UltraMixer: A Compliant Zero-Knowledge Privacy Layer for Tokenized Real-World Assets

Zonglun Li[1], Hong Kang[1], and Xue Liu[1,2]

[1] McGill University
{zonglun.li,hong.kang}@mail.mcgill.ca, xueliu@cs.mcgill.ca
[2] Mohamed Bin Zayed University of Artificial Intelligence

Abstract. Real-world-asset (RWA) tokens endow underlying assets with fractional ownership and more continuous settlement, yet recording these claims on transparent public ledgers exposes flows and positions, undermining market confidentiality. Practical deployments must reconcile enforceable access control with principled privacy once assets are shielded. We present *UltraMixer*, a noncustodial privacy layer natively compatible with ERC-3643. Compliance is enforced at the boundary via zero-knowledge proofs of whitelist membership, while in-mixer transfers and atomic trades operate over commitments with nullifiers to prevent double-spend. A generalized UTXO encoding supports heterogeneous assets (fungible and non-fungible) under a unified commitment scheme. For selective disclosure, *UltraMixer* provides a verdict-only Δ-Window Proof of Holding that attests to continuous ownership across a time interval without revealing balances, identities, or linkages. Gas-aware batching and composable emergency controls (pause, freeze/unfreeze, force-transfer) preserve practicality and governance. The resulting architecture delivers regulator-compatible confidentiality for permissioned RWA markets.

Keywords: Real-World Assets · Mixing Service · ERC-3643-Compliant Privacy.

1 Introduction

"Real-world asset" (RWA) tokens [28] represent claims on real-world goods or financial instruments such as government bonds, real estate, commodities, machinery or even revenue streams. RWA tokens act like digital twins of off-chain assets; they can be traded on a blockchain or used as collateral, but are always backed by an underlying asset [27]. Tokenization provides fractional ownership, automatic settlement and 24/7 markets, features that are not available in traditional systems. Thanks to the properties above, the RWA market has grown rapidly. Industry analyses estimate the market exceeded $24 billion in June 2025, up 380% since 2022 [22]. The same report notes that the total addressable market is huge—only a tiny fraction of the $398 trillion real-estate market and $130 trillion bond market has been tokenized [26].

But strong compliance requirements and the transparent nature of blockchain technology pose a threat to the further development of RWA. Privacy concerns

have even been regarded as one of the main barriers to RWA adoption [8]. Traditional finance makes extensive use of confidential transactions. Block trades and dark pools let large investors buy or sell securities without revealing their intentions; banks and custodians are bound by privacy regulations and internal policies. Without such confidentiality, competitors could front-run trades or glean sensitive information about investment strategies. When assets are tokenized, however, the blockchain's transparency exposes flows and holdings. Open ledgers make it easier to front-run or manipulate markets because trades and values are visible to everyone [10]. Regulatory compliance adds to these concerns. Data-protection frameworks such as the General Data Protection Regulation (GDPR) [1] require organizations to minimize data collection, inform investors how their data is used, encrypt personal information, and obtain consent for cross-border transfers [25]. Tokenization must therefore provide a way to hide sensitive data from the public while still allowing regulators and issuers to verify identities and enforce Anti-Money-Laundering (AML) rules. Achieving a balance between transparency and confidentiality is critical for bringing high-value real-world assets on-chain.

To address this gap, we present *UltraMixer*, a zero-knowledge, privacy-preserving mixing and trading layer for permissioned RWAs. UltraMixer is natively compatible with ERC-3643 (T-REX), an open source EVM standard for issuing, managing, and transferring permissioned tokens with identity and compliance built in [12]. In practice, ERC-3643 couples an ERC-20–compatible token interface with ONCHAINID based identities and an Identity Registry that binds addresses to verifiable claims, plus a separate Compliance contract that performs a pre-transfer eligibility check (`canTransfer`) while preserving ERC-20 interoperability with DeFi tools [12,13]. These features make ERC-3643 a good base for a compliant, privacy aware mixer, and public reports attribute more than $32 billion in tokenized assets to ERC-3643 deployments [5]. Within this framework, a *deposit* in UltraMixer is a standard ERC-20 transfer of an ERC-3643 token from an eligible holder to the mixer contract, gated by the token's identity and compliance checks. A *withdrawal* is the reverse, subject to the same eligibility and any issuer controls such as pause or freeze. *In-mixer transfers* and *atomic trades* occur entirely as movements of commitments and nullifiers inside UltraMixer, so ERC-3643 policies apply only at the boundaries where value enters or exits. In exceptional cases required by policy or law, issuer or agent powers such as forced transfer or recovery can reconcile external state without exposing in-mixer linkages [12].

Shuffle protocols and payment hubs deliver unlinkable payments but stop short of a general asset mixer with private transfer and exchange. CoinShuffle++ achieves unlinkable peer shuffles compatible with Bitcoin consensus, and TumbleBit realizes an untrusted but unlinkable hub, yet both target one shot payments and provide neither multi asset support nor compliance surfaces [23,15]. On Ethereum, Möbius and MixEth formalize efficient deposit–withdraw tumblers but do not support in-mixer transfer or trade, and they offer no notion of proof of holding or permissioned token eligibility [18,24]. Tornado-style mix-

Methods	A	Ops	Trade	Comp./Audit	Proof of Holding
CoinShuffle++ [23]	F	M	×	∅	∅
TumbleBit [15]	F	P	×	∅	∅
Möbius [18]	F	D,W	×	∅	∅
MixEth [24]	F	D,W	×	∅	∅
Tornado-style [21]	F	D,W	×	∅	∅
Privacy Pools [4]	F	D,W	×	PoI	∅
Aegis (NFT) [16]	N	D,W,NFT-S	✓	∅	Snap
Zcash [29]	F	S	×	VK	∅
zkLedger [20]	F	S	×	AV	∅
RAILGUN [6,7]	F+N	S	⋄	VK, PoI	∅
UltraMixer (ours)	**F+N**	**D,W,IM,AT**	✓	**ERC**	Δ

Table 1. Compact comparison across privacy systems. **A**: asset types (F = fungible, N = non-fungible). **Ops**: D = deposit, W = withdraw, S = shielded transfer, M = peer shuffle, P = hub payments, IM = in-mixer transfer, AT = atomic trade, NFT-S = private NFT swap. **Trade**: ✓ = supported, × = not supported, ⋄ = partial. **Comp./Audit**: ∅ = none, PoI = proof of innocence (association sets), VK = viewing keys, AV = auditor-verifiable aggregates, ERC = ERC-3643 gates at boundaries. **Proof-of-Holding**: ∅ = none, Snap = point-in-time proof, Δ = continuous window proof over $[t_1, t_2]$ returning only a boolean verdict.

ers provide strong unlinkability through Merkle commitments and nullifiers but lack a native compliance boundary and offer no continuous time proof of holding [21]. Privacy Pools introduces association set proofs of innocence for selective disclosure, but still focuses on withdrawals rather than interval ownership [4]. For NFTs, Aegis enables private swaps and hides traders, items, and prices, but its ownership attestations are point in time and later actions can correlate with a prior proof [16]. Systems aimed at auditability take different routes: Zcash supports viewing keys for selective disclosure, and zkLedger lets auditors verify aggregates on private ledgers, but does not provide a gate compatible with ERC-3643 style eligibility [29,20]. Railgun offers shielding for ERC-20 and ERC-721 with viewing keys and optional proof of innocence workflows, yet trade support is partial and it does not provide a continuous interval proof of holding or enforce permissioned entry and exit [6,7].

In this paper, we introduce *UltraMixer*, a noncustodial, ERC-3643 compatible privacy layer that supports private *deposit*, *withdraw*, *in-mixer transfer*, and *atomic trade* of heterogeneous assets (FTs and NFTs) under a formal minimal leakage model. At its core, *UltraMixer* combines (i) a Merkle committed whitelist of KYC-approved addresses proved in zero-knowledge, (ii) a generalized UTXO encoding that binds assets to unlinkable commitments and nullifiers, and (iii) two auxiliary zero-knowledge components: a *Proof of Maintain (PoM)* that lets permissionless maintainers update a global Sparse Merkle Tree of consumed nullifiers without root mismatches, and a *Δ-Window Privacy-Preserving Proof of Holding (PoH)* that attests to continuous ownership over a time interval while revealing only a boolean verdict by using the Sparse Merkle Tree of consumed nullifiers. *UltraMixer* is engineered for practicality on EVM chains: multi leaf inserts are aggregated into prover-supplied subtree roots (verified in circuit) to reduce gas, and regulator mandated controls (freeze and unfreeze, token pause, force transfer) are preserved through the whitelist commitment and ERC-3643 semantics, without re identifying ordinary users.

