Cardano Privacy Layer: Zero-Knowledge Proof-Based Membership Verification and Anonymous Voting & Signaling PoC.

Modulo-p

September 2024

Contents

Chapter 1

Introduction and protocol overview.

1.1 Introduction

This document is part of our Fund 11 Catalyst proposal named *Cardano Privacy Layer: Zero-Knowledge Proof-Based Membership Verification and Anonymous Voting & Signaling.* The purpose of this project is to improve the privacy capabilities of the Cardano blockchain. To achieve this, we aim to port a widely-known privacy protocol called Semaphore developed by the pse.dev group, which leverages Zero-Knowledge technology to embed privacy features directly into the Ethereum L1. It has proven to have a multitude of use-cases. Among its many applications worth mentioning are anonymous voting, coin-mixing, and private authentication. The protocol's customizability and versatility make it an appealing choice for integration and development in Cardano's ecosystem.

This research marks the first step of a comprehensive plan, which will unfold across multiple stages. The objetive of this first proposal is to adapt the on-chain components of the Semaphore protocol. That is to say, to establish the theoretical and design foundations required for the reimplementation of the core Semaphore smart contracts. Subsequent phases of the project will involve the adaptation of off-chain components, essential for the transaction construction API necessary to build any application on top of this port, and the relayers that are a key off-chain component to interact with the protocol.

The research will be divided into three chapters. In the first chapter, we will provide an introduction and a general overview of Semaphore, including a comparison of its different versions. The objective of this phase is to offer the reader a comprehensive understanding of the protocol.

In the second chapter, we will analyze the protocol's basic elements. Here, we will detail explanations of cryptographic primitives, arithmetic circuits, and the base smart contracts of the protocol. The purpose of this chapter is to break down the protocol and provide an in-depth explanation of its components and the rationale behind its design.

Lastly, in the third chapter, we will outline the adaptations made to the protocol to align it with the Cardano blockchain and its Extended UTxO (EUTxO) model. We will explain in detail how these adaptations were made, including the technical challenges encountered, and provide pseudo-code for the reimplementation of the base Semaphore contracts. This chapter will conclude with a summary of the progress and advancements achieved in the task of porting the Semaphore protocol.

1.2 What is the Semaphore protocol?

Semaphore is a Zero-Knowledge based protocol that allows users to privately proof their membership to a group and send messages like preferences or votes anonymously. Essentially, the protocol allows you to accomplish three things:

- Create and register an identity within a group.
- Prove that your identity is a member of the group without revealing exactly who.
- Anonymously broadcast an arbitrary string that can represent a preference, vote or opinion.

1.2.1 Key protocol concepts

To understand the protocol we have to take in account three important concepts.

Identities A user can represent themselves by creating an identity. This identity contains unique secret values that are generated locally by the user. These values allow them to prove ownership of the identity and to construct proofs and messages tied to this identity. These secret values are important because they prevent impersonating the user by appending unrelated corrupt messages or proofs.

Groups A group is a set of members, where each member has their identity. A group in Semaphore is represented by a Merkle tree that stores the public identity commitment of each member (the leaves). Semahpore uses an incremental merkle tree to update the leaves efficiently. The membership flow is like this: Users hash their identities to produce an identity commitment which is then added as a leaf of the Merkle tree to join a group. Later a user can construct a proof showing that their identity is part of the group but without specifying who. The users can join groups and leave them by themselves, or the user management can be done by an adminitrator too. These administrators can be centralized entities, Ethereum accounts, multi-sig wallets or even smart concracts.

Signals (messages) The signal is a message that contains an arbitrary string representing a preference, vote, or opinion. This message includes proofs that demonstrate the user is a valid member of the group and that the information sent (message and proofs) was created by an identity. Additionally, this message has a nullifier which nullifies the message if it has been used previously.

Relayers One of the first questions that one can have when understanding the protocol is the following: If each user transaction in a blockchain is necessarily linked to a specific wallet address, how the protocol prevents to link the transaction with an address? Wouldn't the protocol be pseudonymous instead of anonymous?

The protocol uses the idea of relayers, which are parties that can receive user signals (along with the proofs) and then send them to the blockchain. Semaphore allows signals to be received from any address, allowing a relayer to broadcast a signal on behalf of a user. Anybody can set up their own relayer and start broadcasting the signals in a decentralized fashion, and a smart contract verifies wether the signals are valid or not by looking at their cryptographic proofs. Additionally, applications might provide rewards for relayers and implement front-running prevention mechanisms. Thus, in this way, the linkability that exist between a transaction and the real user is breaked.

1.2.2 Zero Knowledge cryptography: The cornerstone of the protocol

The Semaphore protocol heavily depends on Zero-Knowledge cryptography to achieve its privacy capabilities. This cryptography allows someone to construct a proof that supports truthful statements over secret data; instead of directly checking the information, a third party only needs the proof to verify a given statement. Thus, the private information is maintained secret, and statements about them can be checked by examining the proof. In the case of Semaphore, it is used to prove membership or whether the signal is associated with an identity.

