Hybrid PoA on Aztec: Proof of Asset Ownership over Public and Private Balances via Hierarchical Proof-Carrying Data

Rio Kanehiro[1], Yohei Watanabe[1,2], and Mitsugu Iwamoto[1]

[1] The University of Electro-Communications,
1-5-1 Chofugaoka, Chofushi, 182-8585 Tokyo, Japan
uec@rknhr.jp, {watanabe, mitsugu}@uec.ac.jp
[2] AIST,
2-3-26 Aomi, Koto-ku, 135-0064 Tokyo, Japan

Abstract. Proof of Assets (PoA) protocols enable custodians to prove ownership of digital assets without revealing their account addresses or corresponding balances. While existing PoA protocols focused on either private or public balances, hybrid-state blockchains such as the Aztec Network involve both. In these systems, private balances are managed by encrypted notes that work similarly to the UTXO model, with only commitments stored on-chain. We present a PoA protocol that supports hybrid balances by combining public-state membership proofs with proofs of ownership over private notes. Since a custodian may control multiple accounts and numerous notes, we employ hierarchical proof-carrying data via recursive zk-SNARK, enabling scalable proving and efficient batch verification. We implement our system using the Noir DSL with the UltraHonk proving backend, and evaluate the performance.

1 Introduction

Cryptocurrency exchanges and custodians play a key role in the blockchain ecosystem by holding assets on behalf of users and providing advanced trading services, fiat on-ramps, and account recovery. This model, however, comes at a cost: users must trust the custodians to keep their funds secure and manage them responsibly. There have been several cases, notably Mt. Gox and FTX, where these platforms collapsed due to fraud or mismanagement, resulting in catastrophic losses for the platform users.

To address these risks and retain user trust, *Proof of Reserves* (PoR) protocols have been proposed [6] to enable custodians to prove solvency without revealing sensitive information such as individual account balance. PoR is a combination of two sub-protocols: *Proof of Assets* (PoA), which proves ownership of on-chain assets, and *Proof of Liabilities* (PoL), which proves the total liabilities owed to users. If the total assets exceed the liabilities, the custodian is considered solvent.

In this paper, we focus solely on the PoA component for the Aztec Network (Aztec for short), a hybrid-state blockchain that supports both public and private balances. Public balances follow an account-based model similar to Ethereum, where a mapping from account address to balances is maintained in the *Public Data Tree*. In contrast, private balances are represented as a set of *notes*, whose commitments are stored in the *Note Hash Tree*. In addition, each note has an associated *nullifier* that marks the note as *consumed*, with all nullifiers recorded in the *Nullifier Tree*. The total *asset* of an account is defined as the sum of its public balance and the values of all *unconsumed* notes.

To prove ownership of assets, the prover must demonstrate that the public balance is included in the Public Data Tree, that the corresponding note commitments are included in the Note Hash Tree, and that the associated nullifiers are *not* included in the Nullifier Tree. Furthermore, the prover must show ownership of the account and that the total balance is computed correctly without *double counting*, all without revealing public balances, note contents, nullifiers, and account addresses.

The main challenge here is scalability. In a PoA system, a custodian may control thousands of accounts and notes and must be able to prove ownership across all of them in a practical time frame. Using the state-of-the-art proving system UltraHonk, encoding a PoA statement for one account with a single note into a zk-SNARK circuit results in roughly 16,000 gates. As the number of accounts and notes increases, the circuit size grows linearly. As a reference, encoding PoA for 100 accounts yields a circuit with roughly 1,400,000 gates. Moreover, encoding the statement into a circuit requires the prover to fix the input size at compilation time, further limiting practicality.

Table 1. Comparison of related work. BC denotes the target blockchain, PW denotes the prover work, CP denotes the concurrent proving time, and VW denotes the verifier work. n denotes number of accounts or UTXOs, m denotes anonymity set size ($m > n$).

Scheme	BC	PW	CP	VW	Crypto
Provisions [6]	Bitcoin (Public)	$O(m)$	Not Supported	$O(m)$	Sigma Protocol
zk-SNARK [11]	Bitcoin (Public)	$O(m \cdot m \log m)$	$O(m \log m)$	$O(m)$	Groth16
MProve+ [7]	Monero (Private)	$O(n \cdot m)$	Not Supported	$O(n)$	Ring Signature
MProve-Nova [12]	Monero (Private)	$O(n \cdot m \log m)$	Not Supported	$O(1)$	Nova IVC
Hybrid PoA	Aztec (Hybrid)	$O(n \log n \cdot m \log m)$	$O(\log n \cdot m \log m)$	$O(1)$	PCD w/ Recursive ZKP

1.1 Our Contributions

In this work, we present a PoA scheme that addresses the scalability issues of proving asset ownership on Aztec with both public and private balances, which we call *Hybrid PoA*. The key idea is to decompose the PoA statement into smaller sub-proofs that certify ownership and validity of individual notes, public balances, and accounts. These sub-proofs are aggregated using hierarchical proof-carrying data (PCD) instantiated via recursive zk-SNARKs [4]. Hierarchical PCD structures aggregation as a tree, where each node proves correctness over a subset of accounts or notes, and internal nodes combine child proofs. This structure enables scalable proof construction across many accounts and notes, and in principle, proof generation across independent subtrees. To ensure that no duplicate accounts are double-counted, it enforces ordering constraints on note identifiers and account addresses.

Using the UltraHonk proving backend, the aggregated proof can produce a constant-size proof and can be verified in constant time. Verification requires only the roots of the Note Hash Tree, Nullifier Tree, and Public Data Tree, which can be obtained from the blockchain node. As a result, the verifier only needs succinct information about the blockchain state and does not require any protocol-defined anonymity set.

Hybrid PoA has several limitations. First, it requires a trusted setup, as UltraHonk is based on the KZG commitment scheme. Second, since there is no uniqueness across different PoAs, multiple custodians can collude to boost their assets. Finally, custodians may choose to prove ownership of fewer assets than they actually have. This does not violate the protocol soundness and is generally acceptable in PoA, as the custodian's ultimate goal is to prove solvency.

We implemented the Hybrid PoA protocol using Noir DSL and the UltraHonk proving backend. Noir is a domain-specific language for writing zk-SNARK circuits, and UltraHonk is a PLONK-based SNARK that produces constant-sized proofs. Running our protocol implementation on a 16-core server, we generated an aggregated proof for 1,000 accounts, each holding 112 notes along with some public balance. With four accounts proving concurrently, it takes 6 hours to generate the aggregated proof. The proof is approximately 16 KB and can be verified in 83 ms.

1.2 Related Work

Provisions [6] is one of the first constructions of PoR designed for Bitcoin exchanges. It enables custodians to prove solvency without revealing reserve addresses or individual user balances by committing to customer liabilities using Pedersen commitments and proving ownership of funds over an anonymity set of public keys using zero-knowledge proofs. However, its prover and verifier complexities are both linear in the size of the anonymity set and rely on public-key anonymity sets rather than Bitcoin addresses, which makes it challenging to use in real-world settings.

Following Provisions, various PoR protocols have emerged. Reddy [11] presents a PoA protocol for Bitcoin that avoids explicit public-key anonymity sets by using generic zero-knowledge proofs, specifically zk-SNARKs. IZPR [5] introduces an efficient, incrementally updatable PoA protocol using lookup arguments, achieving constant-sized proofs and fast verification. These works primarily target blockchains with public balances and rely on a monolithic prover.

PoA has been studied for privacy-preserving blockchains. MProve [8] and MProve+ [7] propose PoA protocols for Monero, using ring signature as the cryptographic backbone. MProve-Nova [12] extends this line of work by employing an *Incrementally Verifiable Computation* (IVC) based on the *folding scheme*

to achieve scalable proving and constant-time verification across thousands of UTXOs. To prevent double-counting, MProve-Nova introduces the *Double Spend Tree*, which records outputs that have already been used during aggregation and enforces uniqueness across the recursion step.

Table 1 presents a brief comparison between Hybrid PoA and prior PoA protocols. Compared to existing schemes, Hybrid PoA supports both public and private balance under a single proof. Although the total prover work is high, the protocol allows concurrent proof generation by leveraging hierarchical proof-carrying data and produces proofs of constant size.

2 Preliminaries

We denote $[n]$ to represent $\{1, \ldots, n\}$. For a set of elements, we use the notation $\{a_i\}_{i=1}^{n}$ or $\{a^{(i)}\}_{i=1}^{n}$.

2.1 The Aztec Network

The Aztec Network (Aztec) [1] is a Layer 2 blockchain built on Ethereum that supports both private and public state and private and public smart contract execution. Using this property, tokens in Aztec can support both public and private balances, as standardized by the AIP-20 specification [15]. In this section, we outline the core components of the system to understand our protocol: Aztec account, public state, private state, and AIP-20 tokens. A more detailed description is provided in section A.

Aztec Account. An Aztec account is defined by a master secret key sk. The master secret key is used to derive the four sub-keys: the outgoing viewing secret key $\mathsf{ovsk_m}$, the incoming viewing secret key $\mathsf{ivsk_m}$, and the tagging secret key $\mathsf{tsk_m}$. Each of these has a corresponding public key $(\mathsf{Npk_m}, \mathsf{Ovpk_m}, \mathsf{Ivpk_m}, \mathsf{Tpk_m})$. The nullifier secret key $\mathsf{nsk_m}$ has an application-specific key, called the *app-siloed nullifier secret key* $\mathsf{nsk_{app}}$, which is derived from $\mathsf{nsk_m}$ and the application address. The account address a_{acc} is computed from a set of public keys $(\mathsf{Npk_m}, \mathsf{Ovpk_m}, \mathsf{Ivpk_m}, \mathsf{Tpk_m})$ together with a partial address $\mathsf{a_{par}}$. All secret keys are stored locally in the user's *private execution environment (PXE)*, a private environment for storing keys, managing private states, and executing private functions.

Public State. The public state operates similarly to Ethereum Virtual Machine (EVM) based blockchains, where each node maintains a copy of the global data. These public states are managed by the *Public Data Tree* (PDT), an indexed Merkle tree whose leaves represent the key-value pair of public data. The public data key $\mathsf{p_{key}}$ corresponds to the storage slot of an application, and is defined as the hash of the storage slot and the application address. On the other hand, the public data value $\mathsf{p_{val}}$ is the value associated with the public data.

Private State. Private states are represented as *notes*. A note n is typically defined as a tuple of $(\mathsf{n_{owner}}, \mathsf{n_{rnd}}, \mathsf{n_{val}})$, where $\mathsf{n_{owner}}$ denotes the owner of the note, $\mathsf{n_{rnd}}$ is randomness used to compute its commitments, and $\mathsf{n_{val}}$ is the note's value. These notes are stored locally in the user's PXE, while their commitments are stored on-chain in the *Note Hash Tree* (NHT). Note commitments are also called *unique note hash* $\mathsf{n_{uhash}}$ and are derived from the fields of n, the application's storage slot, application address, and a unique nonce $\mathsf{n_{nonce}}$. The Note Hash Tree is an *append-only Merkle Tree*, meaning that once a note commitment is inserted, it remains immutable. To mark that a note is consumed (or spent), a corresponding *nullifier* is inserted into the *Nullifier Tree* (NT), an index merkle tree. The nullifier ρ is derived from note commitment $\mathsf{n_{uhash}}$, the app-siloed nullifier secret key $\mathsf{nsk_{app}}$, and the application address. A note is considered *consumable* if its note commitment is included in the Note Hash Tree and its nullifier is *not* included in the Nullifier Tree.

AIP-20 Token. Public balances follow the account-balance model of ERC-20 tokens on EVM-based blockchains, where there is a mapping from user address a to token balance $\mathsf{b_{pub}}$, maintained in the Public Data Tree. The storage slot for a user's balance is defined as the hash of the user's address and the storage slot for the public balances mapping, $\mathsf{slot_{pubMap}}$. Private balances follow a UTXO-based model, where a user's balance is represented as a set of notes. Each user address maps to a set of notes, and the private balance $\mathsf{b_{pri}}$ of a user is the sum of all consumable notes. The storage slot of the note set is defined as the hash of the user address and the storage slot of the private balances mapping $\mathsf{slot_{priMap}}$.