UltraMixer differs along three dimensions. First, it is asset general: fungibles and non fungibles can be deposited, withdrawn, transferred privately inside the mixer, and exchanged through atomic trades, whereas most prior systems provide only deposit–withdraw or payment flows [23,15,18,24,21]. Second, it integrates a compliance boundary by enforcing ERC-3643 eligibility at entry and exit; Tornado-style mixers do not natively enforce such gates, and zkLedger addresses compliance in a permissioned ledger rather than a public EVM context [21,20]. Third, it introduces a verdict-only $\Delta-$Window Proof of Holding that certifies continuous ownership over $[t_1, t_2]$ without leaking balances, identities, or linkages; Privacy Pools focuses on association set innocence rather than interval ownership, and Aegis offers snapshot proofs that can correlate with later actions [4,16]. Together with PoM for safe Sparse Merkle Trees (SMT) updates of consumed nullifiers, these features yield a compliance-aware and audit-friendly privacy layer for permissioned RWA tokens. Our contributions are summarized as follows:

1. We present *UltraMixer*, to our knowledge, the first noncustodial mixing protocol designed for permissioned tokenized real world assets. It supports private deposits, withdrawals, transfers, and atomic trades of heterogeneous assets while strictly enforcing ERC-3643 whitelist membership through zero-knowledge authorization, achieving strong unlinkability under a formal minimal leakage model.
2. We design two new zero-knowledge mechanisms enabling user to prove their holdings in a Privacy-Preserving manner: (i) *PoM*, which allows permissionless maintainers to securely update a global Sparse Merkle Tree of consumed nullifiers without root mismatches, and (ii) *Δ-Window Privacy-Preserving PoH*, which proves continuous ownership of assets over a time interval based on SMT of consumed nullifiers while revealing only a boolean outcome. These mechanisms enable regulatory audits, collateral checks, and market participation proofs without disclosing balances or linkages.
3. UltraMixer integrates gas-efficient Merkle insertion through prover-supplied subtree roots to reduce on chain costs, formalizes an ideal functionality $\mathcal{F}_{\mathsf{UltraMixer}}$ that captures its leakage and security properties under rational adversaries, and embeds regulator required controls such as freeze, pause, and force transfer through whitelist commitments, achieving both privacy and enforceability without custodianship.

2 *UltraMixer*: Definitions

This section defines *UltraMixer* and its ideal functionality.

Definition 1 (*UltraMixer*). Let $\mathcal{H}$ be a cryptographic hash function [19]. Let Sig be a digital signature scheme. Let $(\mathcal{P}_d, \mathcal{V}_d), (\mathcal{P}_w, \mathcal{V}_w), (\mathcal{P}_{id}, \mathcal{V}_{id}), (\mathcal{P}_t, \mathcal{V}_t), (\mathcal{P}_{td}, \mathcal{V}_{td}), (\mathcal{P}_{PoM}, \mathcal{V}_{PoM}), (\mathcal{P}_{PoH}, \mathcal{V}_{PoH})$ be seven ZK-SNARK prover and verifier pairs. Then we define *UltraMixer* as the protocol $\Pi_{UltraMixer} = \big(\mathcal{H}, Sig, (\mathcal{P}_d, \mathcal{V}_d), (\mathcal{P}_w, \mathcal{V}_w), (\mathcal{P}_{id}, \mathcal{V}_{id}), (\mathcal{P}_t, \mathcal{V}_t), (\mathcal{P}_{td}, \mathcal{V}_{td}), (\mathcal{P}_{PoM}, \mathcal{V}_{PoM}), (\mathcal{P}_{PoH}, \mathcal{V}_{PoH})\big)$.

Ideal Functionality $\mathcal{F}_{\mathsf{UltraMixer}}$. *Parties and Time.* Let $\mathcal{P}$ be the set of users; the adversary $\mathcal{A}$ may adaptively corrupt parties in $\mathcal{P}$. Time is discretized into

epochs $t \in \mathbb{N}$ (e.g., block heights). A trusted KYC authority maintains a whitelist $W \subseteq \mathcal{P}$ of authorized addresses; $\mathcal{F}_{\mathsf{UltraMixer}}$ can query membership in W.

Assets and Coins. There is a universe of asset identifiers $\mathsf{id} \in \mathcal{I}$. A *coin* (UTXO) is a tuple $u = (\mathsf{id}, n, \mathsf{owner}, \ell)$, where id is the asset id, $n \in \mathbb{N}$ the amount (for NFTs, $n = 1$), $\mathsf{owner} \in \mathcal{P}$ the holder, and ℓ a unique *label* (abstracting the on-chain commitment/leaf).

Internal State. $\mathcal{F}_{\mathsf{UltraMixer}}$ maintains:

– An *unspent-coin pool* U: a multiset of coins $u = (\mathsf{id}, n, \mathsf{owner}, \ell)$.
– A *spent-label set* $S \subseteq \{\ell\}$ (nullifiers) recording consumed labels.

All labels ℓ generated by $\mathcal{F}_{\mathsf{UltraMixer}}$ are unique.

Authorization. A request from party P is processed only if $P \in W$ at the processing epoch; otherwise it is ignored.

Leakage Model (to $\mathcal{A}$). $\mathcal{F}_{\mathsf{UltraMixer}}$ leaks only what an on-chain observer inherently learns. No identities, labels ℓ, internal linkages, or exact in-mixer amounts are leaked.

Operations. At epoch t, $\mathcal{F}_{\mathsf{UltraMixer}}$ handles the following commands.

1. Deposit. On input $(\mathsf{Deposit}, \mathsf{id}, n)$ from P:

1. If $P \notin W$, abort.
2. Sample fresh unique label $\ell \leftarrow \{0,1\}^\lambda$.
3. Add $u := (\mathsf{id}, n, P, \ell)$ to U.
4. *Leak:* output to $\mathcal{A}$: "a whitelisted user deposited (id, n)".

2. Withdraw. On input $(\mathsf{Withdraw}, \mathsf{id}, k, \mathsf{addr}_{\text{out}})$ from P:

1. If $P \notin W$, abort.
2. Let $U_P(\mathsf{id}) := \{u \in \mathsf{U} : u.\mathsf{owner} = P,\ u.\mathsf{id} = \mathsf{id}\}$ and $A := \sum_{u \in U_P(\mathsf{id})} u.n$. If $A < k$, abort.
3. Select $\{u_i = (\mathsf{id}, n_i, P, \ell_i)\}_{i=1}^{x} \subseteq U_P(\mathsf{id})$ with $\sum_i n_i \geq k$.
4. For each i: remove u_i from U, add ℓ_i to S
5. Let $T := \sum_i n_i - k$. If $T > 0$, sample fresh ℓ' and insert $(\mathsf{id}, T, P, \ell')$ into U.
6. Transfer k units of id to $\mathsf{addr}_{\text{out}}$ (external).
7. *Leak:* output to $\mathcal{A}$: "a withdrawal of (id, k) to $\mathsf{addr}_{\text{out}}$ occurred".

3. In-Mixer Transfer. On input $(\mathsf{Transfer}, \mathsf{id}, k, Q)$ from P:

1. If $P \notin W$ or $Q \notin W$, abort.
2. Let $U_P(\mathsf{id})$ and A as above. If $A < k$, abort.
3. Select $\{u_i = (\mathsf{id}, n_i, P, \ell_i)\}_{i=1}^{x}$ with $\sum_i n_i \geq k$.
4. For each i: remove u_i from U, add ℓ_i to S
5. Let $T := \sum_i n_i - k$. If $T > 0$, sample fresh ℓ'' and insert $(\mathsf{id}, T, P, \ell'')$ into U.
6. Sample fresh ℓ' and insert $(\mathsf{id}, k, Q, \ell')$ into U.
7. Notify Q privately of receipt (id, k).
8. *Leak:* output to $\mathcal{A}$: "an in-mixer transfer occurred".

4. Atomic Trade. On input $(\mathsf{Trade}, \gamma, \mathsf{id}_A, \theta, \mathsf{id}_B, Q)$ from P and a matching $(\mathsf{TradeAccept}, \gamma, \mathsf{id}_A, \theta, \mathsf{id}_B, P)$ from Q:

1. If $P \notin W$ or $Q \notin W$, abort.
2. Check P has $\geq \gamma$ of id_A and Q has $\geq \theta$ of id_B (via their unspent coins). If not, abort.

3. For P: select $\{u_i^A = (\mathsf{id}_A, n_i^A, P, \ell_i^A)\}$ with $\sum_i n_i^A \geq \gamma$; consume them (remove from U; add ℓ_i^A to S; let $T_A := \sum_i n_i^A - \gamma$. If $T_A > 0$, insert $(\mathsf{id}_A, T_A, P, \ell_A'')$ with fresh ℓ_A''.
4. For Q: select $\{u_j^B = (\mathsf{id}_B, n_j^B, Q, \ell_j^B)\}$ with $\sum_j n_j^B \geq \theta$; consume them; let $T_B := \sum_j n_j^B - \theta$. If $T_B > 0$, insert $(\mathsf{id}_B, T_B, Q, \ell_B'')$ with fresh ℓ_B''.
5. Insert $(\mathsf{id}_A, \gamma, Q, \ell_A')$ and $(\mathsf{id}_B, \theta, P, \ell_B')$ with fresh labels ℓ_A', ℓ_B'.
6. Notify P and Q privately of their new holdings.
7. *Leak:* output to $\mathcal{A}$: "an atomic trade occurred ".

5. Δ-Window Privacy-Preserving Proof of Holding. On input $(\mathsf{ProveHolding}, P, \mathsf{id}, \theta, t_1, t_2)$ from P, with $t_1 < t_2 \leq t$ where t is current time:

1. For each epoch $t' \in [t_1, t_2]$, define $H_{P,\mathsf{id}}(t') := \sum_{u \in \mathsf{U}(t'):\, u.\mathsf{owner}=P,\, u.\mathsf{id}=\mathsf{id}} u.n$, where $\mathsf{U}(t')$ is the set of coins considered unspent at epoch t' (coins exist from creation time until their consumption epoch).
2. If $\min_{t' \in [t_1,t_2]} H_{P,\mathsf{id}}(t') \geq \theta$, output VALID; else output $\mathsf{INVALID}$.
3. *Leak:* only the boolean result and the public tuple $(\mathsf{id}, \theta, t_1, t_2)$.

Privacy and Correctness Guarantees. By construction:

- **No double-spend:** each label ℓ is consumed at most once (enforced by S).
- **Whitelist enforcement:** only $P \in W$ can invoke state-changing operations.
- **Minimal leakage:** beyond on-chain inevitable facts (asset ids, external amounts, occurrence of operations, aggregate fees, public roots, and holding-proof result), no identities, labels, or linkages are revealed.
- **Privacy-Preserving of Δ-Window Proof of Holding:** the query returns only $\mathsf{VALID}/\mathsf{INVALID}$ for $(\mathsf{id}, \theta, t_1, t_2)$, revealing nothing about which coin(s) support the claim.

