Computer Security Symposium 2025
27-31 October 2025

Hybrid PoA on Aztec: Proof of Asset Ownership over Public and Private Token Balances via Recursive zk-SNARKs

Rio Kanehiro[1,a)] Yohei Watanabe[1,2] Mitsugu Iwamoto[1]

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. On a 16-core server, it can prove 1000 accounts with 112 notes in 6 hours, with 4 accounts proving in parallel. Verification requires only 83 ms, with a proof size of 16 KB.

1. Introduction

There has been significant interest in blockchain and Web3 space in recent years, with the total cryptocurrency market surpassing $4 trillion at the time of writing. Blockchain enables self-custody of digital assets and trustless asset transfers, allowing users to control their assets without relying on intermediaries. In practice, however, many users prefer to store their assets in centralized platforms such as cryptocurrency exchanges and custodians. This is mainly due to challenges of securely managing private keys, in which loss, theft, or user error results in loss of funds. These platforms hold the private keys on behalf of the users and provide a Web2 authentication interface, such as email and password. They also provide services that are not available on the blockchain, such as converting fiat to cryptocurrency. This convenience, however, comes at a cost where users must trust the custodians to keep their funds secure and also to 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. 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 assets it owes to users. If the asset exceeds the liabilities, the custodian is considered to be solvent.

1.1 Previous Works

Provisions [7] is one of the first constructions of PoR designed for Bitcoin exchanges. It enabled custodians to prove solvency without revealing the Bitcoin reserve account addresses, the corresponding balances of those accounts, and individual user balances. This is achieved by committing to customer liabilities using Pedersen commitments and proving ownership of funds over an anonymity set of addresses using zero-knowledge proofs. However, one limitation is that the anonymity set consisted of public keys rather than Bitcoin addresses, which made it difficult for real-world use cases.

Following Provisions, various PoR-related protocols have emerged. In the paper by B.S. Reddy [15] presents a PoA protocol for Bitcoin that avoids the use of pubic key anonymity set by using generic zero-knowledge proofs, specifically zk-SNARKs. IZPR [6] introduces an efficient, incrementally updatable PoA protocol using lookup arguments, achieving constant-sized proofs and fast verification. MProve [10], MProve+ [9], and MProve-Nova [16] extend PoR to privacy-focused blockchains, particularly Monero. The most recent work, MProve-Nova, employs an *Incrementally Verifiable Computation* (IVC) scheme based on the *folding scheme* to achieve scalable proving and constant-time verification, even across thousands of UTXOs. Lastly, DAPOL [4] and DAPOL+ [12] provide

[1] The University of Electro-Communications, Japan
[2] AIST, Japan
[a)] riokanehiro@uec.ac.jp

Table 1 Comparison of related work. n: number of accounts / UTXOs, N: anonymity set size ($N > n$), $\tilde{O}(C)$: SNARK proving time for the protocol circuit C.

Scheme	Blockchain	Prover Work	Concurrent Proving	Verifier Work	Crypto Scheme
Provisions [7]	Bitcoin (Public)	$O(N)$	Supported	$O(\log N)$	Sigma Protocol
zk-SNARK [15]	Bitcoin (Public)	$\tilde{O}(nC)$	$\tilde{O}(C)$	$O(n)$	Groth16
MProve-Nova [16]	Monero (Private)	$\tilde{O}(nC)$	Not Supported	$O(1)$	Nova IVC
Hybrid PoA	Aztec (Hybrid)	$\tilde{O}(n \log n \cdot C)$	$\tilde{O}(\log n \cdot C)$	$O(1)$	PCD w/ UltraHonk

a generalized framework for PoL, which can be combined with various PoA constructions to form a PoR.

1.2 PoA for Hybrid-State Blockchains

We now turn our attention to PoA for *hybrid-state blockchains*. In a hybrid-state blockchain, account balances exist in both public and private states. The most famous example is Zcash, which supports both public and shielded (private) transactions since 2016. In this paper, we focus on the Aztec Network (Aztec for short), a more recent hybrid-state blockchain currently in testnet. Despite still being in testnet, Aztec has gained significant interest, with more than 20,000 nodes running on testnet at the time of writing [8].

The main contribution of this work is a PoA framework designed for hybrid-state blockchains. Although we implement and evaluate our protocol solely for Aztec, the same framework can be applied to other hybrid blockchains, such as Zcash. Our design introduces several key distinctions from prior PoA protocols:

1. **Support for both public and private balances.**
2. **No need for a dedicated anonymity set.** Unlike Provisions [7], [15], and previous MProve protocols [9], [10], our design does *not* require a dedicated anonymity set specifically for the protocol. A dedicated anonymity set would often require the verifier to build the anonymity set (often the Merkle root). In contrast, our protocol uses the entire blockchain state as the anonymity set; therefore, the verifier can rely on a trusted blockchain node to provide the state commitment.
3. **Hierarchical Proof-Carrying Data via Recursive zk-SNARK.** We use hierarchical Proof-Carrying Data (P) via recursive zk-SNARK as a way to aggregate proofs. This enables custodians to prove ownership over an arbitrary number of accounts and UTXOs, as well as support for concurrent proof generation. In contrast, MProve-Nova supports only sequential proof generation, which uses IVC.