2.2 Proof System

Definition 1 (Non-interative Proving System). *A proving system is a protocol between a prover $\mathcal{P}$ and an efficient verifier $\mathcal{V}$, in which $\mathcal{P}$ tries to convince $\mathcal{V}$ of the knowledge of a secret witness w such that an*

$\mathcal{R}(x, w) = 1$ *for a public instance* x*. If the proof is a single message from* $\mathcal{P}$ *to* $\mathcal{V}$*, the system is* non-interactive*. A non-interactive proving system has the following syntax:*

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*: Generates public parameters* pp*.*

$(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \mathcal{R})$*: Generates proving key* pk *and verification key* vk*.*

$\pi/\perp \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$*: Outputs* π *if* $\mathcal{R}(x, w) = 1$*; otherwise, outputs* $\perp$*.*

$0/1 \leftarrow \mathsf{Verify}(\mathsf{vk}, x, \pi)$*: Outputs* 1 *if* π *is valid; otherwise, outputs* 0*.*

Definition 2 (zk-SNARK (informal) [10]). *A non-interactive proof* π *for* $\mathcal{R}(x, w)$ *is called a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) if it satisfies the following properties:*

Perfect Completeness: If $\mathcal{R}(x, w) = 1$*, then* $\mathcal{P}$ *can generate a proof* π *that is always accepted by* $\mathcal{V}$*.*

Knowledge Soundness: If $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$*, then there exists an extractor that can extract a valid witness* w *from* $\mathcal{P}$*.*

Zero-Knowledge: The proof (π, x) *reveals nothing about* w*.*

Succinctness: The proof size and verification time are at most polylogarithmic in the size of the relation $|\mathcal{R}|$ *and the witness* $|w|$*.*

Formal definitions of *perfect completeness*, *knowledge soundness*, and *zero-knowledge* are provided in section B. In practice, relations are represented using *arithmetic circuits*. An arithmetic circuit is a directed acyclic graph that consists of addition and multiplication gates, where the wires correspond to the instance x and witness w. Proving $\mathcal{R}(x, w) = 1$ reduces to demonstrating the knowledge of a witness w that satisfies all constraints enforced by the circuit.

Definition 3 (Recursive zk-SNARK (informal)). *A recursive zk-SNARK is a proof that, in part, verifies other zk proofs inside the circuit being proved. Here,* $\mathcal{P}$ *proves that they have verified a set of inner proofs:*

$$\pi \leftarrow \mathsf{Prove}\left(\mathsf{pk}, (x_1, \ldots, x_n), (\pi_1, \ldots, \pi_n)\right).$$

The outer proving circuit implements the verifier logic for these inner proofs. When $\mathcal{V}$ *verifies the outer proof* π*, the inner proofs* $\pi_1, \ldots, \pi_n$ *are also verified. Hence, if the following holds, then each* π_i *is a valid proof for its* x_i*.*

$$\mathsf{Verify}\left((x_1, \ldots, x_n), \pi\right) = 1.$$

2.3 Proof-Carrying Data

Proof-Carrying Data (PCD) [4] is a cryptographic framework that enables parties to perform distributed computations that run indefinitely while ensuring that the correctness of every intermediate state can be efficiently verified. In a PCD system, each step of the computation produces an output called the *message*, denoted by m, and a proof π that proves the validity of that step and its connection to all previous computations. A special case of PCD is *incrementally verifiable computation* (IVC) [14], in which nodes form a linear chain and each node's computation step depends only on its previous one.

Definition 4 (Proof Carrying Data [3,16]). *A proof-carrying data scheme for a class of compliance predicates* F *is defined by a tuple of algorithms* $\mathsf{PCD} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$*, where:*

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$*: Inputs the security parameter* λ *in unary form and outputs public parameters* pp*.*

$(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \varphi)$*: Inputs the public parameters* pp *and a compliance predicate* $\varphi \in \mathsf{F}$*, and outputs a prover key* pk *and a verifier key* vk*.*

$\pi/\perp \leftarrow \mathsf{Prove}(\mathsf{pk}, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M)$*: Inputs the prover key* pk*, the outgoing message* z*, the local data* z_{loc}*, and incoming messages* $\{z_m\}_{m=1}^M$ *along with its proof* $\{\pi_m\}_{m=1}^M$*. It outputs a new proof* π *attesting the correctness of* z*.*

$0/1 \leftarrow \mathsf{Verify}(\mathsf{vk}, z, \pi)$*: Inputs the verifier key* vk*, the message* z*, and the proof* π*, and outputs* $0/1$ *to reject or accept.*

PCD must satisfy *perfect completeness* and *knowledge soundness*, and optionally *zero-knowledge* defined in section C. When the size of the incoming message is one ($M = 1$), we use z_{in} to denote the incoming message.

One construction of PCD is via recursive zk-SNARK. In this approach, each node corresponds to a zk-SNARK circuit that proves the correctness of a local computation and verifies the proofs of its previous nodes. These proofs are verified inside the circuit itself using the recursive zk-SNARK technique. For further details on the construction, refer to [2].

3 Hybrid PoA Protocol

We propose a proof of assets protocol Π_{PoA} for a hybrid-state blockchain, specifically the Aztec Network. Π_{PoA} consists of four entities: the *Aztec node*, a *trusted third party*, the *prover* (cryptocurrency exchange or custodian) $\mathcal{P}$, and the *verifier* $\mathcal{V}$. The Aztec node is responsible for providing authentic blockchain data, such as the Note Hash Tree, the Nullifier Tree, and transaction data. The trusted third party (TTP) generates public parameters for the UltraHonk proving system. After generating the public parameters, TTP plays no further role. The prover, controlling one or more Aztec accounts, proves the total asset value of AIP-20 tokens, without revealing account addresses or individual balances. The verifier checks the proof, given access to the public blockchain state.

3.1 Protocol Prerequisites

Let $\{\mathsf{acc}^{(a)}\}_{a=1}^{A}$ denote the set of accounts controlled by the prover. Each account has its associated keys and addresses, detailed in the background.

Let a_{token} be the address of an AIP-20 token targeted for this protocol. In the token contract, each account is mapped to a public balance $\mathsf{b}_{\mathsf{pub}}$ and a private balance $\mathsf{b}_{\mathsf{pri}}$. The private balance $\mathsf{b}_{\mathsf{pri}}$ of an account $\mathsf{acc}^{(a)}$ is the sum of consumable notes $\{\mathsf{n}^{(a,n)}\}_{n=1}^{N_a}$. For each account, the total balance is defined as $\mathsf{b}_{acc} = \mathsf{b}_{\mathsf{pub}} + \mathsf{b}_{\mathsf{pri}}$. The prover's total assets are $\mathsf{Assets} = \sum_{a=1}^{A} \mathsf{b}_{acc}^{(a)}$, i.e., the sum of all notes and public balances across all accounts.

The blockchain node maintains the following public states: Note Hash Tree (NHT), Nullifier Tree (NT), and Public Data Tree (PDT). We denote the root hash of a tree T as T_{root}, for example, $\mathsf{NHT}_{\mathsf{root}}$ represents the root of NHT. For some leaf ℓ of a tree T, $\mathsf{MemberProof}(\ell, T)$ denotes a membership proof. And for some leaf ℓ' of a tree T', $\mathsf{NonMemberProof}(\ell', T')$ denotes a non-membership proof. These proofs consist of the index of the leaf and the sibling paths from the leaf to the root. They can be verified by $\mathsf{MemberVerify}\big(\ell, T_{\mathsf{root}}, \mathsf{MemberProof}(\ell, T)\big)$ and $\mathsf{NonMemberVerify}\big(\ell', T'_{\mathsf{root}}, \mathsf{NonMemberProof}(\ell', T')\big)$, respectively.

3.2 Protocol Overview

We first present an overview of Π_{PoA}. The goal of the prover is to prove the total assets Assets of a token with address a_{token}, without revealing individual account addresses $\{\mathsf{a}_{acc}^{(a)}\}_{a=1}^{A}$, the public balance values $\{\mathsf{p}_{\mathsf{val}}^{(a)}\}_{a=1}^{A}$, or the private notes $\{\mathsf{n}^{(a,n)}\}_{n=1}^{N_a}$ associated with each account $\mathsf{a}_{acc}^{(a)}$. The verifier, on the other hand, only has access to a succinct representation of the blockchain state, namely the tree roots $\mathsf{NHT}_{\mathsf{root}}$, $\mathsf{NT}_{\mathsf{root}}$, and $\mathsf{PDT}_{\mathsf{root}}$, as well as the mapping slots $\mathsf{slot}_{\mathsf{priMap}}$ and $\mathsf{slot}_{\mathsf{pubMap}}$ of the token. Hence, the public input x_{PoA} to Π_{PoA} is:

$$x_{\mathsf{PoA}} = \left[\mathsf{Assets}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}\right].$$

From a high level, the prover demonstrates knowledge of the notes, public balances, and account addresses that make up Assets in the following steps:

1. **Public Balance Validity**: Prove the inclusion of the Public Data Tree leaf $(\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}})$ in PDT corresponding to the public balance by providing the relevant member proof $\mathsf{MemberProof}((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT})$.
2. **Note Validity**: Prove inclusion of each note commitment $\mathsf{n}_{\mathsf{uhash}}$ in NHT and non-inclusion of its corresponding nullifier ρ in NT by providing the member proof $\mathsf{MemberProof}(\mathsf{n}_{\mathsf{uhash}}, \mathsf{NHT})$ and $\mathsf{NonMemberProof}(\rho, \mathsf{NT})$, respectively.
3. **Account Ownership**: Prove ownership of the account used in 1 and 2 by providing the secret key. Here, we use the master nullifier secret key $\mathsf{nsk}_{\mathsf{m}}$ instead of the core master secret key sk. This simplifies the design, and since $\mathsf{nsk}_{\mathsf{m}}$ is considered private and derived from sk, it is a safe assumption.
4. **Note and Account Uniqueness**: Prove that no duplicate notes and accounts exist by enforcing strict ordering by their Note Hash Tree leaf index and account address, respectively.
5. **Total assets**: Prove the correct computation of $\mathsf{Assets} = \mathsf{b}_{\mathsf{pri}} + \mathsf{b}_{\mathsf{pub}}$, where $\mathsf{b}_{\mathsf{pri}}$ is the sum of all notes.

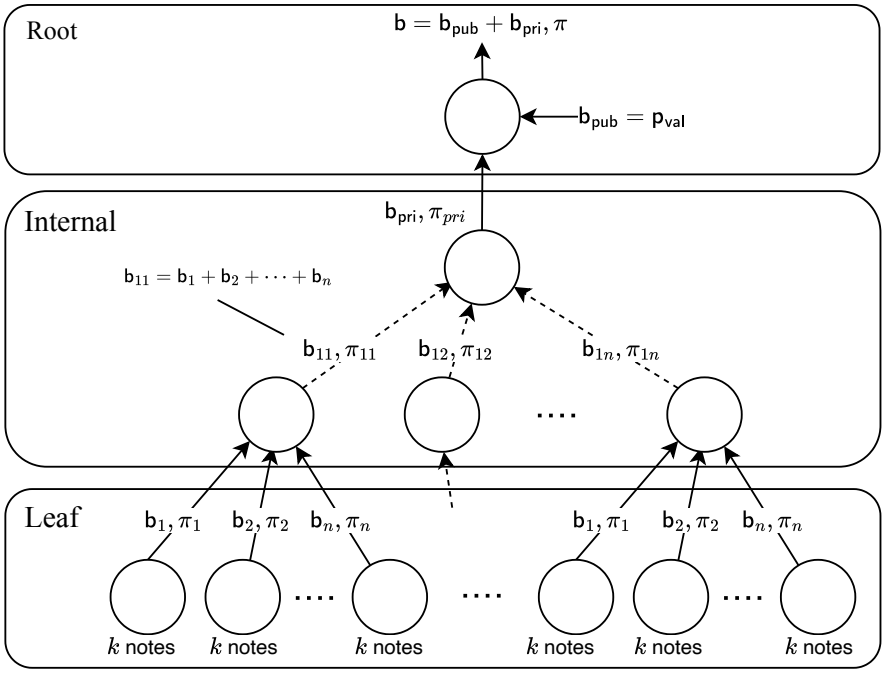

Fig. 1. Hierarchical PCD for Account Tree

We formalize the above description using NP relation $\mathcal{R}_{\mathsf{PoA}}(x_{\mathsf{PoA}}, w_{\mathsf{PoA}})$ where the witness w_{PoA} consists of the private notes, public balance, secret keys, and other auxiliary proofs:

$$w_{\mathsf{PoA}} = \left[\left\{ \begin{array}{c} \left\{ \begin{array}{c} \mathsf{n}^{(a,n)}, \mathsf{n}^{(a,n)}_{\mathsf{nonce}}, \mathsf{MemberProof}\big(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{NHT}\big), \\ \mathsf{NonMemberProof}\big(\rho^{(a,n)}, \mathsf{NT}\big) \end{array} \right\}_{n=1}^{N_a}, \\ \mathsf{a}^{(a)}_{\mathsf{par}}, \mathsf{nsk}^{(a)}_{\mathsf{m}}, \mathsf{Ivpk}^{(a)}_{\mathsf{m}}, \mathsf{Ovpk}^{(a)}_{\mathsf{m}}, \mathsf{Tpk}^{(a)}_{\mathsf{m}}, \mathsf{p}^{(a)}_{\mathsf{val}}, \\ \mathsf{MemberProof}\big((\mathsf{p}^{(a)}_{\mathsf{key}}, \mathsf{p}^{(a)}_{\mathsf{val}}), \mathsf{PDT}\big) \end{array} \right\}_{a=1}^{A} \right].$$

The full description of $\mathcal{R}_{\mathsf{PoA}}$ using pseudocode is described in section D.