Threat Model & Security Assumptions. We assume all participants (users, maintainers, traders) are potentially **malicious but rational**, meaning they will follow or deviate from the protocol only if such action yield financial advantage. They may collude, adaptively corrupt, or attempt front-running and double-spending, but their incentives are bounded by profitability. Regulators, including the ERC-3643 registry agent and whitelist authority, are fully trusted to enforce compliance and maintain the integrity of the whitelist. The underlying blockchain is assumed secure, providing consensus safety, persistence, and liveness, ensuring that all valid transactions are eventually recorded and cannot be reverted except with negligible probability.

Extra Notations. We denote Merkle Tree by $\mathcal{MT}$ and Sparse Merkle Tree [9] by $\mathcal{SMT}$. Given a root digest $root$, a message m, and a Merkle path $path$, the verification procedure in a standard Merkle Tree is expressed as $\mathcal{MT}.\mathsf{Verify}(root, m, path) \in \{0, 1\}$, which outputs 1 if and only if m is a valid leaf under $root$ with the $path$.

For Sparse Merkle Trees, the verification procedure is defined as $\mathcal{SMT}.\mathsf{Verify}(root, path, m, b) \in \{0, 1\}$, where $b \in \{0, 1\}$ indicates whether the leaf corresponding to m is present ($b = 1$) or absent ($b = 0$). The procedure outputs 1 if and only if the claim is consistent with $root$ and the provided $path$.

3 *UltraMixer*: Implementation

UltraMixer provides privacy-preserving asset mobility: users can **deposit**, **withdraw**, **transfer** within the mixer, and **trade** among participants without revealing identities, balances, or asset types. It also supports selective proof of holding—*Δ-Window Privacy-Preserving Proof of Holding*—to attest ownership of specific assets over a time interval $[t_1, t_2]$ for audits, collateral checks, or market-participation proofs without disclosing anything else. A security analysis is in the Appendix.

Whitelist Compatibility and Address Authorization. To ensure regulatory compliance while preserving privacy, *UltraMixer* extends the ERC-3643 whitelist mechanism. In ERC-3643, authorized agents onboard users by verifying their identity through KYC and registering their blockchain addresses; *UltraMixer* preserves this delegation model but augments it with a dynamic Merkle tree $\mathcal{MT}^{\text{addr}}$ committing to all whitelisted addresses. When a user passes KYC, the agent inserts her address addr into $\mathcal{MT}^{\text{addr}}$, updates the root $\mathcal{MT}^{\text{addr}}.\mathsf{Root}$, and publishes it on-chain, allowing users to later prove membership without revealing their identity. Concretely, the user signs the current block height h with secret key sk to produce Sig^{addr}, and generates a zero-knowledge proof (***WhitelistCheck*** sub-circuit) attesting that $\mathcal{MT}.\mathsf{Verify}(\mathcal{MT}^{\text{addr}}.\mathsf{Root}, \mathsf{addr}, \mathsf{Path}) = 1 \wedge \mathsf{addr} = \mathsf{computeAddress}(pk) \wedge Sig.Verify(Sig^{addr}, h, pk)$ where $\mathcal{MT}^{\text{addr}}.\mathsf{Root}$ and h are public inputs, and all others are secret, with h serving as a nonce to prevent proof reuse. If an address's KYC expires or is revoked, the agent replaces its leaf in $\mathcal{MT}^{\text{addr}}$ with a null placeholder (e.g., $\mathsf{0x0}$), recomputes and publishes the updated root, and simultaneously unregisters the address in the ERC-3643 registry, ensuring consistency between the compliance layer and the *UltraMixer* whitelist commitment.

Asset Construction. *UltraMixer* adopts the Unspent Transaction Output (UTXO) model and extends the cryptographic primitives pioneered by Tornado Cash, particularly the use of *leaves* and *nullifiers*. To support broader asset types and composability, *UltraMixer* generalizes the UTXO model to encode arbitrary on-chain assets. Each UTXO encapsulates four components: a user-generated secret value s, an asset identifier id (e.g., RWA token ID or other on-chain native token ID), a quantity n denoting the amount of that asset and the address addr which the user uses for registration. From these, the leaf is constructed as $L = \mathcal{H}(\mathcal{H}(s, 1, \mathsf{addr}), \mathsf{id}, n)$ where $\mathcal{H}$ is a cryptographic hash function. The corresponding nullifier is derived as $N = \mathcal{H}(\mathcal{H}(s, 2, \mathsf{addr}), \mathsf{id}, n)$. This UTXO-based construction is expressive enough to represent all categories of on-chain assets. For *fungible tokens*, including native tokens (e.g., stablecoins) and tokenized RWA (e.g., tokenized stocks), the field id uniquely identifies the token types, while n denotes the quantity of tokens committed. For *non-fungible tokens* (NFTs), which include both standard on-chain collectibles and tokenized RWA such as digital property deeds or certificates, the convention is that $n = 1$ and the field id uniquely identifies the NFT itself.

3.1 On-Chain Activities

Deposit & Withdrawal Protocols. A registered user may initiate private interactions with assets by depositing them into the *UltraMixer* through the *Deposit Protocol*. To begin, the depositor samples a random secret s and computes the corresponding leaf $L = \mathcal{H}(\mathcal{H}(s, 1, \mathsf{addr}), \mathsf{id}, n)$. To avoid collisions and ensure unlinkability, the user must regenerate s if L has appeared previously on-chain.

To authorize the deposit, the user generates a zero-knowledge proof π_d through $\mathcal{P}_d$ attesting to the following statement: $L = \mathcal{H}(\mathcal{H}(s, 1, \mathsf{addr}), \mathsf{id}, n) \wedge$ *Whitelist-Check* where L, id, n and public inputs in *WhitelistCheck* are public witness; meanwhile, others are private. The addr is the same as the addr in *WhitelistCheck*. This proof ensures that the leaf L has been correctly constructed for the specified asset and amount, and that the depositing party is indeed a whitelisted user whose address is included in the latest root of the address Merkle tree. The user then sends a transaction that attaches π_d and enough assets to the smart contract. Upon successful verification by on-chain $\mathcal{V}_d$, the *UltraMixer* contract records the leaf L in the asset Merkle Tree $\mathcal{MT}^{ast}$ and transfers the specified asset (id, n) from the depositor's address into the custody of the mixer.

UltraMixer allows registered users to reclaim assets via the *Withdraw Protocol*, which enables the secure and private redemption of tokens held within the mixer. Suppose a user wishes to withdraw k units of an asset identified by token ID id. To do so, the user selects a set of previously deposited leaves and corresponding Merkle Paths in $\mathcal{MT}^{asst}$: $\{L_i\}_{i=1}^{x}$ and $\{\mathsf{path}_i\}_{i=1}^{x}$, each corresponding to a UTXO encoding ownership of (id, n_i), such that the total sum satisfies $\sum_{i=1}^{x} n_i \geq k$. Let $T = \sum_{i=1}^{x} n_i - k$ denote the remainder, which must be preserved for future use.

The user constructs a new leaf $L' = \mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}), \mathsf{id}, T)$, representing the leftover balance. Additionally, for each consumed leaf L_i, the user computes the corresponding nullifier $N_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}, n_i)$, The sender should also provide payment to maintainers by following the process shown in Section 3.2: *Maintainer Fee Source*. To authorize the withdrawal, the user generates a zero-knowledge proof π_w through $\mathcal{P}_w$ attesting to the following statement:

$$\bigwedge_{i=1}^{x} \Big(\mathcal{MT}.\mathsf{Verify}(\mathcal{MT}^{\text{ast}}.Root, \mathsf{path}_i, L_i) = 1 \wedge L_i = \mathcal{H}(\mathcal{H}(s_i, 1, \mathsf{addr}), \mathsf{id}, n_i) \wedge N_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}, n_i) \Big) \bigwedge Maintenance \bigwedge WhitelistCheck \bigwedge \Big(\mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}), \mathsf{id}, T) = L' \Big) \bigwedge \Big(T = \sum_{i=1}^{x} n_i - k \geq 0 \Big)$$

where $\{s_i\}_{i=1}^{x}$, $\{n_i\}_{i=1}^{x}$,$\{L_i\}_{i=1}^{x}$,s', $\{\mathsf{path}_i\}_{i=1}^{x}$, addr and other secrets in *WhitelistCheck* and *Maintenance* (demonstrate in Section 3.2) are private inputs to the circuit, while others are public. The user (or a relayer) then sends a transaction with π_w to the smart contract. After verifying the proof by using $\mathcal{V}_w$, the *UltraMixer* contract checks that none of the nullifiers $\{N_i\}_{i=1}^{x}$ have been spent before. If success, it adds $\{N_i\}_{i=1}^{x}$ into ***Pool*** (a nullifier pool to store all nullifiers that had already been consumed, but had not been inserted into the Sparse Merkle tree. This concept will be explained in Section 3.2), transfers k tokens of asset id to the user's address, and records all $\{N_i\}_{i=1}^{x}$ as consumed.