We implemented our Hybrid PoA protocol using Noir DSL and the UltraHonk proving backend. Noir is a domain-specific language designed for writing zk-SNARK circuits. UltraHonk is a PLONK-based SNARK that can generate constant-sized proofs. Table 1 presents a brief comparison of our protocol with related works using asymptotic. Running our protocol implementation on a 16-core server, we generated an aggregated proof for 1000 accounts, each holding 112 UTXOs along with some public balance. With four accounts proving concurrently, the aggregated proof was generated in approximately 6 hours. The proof size is 16 KB, can be verified in 83 ms, and costs 2.9 million gas when verified on an Ethereum smart contract. Although not implemented and tested, our protocol can be combined with existing PoL frameworks such as DAPOL+ to form a PoR.

2. Background

This section describes the background of Aztec Network and the proving system that the Hybrid PoA protocol utilizes. Due to the Aztec's complex system architecture, we first describe the high-level overview of the system in section 2.1, then explain the relevant part in detail in the subsequent sections. We assume that the reader is familiar with terms like blockchain, smart contracts, and transactions, although a deep understanding is not required.

2.1 Aztec Overview

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.

2.2 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 [11], $\mathsf{SHA} : \mathbb{F}_r^t \to \mathbb{F}_r$ 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.[*1] Finally, we use **bold** font to denote vectors.

2.3 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.

2.3.1 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.

2.3.2 Indexed Merkle Tree

First introduced in [18], 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 $(v, v_{\text{next}}, \mathsf{leaf}_{\mathsf{idx}}(v_{\text{next}}))$, where v is the stored value, v_{next} is the next-highest value in the tree, and $\mathsf{leaf}_{\mathsf{idx}}(v_{\text{next}})$ 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.

2.4 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* (see section 2.4.4).

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* (see section 2.4.2). The Note Hash Tree is an *append-only Merkle tree* (see section 2.3.1), 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* (see section 2.4.3). A note is spendable if its corresponding nullifier is not included in the *Nullifier Tree*, which can be proven through a non-membership proof.

[*1] 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 2 Keys used in Aztec. SK stands for "Secret Key". Subscript m denotes "Master".

Key	Name	Derivation
sk	Master SK	$\xleftarrow{\$} \mathbb{F}_r$
$\mathsf{nsk_m}$	Nullifier SK	$\mathsf{SHA}(\mathtt{AZ\_NSK\_M}, \mathsf{sk}) \in \mathbb{F}_r$
$\mathsf{ovsk_m}$	Outgoing Viewing SK	$\mathsf{SHA}(\mathtt{AZ\_OVSK\_M}, \mathsf{sk})$
$\mathsf{ivsk_m}$	Incoming Viewing SK	$\mathsf{SHA}(\mathtt{AZ\_IVSK\_M}, \mathsf{sk})$
$\mathsf{tsk_m}$	Tagging SK	$\mathsf{SHA}(\mathtt{AZ\_TSK\_M}, \mathsf{sk})$
$\mathsf{Npk_m}$	Nullifier PK	$\mathsf{ToFq}(\mathsf{nsk_m}) \cdot G \in \mathbb{G}$
$\mathsf{Ovpk_m}$	Outgoing Viewing PK	$\mathsf{ToFq}(\mathsf{ovsk_m}) \cdot G$
$\mathsf{Ivpk_m}$	Incoming Viewing PK	$\mathsf{ToFq}(\mathsf{ivsk_m}) \cdot G$
$\mathsf{Tpk_m}$	Tagging PK	$\mathsf{ToFq}(\mathsf{tsk_m}) \cdot G$
$\mathsf{nsk_{app}}$	App-siloed Nullifier SK	$\mathsf{Pos2}(\mathtt{AZ\_NSK\_APP}, \mathsf{nsk_m}, \mathsf{a}_{app}) \in \mathbb{F}_r$

2.4.1 Keys and Addresses

This section lists the keys involved in constructing the Note Hash Tree and the Nullifier Tree, summarized in Table 2. Due to space limitations, we omit the explanation for each key. Detailed descriptions 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 $\mathsf{a_{par}} \in \mathbb{F}_r$,[*2] the Aztec address a of an account is computed as:

$$\begin{aligned}
\mathsf{pk_{hash}} &= \mathsf{Pos2}(\mathtt{AZ\_PUBKEY\_HASH}, \mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m}),\\
\mathsf{a_{pre}} &= \mathsf{Pos2}(\mathtt{AZ\_CONTRACT\_ADDRESS\_V1}, \mathsf{pk_{hash}}, \mathsf{a_{par}}),\\
\mathsf{a_{point}} &= \mathsf{ToFq}(\mathsf{a_{pre}}) \cdot G + \mathsf{Ivpk_m},\\
\mathsf{a} &= (x\text{-coordinate of } \mathsf{a_{point}}) \in \mathbb{F}_r.
\end{aligned}$$