More concretely, Semaphore uses zk-SNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) that are sent to the smart contract to prove certain conditions privately. Although there are many zk-SNARK protocols available, Semaphore relies on the Groth-16 protocol using the BN-254 elliptic curve. Comparing this protocol with others, Groth-16 is known for its advantage of producing proofs that are small in size and efficient to verify, making it suitable for use in blockchains where storage and computational resources need to be efficient. However, the downside is that it requires a ceremony to create the trusted proving and verification keys. To accomplish this, a multiparty process has to be conducted called powers of tau, this will be treated at the end of this research.

1.2.3 Security guarantees of the protocol

Considering the above the protocol has cryptographic mechanisms that allows to check:

1. **Proof of ownership:**

   Verify that the proofs and messages are being created by the owner of an identity. Note that the protocol works with a system of relayers, consequently the proof of ownership is key since a malicious actor can alter the signal in some way, e.g. broadcasting a different signal than the one the user created. Thus, by proving that there is a correspondence between the signals, proofs and identity, any alteration is prevented.

2. **Proof of membership:**

   Verify that the proof and messages are being created by a valid member of the group. A user appends to the signal a proof that states that its identity is a secret value of one leaf of the Merkle tree; however, this proof doesn't specify or make explicit exactly who it is. Thus, we prove that the message was sent by a valid member in an anonymous manner.

3. **Double signaling:**

   Detect wether a message has been sent previously by an user, this is important since avoids to broadcast a signal twice in a way that can be malicious to the Dapp.

1.2.4 Design characteristics and applications

The protocol has some key design traits that make it very versatile and customizable:

1. **Extendable:**

   It provides some base features which are intended to be extended further by other applications, developers can add more capabilities to the protocol and customize it to their specific needs.

2. **Modular:**

   It is designed in a modular fashion that allows third parties to use only the smart contracts of the core protocol that they need.

3. **Multi-platform:**

   It can be run both on-chain and off-chain, so that it can be adapted to different use contexts. It can run in a browser, server or a smart contract.

The Semaphore protocol, with its versatility and customization capabilities, has led to a wide range of applications across various fields. One of the primary areas where Semaphore has found application is in digital identity management and secure authentication. With its ability to efficiently and privately generate and verify proofs, Semaphore has become an invaluable tool for ensuring the authenticity and integrity of identities in digital environments.

In addition to identity management, Semaphore has also proven useful in voting and decentralized decision-making. By allowing users to securely and verifiably cast votes, the Semaphore protocol enables applications in governance. This has the potential to enhance the transparency by default conditions of L1 blockchains, improving the way elections and community decisions are conducted, while also ensuring the privacy and security of voters.

1.3 Semaphore versions

Semaphore V1 Focuses on providing a zero-knowledge gadget for Ethereum, allowing users to prove membership in a set without revealing identities. It supports anonymous signaling and ensures unique signals using external nullifiers to prevent double-signaling.

Semaphore V2 Introduces optimizations and additional features such as improved integration with other privacy tools, enhanced usability, and security updates.

Semaphore V3 Builds upon V2 with better scalability, performance improvements, and extended support for complex use cases, making it more robust and efficient.

Semaphore V4-beta (latest) Includes the most recent advancements, featuring enhanced security mechanisms, more flexible integration options, and beta features for future-proofing the protocol.

Chapter 2

Protocol technical breakdown

In this chapter, we will explain in detail how the protocol works and what are its components.

2.1 Creating the Identity

Whenever a user wants to join a group, he/she needs to create an identity. As a first step, the user needs to create two secret random values that will later be digested (identity secret):

- **Identity trapdoor:** A 32-byte identity secret used as a trapdoor. This value is used as an additional security measure, making it more difficult to corrupt the pre-image resistance of the identity commitment.
- **Identity nullifier:** A 32-byte identity secret used as a nullifier. Later, this value will be important in the mechanism that prevents double-signaling, specifically, to generate the nullifier that will make each message or signal null.

To prevent fraud, the owner should keep both values secret. Both private values will be digested twice using the Poseidon Hash. The Poseidon Hash is a digest algorithm that consists of arithmetic operations, making the algorithm well-suited and efficient for use in the arithmetic circuits that Zero-Knowledge proofs rely on. The result of this digest is the *identity commitment*, which will be the public value registered in the group.

2.2 Register the Identity into the Group

To register the identity commitment of the user, the value is appended as a new leaf of the incremental Merkle tree which represents the group to register.

Incremental Merkle Trees In principle, a regular Merkle tree is a data structure that can be used to represent a set of elements. The tree is constructed with leaves that represent each member of the set, and sibling leaves are hashed to produce the set digest, namely, the hash root. The key benefit of this data structure is that it provides a computationally efficient way to verify members of a set, in this case, to check whether identities belong to a group.

Thus, when a user joins a group, they will add their identity commitment as a new leaf, which will lead to computing a new Merkle root hash. In the context of a blockchain, computing a new Merkle root hash each time a new user joins can be an expensive computational operation. This is because as the group scales, more hashing operations are needed to compute the new root hash. In order to address this problem, *incremental Merkle trees* were developed. This concept implies that whenever a new leaf is added to the tree, only a minimum of branches are recomputed in order to get the new hash root. This way, unnecessary computations are avoided, making the process more efficient.

