

RESEARCH ARTICLE OPEN ACCESS

Token-Guided Flow Tracing for Auditable Zero-Knowledge Smart Contract Transfers

Junhee Lee¹  | Jihye Kim^{2,3}  | Hyunok Oh^{3,4}  | Heejin Park¹ 

¹Department of Computer Science, Hanyang University, Seoul 04763, Republic of Korea | ²Electronics and Information System Engineering Major, Kookmin University, Seoul 02707, Republic of Korea | ³Zkrypto, Seoul 04763, Republic of Korea | ⁴Department of Information Systems, Hanyang University, Seoul 04763, Republic of Korea

Correspondence: Jihye Kim (jihyek@kookmin.ac.kr) | Hyunok Oh (hoh@hanyang.ac.kr)

Received: 28 May 2026 | **Revised:** 23 July 2026 | **Accepted:** 27 July 2026

Academic Editor: Thomas Haines

Keywords: auditable transfers | private smart-contract transfers | scoped auditability | token-guided tracing | transaction tracing

ABSTRACT

Private transfers on public smart contract blockchains hide transaction values and private transfer links, but accountable privacy requires controlled auditability. Existing auditable zero-knowledge transfer systems make per-transaction audit information available to an authorized auditor, but their audit model may allow the auditor to inspect transactions beyond the flow connected to the authorized audit target. In such a model, granting audit capability for one investigation can expose unrelated transactions or allow tracing to continue farther than intended. In this paper, we propose token-guided flow tracing for auditable zero-knowledge smart contract transfers, enabling audit authorization to be scoped to a transaction flow and epoch range. The construction separates audit authorization from audit capability by introducing a tracing-token manager and an auditor. The auditor can open audit information only when it holds the auditor secret key, an epoch secret key for an authorized epoch, and a tracing token associated with the target transaction flow. Direction-specific tracing tokens confine tracing to the authorized direction, while epoch public keys and a binary key-derivation tree support compact authorization of audit intervals. We analyze security through ledger indistinguishability, transaction nonmalleability, balance, audit correctness, and restricted audit authorization. Restricted audit authorization is established in an honest authorization model that assumes the tracing-token manager and the auditor do not collude beyond explicit audit authorizations; under these assumptions, the auditor cannot open or trace transactions outside the authorized flow, direction, and epoch scope. Our implementation adds 16,349 transfer-circuit constraints, about 178,000 gas to each accepted transfer, and 448 bytes to the serialized transfer transaction, showing that scoped auditability can be added with moderate on-chain overhead.

1 | Introduction

Public blockchains maintain a distributed ledger without relying on a centralized operator, and the ledger contents are visible to network participants. This ledger transparency is useful for public verifiability and data integrity, but it also exposes transaction information that would normally be considered private. In public ledgers, transaction values, account activities, and transaction

links can be observed by parties that are not involved in the transaction. Privacy-preserving payment systems therefore aim to provide confidentiality, which hides transaction values, and anonymity or unlinkability, which hides participants or the links among their transactions. To achieve these goals while preserving public verifiability, existing systems have used several cryptographic techniques, including mixing protocols [1],

This is an open access article under the terms of the [Creative Commons Attribution](https://creativecommons.org/licenses/by/4.0/) License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

Copyright © 2026 Junhee Lee et al. *IET Information Security* published by John Wiley & Sons Ltd.

ring signatures [2], homomorphic encryption [3], commitment schemes, and zero-knowledge proofs [4].

Existing systems combine these techniques in different ways to support private transactions across ledger models and deployment targets. Zerocash [5] provides unspent transaction output (UTXO)-based anonymous payments by committing to private coin information and proving validity with zero-knowledge succinct noninteractive arguments of knowledge (zk-SNARKs). Zeth [6] adapts the Zerocash approach to Ethereum by implementing a smart contract mixer without changing the base ledger. Block-Maze [7] targets stronger privacy in account-model blockchains using a dual balance model and zk-SNARKs to hide account balances, transfer amounts, and links between senders and recipients.