2.4.2 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 struct 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:

$$\begin{aligned}
\mathsf{n_{pc}} &= \mathsf{Pos2}(\mathtt{AZ\_NOTE\_HASH}, \mathsf{n_{owner}}, \mathsf{n_{rnd}}, \mathsf{slot}),\\
\mathsf{n_{hash}} &= \mathsf{Pos2}(\mathtt{AZ\_NOTE\_HASH}, \mathsf{n_{pc}}, \mathsf{n_{val}}),\\
\mathsf{n_{shash}} &= \mathsf{Pos2}(\mathtt{AZ\_SILOED\_NOTE\_HASH}, \mathsf{a}_{app}, \mathsf{n_{hash}}),\\
\mathsf{n_{uhash}} &= \mathsf{Pos2}(\mathtt{AZ\_UNIQUE\_NOTE\_HASH}, \mathsf{nonce}, \mathsf{n_{shash}}).
\end{aligned}$$

[*2] 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`

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

2.4.3 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.

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

$$\rho_{\mathsf{inner}} = \mathsf{Pos2}(\mathtt{AZ\_NOTE\_HASH}, \mathsf{n}_{\mathsf{uhash}}, \mathsf{nsk}_{\mathsf{app}}),$$
$$\rho = \mathsf{Pos2}(\mathtt{AZ\_SILOED\_NOTE\_HASH}, \mathsf{a}_{app}, \rho_{\mathsf{inner}}).$$

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

2.4.4 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 $\mathsf{p}_{\mathsf{key}}$ 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:

$$\mathsf{p}_{\mathsf{key}} = \mathsf{Pos2}(\mathtt{AZ\_PUBLIC\_DATA\_LEAF}, \mathsf{a}_{app}, \mathsf{slot})$$

The public data leaf of $\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}$ is a tuple $(\mathsf{p}_{\mathsf{key}}, \mathsf{p}_{\mathsf{val}}, \mathsf{p}_{\mathsf{next\_key}}, \mathsf{idx}_{\mathsf{next}})$ where $\mathsf{p}_{\mathsf{next\_key}}$ is the next key and $\mathsf{idx}_{\mathsf{next}}$ 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.

2.5 AIP-20 Token Standard

The AIP-20 token standard [20], 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 $\mathsf{b}_{\mathsf{pub}}$ that is maintained in public storage. The storage slot of a user balance is defined as $\mathsf{Pos2}(\mathsf{slot}_{pubMap}, \mathsf{a})$, where 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 $\mathsf{b}_{\mathsf{pri}}$ 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 slot_{priMap} is the logical storage slot of the private balances mapping.

2.6 Non-interactive Proving System

A proof system for a circuit C consists of a prover $\mathcal{P}$ and a verifier $\mathcal{V}$. The aim of $\mathcal{P}$ is to convince $\mathcal{V}$ that, for a given public input pub and private input pri, the circuit outputs out; therefore $C(\mathsf{pub}, \mathsf{pri}) = \mathsf{out}$. Here, pub and out are public information and known to both parties, while pri is private and known only to $\mathcal{P}$. When C is executed, it also produces a set of intermediate values, denoted by int.

We encode this statement as an NP-relation $\mathcal{R}_C(x, w)$, where the instance is $x = (\mathsf{pub}, \mathsf{out})$, and the witness is $w = (\mathsf{pri}, \mathsf{int})$. The relation $\mathcal{R}_C(x, w) = 1$ holds if and only if circuit C outputs the correct out when given inputs $(\mathsf{pub}, \mathsf{out})$ and int is consistent with the circuit execution.

If the proof is a single message from $\mathcal{P}$ to $\mathcal{V}$, the system is *non-interactive*. A non-interactive proving system Π has the following syntax:

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

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

$\mathsf{Execute}(C, \mathsf{pub}, \mathsf{pri}) \to (x, w)$: Runs C to obtain (x, w), where $x = (\mathsf{pub}, \mathsf{out})$ and $w = (\mathsf{pri}, \mathsf{int})$

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

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

2.6.1 zk-SNARK

A proof is said to be a *zk-SNARK* (Zero-Knowledge succinct non-interactive arguments of knowledge) if it satisfies the following properties [14]:

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

Zero Knowledge: Informally, the tuple $(\pi, x, \mathsf{pk}, \mathsf{vk}, \mathsf{pp})$ reveals nothing about the witness w.

