A}) \to \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A$. After receiving the signed invite $(\mathsf{inv}_A, \sigma_A)$ the user $A \in \mathcal{I}$ can derive the ART tree $\mathcal{T}_\xi$, stage stk_ξ key along with their leaf keypair (λ_A, Q_A). See Algorithm 3.

(3.3) $\mathsf{KeyUpdate} : \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, \lambda'_A, Q'_A \to (\vec{Q}'_A, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$. To become a member of the group $\mathcal{P}$ the user A updates their keypair in the group ART $\mathcal{T}_\xi$, in a way similar to (1.3) and (2.1). See Algorithm 4.

(4) RemoveMember. The user $R \in \mathcal{P}$ leaves or is being removed from the group:

(4.0) SelfDestroyInit. The group member R notifies the group about their will to be removed from the group by sending a special signed message to the group. *The operation is optional and does not require any ART changes.*

(4.1) $\mathsf{RemoveMember} : \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, Q_R \to (\vec{Q}'_{R,eph}, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$. The eligible user $A \in \mathcal{P}$ initiates the removal of the user $R \in \mathcal{P}$ from the group by updating the leaf path for the corresponding leaf Q_R and marking R's leaf as *blank*. The algorithm is similar to KeyUpdate() with the exception that A updates the leaf path that they didn't originally control. See Algorithm 6.

(4.2) $\mathsf{ConfirmRemove} : \mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, Q_R \to (\vec{Q}'_R, \pi_{\mathsf{upd}}), (\mathcal{T}'_{\xi+1}, stk_{\xi+1})$. This operation removes the sole ownership of the blank R's leaf by the initiator of the removal operation. Each group member $A \in \mathcal{P}$ can generate a new ephemeral key pair and create their own version of the tree. All versions will be finally merged, presuming the absence of the leaf control by the quorum of $r - 1$ members, where r is the number of users participating in the removal process. See Algorithm 7.

All the above-mentioned operations are designated to propose changes in the group. But there is a single operation that allows each user to apply the proposed changes in their own tree version (leading to an asynchronous update in the entire group):

(5) $\mathsf{ProcessOperation} : \xi_A, \mathcal{T}_A, stk_{\xi_A}, \lambda_A, o_S, \xi_S, (\vec{Q}'_S, \pi) \to \xi_A, \mathcal{T}_A, stk$. The user $A \in \mathcal{P}$ in the epoch ξ_A with ART state $\mathcal{T}_A, stk_{\xi_A}$ receives the operation o_S changing some branch $\vec{Q}'_S$ from the counterparty S and a proof of update correctness π. If the proof is correct and the incoming epoch ξ_S is equal to current A's epoch ξ_A, they execute the merge operation. If the proof is correct and the incoming epoch ξ_S is greater, they basically apply the changes. Otherwise, A rejects the group update. See Algorithm 8.

Additionally, one can easily aggregate several group operations from the set {KeyUpdate, AddMember, ConfirmRemove, RemoveMember} into a single group operation according to the Algorithm 9.

Algorithm 2 $\mathsf{Init}(\cdot)$

// $\mathcal{C}$ initializes a group and creates invites for initial group members
Input:
  $(d_{\mathcal{C}}, Q_{\mathsf{ID},\mathcal{C}})$ // creator's identity keypair
  $(Q_{\mathsf{ID},i}, Q_{\mathsf{inv},i})|_{i \in \mathcal{V}' \subseteq \mathcal{V}}$ // identified invitees
  $(k_{\mathsf{inv}',i}, Q_{\mathsf{inv}',i})|_{i \in \mathcal{U}' \subseteq \mathcal{U}}$ // unidentified invitees
Execute:
$k_{\mathsf{suk}} \xleftarrow{\$} \mathbb{Z}_q, Q_{\mathsf{suk}} \leftarrow [k_{\mathsf{suk}}]G$ // ephemeral setup key pair
$\lambda_{\mathcal{C}} \xleftarrow{\$} \mathbb{Z}_q, \Lambda \leftarrow [\lambda_{\mathcal{C}}]$ // init leaf nodes' array with a creator's secret key
for $i \in \mathcal{V}' \cup \mathcal{U}'$ **do**
  if $i \in \mathcal{V}$ **then**
    $\lambda_i = \mathsf{X3DH}(d_{\mathcal{C}}, k_{\mathsf{suk}}, Q_{\mathsf{ID},i}, Q_{\mathsf{inv},i})$
  else
    $\lambda_i = \mathsf{X3DH}(d_{\mathcal{C}}, k_{\mathsf{suk}}, Q_{\mathsf{inv}',i}, Q_{\mathsf{inv}',i})$
  end if
  $\Lambda.\mathsf{append}(\lambda_i)$ // gen keys for all group mebers
end for
$\mathcal{T}_0 \leftarrow \mathsf{CreateART}(\Lambda)$ // create ART with leaves λ_i
$tk_0 \leftarrow \mathsf{ARTRoot}(\mathcal{T}_0, \lambda_{\mathcal{C}})$
$stk_0 \leftarrow \mathcal{H}_{\mathrm{KDF}}(0 \ldots 0 || tk_0)$ // stage group keys for 0-th epoch
$\xi \leftarrow 0$ // epoch counter
$\sigma_{\mathcal{T}} \leftarrow \mathsf{SigGen}(\mathcal{T}, d_{\mathcal{C}})$
for $i \in \mathcal{V}' \cup \mathcal{U}'$ **do**
  $\mathsf{data} = \{\mathsf{group_id}, \xi, \mathsf{stk}_{\xi}\}$
  $\mathsf{container} \leftarrow \mathsf{Enc}(\mathsf{data}, \lambda_i)$
  if $i \in \mathcal{V}$ **then**
    $\mathsf{inv}_i = (Q_{\mathsf{ID},\mathcal{C}}, Q_{\mathsf{suk}}, Q_{\mathsf{ID},i}, Q_{\mathsf{inv},i}, \mathsf{container})$ // identified invite
  else
    $\mathsf{inv}_i = (Q_{\mathsf{ID},\mathcal{C}}, Q_{\mathsf{suk}}, k_{\mathsf{inv}',i}, \mathsf{container})$ // unidentified invite
  end if
  $\sigma_i \leftarrow \mathsf{SigGen}(\mathsf{inv}_i, d_{\mathcal{C}}))$
end for
return $(\mathcal{T}_0, \sigma_{\mathcal{T}}), (\mathsf{inv}_i, \sigma_i)|_{i \in \mathcal{V}' \cup \mathcal{U}'}$

Algorithm 3 AcceptInvite($\cdot$)