2.3 Create a Group

To create a new group, you initialize the Incremental Merkle Tree with the following parameters:

- **Group-id:** A unique identifier for the group.
- **Tree depth:** Represents the maximum quantity that a group will accept as valid. This number will be some exponent of two; by default, the tree depth is 20 (2^{20}).
- **Members:** The list of members to initialize the group. This value will be a list of identity commitments; by default, it is an empty list (`[]`).

2.4 Adding a New Identity

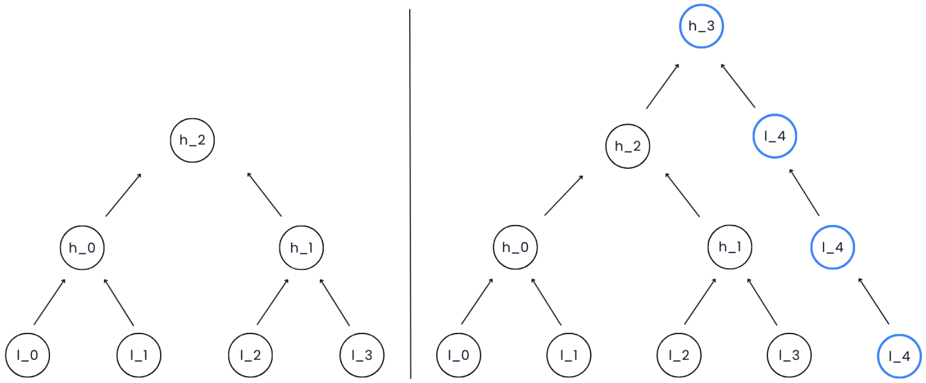
To add a new member, a new leaf must be appended to the Merkle Tree data structure. In this regard, two possibilities may occur:

- When the new node is a left node
- When the new node is a right node

We will always see one of these cases at each level when we are inserting a node: when you insert a node, a) if that node is a left node, the parent node at the next level will be the same node; b) if it is a right node, the parent node will be the hash of this node with the node on its left. This algorithm will be the same at each level, not only at level 0.

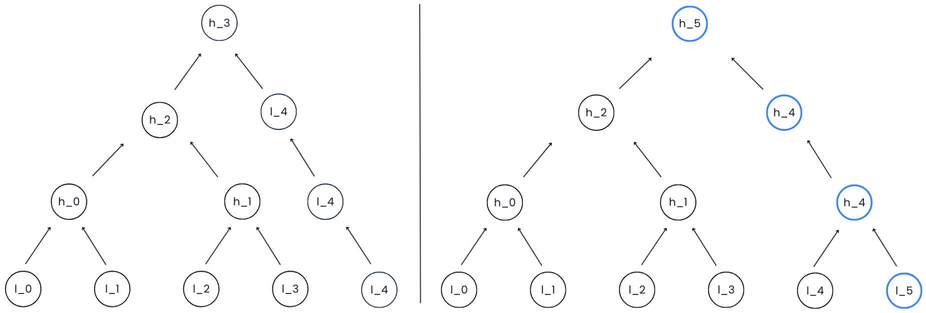
Case 1: The New Node is a Left Node We can see an example of when we insert a new leaf as a left node. It will not be hashed; it will be sent to the next level itself.

If we add l_4..



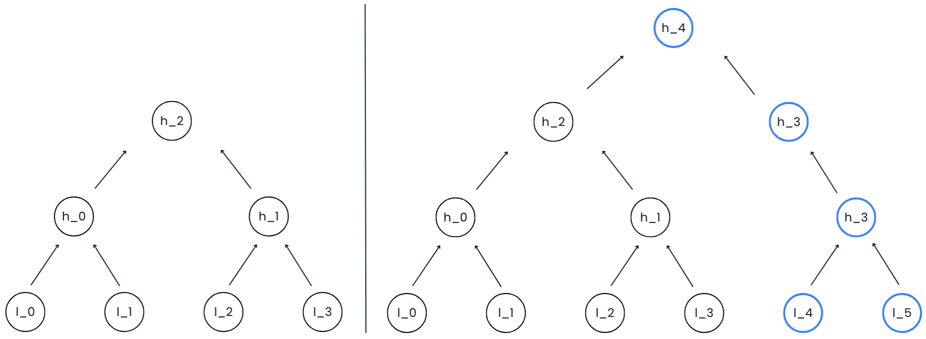
Case 2: The New Node is a Right Node We can see an example of when we insert a new leaf as a right node.

If we add l_5..



2.4.1 Insert Many (Batch Insertion)

Many users can be registered to the group at once. In the case of our example, using the first IMT, let's insert l_4 and l_5 with the insert many function to see the difference.



The idea of inserting many members at the same time and not inserting one, then the other, etc. in a loop is that there are fewer hashings, so it is faster and more optimized.