If the number of accounts (A) and notes (N_a) is relatively small and is pre-determined, then $\mathcal{R}_{\mathsf{PoA}}$ can be compiled into a single zk-SNARK circuit, and the proof can be generated in a realistic time. However, since the size of the circuit grows linearly with A and N_a, when A or N_a is large, the prover would not be able to generate proofs in a reasonable time. Furthermore, since SNARK circuits require fixed input sizes, the value of A and N_a must be determined at compile time, making it impractical to use.

To overcome these limitations, we decompose the relation into smaller sub-relations and then recursively aggregate the proofs into a single, succinct proof. This is achieved using Proof-Carrying Data (PCD), which enables scalable proof composition while preserving succinct verification.

3.3 Π_{PoA} via PCD

The protocol is structured in two layers of trees that together certify the custodian's total assets:

- *Account Tree*: Aggregate all private notes and the public balance of a single account, ensuring the validity and the uniqueness of each note, the correctness of the public balance, and the ownership of the account (see Figure 1).
- *Custodian Tree*: Aggregate multiple Account Trees and sum their balances into a single value, ensuring each account is unique.

The final output of the protocol is the root of the Custodian Tree, which is the custodian's aggregated balance (i.e., Assets) along with its proof. It proves that every underlying note, public balance, and account

ownership is valid, and the total assets are computed correctly. Below, we describe each node in both trees using notation from PCD. A detailed description and the following sub-protocols are defined in the section E: $\mathsf{CompNoteHash}$, $\mathsf{CompNullifer}$, $\mathsf{CompNskApp}$, $\mathsf{CompSlot}$, $\mathsf{CompPublicDataKey}$.

Account Tree Leaf (ATL). Prove the validity and the uniqueness of K ($\leq N_a$) notes and the correctness of their total value. It takes the following messages:

$$z = \left[\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}, \mathsf{b}_{node}\right];$$
$$z_{\mathsf{loc}} = \left[\left\{\begin{matrix}\mathsf{n}^{(k)}, \mathsf{n}^{(k)}_{\mathsf{nonce}}, \mathsf{MemberProof}(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT}) \\ \mathsf{NonMemberProof}(\rho^{(k)}, \mathsf{NT})\end{matrix}\right\}_{k=1}^{K}\right];$$
$$\forall m \in [M],\ z_m = \perp.$$

For each note, it recomputes the note commitment $\mathsf{n}^{(k)}_{\mathsf{uhash}} \leftarrow \mathsf{CompNoteHash}(\mathsf{n}^{(k)}, \mathsf{n}^{(k)}_{\mathsf{nonce}}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}})$ and checks inclusion in NHT using the membership proof $\mathsf{MemberProof}(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT})$ against $\mathsf{NHT}_{\mathsf{root}}$. Each note owner $\mathsf{n}^{(k)}_{\mathsf{owner}}$ must match the account address a_{acc}. It then computes the corresponding nullifier $\rho^{(k)} \leftarrow \mathsf{CompNullifer}(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{nsk}_{\mathsf{app}})$, and verifies non-membership in NT using $\mathsf{NonMemberProof}(\rho^{(k)}, \mathsf{NT})$ against $\mathsf{NT}_{\mathsf{root}}$. The private balance slot $\mathsf{slot}_{\mathsf{pri}}$ and the app-siloed nullifier secret key $\mathsf{nsk}_{\mathsf{app}}$ are treated as Account Tree global parameters and propagated to the root for verification. To prevent duplicate valid notes, it enforces note ordering by their NHT leaf indices: $\mathsf{n}^{(k-1)}_{\mathsf{idx}} < \mathsf{n}^{(k)}_{\mathsf{idx}}$. Finally, it checks that the node balance b_{node} equals $\sum_{k=1}^{K} \mathsf{n}^{(k)}_{\mathsf{val}}$, and that the index range of the note hash $(\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}})$ matches $(\mathsf{n}^{(1)}_{\mathsf{idx}}, \mathsf{n}^{(K)}_{\mathsf{idx}})$.

Account Tree Internal (ATI). Prove the correct aggregation of M child nodes (ATL or ATI nodes) and the enforcement of account-wise note ordering. It takes the following messages:

$$z = \left[\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}, \mathsf{b}_{node}\right];$$
$$z_{\mathsf{loc}} = \perp; \quad \forall m \in [M],\ z_m = \begin{bmatrix}\mathsf{a}^{(m)}_{acc}, \mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pri}}, \mathsf{nsk}^{(m)}_{\mathsf{app}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \\ \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{idx}^{(m)}_{\mathsf{min}}, \mathsf{idx}^{(m)}_{\mathsf{max}}, \mathsf{b}^{(m)}_{node}\end{bmatrix}.$$

It checks that all child nodes share the same global parameters $(\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}})$. It then enforces the note ordering: $\forall m \in [M]$, $\mathsf{idx}^{(m-1)}_{\mathsf{max}} < \mathsf{idx}^{(m)}_{\mathsf{min}}$. Finally, it checks correctness of the aggregated balance $\mathsf{b}_{node} = \sum_{m=1}^{M} \mathsf{b}^{(m)}_{node}$ and the index range $(\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}) = (\mathsf{idx}^{(1)}_{\mathsf{min}}, \mathsf{idx}^{(M)}_{\mathsf{max}})$.

Account Tree Root (ATR). Prove the account ownership, the validity of the public balance, and the correctness of the total account balance. It takes the following messages:

$$z = \left[\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{b}_{acc}\right];$$
$$z_{\mathsf{loc}} = \left[\mathsf{a}_{\mathsf{par}}, \mathsf{nsk}_{\mathsf{m}}, \mathsf{Ivpk}_{\mathsf{m}}, \mathsf{Ovpk}_{\mathsf{m}}, \mathsf{Tpk}_{\mathsf{m}}, \mathsf{p}_{\mathsf{val}}, \mathsf{MemberProof}\left((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT}\right)\right];$$
$$z_{\mathsf{in}} = \left[\mathsf{a}^{(\mathsf{in})}_{acc}, \mathsf{a}^{(\mathsf{in})}_{token}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{pri}}, \mathsf{nsk}^{(\mathsf{in})}_{\mathsf{app}}, \mathsf{NHT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{NT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{idx}^{(\mathsf{in})}_{\mathsf{min}}, \mathsf{idx}^{(\mathsf{in})}_{\mathsf{max}}, \mathsf{b}^{(\mathsf{in})}_{node}\right].$$

First, it checks the consistency of $\mathsf{a}^{(\mathsf{in})}_{acc}$, $\mathsf{nsk}^{(\mathsf{in})}_{\mathsf{app}}$, and $\mathsf{slot}^{(\mathsf{in})}_{\mathsf{pri}}$ across the Account Tree. It derives the master nullifier public key $\mathsf{Npk}_{\mathsf{m}}$ from the master nullifier secret key $\mathsf{nsk}_{\mathsf{m}}$, computes the account address a_{acc} from the public keys $(\mathsf{Npk}_{\mathsf{m}}, \mathsf{Ivpk}_{\mathsf{m}}, \mathsf{Ovpk}_{\mathsf{m}}, \mathsf{Tpk}_{\mathsf{m}})$ and the partial address $\mathsf{a}_{\mathsf{par}}$, and verifies that $\mathsf{a}^{(\mathsf{in})}_{acc}$ matches the computed address. It also checks that $\mathsf{slot}^{(\mathsf{in})}_{\mathsf{priMap}} = \mathsf{CompSlot}(\mathsf{slot}_{\mathsf{priMap}}, \mathsf{a}_{acc})$ and $\mathsf{nsk}^{(\mathsf{in})}_{\mathsf{app}} = \mathsf{CompNskApp}(\mathsf{nsk}_{\mathsf{m}}, \mathsf{a}_{token})$. Since these parameters are all derived from $\mathsf{nsk}_{\mathsf{m}}$, providing the correct $\mathsf{nsk}_{\mathsf{m}}$ ensures the consistency of a_{acc}, $\mathsf{slot}_{\mathsf{pri}}$, and $\mathsf{nsk}_{\mathsf{app}}$ across the Account Tree.
Second, it checks that the child node uses the provided public parameters $(\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}})$. Third, it verifies the public balance by deriving the public balance slot $\mathsf{slot}_{\mathsf{pub}} \leftarrow \mathsf{CompSlot}(\mathsf{slot}_{\mathsf{pubMap}}, \mathsf{a}_{acc})$ and the public data key $\mathsf{p}_{\mathsf{key}} \leftarrow \mathsf{CompPublicDataKey}(\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pub}})$. It then checks the inclusion of $(\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}})$ in PDT using $\mathsf{MemberProof}\big((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT}\big)$ against $\mathsf{PDT}_{\mathsf{root}}$. Finally, it checks that the account balance b_{acc} equals $\mathsf{b}^{(\mathsf{in})}_{node} + \mathsf{p}_{\mathsf{val}}$, where $\mathsf{b}^{(\mathsf{in})}_{node}$ is the private balance of the account.

Custodian Tree Leaf (CTL). Prove the correct aggregation and the uniqueness of M Account Trees. It takes the following messages:

$$z = \left[\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{a}_{\mathsf{min}}, \mathsf{a}_{\mathsf{max}}, \mathsf{b}_{node}\right];$$
$$z_{\mathsf{loc}} = \bot; \quad \forall m \in [M],\ z_m = \begin{bmatrix} \mathsf{a}_{acc}^{(m)}, \mathsf{a}_{token}^{(m)}, \mathsf{slot}_{\mathsf{pubMap}}^{(m)}, \mathsf{slot}_{\mathsf{priMap}}^{(m)}, \\ \mathsf{NHT}_{\mathsf{root}}^{(m)}, \mathsf{NT}_{\mathsf{root}}^{(m)}, \mathsf{PDT}_{\mathsf{root}}^{(m)}, \mathsf{b}_{acc}^{(m)} \end{bmatrix}.$$

It checks that all account trees share the same global parameters ($\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$). It then enforces account ordering by their addresses: $\forall m \in [M],\ \mathsf{a}_{acc}^{(m-1)} < \mathsf{a}_{acc}^{(m)}$. Finally, it verifies the correctness of the aggregated balance $\mathsf{b}_{node} = \sum_{m=1}^{M} \mathsf{b}_{acc}^{(m)}$, and the account address range ($\mathsf{a}_{\mathsf{min}}$, $\mathsf{a}_{\mathsf{max}}$) = ($\mathsf{a}^{(1)}, \mathsf{a}^{(M)}$).

Custodian Tree Internal (CTI). Prove the correct aggregation of M child nodes (CTL or CTI nodes) and enforce the ordering of addresses. The logic is analogous to the ATI node and is omitted due to space constraints.

Custodian Tree Root (CTR). Prove the custodian's total assets. It takes the following messages:

$$z = \left[\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{Assets}\right];$$
$$z_{\mathsf{loc}} = \bot; \quad z_{\mathsf{in}} = \begin{bmatrix} \mathsf{a}_{token}^{(\mathsf{in})}, \mathsf{slot}_{\mathsf{pubMap}}^{(\mathsf{in})}, \mathsf{slot}_{\mathsf{priMap}}^{(\mathsf{in})}, \mathsf{a}_{\mathsf{min}}^{(\mathsf{in})}, \mathsf{a}_{\mathsf{max}}^{(\mathsf{in})}, \\ \mathsf{NHT}_{\mathsf{root}}^{(\mathsf{in})}, \mathsf{NT}_{\mathsf{root}}^{(\mathsf{in})}, \mathsf{PDT}_{\mathsf{root}}^{(\mathsf{in})}, \mathsf{b}_{node}^{(\mathsf{in})} \end{bmatrix}.$$

It checks that the child node uses the provided public parameters ($\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}$, $\mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$). Finally, it verifies that the provided custodian's total assets match the aggregated balance: $\mathsf{Assets} = \mathsf{b}_{node}^{(\mathsf{in})}$.

3.4 The Flow of Π_{PoA}

Public Parameters Setup. TTP generates the public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$.