// each $A \in \mathcal{P}$ accepts their invite to enter the group
Input:
  $(\mathsf{inv}, \sigma^{\mathsf{inv}})$ // signed invite
  $(d_A, k_{\mathsf{inv},A})$ // optional if $A \notin \mathcal{V}$
Execute:
if $A \in \mathcal{V}$ **then**
  $Q_{\mathsf{ID},\mathcal{C}}, Q_{\mathsf{suk}}, Q_{\mathsf{ID},A}, Q_{\mathsf{inv},A}, \mathsf{container} \leftarrow \mathsf{Decode}(\mathsf{inv})$ // identified invite
else
  $Q_{\mathsf{ID},\mathcal{C}}, Q_{\mathsf{suk}}, k_{\mathsf{inv},A}, \mathsf{container} \leftarrow \mathsf{Decode}(\mathsf{inv})$ // unidentified invite
end if
if $\mathsf{SigVerify}(\mathsf{inv}, \sigma^{\mathsf{inv}}, Q_{\mathsf{ID},\mathcal{C}}) = 0$ **then**
  abort
end if
$\lambda_A = \mathsf{X3DH}(k_{\mathsf{inv},A}, d_A, Q_{\mathsf{suk}}, Q_{\mathsf{ID},\mathcal{C}})$
$\mathsf{group_id}, \xi, stk_\xi \leftarrow \mathsf{Decode}(\mathsf{Dec}(\mathsf{container}, \lambda_A))$
$Q_A = [\lambda_A]G$ // A computes their public key
$\pi_{\mathsf{PoK}} \leftarrow \Pi_{\mathsf{PoK}}.\mathsf{Prove}(\lambda_A, Q_A)$ // generate non-interactive proof of knowledge (usually with some **SP**-provided challenge included in transcript)
// the following part is performed by **SP** or alternatively, $\mathcal{T}_\xi$ could be transmitted to A in invite inv itself or other trusted channel
if $\Pi_{\mathsf{PoK}}.\mathsf{Verify}(\pi_{\mathsf{PoK}}, Q_A) = 1$ **then**
  SP sends $\mathcal{T}_\xi$ to A
else
  abort
end if
$tk_\xi \leftarrow \mathsf{ARTRoot}(\mathcal{T}_\xi, \lambda_A)$ // compute current ART root private key
$stk_\xi \leftarrow \mathcal{H}_{\mathrm{KDF}}(stk_\xi || tk_\xi)$
return $\mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A$

Algorithm 4 $\mathsf{KeyUpdate}(\cdot)$

// a group member $A \in \mathcal{P}$ rotates their secret key and updates ART state
Input:
  $\mathcal{T}_\xi, stk_\xi$ // current ART state
  λ_A, Q_A // current private and public leaf keys
  $\lambda'_A \xleftarrow{\$} \mathbb{Z}_q, Q'_A = [\lambda'_A]G$ // new private and public keys, could alternatively be $\lambda'_A \leftarrow \mathcal{H}_{\mathsf{KDF}}(r||\lambda_A)$ or $\lambda'_A \leftarrow \mathcal{H}_{\mathsf{KDF}}(stk_\xi||\lambda_A)$
Execute:
$\lambda'_{A,1} \leftarrow \lambda'_A, Q'_{A,1} \leftarrow Q'_A$
$\vec{Q}_B \leftarrow \mathsf{ExtractCoPath}(\mathcal{T}_\xi, \vec{Q}'_A)$ // get co-path to the branch $\vec{Q}'_A$
for $j \in [\ell - 1]$ **do**
  $\lambda'_{A,j+1} \leftarrow \iota([\lambda_{A,j}]\vec{Q}_{B,j})$
  $Q'_{A,j} \leftarrow [\lambda'_{A,j}]G$
end for
$\vec{\lambda}_A \leftarrow (\lambda_{A,j})|_{j\in[\ell]}, \vec{Q}_A \leftarrow (Q_{A,j})|_{j\in[\ell]}$
$\pi_{\mathsf{upd}} \leftarrow \Pi_{\mathsf{upd}}.\mathsf{Prove}(\vec{Q}'_A, \vec{Q}_B, Q_A; \vec{\lambda}'_A, \lambda_A)$
$\mathcal{T}_{\xi+1} \leftarrow \mathsf{ARTUpdate}(\mathcal{T}_\xi, \vec{Q}'_A)$ // update changed branch $\vec{Q}'_A$ in tree $\mathcal{T}_\xi$
$tk_{\xi+1} \leftarrow \lambda'_{A,\ell}$
$stk_{\xi+1} \leftarrow \mathcal{H}_{\mathsf{KDF}}(stk_\xi||tk_{\xi+1})$
return $(\vec{Q}'_A, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$ // the first tuple is sent to the group, the second represents current local A's state

Algorithm 5 AddMember($\cdot$)

// $A \in \mathcal{P}$ adds new member $P \in \mathcal{I}$ to the group
Input:
  $\mathcal{T}_\xi, stk_\xi$ // current state
  (λ_A, Q_A) // inviter's leaf keypair
  $(d_A, Q_{\mathsf{ID},A})$ // inviter's identity keypair
  if $P \in \mathcal{V}$ **then** $(Q_{\mathrm{ID},P}, Q_{\mathsf{inv},P})$ **else** $k_{\mathsf{inv}',P}$, // identified if $P \in \mathcal{V}$ or unidentified invite otherwise
Execute:
if $P \notin \mathcal{V}$ **then**
  $(Q_{\mathrm{ID},P}, Q_{\mathsf{inv},P}) \leftarrow [k_{\mathsf{inv}',P}]G, [k_{\mathsf{inv}',P}]G$
end if
$k_{\mathsf{suk}} \xleftarrow{\$} \mathbb{Z}_q, Q_{\mathsf{suk}} \leftarrow [k_{\mathsf{suk}}]G$ // ephemeral setup key pair
$\lambda_P = \mathsf{X3DH}(d_A, k_{\mathsf{suk}}, Q_{\mathrm{ID},P}, Q_{\mathsf{inv},P})$ // setting up the invitee's leaf key
$Q_P \leftarrow [\lambda_P]G$
$\mathcal{T}_\xi \leftarrow \mathsf{AppendNode}(\mathcal{T}_\xi, Q_P)$ // append new node or take blank occupying it with Q_P
if $A = \mathcal{C}$ **then**
  $(\vec{Q}'_P, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow \mathsf{KeyUpdate}(\mathcal{T}_\xi, \lambda_A, Q_A, \lambda_P, Q_P)$ // if A is a group owner then we just prove their secret's knowledge
else if $A \in \mathcal{A}$ **then**
  $\lambda_\mathcal{T} \leftarrow \mathsf{ARTRoot}(\mathcal{T}_\xi, \lambda_A)$ // get current root key
  $Q_\mathcal{T} \leftarrow [\lambda_\mathcal{T}]G$
  $(\vec{Q}'_P, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow \mathsf{KeyUpdate}(\mathcal{T}_\xi, \lambda_\mathcal{T}, Q_\mathcal{T}, \lambda_P, Q_P)$
  $\pi_{\mathsf{el}} \leftarrow \Pi_{\mathsf{el}}.\mathsf{Prove}(\perp, A)$
  $\pi \leftarrow \pi_{\mathsf{upd}} \cup \pi_{\mathsf{el}}$ // the group creator/administrator must prove their eligibility to make the group update operation
else
  abort
end if
$\mathsf{data} = \{\mathsf{group_id}, \xi + 1, \mathsf{stk}_{\xi+1}\}$
$\mathsf{container} \leftarrow \mathsf{Enc}(\mathsf{data}, \lambda_P)$
if $P \in \mathcal{V}$ **then**
  $\mathsf{inv}_P \leftarrow (Q_{\mathsf{ID},A}, Q_{\mathsf{suk}}, Q_{\mathsf{ID},P}, Q_{\mathsf{inv},P}, \mathsf{container})$ // identified invite
else
  $\mathsf{inv}_P \leftarrow (Q_{\mathsf{ID},A}, Q_{\mathsf{suk}}, k_{\mathsf{inv}',P}, \mathsf{container})$ // unidentified invite
end if
$\sigma_P^{\mathsf{inv}} \leftarrow \mathsf{SigGen}(\mathsf{inv}_P, d_A)$
return $(\vec{Q}'_P, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1}), (\mathsf{inv}_P, \sigma_P^{\mathsf{inv}})$ // the first tuple is sent to the group, the second represents current local A's state, the third(invite) is given to P