2.5 Managing the Group

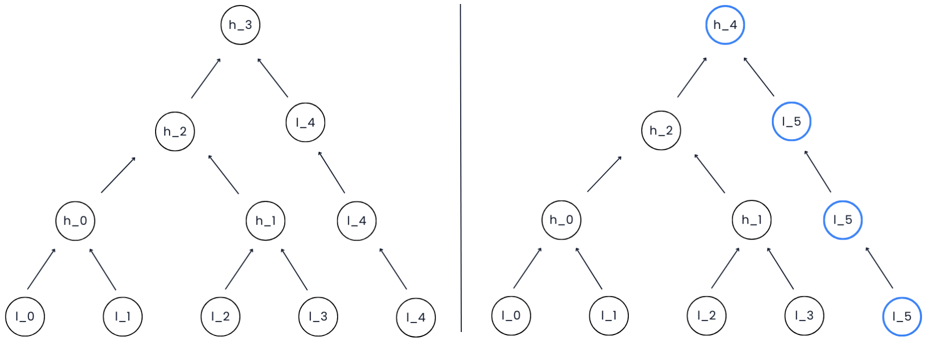
The administrators can manage the groups by updating the members or removing them.

2.5.1 Updating New Members

Similar to the above, when updating the Merkle tree, two cases may occur:

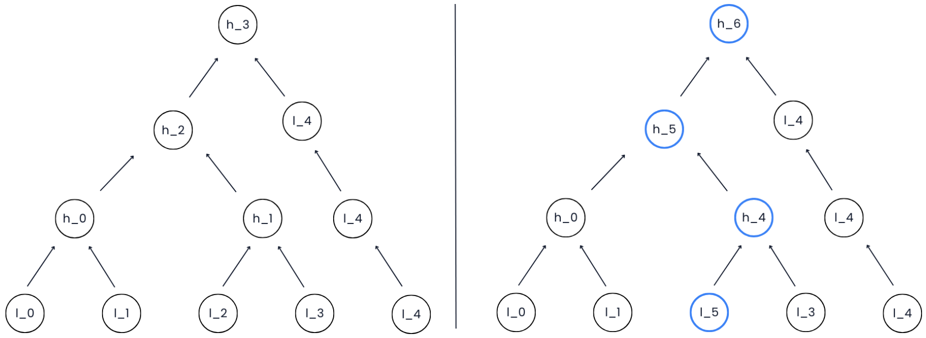
Update when there is no right sibling When there is no right sibling, simply you update the value and you hash it with old branches. Note that these old branches doesn't require to be computed again.

Let's see an example updating l_4 for l_5..



Update when there is right sibling When there is actually a right sibling, the path to the node has to be recomputed along with its siblings.

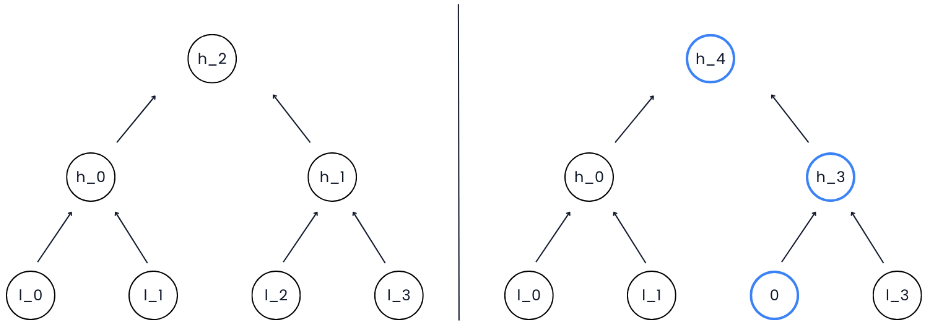
Let's see an example updating l_2 for l_5..



2.5.2 Deleting New Members

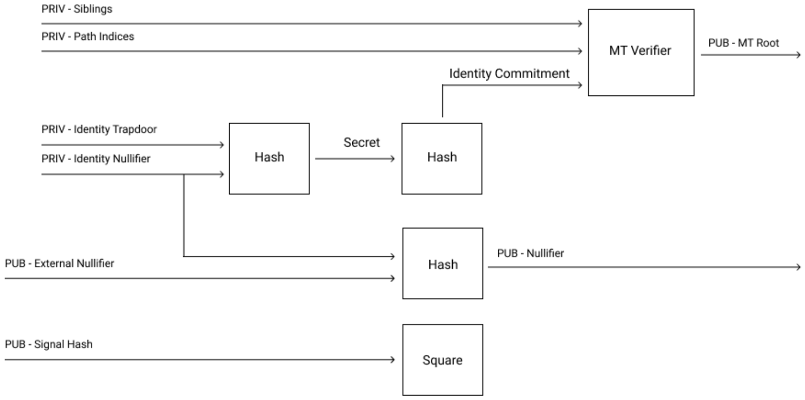
The delete function is the same as the update function but the value used to update is 0.

For example, this is what happens if we delete l_2..



2.6 Sending the Proofs and Signal

Once group registration has been completed, signals can be sent. This first implies constructing the Zero-Knowledge proofs that guarantee the proof of ownership and membership and avoid double-signaling. The following circuit is used to construct the proof.



This scheme implies some sub-circuits to explain.