Key Generation. Prover runs the key generation algorithm to obtain the proving key and the verification key $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \varphi)$ for each predicates $(\varphi_{\mathsf{ATL}}, \varphi_{\mathsf{ATI}}, \varphi_{\mathsf{ATR}}, \varphi_{\mathsf{CTL}}, \varphi_{\mathsf{CTI}}, \varphi_{\mathsf{CTR}})$. The keys are: $\mathsf{pk} = (\mathsf{pk}_{\mathsf{ATL}}, \mathsf{pk}_{\mathsf{ATI}}, \mathsf{pk}_{\mathsf{ATR}}, \mathsf{pk}_{\mathsf{CTL}}, \mathsf{pk}_{\mathsf{CTI}}, \mathsf{pk}_{\mathsf{CTR}})$, $\mathsf{vk} = (\mathsf{vk}_{\mathsf{ATL}}, \mathsf{vk}_{\mathsf{ATI}}, \mathsf{vk}_{\mathsf{ATR}}, \mathsf{vk}_{\mathsf{CTL}}, \mathsf{vk}_{\mathsf{CTI}}, \mathsf{vk}_{\mathsf{CTR}})$.

Instance and Witness Setup. Prover takes a snapshot S of the blockchain at a chosen block height. From S, the prover extracts the root hashes $\mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$ and public data values $\{\mathsf{p}_{\mathsf{val}}^{(a)}\}_{a=1}^{A}$. For each account $\{\mathsf{acc}^{(a)}\}_{a=1}^{A}$, the prover queries its PXE to obtain notes $\{\mathsf{n}\}_{n=1}^{N_a}$, the partial address $\mathsf{a}_{\mathsf{par}}^{(a)}$, the master nullifier secret key $\mathsf{nsk}_{\mathsf{m}}^{(a)}$, and the public keys $(\mathsf{Ivpk}_{\mathsf{m}}^{(a)}, \mathsf{Ovpk}_{\mathsf{m}}^{(a)}, \mathsf{Tpk}_{\mathsf{m}}^{(a)})$. Then, the prover builds the membership proofs for note hashes, non-membership proofs for nullifiers, and membership proofs for public data. Finally, the prover computes the total assets Assets.

Proof Generation. For each account $\mathsf{a}_{acc}^{(a)}$, the prover sorts the associated notes by leaf index and groups them into batches of size K, each corresponding to a leaf node in the Account Tree. For each batch, it computes the node balance b_{node} and index range ($\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}$), and generates a proof for φ_{ATL} using $\mathsf{pk}_{\mathsf{ATL}}$. These steps can be executed in parallel.

The prover recursively aggregates the Account Tree leaf nodes, M at a time. At each step, it computes the aggregated balance b_{node}, sets the index range ($\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}$), and generates a proof for φ_{ATI} using $\mathsf{pk}_{\mathsf{ATI}}$ together with $\mathsf{vk}_{\mathsf{ATL}}$ and $\mathsf{vk}_{\mathsf{ATI}}$ for recursive verification. When the aggregation reaches the top of the Account Tree, the prover computes the account balance b_{acc} as the sum of the aggregated private balance and the public balance. It generates a proof for φ_{ATR} using $\mathsf{pk}_{\mathsf{ATR}}$ and $\mathsf{vk}_{\mathsf{ATI}}$.

Account Trees for different accounts are constructed independently. The prover then sorts all accounts by address, groups them into batches of size M, where each batch corresponds to a leaf node in the Custodian Tree. Custodian Tree leaves work similarly to the Account Tree: the prover aggregates M child nodes at a time, enforces address ordering, and generates proofs for φ_{CTL} and φ_{CTI}. At the Custodian Tree root, the prover computes the total assets Assets and produces the final proof π_{CTR} for φ_{CTR}. Finally, the prover outputs $(x_{\mathsf{PoA}}, \pi_{\mathsf{PoA}}, \mathsf{vk}_{\mathsf{PoA}})$, where $\pi_{\mathsf{PoA}} = \pi_{\mathsf{CTR}}$ and $\mathsf{vk}_{\mathsf{PoA}} = \mathsf{vk}_{\mathsf{CTR}}$.

Proof Verification. The verifier first validates $\mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$ by querying the Aztec node. It also checks the correctness of $\mathsf{slot}_{\mathsf{pubMap}}$ and $\mathsf{slot}_{\mathsf{priMap}}$ of the token a_{token} by checking the token contract. Finally, the verifier checks the validity of the proof π_{PoA} by $0/1 \leftarrow \mathsf{Verify}(\mathsf{vk}_{\mathsf{PoA}}, x_{\mathsf{PoA}}, \pi_{\mathsf{PoA}})$.

4 Security Analysis

We show that any accepting proof guarantees the correctness of the claimed total assets Assets, the correctness of public balances, and notes, the ownership of accounts, and that no duplicate accounts or notes exist. We also discuss that the proof does not reveal anything beyond the public inputs x_{PoA}.

Assumptions. We assume that the proving system instantiating the PCD framework is complete, knowledge-sound, and zero-knowledge, as defined in section B. In particular, with respect to knowledge soundness, there exists an extractor that, given an accepting PCD proof, extracts the local data and incoming messages. Since the final PCD proof π_{CTR} verifies all previous nodes, the extractor can recursively obtain the local data and incoming messages for the nodes.

We further assume that both the prover and verifier obtain a valid block of data from an Aztec node, including the tree roots $\mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$ and the mapping slots $\mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}$. Finally, we assume that the public parameters pp of the proving system are generated securely by the TTP.

Asset correctness. If $\mathsf{Verify}(\mathsf{vk}_{\mathsf{PoA}}, x_{\mathsf{PoA}}, \pi_{\mathsf{PoA}}) = 1$, where $\mathsf{vk}_{\mathsf{PoA}} = \mathsf{vk}_{\mathsf{CTR}}, x_{\mathsf{PoA}} = x_{\mathsf{CTR}}, \pi_{\mathsf{PoA}} = \pi_{\mathsf{CTR}}$, there exists an extractor that extracts the incoming message of CTR. The message contains the child node's balance, b_{node}, which equals the claimed total assets, Assets. Applying the extraction recursively from CTR to ATL, we obtain all notes and public balances for each account. The internal nodes ensure that every node's balance equals the sum of its children's balances. Therefore, the sum of all extracted notes and public balances must match Assets.

Public Balance Correctness. At every ATR node, it derives the public data key $\mathsf{p}_{\mathsf{key}}$ from the account address a_{acc}, the token address a_{token}, and the mapping slot $\mathsf{slot}_{\mathsf{pubMap}}$, and verifies the inclusion of $(\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}})$ in PDT. Hence, this enforces the inclusion of the extracted $\mathsf{p}_{\mathsf{val}}$ in PDT, thereby binding the value of public balance $\mathsf{b}_{\mathsf{pub}}$ with $\mathsf{PDT}_{\mathsf{root}}$.

Note Correctness. At every ATL node, it derives the note commitment $\mathsf{n}_{\mathsf{uhash}}$ from note n, note nonce $\mathsf{n}_{\mathsf{nonce}}$, a_{token}, and private balance slot $\mathsf{slot}_{\mathsf{priMap}}$ and checks its membership in NHT. It also computes the nullifier ρ from $\mathsf{n}_{\mathsf{uhash}}$ and app-siloed nullifier secret key $\mathsf{nsk}_{\mathsf{app}}$, and verifies the non-inclusion in NT. Hence, each extracted note is included in NHT, and its corresponding nullifier is not present in NT. ATI nodes aggregates the note values, therefore the private balance $\mathsf{b}_{\mathsf{pri}}$ at ATR equals the sum of valid notes.

Account Ownership. At each ATR node, it checks that a_{acc} and $\mathsf{nsk}_{\mathsf{app}}$ are derived correctly from the master nullifier secret key $\mathsf{nsk}_{\mathsf{m}}$, and are used across the Account Tree. It also verifies that $\mathsf{slot}_{\mathsf{pri}}$ corresponds to a_{acc} and $\mathsf{slot}_{\mathsf{priMap}}$. Hence, an accepting proof implies that the prover knows $\mathsf{nsk}_{\mathsf{m}}$ that is consistent with a_{acc} and $\mathsf{nsk}_{\mathsf{app}}$, which proves the ownership of the included accounts.

Inflation Resistance. In the Account Tree, ATL and ATI nodes enforce the ordering of notes by their leaf index, preventing duplicate counting of notes. In the Custodian Tree, CTL and CTI nodes enforce the ordering of accounts by their addresses, preventing duplicate counting of account balances. Hence, every note and every account is counted only once when computing Assets.

Privacy. By the zero-knowledge property of the underlying zk-SNARK, there exists a simulator that, given only the public input x_{PoA}, can generate a proof indistinguishable from the real proof. Therefore, the verifier learn only Assets, the token address a_{token}, the mapping slot $\mathsf{slot}_{\mathsf{pubMap}}$ and $\mathsf{slot}_{\mathsf{priMap}}$, and the tree roots $\mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}$. All other inputs remain hidden, including account address, individual balances, note contents, nullifiers, Merkle paths, and secret keys. Although some of these values, such as $\mathsf{nsk}_{\mathsf{app}}, \mathsf{a}_{acc}$, are part of the outgoing messages between PCD nodes, they are not exposed to the external output. Therefore, the protocol reveals only the claimed total assets while preserving the privacy of accounts, balances, and notes.

5 Implementation and Performance Evaluation

We implemented Hybrid PoA using Noir DSL and UltraHonk proving backend developed by Aztec Labs.[3] Noir is a domain-specific language (DSL) for writing SNARK circuits, and UltraHonk is a PLONK-based proof system that supports recursive proof verification over the AltBN254 and Grumpkin elliptic curves.

The implementation follows the protocol described in section 3. It consists of circuits for the leaf, internal, and root nodes of both the Account Tree and the Custodian Tree. In total, the implementation consists of

[3] Source code of Hybrid PoA: `https://github.com/rknhr-uec/hybrid-poa_prod`

Table 2. Performance of Hybrid PoA. n denotes the number of accounts, where each has 112 notes and some public balance. PT denotes the proving time, PS denotes the proof size, and VT denotes the verification time. Note that PS and VS include non-snark-related processes such as reading/writing cache files, fetching public data from the Aztec node, parsing data, etc.

n	PT (s)	PS (KB)	VT (ms)
1	92	16.10	83
10	257	16.10	82
50	1137	16.10	81
100	2229	16.10	82
1000	21716	16.10	83

Table 3. Proving time (PT) and proof size (PS) of Account Tree consisting of n notes.

n	PT (s)	PS (KB)
32	33.26	14.48
100	37.08	14.48
500	94.74	14.48
1000	186.71	14.48

Table 4. Proving time (PT), peak memory usage (PM) during proving, and circuit gate count for each node in the Account Tree (AT) and Custodian Tree (CT). Note that PM is an approximate measurement.

	PT (s)	PM (GB)	Gates (#)
AT Leaf	3.2	0.86	257,779
AT Internal	24.1	11.0	2,853,242
AT Root	6.1	2.0	671,958
CT Leaf	23.8	10.0	2,836,699
CT Internal	24.0	11.0	2,853,574
CT Root	7.8	2.0	666,022

approximately 1,300 lines of Noir code. We did not apply manual circuit-level optimizations, as our goal is to evaluate the baseline cost and behavior of the construction.

5.1 Experiment Setup

All experiments were conducted on a 2019 Mac Pro with a 3.2 GHz 16-core Intel Xeon W processor and 768 GB of DDR4 memory.

We fix the parameters of the hierarchical aggregation structure throughout the evaluation. For the Account Tree, each leaf aggregates up to $k = 32$ notes from a single account, and each internal node aggregates $m = 4$ child nodes. The Custodian leaf aggregates $m = 4$ Account Tree roots, and internal nodes aggregate $m = 4$ child nodes. These parameters were chosen to balance circuit size, recursion depth, and available computational resources.

As for concurrency, four Account Tree leaves, four Custodian Tree leaves, and two internal node proofs were processed in parallel. This configuration was selected to match the server's CPU core count, assigning at least one core to each process. Although the hierarchical structure allows proof generation for different subtrees, all reported results are evaluated on a single machine. Multiple proof-generation processes ran concurrently on the same machine to reflect practical usage, but we did not evaluate distributed proving across multiple machines.

5.2 Results

Table 2 shows the proving and verification performance for a range of accounts ($n = 1$ to 1000), where each account holds 112 notes. Table 3 represents the proving and verification performance of a single Account Tree for a range of notes ($n = 32$ to 1000). Finally, Table 4 summarizes the proving time, peak memory usage, and gate count in the Account Tree and Custodian Tree.