Transfer Protocol. Registered users may transfer assets privately to other registered users through the *Transfer Protocol*. Suppose a sender intends to transfer k tokens of type id to a designated receiver (with address addr'). To prepare for the transfer, the receiver generates a fresh secret s' and computes a new leaf $L' = \mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}'), \mathsf{id}, k)$, which represents the commitment to the transferred tokens, following the same structure as in the deposit process. To demonstrate her eligibility to receive assets, the receiver then constructs a zero-knowledge proof π_{id} through $\mathcal{P}_{id}$ for the following statement: *WhitelistCheck* $\wedge$ $L' = \mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}'), \mathsf{id}, k)$. Here, $s', \mathsf{id}, k, \mathsf{addr}'$ and other secret inputs in *WhitelistCheck* are private, and others are public. This proof guarantees that the receiver is a registered user and that it is bound to the specific commitment L', thereby preventing reuse and frontrunning attacks. The receiver then transmits π_{id} together with the values $\bar{s} = \mathcal{H}(s', 1, \mathsf{addr}'), \mathsf{id}, k$ to the sender. Upon receiving this package, the sender (with address addr) executes a withdrawal-like procedure. She collects a set of leaves $\{L_i\}_{i=1}^{x}$ corresponding to her unspent UTXOs of token type id, such that $\sum_{i=1}^{x} n_i \geq k$. To preserve any remaining balance, the sender constructs a new remainder leaf: $L'' = \mathcal{H}(\mathcal{H}(s'', 1, \mathsf{addr}), \mathsf{id}, T)$, where $T = \sum_{i=1}^{x} n_i - k$. For each consumed leaf L_i, the sender obtains the corresponding nullifier $N_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}, n_i)$. Finally, the sender generates a second proof, π_t through $\mathcal{P}_t$, to attest to the correctness of the transfer. The proven statement is:

$$\bigwedge_{i=1}^{x} \Big(\mathcal{MT}.\mathsf{Verify}(\mathcal{MT}^{\text{ast}}.\mathit{Root}, \mathsf{path}_i, L_i) = 1 \wedge L_i = \mathcal{H}(\mathcal{H}(s_i, 1, \mathsf{addr}), \mathsf{id}, n_i) \wedge N_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}, n_i) \Big) \bigwedge \mathit{WhitelistCheck} \bigwedge \mathit{Maintenance} \bigwedge \Big(\mathcal{H}(\mathcal{H}(s'', 1, \mathsf{addr}), \mathsf{id}, T) = L'' \wedge \mathcal{H}(\bar{s}, \mathsf{id}, k) = L' \wedge T = \sum_{i=1}^{x} n_i - k \geq 0 \Big)$$

Here, $\mathcal{MT}^{\text{ast}}.\mathit{Root}$, $\{N_i\}_{i=1}^{x}$, L', L'', and the public inputs of *WhitelistCheck* and *Maintenance* are public, while all other values remain private.

Finally, the sender submits both π_t and π_{id} to the mixer smart contract. The contract verifies the validity of both proofs by using $\mathcal{V}_{id}$ and $\mathcal{V}_t$ and then ensures that none of the nullifiers have been previously used. If all checks succeed, it inserts the newly created leaves L' and L'' into $\mathcal{MT}^{ast}$, adds $\{N_i\}_{i=1}^{x}$ to the *Pool*, and marks $\{N_i\}_{i=1}^{x}$ as consumed.

Trading Protocol. *UltraMixer* enables registered users to perform secure and privacy-preserving asset trades, for example, using stablecoins to purchase tokenized RWAs. The trading process consists of two phases: an off-chain phase (price confirmation) and an on-chain phase (finalization).

Consider the following scenario. Alice holds stablecoins id_{stb} in *UltraMixer*, and she wishes to purchase tokenized stock assets id_{ast}. Bob, who owns such assets in the mixer, agrees to trade. Suppose Alice proposes a deal to exchange γ id_{stb} for θ id_{ast}, and Bob accepts this price. They then enter the off-chain phase. First, Alice and Bob agree on a **one-time** commitment key k through a key-exchange protocol. Each party generates a **one-time** signature key pair: (sk_A, pk_A) for Alice and (sk_B, pk_B) for Bob. Both parties compute a commitment using k: $\hat{P} = \mathcal{H}(k, \gamma, \theta, id_{stb}, id_{ast})$, and sign $\hat{P}$ with their respective secret

keys: $Sig^A_{\hat{P}} = Sig.Sign(sk_A, \hat{P}), Sig^B_{\hat{P}} = Sig.Sign(sk_B, \hat{P})$. Next, they exchange signatures. After receiving the other's signature, each party selects enough leaves to cover the trading amounts, denoted $\{L^A_i\}^x_{i=1}$ for Alice and $\{L^B_i\}^y_{i=1}$ for Bob. Each then generates fresh secrets and prepares two additional leaves: (L'_A, L''_A) for Alice and (L'_B, L''_B) for Bob. Here, L' represents the leaf storing the traded assets, while L'' represents the remainder leaf holding the tokens left after the trade. Finally, both Alice and Bob generate zero-knowledge proofs, denoted π^A_{td} and π^B_{td} through $\mathcal{P}_{td}$, respectively. Since the statements proven are analogous, we describe only Alice's version. The proof π^A_{td} establishes the following statement:

$$\bigwedge_{i=1}^{x} \Big(\mathcal{MT}.\mathsf{Verify}(\mathcal{MT}^{\text{ast}}.Root, \mathsf{path}_i, L^A_i) = 1 \ \wedge \ L^A_i = \mathcal{H}(\mathcal{H}(s_i, 1, \mathsf{addr}), \mathsf{id}, n_i) \\ \wedge \ N^A_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}, n_i)\Big) \bigwedge \mathit{WhitelistCheck} \bigwedge \mathit{Maintenance} \bigwedge$$

$$\Big(\hat{P} = \mathcal{H}(k, \gamma, \theta, id_{stb}, id_{ast}) \ \wedge \ T = \sum_{i=1}^{x} n_i - \gamma \ \wedge \ T \geq 0\Big) \bigwedge$$

$$\Big(\mathcal{H}(\mathcal{H}(s'', 1, \mathsf{addr}), \mathsf{id}_{\mathsf{stb}}, T) = L''_A \ \wedge \ \mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}), \mathsf{id}_{\mathsf{ast}}, \theta) = L'_A \Big) \bigwedge$$

$$\Big(Sig.Verify(Sig^A_{\hat{P}}, \hat{P}, pk_A) = 1 \ \wedge \ Sig.Verify(Sig^B_{\hat{P}}, \hat{P}, pk_B) = 1\Big)$$

Here, $\mathcal{MT}^{\text{ast}}$.Root, $\{N^A_i\}^x_{i=1}$, $\{\hat{N}^A_i\}^x_{i=1}$, $\hat{P}$, $Sig^A_{\hat{P}}$, $Sig^B_{\hat{P}}$, L', L'', and the public inputs of *WhitelistCheck* and *Maintenance* are public, while all other values remain private. When switching to Bob's view, the proof statement is identical except that all inputs are replaced with Bob's corresponding values.

After generating both π^A_{td} and π^B_{td}, Bob sends π^B_{td} together with the related public inputs to Alice. Alice then submits both proofs, along with all public inputs, to the blockchain, thereby entering the on-chain phase. The *UltraMixer* smart contract first checks that all nullifiers in both proofs have not been consumed, and that $\hat{P}$ and the signature pairs used in verifying each proof are consistent across both proofs. If these checks pass, the contract verifies the proofs through $\mathcal{V}_{td}$. Once verification succeeds, the contract inserts all four leaves L'_A, L''_A, L'_B, L''_B into $\mathcal{MT}^{\text{ast}}$. It then marks all nullifiers as consumed and adds $\{N^A_i\}^x_{i=1}$ to the *Pool*.

Tricks on $\mathcal{MT}$ insertion. In the above operations, *UltraMixer* inserts one or more leaves into $\mathcal{MT}^{\text{ast}}$ within the smart contract. Inserting a single leaf into a depth-32 Merkle tree consumes over 1.5 million gas, which is considerable. In the *Transfer* protocol, three leaves must be inserted (two for the transfer protocol itself, one for paying the maintenance fee), while in the *Trading* protocol, this number increases to five (four for the trading protocol itself, one for paying the maintenance fee). To reduce the insertion cost, during *Transfer* and *Trading*, the sender calculates a subtree containing all leaves that need to be inserted, and the smart contract records only the root of this subtree in $\mathcal{MT}^{\text{ast}}$. In the corresponding proof, the sender must additionally demonstrate the correctness of the subtree root. As a result, $\mathcal{MT}^{\text{ast}}$ may contain two or three extra layers for certain leaves. To ensure circuit compatibility, users must provide a Merkle Path of maximum depth in $\mathcal{MT}^{\text{ast}}$. If the path does not reach the maximum depth, dummy values are appended to complete it.

3.2 Proof of Holding & Maintain

UltraMixer enables users to prove their holdings to both on-chain and off-chain verifiers under the functionality of *Δ-Window Privacy-Preserving Proof of Holding*. At a high level, a user must demonstrate that she possesses a valid, unconsumed nullifier corresponding to a specific token ID and amount, while efficiently hiding all information about the nullifier itself. An intuitive approach is to use a Sparse Merkle Tree (SMT) to record all consumed nullifiers, with the SMT root stored in the smart contract. More precisely, each leaf in the SMT corresponds to a nullifier: for a consumed nullifier, the leaf value is $L = 1$; for an unconsumed nullifier, the leaf value is $L = 0$. In *UltraMixer*, a group of maintainer nodes actively update the SMT using the *Proof-of-Maintain* protocol. Users are not permitted to update the SMT directly on-chain, since updating a depth-253 SMT would cost over 10 million gas. Alternatively, allowing users to update the SMT within a zero-knowledge proof and requiring the contract to only store the root introduces a **root-mismatch** problem: the SMT root referenced in the zero-knowledge proof may differ from the root at the time the transaction is executed by blockchain miners. To address this issue, *UltraMixer* delegates SMT updates to maintainers, treating the process analogously to mining. This subsection first details how maintainers update the SMT, and then explains how users prove their holdings to others.