2.6.1 Sub-circuit: Proof of Membership

As mentioned, the group is represented as an Incremental Merkle tree, and in order to send a signal, a check that the signal was produced by a member is needed. Thus, this sub-circuit will take the form of a Merkle proof. The Merkle proof is an efficient way to verify that a leaf is within a Merkle tree. Specifically, we don't need to recompute the whole Merkle tree; instead, we just need the leaf, the path from the leaf to the root (whether the path is on the left or right at each tree level), and the siblings or neighbor nodes of the path. These values are private inputs to preserve the anonymity of the member:

- **Identity trapdoor:** The 32-byte integer used as an input of the identity commitment, this is, the leaf.
- **Identity nullifier:** The other 32-byte integer used as an input of the identity commitment.
- **Tree Path Indices:** A value that represents the path indices, this means, whether the path is on the left (0) or the right (1).
- **Tree Path Siblings:** The values along the Merkle path to the user's identity commitment.

And will have the following public value:

- **Merkle Root Hash:** The hash value that results from digesting the whole Merkle tree.

The circuit will take these inputs and will let the protocol check if the output of the Merkle proof corresponds to the root hash. The root hash is a public value that represents the group's Merkle tree unique identification. This way, the proof of membership consists of checking if the output of the proof is equal to the Merkle root.

2.6.2 Sub-circuit: Nullifier-hash (Double Signaling Prevention)

Each time a signaling or messaging event is instantiated in the protocol, a value will be needed to be produced as an identification of the current event. For example, if there is a polling, a value will be needed to create to identify that polling, perhaps as a hash of the proposal. This event identification value is called an external nullifier. This external nullifier will be hashed along with the identity nullifier to produce a unique event value, the nullifier. So, each time a signal is sent, a nullifier of the signal will be required. The smart contract checks if the produced nullifier matches the nullifiers already used in the event; thus, the smart contract has a mechanism to recognize and prevent double signaling. If this feature is needed, the smart contract will check the non-membership of the signal nullifier. If the nullifier is not in the list, then the signal passes.

As stated, the following private inputs are needed for this sub-circuit:

- **Identity nullifier:** The 32-byte identity secret used as a nullifier.

Public input:

- **External Nullifier:** The 32-byte external nullifier.

Public outputs:

- **Nullifier Hash:** The hash of the identity nullifier and the external nullifier; used to prevent double-signaling.

2.6.3 Sub-circuit: Signal

This part of the signal prevents any tampering of the signal. In order to achieve this, first the signal is hashed, then the circuit checks whether the square of the hash value corresponds to the signal hash.

Public inputs:

- **Signal hash:** The hash of the user's signal.

2.6.4 Full-circuit: Proof of Ownership

The fact that these sub-circuits compose a major circuit guarantees that the proof and signal are created by the same user.

2.7 Base Smart Contracts

The protocol has two base contracts and two extended contracts. The base contracts mainly deal with group managing functions and the verification of the proof of memberships. On top of this, the extended contracts allow two common uses of the protocol: voting and whistle-blowing. Smart contract programmers can use the base contracts and disregard the extended contracts if the development intends other different applications. In this section, we'll explain what these two base smart contracts are.

2.7.1 SemaphoreGroups.sol

The `SemaphoreGroups` contract is an abstract contract designed to manage groups of data using Merkle trees. The contract is designed to manage groups using Merkle trees, allowing for the addition, updating, and removal of identity commitments to these groups. It ensures data integrity and provides transparency through event logging. The contract is abstract, serving as a base for more specific implementations. The core functions `_createGroup`, `_addMember`, `_updateMember`, and `_removeMember` handle the creation of groups and the management of members within these groups. Each operation checks if the group exists, performs the necessary action on the Merkle tree, and emits an event to log the change.

Key Components

State Variables

- `mapping(uint256 => IncrementalTreeData) internal merkleTrees;`
  - This mapping stores Merkle tree data for different groups identified by a `groupId`.

Library Usage

- `using IncrementalBinaryTree for IncrementalTreeData;`
  - This line allows `IncrementalTreeData` to use functions defined in the `IncrementalBinaryTree` library.

Functions

_createGroup

```
function _createGroup(uint256 groupId, uint256 merkleTreeDepth) internal virtual {..
```

- **Purpose**: Initializes a new group with a specified `groupId` and Merkle tree depth.
- **Visibility**: Internal and virtual, meaning it can only be called from within this contract or derived contracts, and can be overridden.
- **Operations**:
  - Checks if the group already exists.
  - Initializes a new Merkle tree with the specified depth.
  - Stores the new Merkle tree in the `merkleTrees` mapping.
  - Emits an event to signal the creation of a new group.

_addMember

```
function _addMember(uint256 groupId, uint256 identityCommitment) internal virtual {..
```

- **Purpose**: Adds a new identity commitment to an existing group.
- **Visibility**: Internal and virtual.
- **Operations**:
  - Validates if the group exists by checking the Merkle tree depth.
  - Inserts the new identity commitment into the group's Merkle tree.
  - Retrieves the updated Merkle tree root and the index of the new member.
  - Emits an event to log the addition of the new member.