5.3 Discussion

Across all evaluated instances, the proof size and verification time remain constant at approximately 16.10 KB and 83 ms, respectively, independent of the number of accounts and notes. In contrast, proving time increases with the size of the instance. While Hybrid PoA is designed to be parallelizable and should theoretically scale

logarithmically, Table 2 shows roughly linear scaling with the number of accounts n because we only partially parallelized it due to limited computational resources. We believe that full parallelization across multiple high-performance machines could lead to logarithmic proving time.

The primary prover bottleneck is the recursive proof aggregation. As shown in Table 4, proving the correctness of 32 notes takes 3.2 seconds, whereas aggregating four child nodes took 24.1 seconds and consumes more memory.

Generating a proof for 1,000 accounts, each holding 112 notes, took roughly 6 hours. Given that many cryptocurrency exchanges publish their proof-of-reserves reports daily to monthly, this cost is practically acceptable. For context, MProve-Nova [12], which uses Nova-IVC, required about 7 hours to generate a proof for 10,000 outputs, whereas our system produced a proof for 112,000 outputs (1,000 accounts $\times$ 112 notes) in 6 hours, which is around 11 times improvement. Note that this comparison depends on the hardware, implementation details, and proving backend, but it suggests that hierarchical aggregation with recursive SNARKs can be competitive in practice.

6 Conclusion

We introduced a Proof of Assets (PoA) protocol for hybrid-state blockchains, specifically the Aztec Network. Our construction employs hierarchical proof-carrying data via recursive zk-SNARKs to aggregate balances across multiple accounts, enabling a scalable prover. Regardless of the number of accounts or notes, the protocol can produce proofs of constant size with fast verification.

There remains room for improvement. Although proof generation can be parallelized, the underlying proving system uses elliptic-curve operations, which are a bottleneck when verified inside an SNARK. One solution is to replace the recursive zk-SNARK with a scheme that relies only on finite-field arithmetic, such as an FRI-based SNARK, and use the fast verifier at the final step.

Acknowledgments. I would like to thank Aztec Labs for their support in addressing technical questions on their Discord server and during office hours. This work was supported by JSPS KAKENHI Grant Numbers JP23H00468, JP23H00479, JP23K17455, JP23K21644, JP23K24846, and JST CREST JPMJCR23M2.

References

1. Aztec Labs: Aztec documentation. https://docs.aztec.network/
2. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014. pp. 276–294. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
3. Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12825, pp. 681–710. Springer (2021). https://doi.org/10.1007/978-3-030-84242-0_24, https://doi.org/10.1007/978-3-030-84242-0_24
4. Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: Yao, A.C. (ed.) Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings. pp. 310–331. Tsinghua University Press (2010), http://conference.iiis.tsinghua.edu.cn/ICS2010/content/papers/25.html
5. Conley, T., Diaz, N., Espada, D., Kuruvilla, A., Mayne, S., Fu, X.: Izpr: Instant zero knowledge proof of reserve. In: Financial Cryptography and Data Security. FC 2024 International Workshops. pp. 225–239. Springer Nature Switzerland, Cham (2025)
6. Dagher, G.G., Bünz, B., Bonneau, J., Clark, J., Boneh, D.: Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 720–731. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813674, https://doi.org/10.1145/2810103.2813674
7. Dutta, A., Bagad, S., Vijayakumaran, S.: Mprove+: Privacy enhancing proof of reserves protocol for monero. IEEE Transactions on Information Forensics and Security **16**, 3900–3915 (2021). https://doi.org/10.1109/TIFS.2021.3088035
8. Dutta, A., Vijayakumaran, S.: Mprove: A proof of reserves protocol for monero exchanges. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 330–339 (2019). https://doi.org/10.1109/EuroSPW.2019.00043

9. Grassi, L., Khovratovich, D., Schofnegger, M.: Poseidon2: A faster version of the poseidon hash function. In: International Conference on Cryptology in Africa. pp. 177–203. Springer (2023)
10. Ozdemir, A., Boneh, D.: Experimenting with collaborative zk-snarks: Zero-knowledge proofs for distributed secrets. In: Butler, K.R.B., Thomas, K. (eds.) 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022. pp. 4291–4308. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/ozdemir
11. Reddy, S.B.: A zk-snark based proof of assets protocol for bitcoin exchanges. In: 2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS). pp. 135–140 (2023). https://doi.org/10.1109/COMSNETS56262.2023.10041328
12. Thakore, V., Vijayakumaran, S.: Mprove-nova: A privacy-preserving proof of reserves protocol for monero. Proceedings on Privacy Enhancing Technologies (2025)
13. Tzialla, I., Kothapalli, A., Parno, B., Setty, S.T.V.: Transparency dictionaries with succinct proofs of correct operation. In: 29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022. The Internet Society (2022), https://www.ndss-symposium.org/ndss-paper/auto-draft-252/
14. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Canetti, R. (ed.) Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Lecture Notes in Computer Science, vol. 4948, pp. 1–18. Springer (2008). https://doi.org/10.1007/978-3-540-78524-8_1, https://doi.org/10.1007/978-3-540-78524-8_1
15. wonderland: Request for comments: Aip-20: Aztec token standard. https://forum.aztec.network/t/request-for-comments-aip-20-aztec-token-standard/7737
16. Zhou, Z., Zhang, Z., Dong, J.: Proof-carrying data from multi-folding schemes. IACR Cryptol. ePrint Arch. p. 1282 (2023), https://eprint.iacr.org/2023/1282

A The Aztec Network

Aztec is a Layer 2 blockchain on Ethereum that supports both private and public state, as well as private and public smart contract execution. Private functions are executed on the client side in the *Private Execution Environment* (PXE). In contrast, public functions are executed by the *sequencers* in the *Aztec Virtual Machine* (AVM). On top of executing private functions, the PXE is responsible for holding user private keys, managing private states, and generating proofs for the correctness of private execution. On the other hand, the AVM is responsible for verifying client proofs, managing public states, and rolling up transactions into blocks, which are then submitted to the Layer 1 blockchain for verification.

A.1 Notations and Preliminaries

Aztec operates over two elliptic curves, AltBN254 and Grumpkin, which together form a cycle of curves. Let $\mathbb{F}_r$ denote the scalar field of AltBN254 (i.e., the Grumpkin base field), and let $\mathbb{F}_q$ denote the base field of AltBN254 (i.e., the Grumpkin scalar field). Note that for these curves, $r < q$. We denote by $\mathbb{G}$ the elliptic curve group associated with Grumpkin, i.e., $\mathbb{G} = E(\mathbb{F}_r)$, and by $G = 1_{\mathbb{G}}$ its generator point. The function $\mathsf{Pos2} : \mathbb{F}_r^t \to \mathbb{F}_r$ denotes the Poseidon2 hash function [9], $\mathsf{SHA} : \mathbb{F}_r^t \to \mathbb{F}_r$ denotes the SHA-512 hash function, and $\mathsf{ToFq} : \mathbb{F}_r \to \mathbb{F}_q$ is the mapping from $\mathbb{F}_r$ to $\mathbb{F}_q$. We write $a \leftarrow\!\!\$\ A$ to indicate that an element is sampled uniformly at random from the set A. We use `UPPERCASE TYPEWRITER` font to denote constants. Constants prefixed with `AZ_` refer to Aztec-specific values.[4]

A.2 Merkle Trees

Aztec uses two types of binary Merkle trees: the *append-only Merkle tree* and the *indexed Merkle tree*. They both operate over the field $\mathbb{F}_r$ and use the Poseidon2 hash function $\mathsf{Pos2}$ when building the tree.

The *layer* of a node refers to its depth in the tree: leaves are at layer 0 and the root is at layer height. Internal nodes are computed recursively from their children by hashing the left child ℓ, right child r, the children's node layer d, and a constant tree identifier id: $\mathsf{Pos2}(\mathsf{id}, d, \ell, r)$. A *membership proof* for a leaf consists of the leaf index and the sibling path from the leaf to the root.

Append-only Merkle Tree In an append-only Merkle tree, new leaves are inserted in order from left to right. And once a leaf is inserted, the value is immutable and cannot be modified.

Indexed Merkle Tree First introduced in [13], an indexed Merkle tree is a binary Merkle tree over a sorted linked list and allows for efficient non-membership proofs than sparse Merkle trees. Each leaf stores a tuple $(v, v_{\mathsf{next}}, \mathsf{leaf}_{\mathsf{idx}}(v_{\mathsf{next}}))$, where v is the stored value, v_{next} is the next-highest value in the tree, and $\mathsf{leaf}_{\mathsf{idx}}(v_{\mathsf{next}})$ is the index of the leaf v_{next}. New leaves are inserted from left to right, as in the append-only tree, but the v_{next} and i_{next} fields of the existing leaves are updated to preserve the linked list structure. To prove that a value x is not in the tree, one provides a membership proof for the leaf with the value v less than x and its v_{next} greater than x.

A.3 Public and Private States

The public state operates similarly to other Ethereum Virtual Machine (EVM) based blockchains, where each node maintains a copy of the global data. These public states are organized within the *Public Data Tree*.

On the other hand, the private state works with the Unspent Transaction Output (UTXO) model, similar to Bitcoin. In Aztec, private states are represented as *notes*. Unlike Bitcoin, these notes are stored locally in the PXE, while only their commitments are recorded on-chain in the *Note Hash Tree*. The Note Hash Tree is an *append-only Merkle tree*, meaning that once a commitment is inserted, it remains immutable. To spend (or consume) a note, it must be nullified by computing a deterministic *nullifier* and inserting it into the *Nullifier Tree*. A note is spendable if its corresponding nullifier is not included in the *Nullifier Tree*, which can be proven through a non-membership proof.

[4] A complete list can be found here: `https://raw.githubusercontent.com/AztecProtocol/aztec-packages/9872ad4dcaaf85eebb7ff136ba468d99e4d63e0a/noir-projects/noir-protocol-circuits/crates/types/src/constants.nr`

Table 5. List of Keys used in Aztec

Key	Name	Derivation
sk	Master Secret Key	$\leftarrow\!\!\$\ \mathbb{F}_r$
$\mathsf{nsk_m}$	Nullifier Secret Key	$\mathsf{SHA}(\texttt{AZ_NSK_M}, \mathsf{sk}) \in \mathbb{F}_r$
$\mathsf{ovsk_m}$	Outgoing Viewing Secret Key	$\mathsf{SHA}(\texttt{AZ_OVSK_M}, \mathsf{sk})$
$\mathsf{ivsk_m}$	Incoming Viewing Secret Key	$\mathsf{SHA}(\texttt{AZ_IVSK_M}, \mathsf{sk})$
$\mathsf{tsk_m}$	Tagging Secret Key	$\mathsf{SHA}(\texttt{AZ_TSK_M}, \mathsf{sk})$
$\mathsf{Npk_m}$	Nullifier Public Key	$\mathsf{ToFq}(\mathsf{nsk_m}) \cdot G \in \mathbb{G}$
$\mathsf{Ovpk_m}$	Outgoing Viewing Public Key	$\mathsf{ToFq}(\mathsf{ovsk_m}) \cdot G$
$\mathsf{Ivpk_m}$	Incoming Viewing Public Key	$\mathsf{ToFq}(\mathsf{ivsk_m}) \cdot G$
$\mathsf{Tpk_m}$	Tagging Public Key	$\mathsf{ToFq}(\mathsf{tsk_m}) \cdot G$
$\mathsf{nsk_{app}}$	App-siloed Nullifier Secret Key	$\mathsf{Pos2}(\texttt{AZ_NSK_APP}, \mathsf{nsk_m}, \mathsf{a}_{app}) \in \mathbb{F}_r$

Keys and Addresses This section lists the keys involved in constructing the Note Hash Tree and the Nullifier Tree, summarized in Table 5. Detailed descriptions of the keys are available in the official documentation [1].

Given a set of public keys $(\mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m})$ and a partial address $\mathsf{a_{par}} \in \mathbb{F}_r$,[5] the Aztec address a of an account is computed as:

$$\begin{aligned}
\mathsf{pk_{hash}} &= \mathsf{Pos2}(\texttt{AZ_PUBKEY_HASH}, \mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m}),\\
\mathsf{a_{pre}} &= \mathsf{Pos2}(\texttt{AZ_CONTRACT_ADDRESS_V1}, \mathsf{pk_{hash}}, \mathsf{a_{par}}),\\
\mathsf{a_{point}} &= \mathsf{ToFq}(\mathsf{a_{pre}}) \cdot G + \mathsf{Ivpk_m},\\
\mathsf{a} &= (x\text{-coordinate of } \mathsf{a_{point}}) \in \mathbb{F}_r.
\end{aligned}$$

Note Hash Tree The Note Hash Tree is an append-only Merkle tree whose leaves are note commitments, each a 254-bit element of $\mathbb{F}_r$. The tree depth (or height) is fixed at 40 and the tree ID is 1. A note is typically defined as a tuple with the following fields:

- $\mathsf{n_{owner}} \in \mathbb{F}_r$: the note's owner.
- $\mathsf{n_{rnd}} \in \mathbb{F}_r$: randomness for commitment hiding.
- $\mathsf{n_{val}} \in \mathbb{F}_r$: the note's stored value.