Algorithm 6 RemoveMember($\cdot$)

// a member $A \in \mathcal{P}$ initiates an exclusion of $R \in \mathcal{P}$ from the group
Input:
  $\mathcal{T}_\xi, stk_\xi$ // current ART state
  λ_A, Q_A // remover's leaf keypair
  Q_R // removing node's public key
Execute:
$\lambda'_{R,eph} \xleftarrow{\$} \mathbb{Z}_q$ // generate temporary secret which rewrites λ_R node
$Q'_{R,eph} \leftarrow [\lambda_{A,eph}]G$
if $A = \mathcal{C}$ **then**
  $(\vec{Q}'_{R,eph}, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow \mathsf{KeyUpdate}(\mathcal{T}_\xi, \lambda_A, Q_A, \lambda'_{R,eph}, Q'_{R,eph})$ // if A is a group owner then we just prove his secret's knowledge
else if $A \in \mathcal{A}$ **then**
  $\lambda_\mathcal{T} \leftarrow \mathsf{ARTRoot}(\mathcal{T}_\xi, \lambda_A)$ // get current root key
  $Q_\mathcal{T} \leftarrow [\lambda_\mathcal{T}]G$
  $(\vec{Q}'_{R,eph}, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow \mathsf{KeyUpdate}(\mathcal{T}_\xi, \lambda_\mathcal{T}, Q_\mathcal{T}, \lambda'_{R,eph}, Q'_{R,eph})$
  $\pi_{\mathsf{el}} \leftarrow \Pi_{\mathsf{el}}.\mathsf{Prove}(\perp, A)$
  $\pi \leftarrow \pi_{\mathsf{upd}} \cup \pi_{\mathsf{el}}$
else if $\mathsf{IsBlank}(\mathcal{T}_\xi, Q_R) = 1$ **then**
  $\lambda_\mathcal{T} \leftarrow \mathsf{ARTRoot}(\mathcal{T}_\xi, \lambda_A)$ // get current root key
  $Q_\mathcal{T} \leftarrow [\lambda_\mathcal{T}]G$
  $(\vec{Q}'_{R,eph}, \pi_{\mathsf{upd}}), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow \mathsf{KeyUpdate}(\mathcal{T}_\xi, \lambda_\mathcal{T}, Q_\mathcal{T}, \lambda'_{R,eph}, Q'_{R,eph})$
else
  abort
end if
$\mathcal{T}_{\xi+1} \leftarrow \mathsf{MarkAsBlank}(\mathcal{T}_{\xi+1}, Q_R)$ // mark node with public key Q_R as *blank*
return $(\vec{Q}'_{R,eph}, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$

Algorithm 7 ConfirmRemove($\cdot$)

// any member $A \in \mathcal{P}$ confirm exclusion of $R \in \mathcal{P}$ from the group by ART merging operation
Input:
  $\mathcal{T}_\xi, stk_\xi$ // current state
  λ_A, Q_A // remove confirmer's leaf keypair
  Q_R // removing node's public key
Execute:
$\pi_{\mathsf{upd}}, \mathcal{T}_{\xi+1}, \vec{Q}'_R, stk_{\xi+1} \leftarrow \mathsf{RemoveMember}(\mathcal{T}_\xi, stk_\xi, \lambda_A, Q_A, Q_R)$
$\mathcal{T}'_{\xi+1} \leftarrow \mathsf{MergeARTs}(\mathcal{T}_{\xi+1}, \mathcal{T}_\xi)$
$tk_{\xi+1} \leftarrow \mathsf{ARTRoot}(\mathcal{T}'_{\xi+1}, \lambda_A)$
$stk_{\xi+1} = \mathcal{H}_{\mathsf{KDF}}(stk_\xi || tk_{\xi+1})$
return $(\vec{Q}'_R, \pi_{\mathsf{upd}}), (\mathcal{T}'_{\xi+1}, stk_{\xi+1})$

Algorithm 8 ProcessOperation$(\cdot)$

// the user $A \in \mathcal{P}$ processes the group operation from the counterparty S
Input:
  $\xi_A, \mathcal{T}_A, stk_{\xi_A}, \lambda_A$ // current epoch, state of ART, stage key and leaf key of user A
  $o_S, \xi_S, (\vec{Q}'_S, \pi)$ // incoming operation, epoch, branch update and proof from S
Execute:
if $\xi_A > \xi_S$ **then**
  abort // abort if A's current epoch is greater
end if
$Q_S \leftarrow \mathsf{ExtractLeafKey}(\mathcal{T}_A, S)$ // get current couterparty's leaf public key
$\vec{Q}_B \leftarrow \mathsf{ExtractCoPath}(\mathcal{T}_A, \vec{Q}'_S)$ // get current couterparty's co-path
if $o_S \in \{\mathsf{AddMember}, \mathsf{RemoveMember}\}$ **then**
  $\pi = \pi_{\mathsf{upd}} \cup \pi_{\mathsf{el}}$
  if $\Pi_{\mathsf{el}}.\mathsf{Verify}(\pi_{\mathsf{el}}, \perp, S) = 0$ **then**
    abort // check operation eligibility
  end if
else if $o_S \in \{\mathsf{KeyUpdate}, \mathsf{ConfirmRemove}\}$ **then**
  $\pi = \pi_{\mathsf{upd}}$
end if
if $\Pi_{\mathsf{upd}}.\mathsf{Verify}(\pi_{\mathsf{upd}}, \vec{Q}'_S, \vec{Q}_B, Q_S) = 0$ **then**
  abort // check update correctness proof
end if
if $\xi_A - \xi_S = 1$ **then**
  $\mathcal{T}_A \leftarrow \mathsf{ARTUpdate}(\mathcal{T}_A, \vec{Q}'_S)$ // A simply updates current ART state
else if $\xi_A = \xi_S$ **then**
  $\mathcal{T}_S \leftarrow \mathsf{ARTUpdate}(\mathcal{T}_A, \vec{Q}'_S)$
  $\mathcal{T}_A \leftarrow \mathsf{MergeARTs}(\mathcal{T}_A, \mathcal{T}_S)$ // current and incoming epochs are equal so A performs merge
else
  abort
end if
$tk \leftarrow \mathsf{ARTRoot}(\mathcal{T}_A, \lambda_A)$
$stk = \mathcal{H}_{\mathsf{KDF}}(stk_{\xi_A} || tk)$
$\xi_A \leftarrow \xi_S$ // update the epoch counter
return $\xi_A, \mathcal{T}_A, stk$ // state transition of A