Knowledge Soundness: Informally, for any $\mathcal{P}^*$ there exists an efficient extractor such that whenever $\mathcal{P}^*$ convinces $\mathcal{V}$ to accept, the extractor can extract a witness w from $\mathcal{P}^*$ such that $\mathcal{R}_C(x, w) = 1$.

Succinctness: The proof size and verification time are at most polylogarithmic in $|C|$ and $|w|$.

Non-Interactive: There is no back-and-forth communication between $\mathcal{P}$ and $\mathcal{V}$.

2.6.2 Recursive zk-SNARK

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:

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

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

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

then each π_i is a valid proof for its corresponding x_i.

2.7 Proof-Carrying Data via Recursive zk-SNARK

Proof-Carrying Data (PCD) [5] is a cryptographic

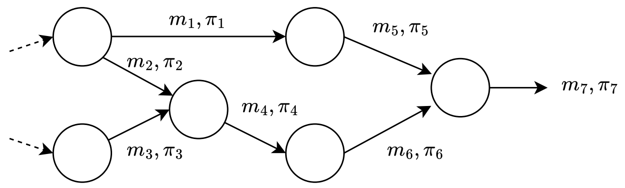

Fig. 1 Proof-Carrying Data (PCD)

framework that enables parties to perform distributed computations that run indefinitely, while ensuring that the correctness of every intermediate state can be verified efficiently. 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. As illustrated in Figure 1, the computations are structured in the form of a directed acyclic graph (DAG), where each node corresponds to a computation, and each edge represents the message and the proof of the computation. A special case of PCD is *incrementally verifiable computation* (IVC) [19], where the nodes form a linear chain, and each step of the computation in the node depends only on the previous one.

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.

2.8 UltraHonk Proving System

The UltraHonk proving system is a zk-SNARK construction developed by Aztec Labs. Unfortunately, at the time of writing, there is no formal paper or detailed documentation, so we provide a high-level overview based on our understanding of the codebase.[*3]

UltraHonk is based on the Plonkish arithmetization [17] which features standard arithmetic circuits, custom gates, and lookup tables. Unlike traditional PLONK, which relies on FFTs (Fast Fourier Transform), UltraHonk uses a sum-check protocol over multilinear polynomials to avoid using FFTs for prover efficiency. For the polynomial commitment scheme, UltraHonk uses Shplemini, which is composed of Gemini [3], Shplonk [2], and KZG [13]. Gemini reduces multilinear polynomial opening claims into univariate ones. Shplonk batches multiple univariate openings into a single claim. KZG provides the final univariate polynomial and constant-size opening proof but requires a trusted setup. The overall proving time is $\tilde{O}(C)$, meaning near linear with small polylog factors in the size of the circuit C.

In summary, UltraHonk can produce constant-size proofs for Plonk-based circuits but requires a trusted third party to generate the public parameters.

[*3] Codebase containing the UltraHonk proving backend: `https://github.com/AztecProtocol/aztec-packages/`

3. Hybrid PoA Protocol

In this section, we present the Hybrid Proof of Assets (PoA) Protocol for the Aztec Network. The protocol consists of four entities: the *Aztec node*, a *trusted third party*, the *custodian* (prover) $\mathcal{C}$, and the *verifier* $\mathcal{V}$. The Aztec node is responsible for providing authentic blockchain data such as Note Hash Tree, Nullifier Tree, and transaction data. The trusted third party generates public parameters for the UltraHonk proving system. The custodian, controlling one or more Aztec accounts, proves the total asset of AIP-20 tokens in zero knowledge, without revealing account addresses or individual balances. Here, since we are working without hybrid-state tokens, this balance is the sum of public and private holdings. The verifier checks the proof, given access to the public blockchain state.

3.1 Notations and Protocol Prerequisites

Let **acc** denote the set of accounts owned by $\mathcal{C}$. Each account has its keys and address, detailed in section 2.4.1.

Let a_{token} be the address of an AIP-20 token targeted for this protocol. In the token contract, there is a mapping from account address to public balance $\mathsf{b}_{\mathsf{pub}}$ and private balance $\mathsf{b}_{\mathsf{pri}}$. For each account, the total balance is $\mathsf{b}_{\mathsf{pub}} + \mathsf{b}_{\mathsf{pri}}$, and $\mathbf{b}$ denotes the set of total balances for each account in **acc**. The custodian's total asset Assets is defined as the sum of all elements in $\mathbf{b}$, i.e., the sum of all private and public balances across **acc** for an AIP-20 token contract. The private balance $\mathsf{b}_{\mathsf{pri}}$ of an account consists of a set of *consumable* (non-nullified) notes $\mathbf{n}$.

The blockchain node maintains the public state: Note Hash Tree NHT, Nullifier Tree NT, and Public Data Tree PDT. We denote the root hash of a tree T as $\mathsf{root}(T)$, for example, $\mathsf{root}(\mathsf{NHT})$ represents the root of NHT. For some leaf leaf, merkle tree T, $\mathsf{MemberProof}(\mathsf{leaf}, T)$ denotes a membership roof. And for some leaf leaf' and merkle tree T', $\mathsf{NonMemberProof}(\mathsf{leaf}', T')$ denotes a non-membership proof. Both proofs consist of the list of sibling nodes along the path from left to the root.