Note has a concept of *storage slot* which is used by smart contracts to locate and manage private notes in a way similar to accessing public state. When a note is created, a unique *nonce* — derived from the transaction identifier— is assigned so that notes with identical contents yield different commitments.

Given $(\mathsf{n_{owner}}, \mathsf{n_{rnd}}, \mathsf{n_{val}})$, storage slot slot, application address a_{app}, and unique nonce $\mathsf{n_{nonce}}$, the unique siloed note commitment $\mathsf{n_{uhash}}$ is computed as:

$$\begin{aligned}
\mathsf{n_{pc}} &= \mathsf{Pos2}(\texttt{AZ_NOTE_HASH}, \mathsf{n_{owner}}, \mathsf{n_{rnd}}, \mathsf{slot}),\\
\mathsf{n_{hash}} &= \mathsf{Pos2}(\texttt{AZ_NOTE_HASH}, \mathsf{n_{pc}}, \mathsf{n_{val}}),\\
\mathsf{n_{shash}} &= \mathsf{Pos2}(\texttt{AZ_SILOED_NOTE_HASH}, \mathsf{a}_{app}, \mathsf{n_{hash}}),\\
\mathsf{n_{uhash}} &= \mathsf{Pos2}(\texttt{AZ_UNIQUE_NOTE_HASH}, \mathsf{nonce}, \mathsf{n_{shash}}).
\end{aligned}$$

The derived $\mathsf{n_{uhash}}$ is included in the transaction data and inserted into the Note Hash Tree by the sequencer.

Nullifier Tree The Nullifier Tree is an indexed Merkle tree that stores nullifier values, each a 254-bit element of $\mathbb{F}_r$. The tree depth is fixed at 40 and the tree ID is 0. It is primarily used to prove the non-existence of a nullifier, ensuring that a note has not already been consumed to prevent double-spending.

[5] The partial address is deterministically computed by hashing the account contract with a random nonce; see `https://docs.aztec.network/aztec/concepts/accounts/keys#address-keys`

Given a note commitment $\mathsf{n_{uhash}}$ and an app-siloed nullifier secret key $\mathsf{nsk_{app}}$, the siloed nullifier ρ is derived as:

$$\begin{aligned}\rho_{\mathsf{inner}} &= \mathsf{Pos2}(\texttt{AZ_NOTE_HASH}, \mathsf{n_{uhash}}, \mathsf{nsk_{app}}),\\ \rho &= \mathsf{Pos2}(\texttt{AZ_SILOED_NOTE_HASH}, \mathsf{a}_{app}, \rho_{\mathsf{inner}}).\end{aligned}$$

Since the Nullifier Tree is an indexed Merkle Tree, the leaf is a tuple: $(\rho, \rho_{\mathsf{next}}, \mathsf{idx_{next}})$ where ρ_{next} is the next nullifier and $\mathsf{idx_{next}}$ is the leaf index of the next nullifier. The leaf is inserted into the Nullifier Tree by the sequencer.

Public Data Tree The Public Data Tree is an indexed Merkle tree that stores public state as key–value pairs, where both are 254-bit elements of $\mathbb{F}_r$. The tree depth is fixed at 40 and the tree ID is 2. The key $\mathsf{p_{key}}$ corresponds to the app storage slot slot, and to prevent one contract from overwriting another contract's state, it is siloed with the app address:

$$\mathsf{p_{key}} = \mathsf{Pos2}(\texttt{AZ_PUBLIC_DATA_LEAF}, \mathsf{a}_{app}, \mathsf{slot})$$

The public data leaf of $\mathsf{p_{key}}, \mathsf{p_{val}}$ is a tuple $(\mathsf{p_{key}}, \mathsf{p_{val}}, \mathsf{p_{next_key}}, \mathsf{idx_{next}})$ where $\mathsf{p_{next_key}}$ is the next key and $\mathsf{idx_{next}}$ is the leaf index of the next key. The sequencer updates the tree during public function execution; if the key exists, the leaf is updated, otherwise a new leaf is appended.

A.4 AIP-20 Token Standard

The AIP-20 token standard [15], proposed by Wonderland, is not yet fully standardized but is widely used as the de facto token standard in Aztec. It supports the transfer of both private and public tokens.

Public balances follow the account-balance model of ERC-20 tokens on EVM-based blockchains, where there is a mapping from user address a to token balance $\mathsf{b_{pub}}$ that is maintained in public storage. The storage slot of a user balance is defined as $\mathsf{Pos2}(\mathsf{slot_{pubMap}}, \mathsf{a})$, where $\mathsf{slot_{pubMap}}$ is the logical storage slot of the public balances mapping.

Private balances follow a UTXO-based model, where the balance of a user is represented as one or many notes. The private balance is a mapping from user address a to a set of notes, and the private balance $\mathsf{b_{pri}}$ for a is the sum of the values of these notes. The storage slot of the user's notes is defined as $\mathsf{Pos2}(\mathsf{slot_{priMap}}, \mathsf{a})$, where $\mathsf{slot_{priMap}}$ is the logical storage slot of the private balances mapping.

B zk-SNARK

Perfect Completness A non-interactive proof π for $\mathcal{R}(x, w) = 1$ has perfect completeness if for every $\mathcal{A}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1 \;\middle|\; \begin{array}{r}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)\\ (\mathcal{R}, x, w) \leftarrow \mathcal{A}(\mathsf{pp})\\ \mathcal{R}(x, w) = 1\\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \mathcal{R})\\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)\end{array}\right] = 1.$$

Knowledge Soundness A non-interactive proof π for $\mathcal{R}(x, w) = 1$ has perfect completeness if for every expected polynomial time adversary $\mathcal{P}^*$, there exists an polynomial time extractor $\mathcal{E}$ such that for every set Z,

$$\begin{aligned}&\Pr\left[\begin{array}{c}(\mathsf{pp}, \mathsf{ai}, \mathcal{R}, x, \mathsf{ao}) \in Z\\ \wedge\, \mathcal{R}(x, w) = 1\end{array}\;\middle|\; \begin{array}{r}\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)\\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp})\\ (\mathcal{R}, x, w, \mathsf{ao}) \leftarrow \mathcal{E}_{\mathcal{P}^*}(\mathsf{pp}, \mathsf{ai})\end{array}\right] \geq\\ &\Pr\left[\begin{array}{c}(\mathsf{pp}, \mathsf{ai}, \mathcal{R}, x, \mathsf{ao}) \in Z\\ \wedge\, \mathcal{R}(x, w) = 1\end{array}\;\middle|\; \begin{array}{r}\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)\\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp})\\ (\mathcal{R}, \mathsf{ao}, \pi, \mathsf{ao}) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{ai})\\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \mathcal{R})\end{array}\right] - \mathsf{negl}(\lambda)\end{aligned}$$

Zero-Knowledge PCD has (statistical) zero knowledge if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that for every honest adversary $\mathcal{A}$ the distributions below are statistically close:

$$\left\{ (\mathsf{pp}, \mathcal{R}, x, \pi) \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\mathcal{R}, x, w) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \mathcal{R}(x, w) = 1 \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \mathcal{R}) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w) \end{array} \right\}$$

$$\text{and} \left\{ (\mathsf{pp}, \mathcal{R}, x, \pi) \;\middle|\; \begin{array}{r} (\mathsf{pp}, \tau) \leftarrow \mathcal{S}(1^\lambda) \\ (\mathcal{R}, x, w) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \pi \leftarrow \mathcal{S}(\tau, \mathcal{R}, x) \end{array} \right\}$$

C Proof Carrying Data

Perfect Completness PCD has perfect completeness if for every $\mathcal{A}$,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, z, \pi) = 1 \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \varphi \in \mathsf{F};\ \varphi(z, z_{\mathsf{loc}}, z_1, \ldots, z_M) = 1 \\ \forall m \in [M], z_m = \bot \text{ or } \mathsf{Verify}(\mathsf{vk}, z_m, \pi_m) = 1 \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \varphi) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M) \end{array} \right] = 1.$$

Knowledge Soundness PCD has knowledge soundness if for every expected polynomial time adversary $\mathcal{P}^*$, there exists an polynomial time extractor $\mathcal{E}$ such that for every set Z,

$$\Pr\left[\begin{array}{c} \varphi \in \mathsf{F} \\ \wedge\ (\mathsf{pp}, \mathsf{ai}, \varphi, \mathsf{o}(\mathsf{T}), \mathsf{ao}) \in Z \\ \wedge\ \mathsf{T} \text{ is } \varphi\text{-compliant} \end{array} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda) \\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp}) \\ (\varphi, \mathsf{T}, \mathsf{ao}) \leftarrow \mathcal{E}_{\mathcal{P}^*}(\mathsf{pp}, \mathsf{ai}) \end{array} \right] \geq$$

$$\Pr\left[\begin{array}{c} \varphi \in \mathsf{F} \\ \wedge\ (\mathsf{pp}, \mathsf{ai}, \varphi, \mathsf{ao}, \mathsf{o}) \in Z \\ \wedge\ \mathsf{Verify}(\mathsf{vk}, \mathsf{ao}, \pi) \end{array} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda) \\ \mathsf{ai} \leftarrow \mathcal{D}(\mathsf{pp}) \\ (\varphi, \mathsf{ao}, \pi, \mathsf{o}) \leftarrow \mathcal{P}^*(\mathsf{pp}, \mathsf{ai}) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \varphi) \end{array} \right] - \mathsf{negl}(\lambda)$$

Zero-Knowledge PCD has (statistical) zero knowledge if there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that for every honest adversary $\mathcal{A}$ the distributions below are statistically close:

$$\left\{ (\mathsf{pp}, \varphi, z, \pi) \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M) \leftarrow \mathcal{A}(\mathsf{pp}) \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, \varphi) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M) \end{array} \right\} \text{ and}$$

$$\left\{ (\mathsf{pp}, \varphi, z, \pi) \;\middle|\; \begin{array}{r} (\mathsf{pp}, \tau) \leftarrow \mathcal{S}(1^\lambda) \\ (\varphi, z, z_{\mathsf{loc}}, \{z_m, \pi_m\}_{m=1}^M) \leftarrow \mathcal{A}(\mathsf{pp}) \\ \pi \leftarrow \mathcal{S}(\tau, \varphi, z) \end{array} \right\}$$

D Relation $\mathcal{R}_{PoA}$

Relation: $\mathcal{R}_{\mathsf{PoA}}(x_{\mathsf{PoA}}, w_{\mathsf{PoA}})$, where:

$$x_{\mathsf{PoA}} = \left[\mathsf{Assets}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}\right];$$

$$w_{\mathsf{PoA}} = \left[\left\{\begin{array}{c}\left\{\begin{array}{c}\mathsf{n}^{(a,n)}, \mathsf{n}^{(a,n)}_{\mathsf{nonce}}, \mathsf{MemberProof}\left(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{NHT}\right), \\ \mathsf{NonMemberProof}\left(\rho^{(a,n)}, \mathsf{NT}\right)\end{array}\right\}_{n=1}^{N_a}, \\ \mathsf{a}^{(a)}_{\mathsf{par}}, \mathsf{nsk}^{(a)}_{\mathsf{m}}, \mathsf{Ivpk}^{(a)}_{\mathsf{m}}, \mathsf{Ovpk}^{(a)}_{\mathsf{m}}, \mathsf{Tpk}^{(a)}_{\mathsf{m}}, \mathsf{p}^{(a)}_{\mathsf{val}}, \\ \mathsf{MemberProof}\left((\mathsf{p}^{(a)}_{\mathsf{key}}, \mathsf{p}^{(a)}_{\mathsf{val}}), \mathsf{PDT}\right)\end{array}\right\}_{a=1}^{A}\right].$$

Constraints:

1. For each $a \in [A]$:
   (a) $\mathsf{Npk}_{\mathsf{m}} \leftarrow \mathsf{CompPublicKey}\left(\mathsf{nsk}^{(a)}_{\mathsf{m}}\right)$.
   (b) $\mathsf{a}^{(a)}_{acc} \leftarrow \mathsf{CompAddress}\left(\mathsf{Npk}_{\mathsf{m}}, \mathsf{Tpk}^{(a)}_{\mathsf{m}}, \mathsf{Ivpk}^{(a)}_{\mathsf{m}}, \mathsf{Ovpk}^{(a)}_{\mathsf{m}}, \mathsf{a}^{(a)}_{\mathsf{par}}\right)$.
   (c) If $a > 1$: $\mathsf{a}^{(a-1)}_{acc} < \mathsf{a}^{(a)}_{acc}$.
   (d) $\mathsf{slot}^{(a)}_{\mathsf{pub}} \leftarrow \mathsf{CompSlot}\left(\mathsf{slot}_{\mathsf{pubMap}}, \mathsf{a}^{(a)}_{acc}\right)$.
   (e) $\mathsf{p}^{(a)}_{\mathsf{key}} \leftarrow \mathsf{CompPublicDataKey}\left(\mathsf{a}_{token}, \mathsf{slot}^{(a)}_{\mathsf{pub}}\right)$.
   (f) $\mathsf{MemberVerify}\Big(\left(\mathsf{p}^{(a)}_{\mathsf{key}}, \mathsf{p}^{(a)}_{\mathsf{val}}\right), \mathsf{PDT}_{\mathsf{root}}, \mathsf{MemberProof}\left(\left(\mathsf{p}^{(a)}_{\mathsf{key}}, \mathsf{p}^{(a)}_{\mathsf{val}}\right), \mathsf{PDT}\right)\Big)$.
   (g) $\mathsf{slot}^{(a)}_{\mathsf{pri}} \leftarrow \mathsf{CompSlot}\left(\mathsf{slot}_{\mathsf{priMap}}, \mathsf{a}^{(a)}_{acc}\right)$.
   (h) $\mathsf{nsk}^{(a)}_{\mathsf{app}} \leftarrow \mathsf{CompNskApp}\left(\mathsf{nsk}^{(a)}_{\mathsf{m}}, \mathsf{a}_{token}\right)$.
   (i) For each $n \in [N_a]$:
      i. $\mathsf{a}_{acc} = \mathsf{n}^{(a,n)}_{\mathsf{owner}}$
      ii. $\mathsf{n}^{(a,n)}_{\mathsf{uhash}} \leftarrow \mathsf{CompNoteHash}\left(\mathsf{n}^{(a,n)}, \mathsf{n}^{(a,n)}_{\mathsf{nonce}}, \mathsf{a}_{token}, \mathsf{slot}^{(a)}_{\mathsf{pri}}\right)$.
      iii. $\mathsf{MemberVerify}\Big(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{MemberProof}\left(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{NHT}\right)\Big)$.
      iv. $\rho^{(a,n)} \leftarrow \mathsf{CompNullifer}\left(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{nsk}^{(a)}_{\mathsf{app}}\right)$.
      v. $\mathsf{NonMemberVerify}\Big(\rho^{(a,n)}, \mathsf{NT}_{\mathsf{root}}, \mathsf{NonMemberProof}\left(\rho^{(a,n)}, \mathsf{NT}\right)\Big)$.
      vi. If $n > 1$:
         A. $\mathsf{n}^{(a,n-1)}_{\mathsf{idx}} \leftarrow \mathsf{ExtLeafIdx}\Big(\mathsf{MemberProof}\left(\mathsf{n}^{(a,n-1)}_{\mathsf{uhash}}, \mathsf{NHT}\right)\Big)$.
         B. $\mathsf{n}^{(a,n)}_{\mathsf{idx}} \leftarrow \mathsf{ExtLeafIdx}\Big(\mathsf{MemberProof}\left(\mathsf{n}^{(a,n)}_{\mathsf{uhash}}, \mathsf{NHT}\right)\Big)$.
         C. $\mathsf{n}^{(a,n-1)}_{\mathsf{idx}} < \mathsf{n}^{(a,n)}_{\mathsf{idx}}$.
      vii. $b_{a,n} \leftarrow b_{a,n-1} + \mathsf{n}^{(a,n)}_{\mathsf{val}}$, where $b_{a,0} = 0$.
   (j) $\left(\mathsf{b}_{\mathsf{pri}}, \mathsf{b}_{\mathsf{pub}}\right) \leftarrow \left(b_{a,N}, \mathsf{p}^{(a)}_{\mathsf{val}}\right)$.
   (k) $\mathsf{b}^{(a)}_{acc} \leftarrow \mathsf{b}_{\mathsf{pri}} + \mathsf{b}_{\mathsf{pub}}$.
   (l) $B_a \leftarrow B_{a-1} + \mathsf{b}^{(a)}_{acc}$, where $B_0 = 0$.
2. $\mathsf{Assets} = B_A$

E PCD Predicates for Π_{PoA}

E.1 Account Tree Leaf

Predicate: $\varphi_{\mathsf{ATL}}(z, z_{\mathsf{loc}}, z_1, \ldots, z_M)$, where:

$$z = \begin{bmatrix} \mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \\ \mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}, \mathsf{b}_{node} \end{bmatrix};$$

$$z_{\mathsf{loc}} = \left[\begin{Bmatrix} \mathsf{n}^{(k)}, \mathsf{n}^{(k)}_{\mathsf{nonce}}, \mathsf{MemberProof}\big(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT}\big), \\ \mathsf{NonMemberProof}\big(\rho^{(k)}, \mathsf{NT}\big) \end{Bmatrix}_{k=1}^{K} \right];$$

$$\forall m \in [M],\ z_m = \bot.$$

Constraints:

1. For each $k \in [K]$:
   (a) $\mathsf{a}_{acc} = \mathsf{n}^{(k)}_{\mathsf{owner}}$
   (b) $\mathsf{n}^{(k)}_{\mathsf{uhash}} \leftarrow \mathsf{CompNoteHash}\big(\mathsf{n}^{(k)}, \mathsf{n}^{(k)}_{\mathsf{nonce}}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}\big)$.
   (c) $\mathsf{MemberVerify}\Big(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{MemberProof}\big(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT}\big)\Big)$.
   (d) $\rho^{(k)} \leftarrow \mathsf{CompNullifer}(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{nsk}_{\mathsf{app}})$.
   (e) $\mathsf{NonMemberVerify}\Big(\rho^{(k)}, \mathsf{NT}_{\mathsf{root}}, \mathsf{NonMemberProof}\big(\rho^{(k)}, \mathsf{NT}\big)\Big)$.
   (f) If $k > 1$:
      i. $\mathsf{n}^{(k-1)}_{\mathsf{idx}} \leftarrow \mathsf{ExtLeafIdx}\Big(\mathsf{MemberProof}\big(\mathsf{n}^{(k-1)}_{\mathsf{uhash}}, \mathsf{NHT}\big)\Big)$.
      ii. $\mathsf{n}^{(k)}_{\mathsf{idx}} \leftarrow \mathsf{ExtLeafIdx}\Big(\mathsf{MemberProof}\big(\mathsf{n}^{(k)}_{\mathsf{uhash}}, \mathsf{NHT}\big)\Big)$.
      iii. $\mathsf{n}^{(k-1)}_{\mathsf{idx}} < \mathsf{n}^{(k)}_{\mathsf{idx}}$.
   (g) $b_k \leftarrow b_{k-1} + \mathsf{n}^{(k)}_{\mathsf{val}}$, where $b_0 = 0$.
2. $(\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}) = (\mathsf{n}^{(1)}_{\mathsf{idx}}, \mathsf{n}^{(K)}_{\mathsf{idx}})$.
3. $\mathsf{b}_{node} = b_K$.

E.2 Account Tree Internal

Predicate: $\varphi_{\mathsf{ATI}}(z, z_{\mathsf{loc}}, z_1, \ldots, z_M)$, where:

$$z = \begin{bmatrix} \mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \\ \mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}, \mathsf{b}_{node} \end{bmatrix}; \quad z_{\mathsf{loc}} = \bot;$$

$$\forall m \in [M],\ z_m = \begin{bmatrix} \mathsf{a}^{(m)}_{acc}, \mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pri}}, \mathsf{nsk}^{(m)}_{\mathsf{app}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \\ \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{idx}^{(m)}_{\mathsf{min}}, \mathsf{idx}^{(m)}_{\mathsf{max}}, \mathsf{b}^{(m)}_{node} \end{bmatrix}.$$

Constraints:

1. For each $m \in [M]$:
   (a) $\Big(\mathsf{a}^{(m)}_{acc}, \mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pri}}, \mathsf{nsk}^{(m)}_{\mathsf{app}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \mathsf{NT}^{(m)}_{\mathsf{root}}\Big)$
   $= (\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}})$.
   (b) If $m > 1$: $\mathsf{idx}^{(m-1)}_{\mathsf{max}} < \mathsf{idx}^{(m)}_{\mathsf{min}}$.
   (c) $b_m \leftarrow b_{m-1} + \mathsf{b}^{(m)}_{node}$, where $b_0 = 0$.
2. $(\mathsf{idx}_{\mathsf{min}}, \mathsf{idx}_{\mathsf{max}}) = (\mathsf{idx}^{(1)}_{\mathsf{min}}, \mathsf{idx}^{(M)}_{\mathsf{max}})$
3. $\mathsf{b}_{node} = b_M$.

E.3 Account Tree Root

Predicate: $\varphi_{\mathsf{ATR}}(z, z_{\mathsf{loc}}, z_{\mathsf{in}})$, where:

$$z = \begin{bmatrix} \mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \\ \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{b}_{acc} \end{bmatrix};$$

$$z_{\mathsf{loc}} = \begin{bmatrix} \mathsf{a}_{\mathsf{par}}, \mathsf{nsk}_{\mathsf{m}}, \mathsf{Ivpk}_{\mathsf{m}}, \mathsf{Ovpk}_{\mathsf{m}}, \mathsf{Tpk}_{\mathsf{m}}, \mathsf{p}_{\mathsf{val}}, \\ \mathsf{MemberProof}\big((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT}\big) \end{bmatrix};$$

$$z_{\mathsf{in}} = \begin{bmatrix} \mathsf{a}^{(\mathsf{in})}_{acc}, \mathsf{a}^{(\mathsf{in})}_{token}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{pri}}, \mathsf{nsk}^{(\mathsf{in})}_{\mathsf{app}}, \mathsf{NHT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{NT}^{(\mathsf{in})}_{\mathsf{root}}, \\ \mathsf{idx}^{(\mathsf{in})}_{\mathsf{min}}, \mathsf{idx}^{(\mathsf{in})}_{\mathsf{max}}, \mathsf{b}^{(\mathsf{in})}_{node} \end{bmatrix}.$$

Constraints:

1. $\mathsf{Npk}_{\mathsf{m}} \leftarrow \mathsf{CompPublicKey}(\mathsf{nsk}_{\mathsf{m}})$.
2. $\mathsf{a}_{acc} = \mathsf{CompAddress}(\mathsf{Npk}_{\mathsf{m}}, \mathsf{Tpk}_{\mathsf{m}}, \mathsf{Ivpk}_{\mathsf{m}}, \mathsf{Ovpk}_{\mathsf{m}}, \mathsf{a}_{\mathsf{par}})$.
3. $\mathsf{slot}_{\mathsf{pri}} \leftarrow \mathsf{CompSlot}(\mathsf{slot}_{\mathsf{priMap}}, \mathsf{a}_{acc})$.
4. $\mathsf{nsk}_{\mathsf{app}} \leftarrow \mathsf{CompNskApp}(\mathsf{nsk}_{\mathsf{m}}, \mathsf{a}_{token})$.
5. $\Big(\mathsf{a}^{(\mathsf{in})}_{acc}, \mathsf{a}^{(\mathsf{in})}_{token}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{pri}}, \mathsf{nsk}^{(\mathsf{in})}_{\mathsf{app}}, \mathsf{NHT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{NT}^{(\mathsf{in})}_{\mathsf{root}}\Big)$
   $= (\mathsf{a}_{acc}, \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pri}}, \mathsf{nsk}_{\mathsf{app}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}})$.
6. $\mathsf{slot}_{\mathsf{pub}} \leftarrow \mathsf{CompSlot}(\mathsf{slot}_{\mathsf{pubMap}}, \mathsf{a}_{acc})$.
7. $\mathsf{p}_{\mathsf{key}} \leftarrow \mathsf{CompPublicDataKey}(\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pub}})$.
8. $\mathsf{MemberVerify}\Big((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT}_{\mathsf{root}}, \mathsf{MemberProof}\big((\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}), \mathsf{PDT}\big)\Big)$.
9. $\big(\mathsf{b}_{\mathsf{pri}}, \mathsf{b}_{\mathsf{pub}}\big) \leftarrow \big(\mathsf{b}^{(\mathsf{in})}_{node}, \mathsf{p}_{\mathsf{val}}\big)$
10. $\mathsf{b}_{acc} = \mathsf{b}_{\mathsf{pri}} + \mathsf{b}_{\mathsf{pub}}$

E.4 Custodian Tree Leaf

Predicate: $\varphi_{\mathsf{CTL}}(z, z_{\mathsf{loc}}, z_1, \ldots, z_M)$, where:

$$z = \begin{bmatrix} \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \\ \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{a}_{\mathsf{min}}, \mathsf{a}_{\mathsf{max}} \mathsf{b}_{node} \end{bmatrix};\ z_{\mathsf{loc}} = \perp;$$

$$\forall m \in [M],\ z_m = \begin{bmatrix} \mathsf{a}^{(m)}_{acc}, \mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pubMap}}, \mathsf{slot}^{(m)}_{\mathsf{priMap}}, \\ \mathsf{NHT}^{(m)}_{\mathsf{root}}, \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{PDT}^{(m)}_{\mathsf{root}}, \mathsf{b}^{(m)}_{acc} \end{bmatrix}.$$

Constraints:

1. For each $m \in [M]$:
   (a) $\Big(\mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pubMap}}, \mathsf{slot}^{(m)}_{\mathsf{priMap}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{PDT}^{(m)}_{\mathsf{root}}\Big)$
   $= (\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}})$.
   (b) If $m > 1$: $\mathsf{a}^{(m-1)}_{acc} < \mathsf{a}^{(m)}_{acc}$.
   (c) $b_m \leftarrow b_{m-1} + \mathsf{b}^{(m)}_{acc}$, where $b_0 = 0$.
2. $(\mathsf{a}_{\mathsf{min}}, \mathsf{a}_{\mathsf{max}}) = (\mathsf{a}^{(1)}_{acc}, \mathsf{a}^{(M)}_{acc})$.
3. $\mathsf{b}_{node} = b_M$.