Algorithm 9 Aggregate$(\cdot)$

// a member $A \in \mathcal{P}$ performs several ART operations $\vec{o}$ at once
Input:
$\mathcal{T}_\xi, stk_\xi$
$\vec{o} = [o_1, o_2, \ldots, o_u]$, where $o_i \in \{\mathsf{KeyUpdate}, \mathsf{AddMember}, \mathsf{RemoveMember}, \mathsf{ConfirmRemove}\}$
Execute:
$\mathcal{T}_{\mathsf{agg}} \leftarrow \mathsf{CreateTree}()$ create empty tree for aggregated changes
$\mathcal{T}_{\xi+1} \leftarrow \mathcal{T}_\xi$
for $i \in [u]$ **do**
    $(\vec{Q}'_i, \pi_i), (\mathcal{T}_{\xi+1}, stk_{\xi+1}) \leftarrow o_i(\mathcal{T}_{\xi+1}, \cdot)$ // execute each operation propagating current state of the tree $\mathcal{T}_{\xi+1}$ passing o_i's inputs except $\mathcal{T}_{\xi+1}$ as a dot
    $\mathcal{T}_{\mathsf{agg}} \leftarrow \mathsf{AddBranch}(\mathcal{T}_{\mathsf{agg}}, \vec{Q}'_i)$ // add new branch to aggregated tree
end for
$tk_{\xi+1} \leftarrow \mathsf{ARTRoot}(\mathcal{T}_{\xi+1}, \cdot)$ // obtaining current private root key passing o_u operation secret as a dot
$stk_{\xi+1} \leftarrow \mathcal{H}_{\mathsf{KDF}}(stk_\xi || tk_{\xi+1})$
$\pi \leftarrow \bigcup_{i=1}^{u} \pi_i$ // $\bigcup_{i=1}^{u} \pi_i$ – union of operations proofs considering each proof π_i could include $\pi_{\mathsf{upd},i}$, which itself is a union of ART level proofs $\pi_{\mathsf{upd},i} = \bigcup_j \pi^{(j)}_{\mathsf{ART}}$
return $(\mathcal{T}_{\mathsf{agg}}, \pi), (\mathcal{T}_{\xi+1}, stk_{\xi+1})$

6 Anonymous Role Management in groups based on 0-ART

Usually, applications that use ART and 0-ART constructions presume an encryption of the group content with a stage key and its signature by an identity key (to achieve integrity and authenticity control). However, it is not challenging to achieve anonymity in such settings by removing the signature. At the same time, it may be essential to have an "anonymous eligibility" property, which involves proving by the user that they have permission to operate without revealing their identity. This section is dedicated to covering how role management can be organized in solutions based on ART, which provides anonymous eligibility.

6.1 Anonymous Credentials

We define the following minimal set of roles for the 0-ART group:

- **owner** – creator of the group that could endure any operation in the group. By convention, the owner of the group possesses the leftmost 0-ART leaf. The owner could endure any group operation
- **admin** – member of a group that's eligible for AddMembers, RemoveMember operations. Admin must hold a valid anonymous credential that certifies their role
- **member** – regular group member without auxiliary permission, could read and write messages as well as endure KeyUpdate operation

To distinguish the roles of participants in the system, we propose an anonymous credential scheme. It is required to satisfy the following properties:

- The **owner** is only authorized to *issue* and *revoke* credentials
- A credential holder could generate a credential presentation proof, asserting to a verifier the validity of some predicate about credential claims without revealing it fully
- A credential can be revoked, and the verifier can check its status
- A credential should have a limited period of validity
- No third party can link a credential holder to credential presentation proof (unlinkability)

We propose to use the scheme that relies on a Schnorr-based digital signature scheme (for credential issuance) and the above-mentioned Bulletproofs protocol (for proving its state).

6.1.1 General construction

Each credential has the following structure:

$$\texttt{cred} = (\texttt{claims}, (R, s)),$$

where $(R, s) \in \mathbb{G} \times \mathbb{F}_q$ - is a valid Schnorr signature of encoded claims block cast by the credential issuer and claims have the following structure:

$$\texttt{claims} = (Q, \texttt{role}, \texttt{id}, \texttt{exp}),$$

where Q – public key of the credential holder, `role` – the role of the credential holder, `id` – a credential's nonce, `exp` – credential expiration time.

6.1.2 Basic Functions

An anonymous credential system consists of the following algorithms: Issue, Verify, Revoke, Present, VerifyPresentation.

Issue. $\mathcal{C}$ is the group owner and let B be the group member. Suppose that $\mathcal{C}$ wants to give administrator rights to a user B in a group $\mathcal{P}$. To do this, $\mathcal{C}$ generates credential claims:

$\mathcal{C}$ generates valid $\texttt{claims}$ for user B: $\texttt{claims} = (Q \leftarrow Q_{\mathsf{ID},B}, \texttt{id} \xleftarrow{R} \mathbb{F}_q, \texttt{role} \leftarrow \texttt{admin}, \texttt{exp})$. Then, $\mathcal{C}$ computes:

$$r \xleftarrow{R} \mathbb{F}_q, R \leftarrow [r]P,$$
$$c = \mathcal{H}(Q_{\mathsf{ID},\mathcal{C}}, R, \texttt{claims}), \quad s = r + d_{\mathcal{C}} \cdot c,$$
$$\texttt{cred} = (\texttt{claims}, (R, s)).$$

After that $\mathcal{C}$ could send cred to B privately (e.g., through a p2p channel or another trusted channel).

It's also supposed that credentials need to be updated regularly; to update credentials $\mathcal{C}$, Issue is run repeatedly.

Verify. A credential holder can verify a held credential by simply verifying the Schnorr digital signature $R \stackrel{?}{=} [s]P - [c]Q_{\mathsf{ID},\mathcal{C}}$ and asserting the validity of each claim according to the policy.

Revoke. To revoke a credential, the issuer must include its claims.id into the revocation list $\mathcal{B}$. The revocation list must also be signed by the issuer and include expiration info. It must also be updated regularly in accordance with a policy. We can additionally create a revocation list as described in [Kur+24], allowing users to make a lightweight non-inclusion proof.

Present. Suppose user B is an admin of a group $\mathcal{P}$ and holds a credential $\texttt{cred} = ((Q, \texttt{role}, \texttt{id}, \texttt{exp}), (R, s))$ issued by the group owner $\mathcal{C}$. To prove eligibility to perform privileged actions in a system, user B generates a credential presentation proof π_{Cred} for the following relation:

$$\mathcal{R}_{\mathsf{Cred}} = \left\{ \begin{array}{l|l} & Q = [k]P, \\ (Q_{\mathsf{ID},A}, \texttt{time.now}, \mathcal{B}); & \texttt{role} = \texttt{admin}, \\ (k, Q_{\mathsf{ID},B}, \texttt{role}, \texttt{id}, \texttt{exp}, R, s) & \texttt{id} \notin \mathcal{B}, \\ & \texttt{time.now} < \texttt{exp}, \\ & c = \mathcal{H}(Q_{\mathsf{ID},A}, R, Q, \texttt{role}, \mathsf{id}, \mathsf{exp}), \\ & R = [s]P - [c]Q_{\mathsf{ID},A} \end{array} \right\}$$

where $\mathcal{B}$ – set of identifiers of revoked credentials, $\texttt{time.now}$ – present moment of time.

Informally, the prover must prove the validity of the signature, the possession of the certified key, the validity of the credential at the present moment in time, and that the credential identifier is not a member of a set of revoked credentials.

7 Security

We divide this section into two parts. In the first part, we define a malicious insider and construct an attack, showing that the ART and MLS protocols are vulnerable in this setting. In the second part, we demonstrate that Zero-ART can satisfy the security requirements in this case.

7.1 Malicious-insider attack

7.1.1 Definition

Since group users are not explicitly identified, no one in the group can fully trust another member of the system. If a member of the system is compromised, the protocol must be resilient to attacks on the group that can be executed through otherwise legitimate group operations.

We define a **malicious insider** as a user who is a legitimate member of the group but behaves adversarially toward it. We model the **malicious-insider attack** as the system's vulnerability to internal path node forgery for a specific set of group members, causing them to desynchronize according to other group members by deriving different views of the tree root.

7.1.2 Attack scenario

To simulate this scenario, consider a group of n users represented by an ART tree. Each user occupies a single leaf and controls the nodes along its leaf-to-root path in the tree. Let U_i be a malicious insider, $i \in [n]$. For simplicity, we relax conditions on the KeyUpdate function and allow the user to update the tree at arbitrary times.