3.2 Simplified Hybrid PoA

Before presenting the full version of Hybrid PoA, we first describe a simplified version of the PoA protocol, which consists of a single Aztec account and a single private note. The goal of the custodian is to prove the total asset (or total balance b in this simplified case) of an AIP-20 token, which consists of public balance and a note value (private balance), without revealing the account address, or the note contents. This is achieved by encoding the above statement into an SNARK circuit C.

The circuit C proceeds in four steps:

1. **Public balance check**: Check the inclusion of the public data leaf corresponding to the account's public balance.
2. **Private balance check**: Check the inclusion of the

note commitment in the Note Hash Tree and non-inclusion of its nullifier in the Nullifier Tree.
3. **Account ownership check**: Check the ownership of the account used in 1 and 2.
4. **Total balance computation** Compute the total balance from the note content and pubic balance, and output the balance.

In step 3, we use knowledge of the master nullifier secret key $\mathsf{nsk_m}$ to prove account ownership instead of the full master secret key sk. This simplifies the design, and since $\mathsf{nsk_m}$ is considered to be private and is derived from sk, we can say that this is a safe assumption.

The circuit C takes the following public input pub, private input pri, and outputs out:

$$\mathsf{pub} = \begin{bmatrix} \mathsf{a}_{token}, \mathsf{slot}_{pubMap}, \mathsf{slot}_{priMap}, \\ \mathsf{root}(\mathsf{PDT}), \mathsf{root}(\mathsf{NHT}), \mathsf{root}(\mathsf{NT}) \end{bmatrix}, \quad \mathsf{out} = \begin{bmatrix} \mathsf{b} \end{bmatrix},$$

$$\mathsf{pri} = \begin{bmatrix} \mathsf{a}_{cus}, \mathsf{a_{par}}, \mathsf{nsk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m}, \mathsf{Tpk_m}, \\ \mathsf{n}, \mathsf{n_{nonce}}, \mathsf{p_{val}}, \mathsf{p_{next\_key}}, \mathsf{leaf_{idx}}(\mathsf{p_{next\_key}}), \\ \mathsf{MemberProof}\big(\mathsf{n_{uhash}}, \mathsf{NHT}\big), \\ \mathsf{NonMemberProof}\big(\rho, \mathsf{NT}\big), \\ \mathsf{MemberProof}\left(\begin{pmatrix} \mathsf{p_{key}}, \mathsf{p_{val}}, \mathsf{p_{next\_key}}, \\ \mathsf{leaf_{idx}}(\mathsf{p_{next\_key}}) \end{pmatrix}, \mathsf{PDT}\right) \end{bmatrix}.$$

Here, the note n is a struct with fields $(\mathsf{a}_{cus}, \mathsf{n_{rnd}}, \mathsf{n_{val}})$.

The circuit C performs the following computations. Steps 1–4 correspond to the public balance check, steps 5–10 to the private balance check, steps 11–12 to the account ownership check, and finally, step 13 to the total balance computation.

1. Compute the public balance storage slot: $\mathsf{slot}_{pub} = \mathsf{Pos2}(\mathsf{slot}_{pubMap}, \mathsf{a}_{cus})$.
2. Derive the public data key: $\mathsf{p_{key}} = \mathsf{Pos2}(\mathtt{AZ\_PUBLIC\_DATA\_LEAF}, \mathsf{a}_{token}, \mathsf{slot}_{pub})$.
3. Set the public data leaf leaf_{pub} as: $(\mathsf{p_{key}}, \mathsf{p_{val}}, \mathsf{p_{next\_key}}, \mathsf{leaf_{idx}}(\mathsf{p_{next\_key}}))$.
4. Check that leaf_{pub} above is included in PDT using $\mathsf{MemberProof}\Big(\big(\mathsf{p_{key}}, \mathsf{p_{val}}, \mathsf{p_{next\_key}}, \mathsf{leaf_{idx}}(\mathsf{p_{next\_key}})\big), \mathsf{PDT}\Big)$ and $\mathsf{root}(\mathsf{PDT})$.
5. Check the private balance storage slot: $\mathsf{slot}_{pri} = \mathsf{Pos2}(\mathsf{slot}_{priMap}, \mathsf{a}_{cus})$.
6. Using $\mathsf{n_{nonce}}$, n, and slot_{pri}, compute the note commitment $\mathsf{n_{uhash}}$ (see section 2.4.2).
7. Check that $\mathsf{n_{uhash}}$ is included in NHT using the provided $\mathsf{MemberProof}(\mathsf{n_{uhash}}, \mathsf{NHT})$ and $\mathsf{root}(\mathsf{NHT})$.
8. Derive the app-siloed nullifier secret key: $\mathsf{nsk_{app}} = \mathsf{Pos2}(\mathtt{AZ\_NSK\_APP}, \mathsf{nsk_m}, \mathsf{a}_{token})$.
9. Compute the nullifier ρ using $\mathsf{n_{uhash}}$ and $\mathsf{nsk_{app}}$ (see section 2.4.3).
10. Check non-inclusion of ρ in NT using the provided $\mathsf{NonMemberProof}(\rho, \mathsf{NT})$ and $\mathsf{root}(\mathsf{NT})$.
11. Derive the master nullifier public key: $\mathsf{Npk_m} = \mathsf{ToFq}(\mathsf{nsk_m}) \cdot G$.
12. Compute the Aztec address a_{comp} using public key set $(\mathsf{Npk_m}, \mathsf{Tpk_m}, \mathsf{Ivpk_m}, \mathsf{Ovpk_m})$ and $\mathsf{a_{par}}$ (see section 2.4.1), and check $\mathsf{a}_{comp} = \mathsf{a}_{cus}$.
13. Compute and output total balance: $\mathsf{b} = \mathsf{n_{val}} + \mathsf{p_{val}}$.