Proof of Maintain (PoM). *Proof of Maintain* enables maintainer nodes to earn rewards by updating the SMT that records consumed nullifiers. Let $\mathcal{SMT}^{\mathsf{N}}$ denote the Sparse Merkle Tree that stores all consumed nullifiers. As described in the previous protocols, whenever a user consumes a nullifier, the corresponding nullifier is marked as consumed in the smart contract. Before these newly consumed nullifiers are inserted into $\mathcal{SMT}^{\mathsf{N}}$, they are temporarily stored in a waiting pool, denoted *Pool*, inside the contract. Once inserted into $\mathcal{SMT}^{\mathsf{N}}$, the nullifiers are removed from *Pool*.

We now illustrate the update process with a concrete example. Suppose the *Pool* contains a set of nullifiers $\{N_i\}_{i=1}^{x}$. Each maintainer keeps a full local copy of $\mathcal{SMT}^{\mathsf{N}}$. The maintainer first updates her local tree and records the necessary information for each step, which will later be used to prove the correctness of the update. The update process proceeds as follows. The maintainer locates N_1 in $\mathcal{SMT}_0^{\mathsf{N}}$ (the tree before any updates) and records its Merkle path path_1. She then updates the corresponding leaf in $\mathcal{SMT}^{\mathsf{N}}$ from 0 to 1, and recursively recomputes the node values along path_1, and obtains the updated tree $\mathcal{SMT}_1^{\mathsf{N}}$. Next, she identifies and records the Merkle path path_2 in $\mathcal{SMT}_1^{\mathsf{N}}$ for N_2, updates the corresponding leaf, and derives $\mathcal{SMT}_2^{\mathsf{N}}$. This process continues until all $\{N_i\}_{i=1}^{x}$ are updated, yielding the final tree $\mathcal{SMT}_x^{\mathsf{N}}$. Each recorded path path_i has the same length. In practice, some SMT implementations optimize storage by omitting subtrees in which all leaves hold the default value (representing unconsumed nullifiers) and replacing them with a predetermined root value. In this case, the maintainer must extend the Merkle path down to the leaf.

Once all paths $\{\mathsf{path}_i\}_{i=1}^{x}$ are collected, the maintainer generates a zero-knowledge proof π_{PoM} attesting to the following statement:

$$\bigwedge_{i=1}^{x} \Big(\hat{N}_i = \mathcal{H}(N_i) \wedge \mathcal{SMT}.\mathsf{Verify}(\mathcal{SMT}^{\mathsf{N}}_{i-1}.\mathit{Root}, \mathsf{path}^{\mathsf{N}}_i, \hat{N}_i, 0) = 1 \wedge \mathcal{SMT}.\mathsf{Verify}(\mathcal{SMT}^{\mathsf{N}}_{i}.\mathit{Root}, \mathsf{path}^{\mathsf{N}}_i, \hat{N}_i, 1) = 1 \Big) \bigwedge \mathsf{addr} = \mathsf{addr}$$

Here, addr (the maintainer's address, used to protect against frontrunning attacks), $\{N_i\}_{i=1}^{x}$, $\mathcal{SMT}^{\mathsf{N}}_0$.Root, and $\mathcal{SMT}^{\mathsf{N}}_x$.Root are public inputs, while all other values are private. This proof guarantees that the maintainer has correctly updated $\mathcal{SMT}^{\mathsf{N}}$ with respect to $\{N_i\}_{i=1}^{x}$.

Afterwards, the maintainer submits a transaction to the smart contract containing π_{PoM} and the public inputs. The contract first checks whether $\mathcal{SMT}^{\mathsf{N}}_0$.Root matches the root currently stored on-chain, and then verifies that all $\{N_i\}_{i=1}^{x}$ are present in the pool. If these checks succeed, the contract verifies π_{PoM} through $\mathcal{V}_{PoM}$. Once verification is successful, the contract removes $\{N_i\}_{i=1}^{x}$ from the *Pool* and rewards the maintainer's address. Finally, the contract inserts the $\mathcal{H}(\mathcal{SMT}^{\mathsf{N}}_x.\mathit{Root}, t', t)$ into $\mathcal{MT}^{\mathit{Root}}$, where t' is the block height when the last maintainer successfully maintains the $\mathcal{SMT}^{N}$ and t is the current block height.

Maintainer Fee Source. The profit earned by maintainers originates from users who consume nullifiers. A simple pricing model assumes that inserting each consumed nullifier into $\mathcal{SMT}^{\mathsf{N}}$ costs a fixed fee of m tokens of type id_p, where id_p is a stablecoin only used for maintainer fee. (This work does not attempt to optimize or analyze the pricing strategy, but only demonstrates the protocol for performing the payment.) Accordingly, every user must hold sufficient mixed tokens id_p in order to consume other types of mixed assets. Specifically, when a user consumes x nullifiers, she must also spend $x \cdot m$ tokens of type id_p.

The payment process for maintenance fees follows the same procedure as the *Withdraw* protocol. The user first locates a sufficient set of leaves $\{L_i\}_{i=1}^{z}$ corresponding to tokens of type id_p. From these, she derives the associated nullifiers $\{N_i\}_{i=1}^{z}$ and paths $\{\mathsf{path}_i\}_{i=1}^{z}$. Note that *UltraMixer* does not provide functionality for users to prove holdings of id_p. As a result, maintainers are not required to insert the consumed nullifiers of id_p into $\mathcal{SMT}^{\mathsf{N}}$. Therefore, the zero-knowledge proof used by the user to pay the maintenance fee must demonstrate the following statements, captured in the ***Maintenance*** sub-circuit:

$$\bigwedge_{i=1}^{z} \Big(\mathcal{MT}.\mathsf{Verify}(\mathcal{MT}^{\mathrm{ast}}.\mathit{Root}, \mathsf{path}_i, L_i) = 1 \wedge L_i = \mathcal{H}(\mathcal{H}(s_i, 1, \mathsf{addr}), \mathsf{id}_p, n_i) \wedge N_i = \mathcal{H}(\mathcal{H}(s_i, 2, \mathsf{addr}), \mathsf{id}_p, n_i) \Big) \bigwedge \Big(\mathcal{H}(\mathcal{H}(s', 1, \mathsf{addr}), \mathsf{id}_p, T) = L' \Big) \bigwedge \Big(T = \sum_{i=1}^{x} n_i - m \cdot x \geq 0 \Big)$$

Here, $\mathcal{MT}^{\mathrm{ast}}$.Root, $\{N_i\}_{i=1}^{x}$, and L' are public inputs, while all other values remain private. Zero-knowledge proofs that consume nullifiers (*Withdraw* π_w, *Transfer* π_t, and *Trading* π_{td}) incorporate this sub-circuit. Once the contract verifies the proof, $m \cdot x$ tokens of type id_p in the deposit pool are left without an owner. After the maintainer updates $\mathcal{SMT}^{\mathsf{N}}$, the contract transfers these id_p tokens to the maintainer as compensation.

Δ-Window Privacy-Preserving Proof of Holding (PoH). A user can leverage $\mathcal{SMT}^{\mathsf{N}}$, whose root stored on-chain, to prove asset ownership to verifiers (either on-chain or off-chain). Consider the following scenario: Alice wishes to participate in an event that requires all participants to hold a certain NFT RWA id_{ast} continuously from time t_1 (e.g., a block height) to time t_2. Alice wants to prove that she satisfies this condition **without** revealing any additional information (including exact balance, how and when spending tokens after t_2).

Assume Alice has a nullifier N, corresponding to a leaf L), which corresponds to id_{ast}, throughout the interval $[t_1, t_2]$. Let $\mathcal{SMT}^{\mathsf{N}}_{t_1}$ and $\mathcal{SMT}^{\mathsf{N}}_{t_2}$ denote the SMTs recording consumed nullifiers at block heights t_1 and t_2, respectively. Let $\mathcal{MT}^{\mathsf{ast}}_{t_1}$ and $\mathcal{MT}^{\mathsf{ast}}_{t_2}$ denote the asset Merkle trees at block heights t_1 and t_2.

Alice first obtains the paths $\mathsf{path}^{Root}_{t1}$ and $\mathsf{path}^{Root}_{t2}$ for $\mathcal{SMT}^{N}_{t1}.Root$ and $\mathcal{SMT}^{N}_{t2}.Root$ in $\mathcal{MT}^{Root}$. Alice then obtains the paths $\mathsf{path}^{N}_{t_1}$ and $\mathsf{path}^{N}_{t_2}$ for N in $\mathcal{SMT}^{\mathsf{N}}_{t_1}$ and $\mathcal{SMT}^{\mathsf{N}}_{t_2}$, respectively. She also obtains $\mathsf{path}^{\mathsf{ast}}_{t_1}$ and $\mathsf{path}^{\mathsf{ast}}_{t_2}$, which are the paths for L in $\mathcal{MT}^{\mathsf{ast}}_{t_1}$ and $\mathcal{MT}^{\mathsf{ast}}_{t_2}$. Finally, Alice must prove knowledge of the secret underlying the nullifier, that the nullifier is valid and unspent during the interval $[t_1, t_2]$, and that the corresponding asset amount satisfies the event's threshold requirement. To do this, Alice generates a zero-knowledge proof π_{PoH} by using $\mathcal{P}_{PoH}$, attesting to the following statement:

$$\mathcal{MT}.Verify(\mathcal{MT}^{Root}.Root, \mathsf{path}^{Root}_{t1}, \mathcal{H}(\mathcal{SMT}^{\mathsf{N}}_{t1}.Root, t'_1, t_1)) \wedge$$