A malicious insider is allowed to choose the time to update the group state freely, and thus the time of the attack. When favorable conditions arise, the user U_i initiates the KeyUpdate.

We say that user U_i performs a forgery of node $U_{i,j}$ – the parent of sibling nodes $U_{i,j-1}, U_{k,j-1}$ – if the secret of this node is computed as

$$\lambda_{i,j} = \iota([\lambda_{i,j-1} \cdot \widetilde{\lambda}_{k,j-1}]G),$$

where $\widetilde{\lambda}_{k,j-1}$ is randomly generated by U_i. This manipulation prevents all users whose path to the root contains node $U_{k,j-1}$ from computing the new value of the tree root.

The main issue is that as long as any user of the set who holds the secret of node $U_{k,j-1}$ does not synchronize with the group, the group will continue to appear correct. If the adversary influences such users' network access, they can control the time during which users are unable to read group messages to which they have rights.

Recovery after an attack is possible. User U_k, who was affected by the malicious insider attack, still possesses the secret value of their leaf. To restore the ability to calculate the root of the tree, the user should initiate the update phase. However, it should be noted that they cannot calculate the path to the root, and therefore, all proof variants that include this condition will be invalid.

7.1.3 Terms of success

Let us describe in more detail the conditions under which a malicious insider U_i may succeed in an attack. U_i is primarily interested in preventing the affected users from accessing the tree root as long as possible.

It should be noted that forging a node whose secret is shared by a large number of users will likely result in rapid recovery, since the probability that at least one of these users will resynchronize with the group and detect the forgery is high. Consequently, the lower (i.e., closer to the leaves) the ancestor of leaf U_i is forged, the fewer participants can detect the attack. In terms of subtrees, this can be rephrased as follows: the smaller the subtree chosen for root forgery, the fewer users will be able to identify the malicious modification.

In practice, this means that the attacker will mainly affect users located in the same subtree as themselves. Thus, while these conditions do not reliably allow the attacker to compromise specific users in some cases, they are sufficient to sabotage the group by affecting a particular subset of participants.

7.2 0-ART security model

First, we prove that 0-ART satisfies basic secure definitions of FS and PCS. The core idea of these proofs comes from the work [CCZ21], which shows how the probability of an attack on FS and PCS for ART-based protocols can be reduced to an IND-CCA attack. Since 0-ART preserves the fundamental ART structure, these theorems can also be applied to our system.

Then, we prove that our system is resistant to malicious insider attacks. In particular, we show that this proof can be reduced to one of the fundamental properties in the proving system.

Suppose that we have a 0-ART $\mathcal{T}_\xi$, where ξ is the current stage epoch.

7.2.1 Forward Secrecy

Theorem 7.1. *Let the key derivation function $\mathcal{H}_{\mathsf{KDF}}$ be modeled as a random oracle, $\mathcal{A}$ be a PPT adversary. If the keys of epoch $\xi + 1$ are leaked and adversary $\mathcal{A}$ can break FS of 0-ART, there exists adversary $\mathcal{B}_{\mathsf{CCA}}$, an elementary wrapper around $\mathcal{A}$, that can break the IND-CCA of epoch ξ with advantage:*

$$\mathsf{Adv}^{\mathsf{RO}}_{\mathsf{FS}}[\mathcal{A}, 0\text{-}ART] \leq Q \cdot \mathsf{Adv}^{\mathsf{RO}}_{\mathsf{CCA}}[\mathcal{A}, 0\text{-}ART],$$

where Q is the number of epochs.

Proof. The main idea is to show that we can reduce this advantage to the IND-CCA attack probability. Recall that in each epoch, users derive a stage key stk_ξ for encryption from the current root key tk_ξ and the previous stage key $stk_{\xi-1}$. We distinguish two cases:

- The adversary $\mathcal{A}$ compromises all current system keys except leaf keys. To derive the encryption key of a previous epoch $\xi - 1$, $\mathcal{A}$ needs to know the root of the corresponding tree $\mathcal{T}_{\xi-1}$. Since the root depends on the leaf keys, the adversary cannot compute the session key without them. Hence, in this case, each epoch can be reduced to an IND-CCA security.
- The adversary compromises all current system keys, including at least one leaf key. In this case, the adversary can compute the roots of previous trees up to the epoch where the compromised leaf has not yet been updated. However, $\mathcal{A}$ must obtain the stage keys of previous epochs for deriving session keys. Since each stage key recursively depends on all earlier stage keys, the problem is eventually reduced to breaking the initial stage key, which is derived from the secure AKE protocol. Thus, $\mathcal{A}$ cannot compromise the previous keys in this way.

By allowing $\mathcal{A}$ to query each session key, root key, and plaintext encryption from the IND-CCA challenger, this game can be reduced to GameCCA for each epoch i:

$$\mathsf{Adv}^{\mathsf{RO}}_{\mathsf{FS}}[\mathcal{B}_{\mathsf{FS},i}, s_i] \leq \mathsf{Adv}^{\mathsf{RO}}_{\mathsf{CCA}}[\mathcal{A}, 0\text{-}ART],$$

Since this game can be repeated Q times, the advantage scales accordingly, and thus the theorem holds. □

7.2.2 Post-Compromised Secrecy

Theorem 7.2. *Let the key derivation function $\mathcal{H}_{\mathsf{KDF}}$ be modeled as a random oracle, $\mathcal{A}$ be a PPT adversary. If the keys of epoch $\xi+1$ are leaked and in the challenge epoch all leaf keys are updated, the advantage of adversary $\mathcal{A}$ to break PCS is equal to the advantage of $\mathcal{A}$ to break the IND-CCA of epoch $\xi+1$ such that:*

$$\mathsf{Adv}^{\mathsf{RO}}_{\mathsf{PCS}}[\mathcal{A}, 0\text{-}ART] = \mathsf{Adv}^{\mathsf{RO}}_{\mathsf{CCA},\xi+1}[\mathcal{A}, 0\text{-}ART]$$

Proof. If $\mathcal{A}$ does not know the current stage key stk_ξ, it cannot derive the subsequent stage key $stk_{\xi+1}$. In this case, the attack reduces to an IND-CCA attack in epoch $\xi+1$. Conversely, if $\mathcal{A}$ has compromised the current stage key stk_ξ and possesses leaf keys, the success probability of an attack remains 1 until those keys are refreshed. Once each leaf key is updated, the attack again reduces to an IND-CCA attack, making the advantage negligible. □

7.2.3 Operations Verifiability

Suppose that $\mathcal{A}$ is a malicious insider aiming to compromise the access of certain users by performing the attack described in subsection 7.1.

In 0-ART, the KeyUpdate operation must generate a proof π_{upd} proving that the key update originates from the owner of the corresponding leaf, and that the path from the leaf to the root has been recomputed correctly. Thus, to perform a malicious insider attack, the adversary must produce a forged proof π^*_{upd} corresponding to a corrupted update that still verifies as valid.

We can reduce this to the case of a corrupted prover P^* violating the soundness property. A similar approach is described in the work [DDF21]. To make this precise, we give an attack Game 7.1.