The custodian executes the above circuit and generates the proof of execution. The verifier uses the proof to verify that the checks were performed correctly and the computed balance is correct.

3.3 Security Analysis of Simplified PoA

Here, we provide a high-level security analysis of the simplified PoA protocol. We assume the underlying UltraHonk proving system satisfies the properties of zk-SNARK described in section 2.6.1. Under this assumption, the circuit guarantees the following:

- **Balance correctness.** The circuit constrains the custodian to correctly compute the total balance as the sum of the valid public balance and the non-nullified private note. The public balance $\mathsf{p_{val}}$ must correspond to the Merkle leaf leaf_{pub}, which is proven to be part of PDT. The note n must be included in NHT and its computed ρ must be shown to be not included in NT using the provided Merkle proofs.
- **Account ownership.** The circuit binds both public and private balances to the same account Aztec address a_{cus}. This is achieved by recomputing the account address using $\mathsf{nsk_m}$, public keys, and $\mathsf{a_{par}}$, and checking that it matches the input a_{cus} used for slot_{pub} and slot_{pri} derivations.
- **Privacy.** The protocol does not reveal which account or note was used, nor the public or private balances. The only public output is the total balance b, and all checks are performed within the zk-SNARK circuit.

3.4 Extending to Multiple Accounts and Notes

The simplified protocol introduced in section 3.2 presents the foundation for constructing PoA systems for hybrid-state blockchains. We now extend this protocol to support multiple accounts and multiple notes per account.

The challenge is prover scalability: how do we construct a protocol that can support thousands of accounts and notes? If we naively build a circuit that iterates the simplified PoA protocol for all accounts and notes, the circuit size will grow linearly with the number of inputs, and the prover runs in quasi-linear time, i.e, $\tilde{O}(n)$. Moreover, SNARK circuits do not support dynamically sized inputs as the size of the circuit must be fixed at compile time.

To overcome this limitation, we employ recursive zk-SNARK using a hierarchically-structured PCD (Figure 2). In this scheme, small batches of notes are verified at leaf nodes, each of which outputs the sum of the note values as a local balance along with a proof of correct computation. Then, the internal nodes recursively verify child proofs, aggregate their local balances, and output the result and the proof to their parent node until a single internal node remains. The root node then verifies the aggregated proof from the final internal node and the public balance. It outputs the total balance by summing the public balance

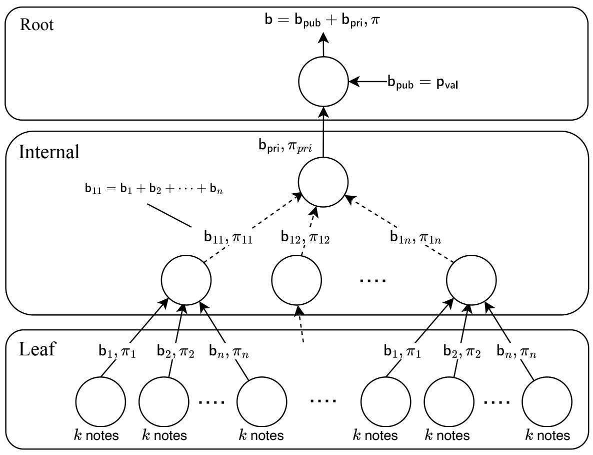

Fig. 2 Hierarchical PCD for Account Tree

and the output of the final internal node, which represents the private balance. This structure enables custodians to prove a balance consisting of many notes associated with a single account. Moreover, each layer of the tree can be processed in parallel, enabling scalable proving. We refer to this structure as the *Account Tree*.