_updateMember

```
function _updateMember(
    uint256 groupId,
    uint256 identityCommitment,
    uint256 newIdentityCommitment,
    uint256[] calldata proofSiblings,
    uint8[] calldata proofPathIndices
) internal virtual {
```

- **Purpose**: Updates an existing identity commitment in a group.
- **Visibility**: Internal and virtual.
- **Operations**:
  - Validates if the group exists by checking the Merkle tree depth.
  - Updates the identity commitment in the group's Merkle tree.
  - Retrieves the updated Merkle tree root and the index of the updated member.
  - Emits an event to log the update of the member.

_removeMember

```
function _removeMember(
    uint256 groupId,
    uint256 identityCommitment,
    uint256[] calldata proof) {..}
```

2.7.2 SemaphoreVerifier.sol

The contract is designed to verify the correctness of Semaphore proofs, crucial for ensuring the integrity and privacy of group membership and signaling processes. It leverages the Pairing library for cryptographic operations and adheres to the `ISemaphoreVerifier` interface, offering a method to verify proofs using predefined verification key points. It is a modified version of the Groth16 verifier template from SnarkJS, specifically adapted for Semaphore.

Functions

verifyProof

```
function verifyProof(
    uint256 merkleTreeRoot,
    uint256 nullifierHash,
    uint256 signal,
    uint256 externalNullifier,
    uint256[8] calldata proof,
    uint256 merkleTreeDepth
) external view override {..
```

This Solidity function `verifyProof` is designed to verify a cryptographic proof related to a Semaphore protocol. Here's a brief explanation of each parameter:

- `merkleTreeRoot`: Represents the root hash of a Merkle tree, which is used to efficiently prove the inclusion or absence of an element.
- `nullifierHash`: Hash of a nullifier, which is a unique identifier used to indicate that a particular commitment has been spent or used in some way.
- `signal`: A signal value related to the proof.
- `externalNullifier`: Another unique identifier, typically used to link proofs or commitments to specific external conditions or actions.
- `proof`: An array containing cryptographic proof elements, likely derived from a zk-SNARK or similar proof system, used to validate the integrity and correctness of the provided data.

- `merkleTreeDepth`: Specifies the depth or height of the Merkle tree being used, which determines the number of layers in the tree structure.

The function is marked as `view` and `external`, indicating that it does not modify state and can be called externally. Its purpose is to validate the provided proof against the specified parameters and return whether the proof is valid or not.

_getVerificationKey

```
function _getVerificationKey(uint256 vkPointsIndex) private view returns (VerificationKey me
    ..
}
```

This Solidity function `_getVerificationKey` is a private `view` function that returns a `VerificationKey` struct. Here's a breakdown of what it does:

- **Initialization**: It initializes a `VerificationKey` struct `vk`.
- **Assigning Values**:
  - `vk.alfa1`: Assigns a value of type `Pairing.G1Point`, likely representing a point in a cryptographic group G1 with specific coordinates.
  - `vk.beta2` and `vk.gamma2`: Assigns values of type `Pairing.G2Point`, representing points in a cryptographic group G2. These points have coordinates specified by arrays of integers.
  - `vk.delta2`: Assigns a value of type `Pairing.G2Point` using values from `VK_POINTS`, fetched based on the `vkPointsIndex` parameter.
  - `vk.IC`: Initializes an array of `Pairing.G1Point` of length 5, likely representing constraints or intermediate points in cryptographic proofs.
- **Return**: Finally, the function returns the `vk` struct containing all assigned values.

Chapter 3

Protocol adaptation

3.1 Overview

In this chapter the adaptation of the Semaphore protocol will be designed for an UTxO ledger model. Originally the Semaphore protocol was designed for the accounting model of Ethereum, however, Cardano uses an extended version of Bitcoin UTxO ledger model. This imply redesigning several parts of the smart contract operation and adapt it to Cardano.

As a guiding principle, we based the adaptation following an iterative design, this means, start from simple to complex. In this sense we did a simpler version that could reproduce the minimal features of Semaphore, and to leave some improvements in terms of security and flexibility for later iterations.

This first iteration has some differences between original one, the more crucial difference is that it follows a monolithic design instead of a modular one. All the features of the base protocol were comprised into in a multi-validator. This simplification can be summarized as following:

Monolithic The Ethereum Semaphore version was composed of two contracts which we reduced in one.

Event-based Each event signaling is tied to the creation of a group instance, this imply creating or duplicating the group each time an event is done. This, in contrast to a group-based instance where the same group handles many events, and there is no need for duplicating the group once is created.

Trustful The group management like add, update and delete users is trusted on the admin, rather than the contract checking if this group managing is correctly performed.

In future iterations of the protocol will be updated to work in a modular, group-based and trustless manner.

Signaling Modification

In the original design of the protocol a list is used to check the nullifiers already used. To make this more efficient we will use a sparse merkle tree; concretely, the reason to use this data structure is to proof non-memberships of new signals, which is key to prevent double signaling.