Attack Game 7.1. ***Operations Verifiability attack***

- *The challenger $\mathcal{C}$ initiates the group, $\mathcal{T}_\xi, tk_\xi, stk_\xi, (\mathsf{inv}_\mathcal{A}, \sigma_\mathcal{A}) \leftarrow \mathsf{InitGroup}(1^\lambda)$, and sends $\mathcal{T}_\xi$ to $\mathcal{A}$ along with unidentified invite $(\mathsf{inv}_\mathcal{A}, \sigma_\mathcal{A})$.*
- *The adversary $\mathcal{A}$ accepts the invite, derives the leaf key pair $(\lambda_\mathcal{A}, Q_\mathcal{A})$ and initiates a forged $\mathsf{KeyUpdate}^*$: firstly, it selects $m \in [\ell]$ corresponding to the node index which it wants to corrupt. Then it computes:*

$$\forall i \in [\ell] \setminus \{m\} : \lambda^*_{\mathcal{A},i} \leftarrow \iota([\lambda^*_{\mathcal{A},i-1}]Q_{B,i-1}), \quad Q^*_{\mathcal{A},i} \leftarrow [\lambda^*_{\mathcal{A},i}]G$$
$$\widetilde{\lambda}_{k,m-1} \xleftarrow{R} \mathcal{A}, \quad \lambda^*_{\mathcal{A},m} \leftarrow \iota([\lambda^*_{\mathcal{A},m-1} \cdot \widetilde{\lambda}_{k,m-1}]Q_{B,m-1}), \quad Q^*_{\mathcal{A},m} \leftarrow [\lambda^*_{\mathcal{A},m}]G$$

  As a result, $\mathcal{A}$ outputs the forged key update $(\vec{Q}^*_\mathcal{A}, \pi^*_{\mathsf{upd}})$:

$$\vec{Q}^*_\mathcal{A} \leftarrow (Q^*_{\mathcal{A},i})|_{i\in[\ell]}$$
$$\pi^*_{\mathsf{upd}} \leftarrow \Pi_{\mathsf{upd}}.\mathsf{Prove}(\vec{Q}^*_\mathcal{A}, \vec{Q}_B, Q_\mathcal{A}; \vec{\lambda}^*_\mathcal{A}, \lambda_\mathcal{A})$$

- *The challenger plays a verifier role. We define the advantage of adversary $\mathcal{A}$ in the operations-verifiability attack, denoted $\mathsf{Adv}_{\mathsf{OV}}[\mathcal{A}, 0\text{-}ART]$, as the probability that*

$$\Pr(\Pi_{\mathsf{upd}}.\mathsf{Verify}(\pi^*_{\mathsf{upd}}, \vec{Q}^*_\mathcal{A}, \vec{Q}_B, Q_\mathcal{A}) = 1.$$

Thus, we could define $\mathsf{Adv}_{\mathsf{OV}}[\mathcal{A}, 0\text{-ART}]$ as:

$$\mathsf{Adv}_{\mathsf{OV}}[\mathcal{A}, 0\text{-ART}] = \Pr\left[\Pi_{\mathsf{upd}}.\mathsf{Verify}(\pi^*, \vec{Q}^*_{\mathcal{A}}, \vec{Q}_B, Q_{\mathcal{A}}) = 1 \,\middle|\, \begin{array}{c} (\mathsf{inv}_{\mathcal{A}}, \sigma_{\mathcal{A}}) \leftarrow \mathcal{C}.\mathsf{InitGroup}(1^\lambda) \\ \mathcal{T}_\xi, st_\xi, \lambda_{\mathcal{A}}, Q_{\mathcal{A}} \leftarrow \mathcal{A}.\mathsf{AcceptInvite}(\mathsf{inv}_{\mathcal{A}}, \sigma_{\mathcal{A}}) \\ (\vec{Q}^*_{\mathcal{A}}, \pi^*) \leftarrow \mathcal{A}.\mathsf{KeyUpdate}^*(\mathcal{T}_\xi, \lambda_{\mathcal{A}}, Q_{\mathcal{A}}) \end{array}\right]$$

Definition 7.3 (security against malicious insiders)**.** *0-ART* is *secure against malicious insider attack* if, for all efficient adversaries $\mathcal{A}$, the value $\mathsf{Adv}_{\mathsf{OV}}$ is negligible.

Theorem 7.4. *If* Π_{upd} *has computational soundness, then 0-ART is secure against malicious insider attack for all efficient adversaries* $\mathcal{A}$

Proof. By the construction of *0-ART* protocol Π_{upd} for relation $\mathcal{R}_{\mathsf{upd}}$ is a zk-NARK as it is built upon *bulletproofs*. Thus, it has *computational soundness* under *the discrete logarithm assumption*. Therefore $\mathsf{Adv}_{\mathsf{OV}}(\mathcal{A}, 0\text{-ART}) < \nu(\lambda)$ for some negligible function $\nu(\cdot)$. Hence, *0-ART* is *secure against malicious insider attack*.

□

8 Implementation and Benchmarks

We have built an implementation[Dis] of the **0-ART** protocol for group management, along with a demo of the private collaborative workspace **Veil**, built with the CRDT library **automerge**[Swi]. In Section 4, we intentionally omitted many implementation details, such as sending messages to a group in an anonymous yet verifiable manner and authentication for obtaining a list of group messages. This can be easily resolved by signing a message with the ART root key tk, which does not disclose which member sent the message or the authentication request.

It's also important to stress that each group operation proof should comprise a hash of the context in which this operation is performed at the beginning of the Fiat-Shamir transform(e.g., tree updates, metadata, headers, other user-provided data, etc.). So that a proof not only asserts the correctness of some group operation but also validates the context in which it is performed, just like a digital signature ensures integrity and authenticity.

8.1 Benchmarks

Here, we present benchmarks of different group operations(KeyUpdate, AddMember, RemoveMember plus anonynous credentials operations) in our implementation (all benchmarks are run on *Apple M2 Pro*):

Group size	Apply KeyUpdate	Apply AddMember	Apply RemoveMember
2^4	0.52s	0.52s	0.52s
2^{10}	0.72s	0.79s	0.72s
2^{14}	0.97s	1.04s	0.97s
2^{16}	1.15s	1.23s	1.15s
2^{18}	1.23s	1.32s	1.23s
2^{20}	1.36s	1.57s	1.36s
Group size	Process KeyUpdate	Process AddMember	Process RemoveMember
2^4	93.24ms	93.1ms	93.02ms
2^{10}	110.41ms	119.53ms	111.66ms
2^{14}	139.22ms	143.68ms	138.62ms
2^{16}	158.13ms	164.74ms	157.78ms
2^{18}	186.33ms	191.94ms	185.5ms
2^{20}	279.78ms	284.18ms	271.54ms

Table 3: Average time of performing an operation and average time spent on processing it by the other member of the group.

Credential operation	Average time
Issue a credential	2.08ms
Verify the credential	2.05ms
Create a credential presentation proof	1.12s
Verify the credential presentation proof	339.43ms

Table 4: Average time of credentials operations.

Group size	Prove KeyUpdate	Verify KeyUpdate	Proof size
2^4	0.53s	93.87ms	4740b
2^{10}	0.76s	114.2ms	11850b
2^{14}	0.97s	135.21ms	17306b
2^{16}	1.1s	146.40ms	19756b
2^{18}	1.26s	158.44ms	22206b
2^{20}	1.39s	171.30ms	24656b

Table 5: Benchmarks of proving system for $\mathcal{R}_{\mathsf{upd}}$.