$$\mathcal{MT}.Verify(\mathcal{MT}^{Root}.Root, \mathsf{path}^{Root}_{t2}, \mathcal{H}(\mathcal{SMT}^{\mathsf{N}}_{t2}.Root, t'_2, t_2)) \wedge$$

$$L = \mathcal{H}(\mathcal{H}(s, 1, \mathsf{addr}), \mathsf{id}_{\mathsf{ast}}, 1) \ \wedge \ \mathcal{MT}.Verify(\mathcal{MT}^{\mathsf{ast}}_{t1}.Root, \mathsf{path}^{\mathsf{ast}}_{t1}, L) = 1 \wedge$$

$$\mathcal{MT}.Verify(\mathcal{MT}^{\mathsf{ast}}_{t2}.Root, \mathsf{path}^{\mathsf{ast}}_{t2}, L) = 1 \ \wedge \ N = \mathcal{H}(\mathcal{H}(s, 2, \mathsf{addr}), \mathsf{id}_{\mathsf{ast}}, 1)$$

$$\hat{N} = \mathcal{H}(N) \ \wedge \ \mathcal{SMT}.\mathsf{Verify}(\mathcal{SMT}^{\mathsf{N}}_{t1}.Root, \mathsf{path}^{\mathsf{N}}_{t1}, \hat{N}, 0) = 1 \wedge$$

$$\mathcal{SMT}.\mathsf{Verify}(\mathcal{SMT}^{\mathsf{N}}_{t2}.Root, \mathsf{path}^{\mathsf{N}}_{t2}, \hat{N}, 0) = 1$$

Here, id_{ast}, $\mathcal{MT}^{\mathsf{ast}}_{t_1}$.Root, $\mathcal{MT}^{\mathsf{ast}}_{t_2}$.Root, $\mathcal{MT}^{Root}.Root$, t_1, and t_2 are public inputs, while all other values remain private. Since in *UltraMixer* the $\mathcal{SMT}$ only supports insertions of consumed nullifiers and does not allow removals, the proof guarantees that the prover held token of type id_{ast} continuously from t_1 to t_2. Moreover, because the public inputs reveal nothing about N or L—nor any commitments or derived values—the verifier cannot infer when the user performed actions involving these assets. In the fungible asset case, proofs can be batched to attest multiple nullifiers, and the time window can be partitioned into segments to accommodate actions within the window that consume a nullifier; time continuity remains guaranteed by the interval committed in $\mathcal{MT}^{Root}$.

3.3 Discussion on Emergency Measures

The above protocol description illustrates the basic functionality of the RWA mixer. However, tokenized RWA standards like ERC-3643 also require strict *Emergency Measures*, which mainly include *account freeze/unfreeze*, *force transfer*, and *token pause*. This section discusses potential approaches to implementing these measures within the *UltraMixer* framework.

For *account freeze/unfreeze*, one option is to control user activity via the whitelist mechanism by temporarily removing the user's address addr from $\mathcal{MT}^{\mathsf{addr}}$.

This prevents the user from performing any actions on her mixed assets. For *token pause*, a possible design is to let the regulator maintain a dedicated Merkle tree of unpaused token identifiers. During on-chain actions, users must provide the proof including that the token id they are acting on with belongs to the regulator's unpaused-token tree. *Force transfer* is more challenging, since *UltraMixer* relies on users' knowledge of secret values to authorize transactions. Suppose the regulator intends to forcibly transfer k tokens of type id from Alice. The regulator first freezes Alice's address in the mixer, preventing her from executing further actions, until she withdraws the k tokens of type id. (Note that this withdrawal does not require a whitelist check, but instead requires approval from the regulator—specifically, a regulator-signed message bound to a random challenge.) After this step, the regulator leverages the ERC-3643 force-transfer function to finish the transfer.

Remark 1. *UltraMixer* can optionally attach an encrypted audit capsule (*Eye* in [17]) to each action. Under a threshold regulator key, the capsule reveals the KYC-bound address and operation metadata to only authorized auditors, enabling due-process tracing without public leakage.

4 Experiment

Experiment Setting. We implement the *UltraMixer* protocol using Golang for circuit design and Solidity for the smart contract. Specifically, we employ Groth-16 [14] as the underlying ZK-SNARK protocol and construct R1CS circuits with Gnark [3]. Proof generation is benchmarked on a MacBook with an M1 Pro chip and 32 GB of memory. For construction details, we adopt MiMC as the hash function and ECDSA as the signature scheme. $\mathcal{H}$. The Merkle depth of $\mathcal{MT}^{\mathsf{ast}}$ and $\mathcal{MT}^{\mathsf{addr}}$ is set to 32, consistent with the Tornado Cash configuration. The Merkle depth of $\mathcal{SMT}^{\mathsf{N}}$ is set to 254, since each nullifier is represented by a single element in R1CS, and the circuit is compiled over curve BN-254, giving a maximum nullifier bit length of 254. We evaluate the system by collecting gas consumption, circuit size, proof generation time, and verification time for both user and maintainer operations.

Experiment Results. Figure 1 presents the circuit statistics for user actions,

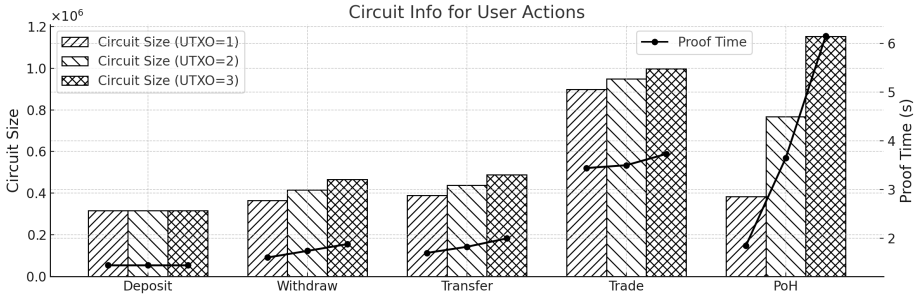

Fig. 1. This figure shows the circuit size and proof time for all user actions' proofs: *Deposit*(π_d), *Withdraw* (π_w), *Transfer* (π_t), *Trade* (π_{td}), *Proof of Holding* (π_{PoH}) with different numbers of UTXO Usages. Note: since the π_{id} in *Transfer* has the same construction with π_d, this figure will only give the info about π_d

while Figure 2 reports the gas cost of each action when using 1, 2, or 3 UTXOs per transaction. We restrict our evaluation to this range because, on average, Bitcoin transactions consume 2.26 UTXOs, and 94.97% of transactions consume fewer than 3 UTXOs [11]. As the number of consumed UTXOs increases, the gas cost, circuit size, and proof generation time also increase for all actions except *Deposit*, which does not consume UTXOs. The overall performance remains practical: proof generation time is below 6 seconds for all actions, while the on-chain cost is approximately 2.5 million gas (around $2.38 on Ethereum [2]). Verification is efficient, since the Groth-16 verifier runs in constant time, with proof verification taking only 0.001 seconds across all proofs. It is worth noting that the circuit size and proof generation time for *PoH* grow faster than for other actions. This is because *PoH* requires verifying a depth-254 Sparse Merkle Tree path for each consumed UTXO. However, from the gas-cost perspective, *PoH* is more efficient, since it does not require inserting any leaves into the Merkle Tree.

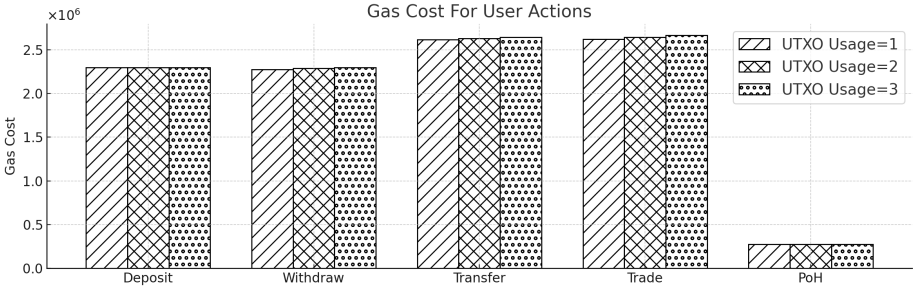

Fig. 2. This figure shows the gas cost for all user actions: *Deposit, Withdraw, Transfer, Trade, Proof of Holding* with different numbers of UTXO Usages

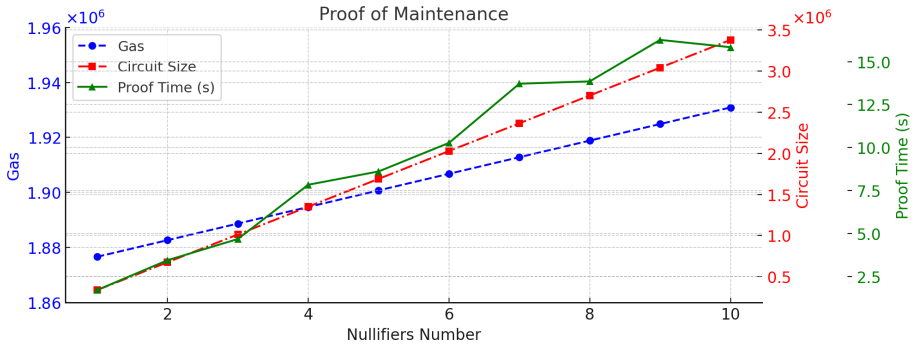

Fig. 3. This figure shows gas cost, circuit size (π_{PoM}), proof time for the Proof of Maintenance Protocol when collecting different numbers of nullifiers.