To support multiple accounts, we apply the same approach using hierarchical PCD. Here, each leaf corresponds to the root of an Account Tree, which is a PoA tree for a single account. The output of the root node is the total assets across all accounts controlled by the custodian. We refer to this structure as the *Custodian Tree*.

In addition to prover scalability, we must take care of *double counting*. In double-counting, a custodian may try to include the same valid note or account multiple times to increase the total balance. To prevent this, we add ordering checks within the circuit: all account addresses must be in increasing order, and all notes must be sorted in ascending order by their note hash leaf indices. This way, duplicate notes and accounts can be detected during circuit execution, which ensures that each notes and account are counted only once.

3.5 The Full Protocol Overview

We are now ready to present the full Hybrid PoA protocol. Due to space limitations, we provide only a high-level description. As described in section 3.4, Hybrid PoA consists of two hierarchical structures: the *Account Tree* and the *Custodian Tree*.

3.5.1 Account Tree

The Account Tree handles both the public balance and the private notes associated with an account acc. First, the custodian begins by retrieving notes $\mathbf{n}$ for acc from the PXE. $\mathbf{n}$ are then sorted by their note hash leaf indices $\mathsf{leaf}_{\mathsf{idx}}(\mathsf{n}_{\mathsf{hash}})$ and grouped into batches of size k.

Each leaf node of the Account Tree verifies ownership and inclusion of its k notes using the provided membership proofs. To prevent double-counting, the notes must be in increasing order by their $\mathsf{leaf}_{\mathsf{idx}}(\mathsf{n}_{\mathsf{hash}})$. The leaf node then outputs the sum of note values as a local balance, along with the local minimum and maximum of $\mathsf{leaf}_{\mathsf{idx}}(\mathsf{n}_{\mathsf{hash}})$ of that batch (denoted $\mathsf{min}_{\mathsf{idx}}$ and $\mathsf{max}_{\mathsf{idx}}$, respectively), and a proof of correct computation.

The system recursively aggregates leaf nodes through the internal nodes until a single internal node remains. Each internal node verifies its child proofs, accumulates the child balances, and checks that all children node uses the same parameters: token address a_{token}, account address acc, private balance storage slot slot_{pri}, and root hashes. We call this the *global parameters*. It also checks the global ordering of the notes such that $\mathsf{max}_{\mathsf{idx}}$ of the child at position i is less than $\mathsf{min}_{\mathsf{idx}}$ of the child at position $i+1$ for all child nodes. The internal node outputs the aggregated local sum together with its $\mathsf{min}_{\mathsf{idx}}$ and $\mathsf{max}_{\mathsf{idx}}$.

The root node performs the final check for the account by combining public and private balance checks. It verifies the account's public balance $\mathsf{b}_{\mathsf{pub}}$ by checking inclusion of PDT, validates the proof from the final internal node, and ensures the correctness of the global parameters. It performs an account ownership check by deriving slot_{pub} and slot_{pri} from $\mathsf{nsk}_{\mathsf{m}}$ and checks that all child nodes used the correct slot_{pri}. The root node outputs the total balance b for the account acc together with the aggregated proof.

3.5.2 Custodian Tree

The Custodian Tree works similarly to the Account Tree, but for accounts. The custodian first sorts the accounts by their address and groups them into batches of size k'.

At each leaf node, k' Account Trees proofs are verified and their balances are aggregated. The accounts must be in increasing order by their addresses a, and the Account Tree must use the same global parameters. The leaf outputs the aggregated balance of its batch, along with the minimum and maximum account addresses (denoted by $\mathsf{min}_{\mathsf{addr}}$ and $\mathsf{max}_{\mathsf{addr}}$, respectively) of that batch and a proof of correct computation.

Internal nodes recursively verify child proofs, accumulate their balances, and enforce ordering across the batches by checking $\mathsf{max}_{\mathsf{addr}}$ of child at position i is less than $\mathsf{min}_{\mathsf{addr}}$ of child at position $i+1$. The root node then validates its child proof and outputs the custodian's total asset Assets along with the final aggregated proof π.

The verifier checks the proof π generated at the Custodian Tree root. The statement instance is $x = [\mathsf{pub}, \mathsf{out}]$, where pub consists of the token address a_{token}, the public and private balance mapping slots (slot_{pubMap} and slot_{priMap}), and the Merkle roots $\mathsf{root}(\mathsf{NHT})$, $\mathsf{root}(\mathsf{NT})$, and $\mathsf{root}(\mathsf{PDT})$. The public output out is the custodian's total asset Assets. Let vk be the verification key for the Custodian Tree root circuit. If $\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1$, the verifier is convinced that the custodian owns Assets.