9 Conclusions

In this paper, we introduced 0-ART, a group key exchange protocol that extends ART with verifiable operations. We formally defined and described its main functions together with their proof relations. Furthermore, we introduced a new scheme, Causal ART, designed to resolve conflicts arising from concurrent updates, thereby making the protocol more practical. Our scheme also improves participant removal by

allowing the deleted user's path not to depend on the one who removed them. In addition, we proposed an anonymous credential mechanism that enables role assignment within groups and grants rights to perform specific operations. We also provide and prove basic security features: forward secrecy, post-compromise secrecy, and verifiability of operations.

The 0-ART protocol supports interaction among a large number of participants while remaining efficient, since the proofs are compact and fast to generate and verify. The reliance on the service provider is minimal for transport, which makes the protocol well-suited for decentralized configurations.

Looking ahead, we identify several promising directions for future work. A key challenge is to extend support for fully decentralized systems, as decentralized systems may naturally form subgroups with localized interactions. It is therefore crucial to ensure the correct handling of such states.

References

[DH76] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22 (6 1976). ISSN: 15579654. DOI: 10.1109/TIT.1976.1055638.

[KE10] Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://www.rfc-editor.org/info/rfc5869.

[MP16] Moxie Marlinspike and Trevor Perrin. "The Double Ratchet Algorithm". In: *Signal* (2016).

[PM16] Trevor Perrin and Moxie Marlinspike. *The Double Ratchet Algorithm*. https://signal.org/docs/specifications/doubleratchet/. Revision 1, 2016-11-20. 2016. URL: https://signal.org/docs/specifications/doubleratchet/.

[BBR18] 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.

[Bha+18] Karthikeyan Bhargavan et al. "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups". In: *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 1–15. DOI: 10.1109/EuroSP.2018.00009. URL: https://ieeexplore.ieee.org/document/8406611.

[Bun+18] Benedikt Bunz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *Proceedings - IEEE Symposium on Security and Privacy*. Vol. 2018-May. 2018. DOI: 10.1109/SP.2018.00020.

[Coh+18] Katriel Cohn-Gordon et al. "On ends-to-ends encryption asynchronous group messaging with strong security guarantees". In: *Proceedings of the ACM Conference on Computer and Communications Security*. 2018. DOI: 10.1145/3243734.3243747.

[Mat19] Matrix.org. *Olm Documentation*. https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/olm.md. 2019.

[Wei19] Matthew Weidner. "Group Messaging for Secure Asynchronous Collaboration". https://mattweidner.com/assets/pdf/acs-dissertation.pdf. MPhil dissertation. Cambridge, UK: University of Cambridge, 2019.

[Ink20] Ink&Switch. *KeyHive: End-to-End Encrypted Group Messaging for Secure Collaboration*. https://www.inkandswitch.com/keyhive/notebook/02/, Accessed: 2025-09-17. 2020.

[CCZ21] Kaiming Chen, Jiageng Chen, and Jixin Zhang. "Anonymous Asynchronous Ratchet Tree Protocol for Group Messaging". In: *Sensors* 21 (Feb. 2021), p. 1058. DOI: 10.3390/s21041058.

[DDF21] Julien Devigne, Céline Duguey, and Pierre Alain Fouque. "MLS Group Messaging: How Zero-Knowledge Can Secure Updates". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 12973 LNCS. 2021. DOI: 10.1007/978-3-030-88428-4_29.

[Kle+21] Karen Klein et al. "Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 268–284. DOI: 10.1109/SP40001.2021.00035.

[Mat22] Matrix.org. *Megolm Documentation*. https://gitlab.matrix.org/matrix-org/olm/-/blob/master/docs/megolm.md. 2022.

[BCG23] David Balbás, Daniel Collins, and Phillip Gajland. *Analysis and Improvements of the Sender Keys Protocol for Group Messaging*. 2023. arXiv: 2301.07045 [cs.CR]. URL: https://arxiv.org/abs/2301.07045.

[Bar+23] Richard Barnes et al. *The Messaging Layer Security (MLS) Protocol*. RFC 9420. July 2023. DOI: 10.17487/RFC9420. URL: https://www.rfc-editor.org/info/rfc9420.

[PPD23] Gerhart Paul, Rösler Paul, and Schröder Dominique. *Security of Ibex*. Security analysis. https://threema.com/press-files/2_documentation/security_analysis_ibex_2023.pdf. Threema, July 2023.

[Kur+24] Oleksandr Kurbatov et al. *IMOK: A compact connector for non-prohibition proofs to privacy-preserving applications*. Cryptology ePrint Archive, Paper 2024/1868. 2024. URL: https://eprint.iacr.org/2024/1868.

[Bon+25] Dan Boneh et al. "Exponent-VRFs and Their Applications". In: *Advances in Cryptology – EUROCRYPT 2025*. Ed. by Serge Fehr and Pierre-Alain Fouque. Cham: Springer Nature Switzerland, 2025, pp. 195–224. ISBN: 978-3-031-91098-2.

[Hru25] Yevhen Hrubiian. *Cross-group Sigma-protocols*. https://github.com/juja256/zkp/blob/main/sigma_cross.pdf. 2025.

[Nex25] NextGraph. *NextGraph — Introduction*. 2025. URL: https://docs.nextgraph.org/en/introduction/ (visited on 09/25/2025).

[Swi25] Ink & Switch. *Group Key Agreement with BeeKEM*. https://www.inkandswitch.com/keyhive/notebook/02/. 2025.

[Wir25] Wire. *Announcing Wire MLS General Availability*. 2025. URL: https://wire.com/en/blog/wire-mls-is-now-generally-available.

[Xia+25] Meihua Xiao et al. "Formal Analysis of Ratchet Protocols Based on Logic of Events". In: *Applied Sciences* 15.13 (2025). ISSN: 2076-3417. DOI: 10.3390/app15136964. URL: https://www.mdpi.com/2076-3417/15/13/6964.

[Dis] DistributedLab. *0-ART implementation*. https://github.com/zero-art-rs. Accessed: 2025-09-30.

[Swi] Ink & Switch. *automerge CRDT library*. https://automerge.org/, Accessed: 2025-09-30.

A Circuit logic for proving scalar multiplication

Here we describe circuit logic for proving scalar multiplication $R = [\lambda_{\mathcal{A}}]Q_{\mathcal{B}}$ where $R = (x_R, y_R)$ could be provided either as committed secret inputs or public inputs:

1. Define high-level witness variables $\mathbf{k} \leftarrow \lambda_{\mathcal{A}}, \mathbf{s} \leftarrow x_R, \mathbf{r} \leftarrow y_R$
2. Define low-level witness variables $\mathbf{a}_i, \mathbf{b}_i$ from bit-decomposition of

$$\lambda_{\mathcal{A}} = \sum_{i=0}^{n-1} k_i 2^i, \mathbf{a}_i \leftarrow k_i, \mathbf{b}_i \leftarrow (1 - k_i)$$

3. For each bit i add quadratic constraint $\mathbf{a}_i \cdot \mathbf{b}_i = 0$ and linear constraint: $\mathbf{b}_i = 1 - \mathbf{a}_i$
4. Add linear constraint $\sum_{i=0}^{n-1} \mathbf{a}_i 2^i = \mathbf{k}$
5. Define sequence $\{c_i\}_{i=0}^{n-1} \subset \mathbb{N}^n$ such that $\sum_{i=0}^{n-1} c_i = 0$, for example

$$c_0 = 2, \forall i \in \{1, \ldots, n-2\} : c_i = c_{i-1} + 1, c_{n-1} = -\frac{n^2 + n - 2}{2}$$