3.2 First Iteration

This first iteration consist in a multi-validator that can mint a group token and manage the group by spending that token. Basic actions of this first iteration can be described as follows:

1. **Create a group**
2. **Modify a group**
3. **Signlaling**

In the next section each step will be explain in detail. The redeemer will be a custom type to represent these actions:

```
type Redeemer {
    Create
    Modify
    Signal(groth16.Proof, mpf.Proof, (ByteArray, ByteArray))
}
```

Inside the Signal value, it is included one value to verify the zk-Proof and values to verify the insertion of the nullifier.

(1) Create a group.

To create a group is the first entry of the protocol. In our first iteration an administrator of some organization can initialize a group by minting a *group token*. The group token is an NFT, a way to tie a specific UTxO to a specific group instance, meaning that one group instance will only correspond to one UTxO in its whole lifetime. This unique token will differenciate the group instances to other instances and UTxOs with malicious datums.

This action corresponds to the minting part of the multi-validator and it is parametized with the following random values to create the NFT:

Input reference A specific UTxO is expected to be used as input when minting the *Group Token*, concretely, this is the transaction-id of the UTxO taken as input. It serves as a random value to create a unique asset and non-fungible asset (Group NFT).

Initial Merkle Root Hash This is the merkle root hash of the initial member set of the group.

The asset name of the Group NFT is computed this way `Hash(Merkle_Root_Hash, Input_Tx_Id)`. The administrator of the group can create a new group by minting a new Group NFT if only if:

- One Group NFT is being minted.
- This Group NFT is sent to the multi-validator script address.
- The Datum is formed correctly.

Attached to the Group NFT UTxO, a Datum will be created in with the following values:

Merkle Root Hash The Merkle root

Group Tree Depth The maximum depth of the merkle tree that represents the group.

Admin PubKeyHash The public key hash of the administrator.

Nullifier Merkle Patricia forestry Root Hash A *Merkle Patricia Forestry* that will represent the used nullifiers of the signal. Crucial to prevent double spending.

Verification key reference input This is the reference input used to get the verification keys used in the Zero-Knowledge proof.

The script will receive a Plutus Data 'Create' of type Redeemer.

(2) Modify the Group

The admin could need to modify the group to perform the following actions:

- Add new members
- Update members
- Delete members

In order to accomplish one of these actions, the spending part of the script must be executed with a redeemer of 'Modify'. The Group UTxO could only be spent if only if:

- The transacton includes a signatures that corresponds to the administrator public key hash setted in the datum.
- The Group NFT is sent back to the script address.
- Before the event starts.

(3) Signaling

This action is performed by the user (or a relayer, see below) to cast a signal to the protocol. Again, the spending part of the validator must be executed with the Redeemer `Signal`. The redeemer will include some needed values to achieve several verifications such as the Zero-Knowledge proof and the *Merkle Patricia Forestry* inclusion proof (these will be detailed below). The Group UTxO could only be spent if only if:

- The Group Token is sent back to the script address.
- The Datum is preserved in the Group NFT output, except the Nullifier Merkle tree.
- A reference input is included in the transaction and it corresponds to the one set in the Datum. The reference input will contain the verifications keys to check the Zero-Knowledge proof.
- A valid Zero-Knowledge proof. A valid Zero-Knowledge proof is provided. To construct the proof, the contract will need the following parameters: the reference input with the verification key, the proof (which will be provided in the redeemer), and three public inputs—the group root hash (provided in the datum), the external nullifier (provided by the policy ID of the contract), and the hash of the nullifier (provided by the user).
- The nullifier hash of the signal hasn't been spent. To verify this, we will use a proof of inclusion in the Merkle Patricia Forestry tree, which contains all used nullifiers. The proof of inclusion can only be made with new members of the set; if a particular value has already been included in the tree, it is impossible to construct a valid tree. This ensures that only nullifiers that haven't been spent are suitable for the proof of inclusion. Therefore, this verification satisfies the double signaling prevention that the Semaphore protocol requires.
- The Tx is within the event schedule.

It is worth noting that since the external nullifier will be the policyId, the minting of a group token is limited to a single signaling event. In order to conduct new events the Group token must be minted again, this means as a new or alternative instance.

(4) Relayers

Signaling requires validation of proof of membership. Since validation is performed on-chain, proof of membership needs to be sent to a smart contract on the blockchain, and so the address of the sender becomes public. To keep anonymity, relayers can get involved.

Signaling involves sending a pair consisting of a) proof of membership, b) the signal (message) itself. Note that verification of proof involves verifying the hash of the signal, and this prevents the relayer from sending a false signal. Relayers will be compensated through fees taken from a common-pool. Funds for this common-pool can be taken from initial registration fees or by some other means controlled, for example the treasury of a DAO. Having more than one relayer competing for the fee is desirable, as this diminishes the possibility of failing to relay the signal sent by a user. The idea is that user will simultaneously send the signaling information (pair of proof and signal) to all registered relayers. Only one of them will succeed, as the nullifier will prevent double signaling. Note that for those that do not succeed, no transaction fee is incurred since a failing verification is rejected as an invalid transaction.

3.3 Future iteration considerations

As mentioned before this project will be executed in iterations; once the first iteration is done, new features, modifications and upgrades will be incorporated to the protocol.