4. Implementation and Performance

We implemented Hybrid PoA using Noir DSL and UltraHonk proving backend. [*4] Noir is a domain-specific lan-

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

Table 3 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, VT denotes the verification time, and VC denotes the verification cost in an Ethereum smart contract. 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)	VC (10^6 gas)
1	92	16.10	83	2.91
10	257	16.10	82	2.91
50	1137	16.10	81	2.91
100	2229	16.10	82	2.91
1000	21716	16.10	83	2.91

Table 4 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 5 Proving time (PT), peak memory usage during proving (PM), 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

guage (DSL) developed by Aztec Labs for writing SNARK circuits. The UltraHonk proving backend is provided within the Barretenberg library, also developed by Aztec Labs. In addition to proof generation and verification, UltraHonk supports Ethereum verifier contract generation.

Our implementation consists of around 1300 lines of Noir code. It implements SNARK circuits for each node in the hierarchical PCD: leaf, internal, and root nodes of both the Account Tree and Custodian Tree. An Account Tree leaf node processes 32 notes, and its internal node aggregates 4 child nodes (either leaf or internal). A Custodian Tree leaf node aggregates 4 Account Tree root nodes, and its internal node similarly aggregates 4 child nodes.

We evaluated the performance on a 2019 Mac Pro with a 3.2 GHz 16-core Intel Xeon W processor and 768 GB of DDR4 memory. Table 3 shows the proving and verification performance for a range of accounts ($n = 1$ to 1000), where each account holds 112 notes. As for concurrency, 4 accounts, 2 internal nodes, and 4 Account Tree leaves were processed in parallel. Table 4 represents the performance of a single Account Tree for a range of notes ($n = 32$ to 1000). Lastly, Table 5 summarizes the proving time, peak memory usage, and gate count for each node in both trees.

5. 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 constant-sized proofs with fast verification that can even be verified on Ethereum.

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 only use the fast verifier at the final aggregation 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 K Program Grant Number JPMJKP24U2, Japan.

References

[1] Aztec Labs: Aztec Documentation, `https://docs.aztec.network/`.
[2] Boneh, D., Drake, J., Fisch, B. and Gabizon, A.: Efficient polynomial commitment schemes for multiple points and polynomials, *ePrint* (2020).
[3] Bootle, J., Chiesa, A., Hu, Y. and Orru, M.: Gemini: Elastic SNARKs for Diverse Environments, *EUROCRYPT*, Springer-Verlag (2022).
[4] Chalkias, K., Lewi, K., Mohassel, P. and Nikolaenko, V.: Distributed Auditing Proofs of Liabilities, ePrint (2020).
[5] Chiesa, A. and Tromer, E.: Proof-Carrying Data and Hearsay Arguments from Signature Cards, *ICS* (Yao, A. C., ed.), Tsinghua University Press (2010).
[6] Conley, T., Diaz, N., Espada, D., Kuruvilla, A., Mayne, S. and Fu, X.: IZPR: Instant Zero Knowledge Proof of Reserve, *FC*, Springer Nature Switzerland (2025).
[7] Dagher, G. G., Bünz, B., Bonneau, J., Clark, J. and Boneh, D.: Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges, *ACM CCS* (2015).
[8] DashLabs: Dashtec - Validator Hub, `https://dashtec.xyz/`.
[9] Dutta, A., Bagad, S. and Vijayakumaran, S.: MProve+: privacy enhancing proof of reserves protocol for Monero, *IEEE Transactions on Information Forensics and Security* (2021).
[10] Dutta, A. and Vijayakumaran, S.: MProve: A proof of reserves protocol for Monero exchanges, *EuroS&PW*, IEEE (2019).
[11] Grassi, L., Khovratovich, D. and Schofnegger, M.: Poseidon2: A faster version of the poseidon hash function, *Africacrypt*, Springer (2023).
[12] Ji, Y. and Chalkias, K.: Generalized proof of liabilities, *ACM CCS* (2021).
[13] Kate, A., Zaverucha, G. M. and Goldberg, I.: Constant-Size Commitments to Polynomials and Their Applications, *ASIACRYPT* (2010).
[14] Ozdemir, A. and Boneh, D.: Experimenting with Collaborative zk-SNARKs: Zero-Knowledge Proofs for Distributed Secrets, *USENIX Security*, USENIX Association (2022).
[15] Reddy, S. B.: A ZK-SNARK based proof of assets protocol for bitcoin exchanges, *COMSNETS* (2023).
[16] Thakore, V. and Vijayakumaran, S.: MProve-Nova: A Privacy-Preserving Proof of Reserves Protocol for Monero, *Proceedings on Privacy Enhancing Technologies* (2025).
[17] The Electronic Coin Company: The halo2 Book, `https://zcash.github.io/halo2/`.
[18] Tzialla, I., Kothapalli, A., Parno, B. and Setty, S. T. V.: Transparency Dictionaries with Succinct Proofs of Correct Operation, *NDSS*, The Internet Society (2022).
[19] Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency, TCC'08 (2008).
[20] wonderland: Request for Comments: AIP-20: Aztec Token Standard, `https://forum.aztec.network/t/request-for-comments-aip-20-aztec-token-standard/7737`.