E.5 Custodian Tree Internal

Predicate: $\varphi_{\mathsf{CTI}}(z, z_{\mathsf{loc}}, z_1, \ldots, z_M)$, where:

$$z = \begin{bmatrix} \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \\ \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{a}_{\mathsf{min}}, \mathsf{a}_{\mathsf{max}}, \mathsf{b}_{node} \end{bmatrix}; \quad z_{\mathsf{loc}} = \perp;$$

$$\forall m \in [M],\ z_m = \begin{bmatrix} \mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pubMap}}, \mathsf{slot}^{(m)}_{\mathsf{priMap}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \\ \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{PDT}^{(m)}_{\mathsf{root}}, \mathsf{a}^{(m)}_{\mathsf{min}}, \mathsf{a}^{(m)}_{\mathsf{max}}, \mathsf{b}_{node} \end{bmatrix}.$$

Constraints:

1. For each $m \in [M]$:
   (a) $\left(\mathsf{a}^{(m)}_{token}, \mathsf{slot}^{(m)}_{\mathsf{pubMap}}, \mathsf{slot}^{(m)}_{\mathsf{priMap}}, \mathsf{NHT}^{(m)}_{\mathsf{root}}, \mathsf{NT}^{(m)}_{\mathsf{root}}, \mathsf{PDT}^{(m)}_{\mathsf{root}}\right)$
   $= (\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}})$.
   (b) If $m > 1$: $\mathsf{a}^{(m-1)}_{\mathsf{max}} < \mathsf{a}^{(m)}_{\mathsf{min}}$.
   (c) $b_m \leftarrow b_{m-1} + \mathsf{b}^{(m)}_{node}$, where $b_0 = 0$.
2. $(\mathsf{a}_{\mathsf{min}}, \mathsf{a}_{\mathsf{max}}) = (\mathsf{a}^{(1)}_{\mathsf{min}}, \mathsf{a}^{(M)}_{\mathsf{max}})$.
3. $\mathsf{b}_{node} = b_M$.

E.6 Custodian Tree Root

Predicate: $\varphi_{\mathsf{CTR}}(z, z_{\mathsf{loc}}, z_{\mathsf{in}})$, where:

$$z = \begin{bmatrix} \mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \\ \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}}, \mathsf{Assets} \end{bmatrix}; \quad z_{\mathsf{loc}} = \perp;$$

$$z_{\mathsf{in}} = \begin{bmatrix} \mathsf{a}^{(\mathsf{in})}_{token}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{pubMap}}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{priMap}}, \mathsf{NHT}^{(\mathsf{in})}_{\mathsf{root}}, \\ \mathsf{NT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{PDT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{a}^{(\mathsf{in})}_{\mathsf{min}}, \mathsf{a}^{(\mathsf{in})}_{\mathsf{max}}, \mathsf{b}^{(\mathsf{in})}_{node} \end{bmatrix}.$$

Constraints:

1. $\left(\mathsf{a}^{(\mathsf{in})}_{token}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{pubMap}}, \mathsf{slot}^{(\mathsf{in})}_{\mathsf{priMap}}, \mathsf{NHT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{NT}^{(\mathsf{in})}_{\mathsf{root}}, \mathsf{PDT}^{(\mathsf{in})}_{\mathsf{root}}\right)$
   $= (\mathsf{a}_{token}, \mathsf{slot}_{\mathsf{pubMap}}, \mathsf{slot}_{\mathsf{priMap}}, \mathsf{NHT}_{\mathsf{root}}, \mathsf{NT}_{\mathsf{root}}, \mathsf{PDT}_{\mathsf{root}})$.
2. $\mathsf{Assets} = \mathsf{b}^{(\mathsf{in})}_{node}$.

F Sub Protocol

Unique Note Hash Computation

Gadget: $\mathsf{n}_{\mathsf{uhash}} \leftarrow \mathsf{CompNoteHash}(\mathsf{n}, \mathsf{n}_{\mathsf{nonce}}, \mathsf{a}, \mathsf{slot})$
where: $\mathsf{n} = (\mathsf{n}_{\mathsf{owner}}, \mathsf{n}_{\mathsf{rnd}}, \mathsf{n}_{\mathsf{val}})$
Constraints:

1. $\mathsf{n}_{\mathsf{pc}} \leftarrow \mathsf{Pos2}(\mathtt{AZ_NOTE_HASH}, \mathsf{n}_{\mathsf{owner}}, \mathsf{n}_{\mathsf{rnd}}, \mathsf{slot})$.
2. $\mathsf{n}_{\mathsf{hash}} \leftarrow \mathsf{Pos2}(\mathtt{AZ_NOTE_HASH}, \mathsf{n}_{\mathsf{pc}}, \mathsf{n}_{\mathsf{val}})$.
3. $\mathsf{n}_{\mathsf{shash}} \leftarrow \mathsf{Pos2}(\mathtt{AZ_SILOED_NOTE_HASH}, \mathsf{a}, \mathsf{n}_{\mathsf{hash}})$.
4. $\mathsf{n}_{\mathsf{uhash}} \leftarrow \mathsf{Pos2}(\mathtt{AZ_UNIQUE_NOTE_HASH}, \mathsf{n}_{\mathsf{nonce}}, \mathsf{n}_{\mathsf{shash}})$.

Nullifier Compuation from $\mathsf{n}_{\mathsf{uhash}}$

Gadget: $\rho \leftarrow \mathsf{CompNullifer}(\mathsf{n}_{\mathsf{uhash}}, \mathsf{a})$
Constraints:

1. $\rho_{\mathsf{inner}} \leftarrow \mathsf{Pos2}(\mathtt{AZ_NOTE_HASH}, \mathsf{n}_{\mathsf{uhash}}, \mathsf{nsk}_{\mathsf{app}})$

2. $\rho \leftarrow \mathsf{Pos2}(\texttt{AZ_SILOED_NOTE_HASH}, \mathsf{a}, \rho_{\mathsf{inner}})$

Address Computation from Public Keys

Gadget: $\mathsf{a} \leftarrow \mathsf{CompAddress}(\mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m}, \mathsf{a_{par}})$
Constraints:

1. $\mathsf{pk_{hash}} \leftarrow \mathsf{Pos2}(\texttt{AZ_PUBKEY_HASH}, \mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m})$
2. $\mathsf{a_{pre}} \leftarrow \mathsf{Pos2}(\texttt{AZ_CONTRACT_ADDRESS_V1}, \mathsf{pk_{hash}}, \mathsf{a_{par}})$
3. $\mathsf{a_{point}} \leftarrow \mathsf{ToFq}(\mathsf{a_{pre}}) \cdot G + \mathsf{Ivpk_m}$
4. $\mathsf{a} \leftarrow (x\text{-coordinate of } \mathsf{a_{point}}) \in \mathbb{F}_r$

Mapping Key Slot Computation

Gadget: $\mathsf{slot} \leftarrow \mathsf{CompSlot}(\mathsf{slot}_{map}, \mathsf{a})$
Constraints:

1. $\mathsf{slot} \leftarrow \mathsf{Pos2}(\mathsf{slot}_{map}, \mathsf{a})$

App-siloed Nullifier Secret Key Computation

Gadget: $\mathsf{nsk_{app}} \leftarrow \mathsf{CompNskApp}(\mathsf{nsk_m}, \mathsf{a})$
Constraints:

1. $\mathsf{nsk_{app}} \leftarrow \mathsf{Pos2}(\texttt{AZ_NSK_APP}, \mathsf{nsk_m}, \mathsf{a})$

Public Key Computation

Gadget: $\mathsf{pk} \leftarrow \mathsf{CompPublicKey}(\mathsf{sk})$
Constraints:

1. $\mathsf{pk} \leftarrow \mathsf{ToFq}(\mathsf{sk}) \cdot G \in \mathbb{G}$

Public Data Key Computation

Gadget: $\mathsf{p_{key}} \leftarrow \mathsf{CompPublicDataKey}(\mathsf{a}, \mathsf{slot_{pub}})$
Constraints:

1. $\mathsf{p_{key}} \leftarrow \mathsf{Pos2}(\texttt{AZ_PUBLIC_DATA_LEAF}, \mathsf{a}, \mathsf{slot_{pub}})$

Membership Verification

Gadget: $\mathsf{MemberVerify}\big(\ell_{\mathsf{pre}}, T_{\mathsf{root}}, \mathsf{MemberProof}(\ell_{\mathsf{pre}}, T)\big)$, where:

- ℓ_{pre}: preimage of the leaf ℓ,
- $\mathsf{MemberProof}(\ell_{\mathsf{pre}}, T) = (\ell_{\mathsf{idx}}, \{\mathsf{sn}_i\}_{i=1}^{n})$.
  - ℓ_{idx}: leaf index,
  - $\{\mathsf{sn}_i\}_{i=1}^{n}$: sibling nodes with length $n = \log_2|T|$

Constraints:

1. $\ell \leftarrow \mathsf{Pos2}(\ell_{\mathsf{pre}})$.
2. $(h, \mathsf{idx}) \leftarrow (\ell, \ell_{\mathsf{idx}})$.
3. For each $i \in [n]$:
   (a) If idx is divisible by 2,
   - Then $h_i \leftarrow \mathsf{Pos2}(\mathsf{id}, i, h_{i-1}, \mathsf{sn}_i)$;

- Otherwise: $h_i \leftarrow \mathsf{Pos2}(\mathsf{id}, i, \mathsf{sn}_i, h_{i-1})$.

(b) $\mathsf{idx} \leftarrow \mathsf{idx}/2$.

4. $h_n = T_{\mathsf{root}}$.

Non-membership Verification

Gadget: $\mathsf{NonMemberVerify}\big(x, T_{\mathsf{root}}, \mathsf{NonMemberProof}(x, T)\big)$, where:

- x: value to prove non-membership
- $\mathsf{NonMemberProof}(x, T) = (\ell'_{\mathsf{pre}}, \ell'_{\mathsf{idx}}, \{\mathsf{sn}_i\}_{i=1}^{n})$.
  - $\ell'_{\mathsf{pre}} = (v, v_{\mathsf{next}}, \mathsf{leaf}_{\mathsf{idx}}(v_{\mathsf{next}}))$: preimage of the low leaf,
  - ℓ'_{idx}: leaf index of the low leaf,
  - $\{\mathsf{sn}_i\}_{i=1}^{n}$: sibling nodes of the low leaf with length $n = \log_2|T|$

Constraints:

1. $\mathsf{MemberVerify}(\ell'_{\mathsf{pre}}, T_{\mathsf{root}}, (\ell'_{\mathsf{idx}}, \{\mathsf{sn}_i\}_{i=1}^{n}))$.
2. $x > v$ and $x < v_{\mathsf{next}}$.

Leaf Index Extraction Extracts the leaf index from the membership proof.