Figure 3 reports the gas cost, circuit size, and proof generation time for the *Proof-of-Maintenance* action. We evaluate transactions containing between 1 and 10 nullifiers. As expected, gas cost, circuit size, and proof time all increase with the number of nullifiers. Nevertheless, both the computational and monetary overhead remain practical even for 10 nullifiers: the proof consumes approxi-

mately 1.93 million gas (about $1.83 on Ethereum [2]) and requires around 16 seconds to generate π_{PoM}.

5 Conclusion

UltraMixer reconciles enforceable compliance with principled privacy for tokenized RWAs. By enforcing ERC-3643 whitelist membership at the boundary and using a generalized UTXO with commitments and nullifiers inside, it supports private deposit, withdraw, in-mixer transfer, and atomic trade across fungible and non-fungible assets. Proof of Maintain enables safe SMT updates, and the Δ-Window Privacy-Preserving Proof of Holding provides selective disclosure without revealing balances, identities, linkages, and following operations. Gas-aware batching and emergency controls preserve practicality and governance.

References

1. General data protection regulation (gdpr), regulation (eu) 2016/679. https://gdpr-info.eu/, official consolidated text as of May 25, 2018; accessed on September 21, 2025
2. (Sep 2025), https://etherscan.io/gastracker
3. Botrel, G., Piellard, T., Housni, Y.E., Kubjas, I., Tabaie, A.: Consensys/gnark: v0.14.0 (Jun 2025). https://doi.org/10.5281/zenodo.5819104, https://doi.org/10.5281/zenodo.5819104
4. Buterin, V., et al.: Blockchain privacy and regulatory compliance: Towards a practical equilibrium. SSRN preprint (2023), https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4563364
5. Cernescu, D.: Erc-3643 leads rise in compliant tokenized asset standards. The Paypers (Jul 2025), https://thepaypers.com/crypto-web3-and-cbdc/news/erc-3643-leads-rise-in-compliant-tokenized-asset-standards, accessed: September 21, 2025
6. contributors, R.: Private proofs of innocence. Protocol documentation (2025), https://docs.railgun.org/wiki/assurance/private-proofs-of-innocence
7. contributors, R.: Wallets and viewing keys. Protocol documentation (2025), https://docs.railgun.org/wiki/learn/wallets-and-keys
8. COTI: Unlocking confidential real-world assets (rwas) with coti. https://cotinetwork.medium.com/unlocking-confidential-real-world-assets-rwas-with-coti-5b5837d53146 (September 2024), medium blog post, accessed 2025-08-10
9. Dahlberg, R., Pulls, T., Peeters, R.: Efficient sparse merkle trees: Caching strategies and secure (non-) membership proofs. In: Nordic Conference on Secure IT Systems. pp. 199–215. Springer (2016)
10. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. arXiv preprint arXiv:1904.05234 (2019)
11. Delgado-Segura, S., Pérez-Sola, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: Analysis of the bitcoin utxo set. In: International Conference on Financial Cryptography and Data Security. pp. 78–91. Springer (2018)
12. ERC3643 Association: Erc-3643 permissioned tokens (2024), https://docs.erc3643.org/erc-3643, accessed: September 21, 2025
13. Ethereum Improvement Proposals: Eip-20: Token standard (erc-20). https://eips.ethereum.org/EIPS/eip-20 (2015), accessed: September 21, 2025

14. Groth, J.: On the size of pairing-based non-interactive arguments. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 305–326. Springer (2016)
15. Heilman, E., Alshenibr, L., Baldimtsi, F., Scafuro, A., Goldberg, S.: Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In: Network and distributed system security symposium (2017)
16. Hisham, S.: Aegis: Privacy-preserving market for non-fungible tokens. Preprint / project description (2023), https://www.semanticscholar.org/paper/8e877b3324c215f4eec5431a6be913a69668dee1, privacy-preserving NFT swap using zkSNARKs
17. Li, Z., Ni, W., Zheng, S., Luo, J., Sun, W., Chen, L., Liu, X., Zheng, T., Qin, Z., Ren, K.: Hurricane mixer: The eye in the storm—embedding regulatory oversight into cryptocurrency mixing services. Cryptology ePrint Archive, Paper 2025/1659 (2025), https://eprint.iacr.org/2025/1659
18. Meiklejohn, S., Mercer, R.: Möbius: Trustless tumbling for transaction privacy (2018)
19. Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC press (2018)
20. Narula, N., Vasquez, W., Virza, M.: zkledger: Privacy-preserving auditing for distributed ledgers. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2018), https://www.usenix.org/system/files/conference/nsdi18/nsdi18-narula.pdf
21. Pertsev, A., Semenov, R., Storm, R.: Tornado cash privacy solution version 1.4. Tornado cash privacy solution version **1**(6) (2019)
22. RedStone: Real-world assets in onchain finance report. https://blog.redstone.finance/2025/06/26/real-world-assets-in-onchain-finance-report/ (June 2025), blog post, accessed 2025-08-10
23. Ruffing, T., Moreno-Sanchez, P., Kate, A.: P2p mixing and unlinkable bitcoin transactions. In: Network and Distributed System Security Symposium (NDSS) (2017), https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss201701-4RuffingPaper.pdf
24. Seres, I.A., Nagy, D.A., Buckland, C., Burcsi, P.: Mixeth: Efficient, trustless coin mixing service for ethereum. In: International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019). OASIcs, vol. 71, pp. 13:1–13:20. Schloss Dagstuhl (2020). https://doi.org/10.4230/OASIcs.Tokenomics.2019.13, https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Tokenomics.2019.13
25. Team, R.: Legal considerations for tokenizing real-world assets. https://www.rwa.io/post/legal-considerations-for-tokenizing-real-world-assets (December 2024), rWA.io article, accessed 2025-08-10
26. Tostevin, P., Rushton, C.: Total value of global real estate: Property remains the world's biggest store of wealth. https://impacts.savills.com/market-trends/the-total-value-of-global-real-estate-property-remains-the-worlds-biggest-store-of-wealth.html (September 2023), savills Impacts – Market Trends, September 2023. Accessed: 2025-08-10
27. Webisoft: Real-world assets (rwa) tokenization: Complete web3 guide. https://webisoft.com/articles/real-world-assets-rwa-tokenization-guide/ (July 2025), blog post, Webisoft, July 3, 2025. Accessed: 2025-08-10
28. Xia, N., Zhao, X., Yang, Y., Li, Y., Li, Y.: Exploration on real world assets and tokenization. arXiv preprint arXiv:2503.01111 (2025)

29. Zcash Improvement Proposals: ZIP 316: Unified addresses and unified viewing keys. Standards document (2022), https://zips.z.cash/zip-0316

Appendix

UC Security Proof of UltraMixer

Theorem 1 (UC Security of UltraMixer). *Under the assumptions of a trusted zk-SNARK setup and standard cryptographic soundness and zero-knowledge, the UltraMixer protocol (in the asynchronous network model with malicious adversaries) UC-realizes the ideal functionality $\mathcal{F}_{UltraMixer}$ (as defined in Section 2.1 of the paper). In particular, for any PPT adversary A controlling a subset of parties, there exists a PPT simulator S such that for all environments Z, the execution of the real protocol with A is indistinguishable from the ideal execution with S and $\mathcal{F}_{UltraMixer}$. Consequently, UltraMixer achieves privacy (unlinkability, identity hiding), correctness (no double-spend, whitelist enforcement), and minimal leakage as specified by $\mathcal{F}_{UltraMixer}$.*

Proof. Our proof uses the UC simulation paradigm. We construct a simulator S that interacts with the ideal functionality $\mathcal{F}_{\text{UltraMixer}}$ and environment Z, and simulates the real-world adversary's view. The high-level strategy is to show that S can reproduce all observable outcomes of the UltraMixer protocol (transfers of assets, verification results, etc.) using only the information that $\mathcal{F}_{\text{UltraMixer}}$ would leak, and using the ZK-SNARK's simulation properties for cryptographic outputs. We break down the simulation by each operation of UltraMixer:

Deposit: When an honest party P deposits asset (id, n), $\mathcal{F}_{\text{UltraMixer}}$ adds a coin for P and leaks only the fact that some whitelisted user deposited (id, n). The simulator S mimics this by producing a fake on-chain deposit from an address (representing P) with amount n of asset id, along with a simulated proof $\tilde{\pi}_d$. By the zero-knowledge property, $\tilde{\pi}_d$ is indistinguishable from a real proof and reveals no secret information. The proof will pass verification (since S can simulate a valid proof for a true statement: P is whitelisted and committing to (id, n)). Thus, the adversary's view is a deposit of (id, n) with a valid proof – exactly as in a real execution. If the adversary's party attempts a deposit, S forwards the request to $\mathcal{F}_{\text{UltraMixer}}$ (if allowed) to maintain consistency. Invalid deposit attempts (e.g. non-whitelisted or bad proofs) are ignored by $\mathcal{F}_{\text{UltraMixer}}$, and S simulates a contract failure (proof rejection) on the real side. Soundness of the SNARK ensures a malicious deposit with no valid witness would fail in real execution just as it does in the ideal model.

Withdraw: For an honest party P withdrawing k of asset id to external address $addr_{out}$, $\mathcal{F}_{\text{UltraMixer}}$ burns P's internal coins (ensuring no double-spend) and outputs a leak of a withdrawal of (id, k) to $addr_{out}$. The simulator S accordingly simulates the mixer contract transferring k of id to $addr_{out}$ on-chain, accompanied by a simulated withdraw proof $\tilde{\pi}_w$. Since the withdrawal conditions hold in the ideal world (sufficient balance, etc.), S can generate $\tilde{\pi}_w$ that

passes verification (using the SNARK's simulator). This fools the adversary into believing a real withdrawal took place. No information beyond (id, k) and $addr_{out}$ is revealed (the proof is zero-knowledge), matching the ideal leakage. If a corrupted party Q (controlled by adversary) withdraws, S lets Q's request go through in $\mathcal{F}_{\text{UltraMixer}}$ (if valid) and the adversary sees the same $(id, k, addr_{out})$ transfer as in real. Any cheating by Q (such as attempting to withdraw more than owned or reuse a spent coin) would result in a proof that fails in real execution (by soundness or on-chain checks) – S simulates the failure, and $\mathcal{F}_{\text{UltraMixer}}$ would have aborted that request as well. Thus, both worlds agree on the outcome (no funds transferred if invalid).