3.3.1 Modularity

The first improvement to the protocol will be decomposing the main smart contract into smaller components. This follows the original design of the protocol and allows for flexible use-cases of Plutus smart contracts; for example, some Dapps could use just one base smart contract of the protocol and omit others.

3.3.2 Group-base instances

In the first iteration, each event, such as a vote, poll, or survey, will require instantiating a new group instance, even if the group already exists. We initially conducted the project this way to achieve a simpler PoC. However, in the future, we want each group instance to be unique and reusable in various signal events. For example, the UTxO that contains the group instance could be simultaneously reused in a poll and a survey without needing to create the same group instance again.

3.3.3 Relayer claiming mechanisms

We are going to design an incentive mechanisms for the Relayers to claim the signals fees.

3.3.4 Token registering

Up until now, we have talked about the permission of administrators to manage the group. However, the new group mechanisms could be used that don't need intermediaries. For example, one may have a membership token that could be used as permission to join or leave a group. This way, the group could be managed in a more decentralized manner.

3.3.5 Trustless add, update and delete actions

Currently, the first iteration assumes some confidence in the way the administrator updates the integrity check of the group. It only requires the signature of the admin to modify the Merkle tree root hash. However, in the future, we can incorporate the verification of these modifications directly into the smart contract. This way, the administrator could modify the group in a less arbitrary manner, ensuring there are no errors in the integrity data update.

3.3.6 Group composability

Certain organizations, like confederations, would need to congregate various groups to conduct a signaling event. We would like a mechanism to merge groups into more complex ones and to function as a single Semaphore group.

3.4 Elliptic curve adaptation

The PSE group has developed a Circom circuit to construct the Zero-Knowledge proof needed for the key verifications of the protocol and to maintain some secret data in order to achieve privacy. This circuit has been audited, ensuring that the protocol will behave as expected and avoid potential exploits. In this sense, one of the key questions of this project was: To what extent can we preserve the circuit?

In this regard, the main component that could imply a problem of interoperability between Cardano and the original protocol design is the hashing algorithm used in the group data structure. It could have been the case that the Poseidon Hash algorithm could not be implemented with the Plutus primitives of the third version. To avoid any difficulty, we chose to use the signature of the administrator to upgrade the group instead of having each Merkle tree upgrade be directly verified by the smart contract. In this way, we can use the original circuit as it is without any modification, since we avoid the need to use the Poseidon Hash algorithm in the Plutus smart contract. The contrary could have meant a significant deviation from the original design, resulting in the existing audits of the protocol becoming inapplicable.

Bibliography

[1] Kobi Gurkan, Koh Wei Jie, Barry Whitehat. *Community Proposal: Semaphore: Zero-Knowledge Signaling on Ethereum* https://semaphore.pse.dev/whitepaper-v1.pdf

[2] Koh Wei Jie. *To Mixers and Beyond: presenting Semaphore, a privacy gadget built on Ethereum* https://medium.com/coinmonks/to-mixers-and-beyond-presenting-semaphore-a-privacy-gadget-built-on-ethereum-4c8b00857c9

[3] *Brief Introduction of Semaphore, a Zero-knowledge Group Membership Protocol* https://hackernoon.com/brief-introduction-of-semaphore-a-zero-knowledge-group-membership-protocol

[4] *Semaphore V2 is Live!* https://medium.com/privacy-scaling-explorations/semaphore-v2-is-live-f263e9372579

[5] *Semaphore Docs* https://docs.semaphore.pse.dev/

[6] File *semaphore.sol* in GitHub repository. https://github.com/semaphore-protocol/semaphore/blob/main/packages/contracts/contracts/base/SemaphoreGroups.sol

[7] File *SemaphoreVerifier.sol* in GitHub repository. https://github.com/semaphore-protocol/semaphore/blob/main/packages/contracts/contracts/base/SemaphoreVerifier.sol

[8] *Incremental Merkle Tree* https://zokyo-auditing-tutorials.gitbook.io/zokyo-tutorials/tutorial-16-zero-knowledge-zk/definitions-and-essentials/incremental-merkle-tree

[9] VPlasencia. *Incremental Merkle Tree Semaphore v4* https://hackmd.io/@vplasencia/S1whLBN16

[10] *Announcing the Perpetual Powers of Tau Ceremony to benefit all zk-SNARK projects - Koh Wei Jie Medium* https://medium.com/coinmonks/announcing-the-perpetual-powers-of-tau-ceremony-to-benefit-all-zk-snark-projects-c3da86a

[11] *Setup Ceremonies - Antony Mpho Matlala* https://zkproof.org/2021/06/30/setup-ceremonies/

[12] *Conclusion of the Powers of Tau Ceremony - Z-Cash Foundation* `https://zfnd.org/conclusion-of-the-powers-of-tau-ceremony/`

[13] *Diving into the zk-SNARKs Setup Phase - Daniel Benarroch Medium* `https://medium.com/qed-it/diving-into-the-snarks-setup-phase-b7660242a0d7`

[14] *What happens in the trusted setup phase of the Groth16 protocol - Phong Lee Medium* `https://leducphong.medium.com/what-happens-in-the-trusted-setup-phase-of-the-groth16-protocol-0a7764ac8080`