6. Let $G \in \mathbb{G}$, Define sequence $\{\Delta_i\}_{i=0}^{n-1}$ as $\Delta_i = [k_i]([2^i]Q_{\mathcal{B}}) + [c_i]G$ so that each $\Delta_i$takes value $\Delta_i^{(0)} = [c_i]G$ or $\Delta_i^{(1)} = [2^i]Q_{\mathcal{B}} + [c_i]G$.
7. Denote partial sum $P_j = \sum_{i=0}^{j} \Delta_i$ so that $P_i = P_{i-1} + \Delta_i$ and $P_{n-1} = \sum_{i=0}^{n-1} \Delta_i = [\lambda_{\mathcal{A}}]Q_{\mathcal{B}}$.
8. Add low-level variables for P_0:

$$\mathbf{x}_0 \leftarrow \mathbf{a}_0 \cdot (x_{\Delta_0^{(1)}} - x_{\Delta_0^{(0)}}) + x_{\Delta_0^{(0)}}, \mathbf{y}_0 \leftarrow \mathbf{a}_0 \cdot (y_{\Delta_0^{(1)}} - y_{\Delta_0^{(0)}}) + y_{\Delta_0^{(0)}}$$

9. For each $i = 1..n-1$:
   - add low-level variables for P_i with following assignments:

$$\mathbf{x}_i \leftarrow x_{P_{i-1}+\Delta_i}, \mathbf{y} \leftarrow y_{P_{i-1}+\Delta_i}$$

   - add low-level variables with 3 quadratic constraints:

$$\mathbf{x}_i^2 \leftarrow \mathbf{x}_i \cdot \mathbf{x}_i, \mathbf{x}_i^3 \leftarrow \mathbf{x}_i^2 \cdot \mathbf{x}_i, \mathbf{y}_i^2 \leftarrow \mathbf{y}_i \cdot \mathbf{y}_i$$

   - check $P_i \in E(\mathbb{F}_q)$ via linear constraint: $\mathbf{y}_i^2 = \mathbf{x}_i^3 + a\mathbf{x}_i + b$
   - define $\mathbf{x}_{\Delta_i}, \mathbf{y}_{\Delta_i}$ as linear combinations of $\mathbf{a}_i$:

$$\mathbf{x}_{\Delta_i} \leftarrow \mathbf{a}_i \cdot (x_{\Delta_i^{(1)}} - x_{\Delta_i^{(0)}}) + x_{\Delta_i^{(0)}}, \mathbf{y}_{\Delta_i} \leftarrow \mathbf{a}_i \cdot (y_{\Delta_i^{(1)}} - y_{\Delta_i^{(0)}}) + y_{\Delta_i^{(0)}}$$

   - check that $P_i = P_{i-1} + \Delta_i$ so that points $-P_i, P_{i-1}, \Delta_i$ are co-linear by adding constraints(2 quadratic and 1 linear):

$$\mathbf{t}_1 \leftarrow (\mathbf{y}_{i-1} + \mathbf{y}_i) \cdot (\mathbf{x}_{\Delta_i} - \mathbf{x}_i), \mathbf{t}_2 \leftarrow (\mathbf{y}_{\Delta_i} + \mathbf{y}_i) \cdot (\mathbf{x}_{i-1} - \mathbf{x}_i), \quad \mathbf{t}_1 = \mathbf{t}_2$$

10. Add final linear constraints: $\mathbf{x}_{n-1} = \mathbf{s}, \mathbf{y}_{n-1} = \mathbf{r}$

B ART Multi-Device

Multi-device support is one of the most important features of modern messengers. Regardless of device used to access the network, the user should be able to participate in their groups.

In most private messengers, each device of a user has its own key pair and essentially acts as an additional group member, though associated with a single identity. Within the 0-ART concept, an increase in the number of participants leads to an increase in the tree. Moreover, allowing multiple leaves for a single user could enable potential manipulation in the future. However, it is critical to note that each device is subject to the rule: "Each user claims only their own history". Therefore, 0-ART must retain the same concept that each device claims only its own history.

B.0.1 Main concept

We propose a completely different approach - **user anchor leaf** - a shared secret that the user shares between all their devices. In this model, all devices of a user form a certain group, which can be constructed using 0-ART. From this point on, we refer to this structure as the *device ART*. This allows users to freely and seamlessly operate with devices, which will only affect the users' leaf secrets and chat keys in the group. In this way, the group tree does not store additional leaves for a user's other devices, or even references to them.

Importantly, each user anchor leaf is unique per group, as the user's leaf differs across groups. Thus, for each group, the user has to create a separate device ART. This may slightly complicate client-side logic for users who participate in many groups. However, it is possible to use a single device ART for all groups, essentially putting an array of keys into appropriate nodes. But for a general case, we will denote in the design as separate trees for each group.

B.0.2 Construction

Suppose that Alice was being invited to the group $\mathcal{P}$ and she has the one device D_1. When she has prepared the second phase of AddMember/InitGroup, she generates a random value $r \xleftarrow{R} \mathbb{Z}_q^*$ and updates the leaf secret as:

$$\lambda_A := \mathsf{HKDF}(r, \lambda_A), \quad Q_A = [\lambda_A]P.$$

If Alice has one device, her device leaf for a group $\mathcal{P}$ is the same as the anchor leaf :

$$\lambda_{D_1} \leftarrow \lambda_A, \quad Q_{D_1} \leftarrow Q_A.$$

Then, interaction and updates are similar to the standard mode.

However, Alice decides to communicate from another device D_2. In this case, D_1 and D_2 establish a shared secret for each group (in a particular case for $\mathcal{T}$). For this purpose, she creates a device ART $\tilde{\mathcal{T}}$ and generates an invitation for D_2. The anchor leaf becomes a new leaf in the group tree $\mathcal{T}$, so Alice performs a KeyUpdate operation to assign her new secrets. The invitation and other parameters are sent exclusively via QR live or a link that can be transmitted via a secure, encrypted channel. In this way, the device ART looks like a binary tree of height 2 (Figure 3).

In the anchor leaf, λ_A serves as the secret value of the leaf within the tree of group $\mathcal{P}$.

Consequently, any device can perform a KeyUpdate, since each of them has equal control over the anchor leaf.

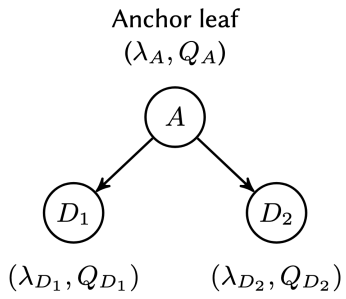


Figure 3: Alice's device tree, where she has two devices: D_1 and D_2.

Additionally, due to the exchanged invitation, devices can establish a private p2p chat, allowing them to exchange messages and synchronize not only locally. Therefore, if Alice joins another group (now is a member of group $\mathcal{P}_1$ and $\mathcal{P}_2$) from the device D_2, it sends the new device tree $\widetilde{\mathcal{T}_2}$ to the D_1 over their private chat.

If Alice wants to add another device, in this case, a device that will generate invitations, i.e., D_1, adds a new leaf to all device trees and completes the KeyUpdate for each group. After that, it sends updated trees to the other Alice devices over their each own private chat.

If Alice wants to delete one of her devices, she can initiate the removal either from the device itself or from one of her other devices. The procedure is similar to a standard removal operation in a group, except that the interaction with each other device from the device tree takes place via private chats. However, to ensure security, if the removal is initiated from the device that is to be deleted, the removal must be finished by another of Alice's devices. This device should blind the leaves corresponding to the removed device in the device trees of all groups and perform a KeyUpdate for each group with the corresponding new anchor leaf.