In-Mixer Transfer: For a private transfer from P to Q of k units of asset id, the ideal functionality moves the coin from P to Q and leaks only that an in-mixer transfer occurred (no identities or amount). The simulator S handles an honest P by simulating a contract call that updates the mixer's state (Merkle tree) with a new coin for Q of value k, using two simulated proofs: $\tilde{\pi}_t$ (proving P had a valid coin of value k and nullifiers) and $\tilde{\pi}_{id}$ (proving Q is whitelisted and the new commitment is correctly formed). Both proofs correspond to true statements in the ideal execution and can be generated without witnesses using the SNARK's zero-knowledge simulator. The adversary sees a transaction that results in a mixer state update (and perhaps a generic "transfer occurred" event) but gains no link between P and Q or the amount k. This is indistinguishable from a real private transfer, as real proofs leak no more information. If Q is corrupt, $\mathcal{F}_{\text{UltraMixer}}$ would have privately informed Q of the incoming (id, k), and indeed the adversary controlling Q will learn k in both real and ideal worlds. If P (sender) is corrupt, then the adversary itself will produce the transfer proof in real execution. S simply relays the transfer request to $\mathcal{F}_{\text{UltraMixer}}$ (which updates state and sends the appropriate leak). Whether the transfer succeeds or fails, the adversary's view is consistent: a success (with no public details) if and only if the ideal transfer was allowed. Any attempt by a corrupt sender to bypass protocol rules (e.g. send coins not owned or to a non-whitelisted receiver) cannot succeed in real due to SNARK soundness and contract verification, and will not succeed in ideal (ideal functionality aborts), so Z sees failure in both cases.

Atomic Trade: $\mathcal{F}_{\text{UltraMixer}}$ implements atomic two-party trades of assets, leaking only that "an atomic trade occurred." The simulator S must simulate the combined proof transaction for the trade. If both parties P and Q are honest, S obtains the trade request from $\mathcal{F}_{\text{UltraMixer}}$ (which would execute it internally and leak the event). S then fabricates two proofs $\tilde{\pi}_P, \tilde{\pi}_Q$ for P and Q's trading statements (each proving ownership of the required asset and authorization) and feeds them in one simulated transaction that updates the mixer state (swapping the assets). The adversary sees a successful trade on-chain but with no identifying info (no addresses or assets are revealed publicly). This matches the ideal leak. If one party is adversarial (say Q is corrupt), there are a few sub-cases: - If the trade is executed properly (both provide valid proofs), then $\mathcal{F}_{\text{UltraMixer}}$ swaps the assets and leaks the event. S shows the corresponding on-chain success with proofs. Q being corrupt means the

adversary knows Q's part of the trade (how much Q gave/received), and indeed in real execution A learns that from controlling Q. In the ideal world, S can output to A the private outcome for Q (which $\mathcal{F}_{\text{UltraMixer}}$ would send to Q). Thus, A's knowledge of Q's assets is consistent. The adversary does not learn anything about honest P beyond the trade happened (and the trade terms, which it set or agreed upon). - If the adversary tries to deviate (e.g. Q submits no proof or an invalid proof), then in real life the atomic swap will fail entirely (the contract won't execute partial trades). In the ideal model, if one input is missing or invalid, $\mathcal{F}_{\text{UltraMixer}}$ aborts the trade. S simply does not simulate a successful trade on-chain. The environment Z sees no trade event in both worlds. Therefore, a malicious Q cannot gain any one-sided advantage: they either complete the trade fairly or it doesn't happen (ensuring fairness). This aligns with both the real protocol's design and the ideal functionality.

In all cases, the adversary's view of an atomic trade is limited to "a trade took place" (or failed to), which is identical to the ideal leakage.

Proof of Holding: When an honest party P proves continuous holding of θ of asset id from t_1 to t_2, $\mathcal{F}_{\text{UltraMixer}}$ returns a boolean result (VALID or INVALID) and leaks only that outcome and the query parameters. S accordingly simulates a proof verification on-chain. If the ideal result is VALID, S produces a simulated proof $\tilde{\pi}_h$ attesting to the holding property (which is a true statement given $\mathcal{F}_{\text{UltraMixer}}$'s state, so S can generate $\tilde{\pi}_h$ that passes). If the ideal result is INVALID, S can either simulate a proof that fails verification or simply no proof accepted. In both cases, the adversary and environment only learn the boolean outcome and (id, θ, t_1, t_2). In a real execution, a SNARK Proof of Holding is zero-knowledge, so A would also only see the outcome (and perhaps a trivial public output of 0/1). Thus, Z observes the same. If a corrupt party attempts to fake a holding proof (claim VALID when it's false), soundness ensures the proof will not be accepted, yielding INVALID on-chain, which matches $\mathcal{F}_{\text{UltraMixer}}$'s response. Conversely, if a corrupt party truly has the holdings, the proof will succeed and both worlds output VALID. Therefore, the adversary gains no advantage; the result visibility is the same in both worlds.

In addition to simulating each operation's outcome, S must handle adaptive corruptions consistently. If the environment Z corrupts an honest party P mid-execution, S is informed and must provide A with P's internal state (coin secrets, etc.) consistent with the transcript so far. S can do this because it knows all P's coins from $\mathcal{F}_{\text{UltraMixer}}$. For each coin (id, n, P, ℓ) that P holds, S can fabricate a secret s and an address (public key) for P such that the coin's commitment $\ell = H(H(s, 1, addr_P), id, n)$ matches the on-chain commitment. Since H is collision-resistant but not invertible by A, S is free to choose any valid preimage $(s, addr_P)$ for each commitment (the probability that A can detect a difference is negligible as long as the values produce the correct ℓ). S can always find such an s if it controls the random oracle (or CRS trapdoor) used in H, or simply by trial until the hash matches (treating H as a random function). This way, the corrupted

party's state handed to A will pass all verification for any past proofs (which A didn't see witnesses for) and will be coherent with future actions. Thus, adaptive corruption does not give Z any distinguishing power: the secrets S provides are distributed like real secrets given the public info.

Now we argue that no environment Z can distinguish the real protocol from the ideal simulation except with negligible probability. All real protocol outputs that Z can observe (directly or via A) have been matched in distribution by S's simulation: - The public ledger events (deposits into the mixer, withdrawals out, etc.) are identical in content: same assets, amounts, external addresses, timing (ordering), and success/failure patterns. $\mathcal{F}_{\text{UltraMixer}}$ was defined to leak exactly those details an on-chain observer would see, and S ensures the simulated blockchain provides those to A. - The zero-knowledge proof data seen by A (and hence by Z through the adversary's view) cannot be used to distinguish worlds due to the indistinguishability property of SNARK simulations. In the real world, all proofs are honestly generated; in the ideal, S produces simulated proofs for true statements. By zero-knowledge, no polytime observer can tell whether a given proof was generated with the real witness or by the simulator – the distributions are statistically/computationally close. We rely here on the assumption that the SNARK's simulation soundness holds (our S never needed to simulate a proof for a false statement – it only simulates proofs for actions that $\mathcal{F}_{\text{UltraMixer}}$ allowed, which are by definition "true" statements in the context of the protocol). - The cryptographic state (Merkle tree roots, nullifier sets) evolves consistently in both worlds. If Z tries to use these to distinguish, it would require inverting or correlating the hash commitments. For example, linking a deposit and withdrawal would require breaking the hiding of either the commitment or the nullifier. Our design ensures that, under collision resistance and hiding assumptions of the hash, these links are not discernible. S updates the simulated Merkle roots exactly as in real (or in a way indistinguishable under random oracle assumptions). The adversary might see identical root values in both worlds at corresponding times, leaving no tell-tale sign. - Any attempt by A to exploit the protocol (double-spend, forge proofs, misuse authorization) fails in both worlds. In the real world, such an attempt would break soundness or other security assumptions; in the ideal world, $\mathcal{F}_{\text{UltraMixer}}$ trivially prevents it. Thus Z never observes a success of an illegal action in one world and a failure in the other – both yield failure (or both would require a cryptographic break which we assume is negligible probability). - The fair exchange nature of atomic trades ensures that partial executions are not observable. Z cannot catch a difference where, say, in real P lost assets but Q didn't gain them – that situation cannot occur in the real protocol (the contract's atomicity prevents it) or in ideal (the functionality would abort). So fairness is maintained across worlds.

If Z had non-negligible advantage in guessing real vs ideal, we could use Z to either distinguish a ZK proof from a simulated one (breaking zero-knowledge), or to violate soundness by creating a transcript that exists in one world but not the other. Since these are assumed secure, Z's advantage must be negligible.

Therefore, S succeeds in simulating A's view perfectly up to negligible statistical distance.

Hence, we conclude the UltraMixer protocol UC-realizes $\mathcal{F}_{\text{UltraMixer}}$. This means that for all practical purposes, the protocol achieves the security properties of the ideal mixer: user privacy (no linkability or identity exposure beyond what is inevitable), correctness of operations (assets are neither created nor destroyed except by authorized actions, and no double-spending occurs), and minimal leakage of information. The proof is complete.