Privacy alone, however, is not sufficient for many financial applications. If all transaction information is hidden from the public, the system must still provide a controlled way to support regulatory compliance and anti-money laundering investigations. Several systems therefore study how to support auditability while preserving the privacy of transactions or ledger records [8–14]. These works show that auditability is an essential feature of private transaction systems, but it also introduces a new concentration of power: an auditor that can open too much information may learn more than is necessary for a specific investigation.

Azeroth is a recent auditable zero-knowledge-transfer framework designed for account-based smart contract blockchains [14]. It uses dual accounts, encrypted transfer records, an accumulator, and zk-SNARK verification to support private transfers through smart contracts without modifying the base ledger. Its unified transfer procedure avoids exposing the transfer type through separate contract entry points because deposit, withdrawal, public transfer, private transfer, and mixed transfers share the same transaction structure, while public value movements remain visible when public inputs or outputs are nonzero. Azeroth also supports auditability by encrypting the transaction information for an authorized auditor. The auditor is an observer in this process and does not obtain the ability to authorize, modify, or invalidate valid transfers. Nevertheless, its audit model remains overly broad. Once the auditor obtains the auditor secret key, the auditor can decrypt transaction ciphertexts across the ledger. Furthermore, opening a selected transaction does not directly identify the spent commitment, so flow analysis over transactions cannot be performed locally from that transaction alone.

This observation motivates our goal of supporting transaction-flow tracing in auditable zero-knowledge transfers while restricting audit access to the authorized flow and epoch range. The central problem is not merely how to decrypt a selected transaction, but how to let an auditor follow only the transaction flow that has been authorized for a concrete investigation. A naive solution is to require the auditor to ask the manager for permission whenever it wants to inspect another transaction. This reduces the auditor's autonomy, but it is unsuitable for flow tracing because an investigation starting from one suspicious transaction may require several adjacent transactions to be inspected. The manager would then have to remain online and approve every tracing step.

In this paper, we present a token-guided flow-tracing construction for auditable zero-knowledge smart contract transfers that limits audit disclosure to an authorized flow, tracing direction, and epoch range. The construction separates the authority to authorize an audit from the ability to open audit information by introducing a tracing token manager and an auditor. The restricted audit guarantee assumes that these two parties do not collude beyond explicit audit authorizations. The tracing token manager authorizes an audit interval by disclosing epoch secret-key material, but it cannot decrypt the transaction contents by itself. The auditor can decrypt audit information only when it has the auditor secret key, an epoch secret key for the transaction epoch, and a tracing token associated with the transaction. Therefore, the auditor does not obtain ledger-wide visibility from the auditor secret key alone; instead, it follows a transaction flow using disclosed tracing tokens within epochs authorized by the disclosed epoch-secret-key material.

The main technical idea is to attach tracing tokens to encrypted transfer records and update them across transfer transactions. A disclosed tracing token enables the auditor to test ledger transactions and derive adjacent tracing tokens only along the corresponding transaction flow, covering both transfers of a note and successive transfers by the same user. To prevent a tracing authorization from extending to the opposite direction, our construction derives direction-specific forward and backward tokens from a base tracing token. Thus, a token for backward tracing does not give the auditor the ability to continue tracing forward from a previously reached transaction. To restrict auditing to authorized epochs, we generate each auditor ciphertext under the active epoch public key. The tracing token manager derives epoch secret keys from a binary key-derivation tree instantiated by a pseudorandom function (PRF), so an authorized audit interval can be represented by a compact set of disclosed tree nodes rather than by one secret key per epoch.

This paper makes the following contributions.

- We observe that audit mechanisms based on auditor-encrypted transaction records can give broader visibility than a case-specific investigation requires: an auditor with the audit key may be able to open unrelated transactions, whereas the intended audit scope should be limited to the relevant flow, tracing direction, and epoch range.
- We propose a token-guided audit mechanism for auditable zero-knowledge smart contract transfers. The construction separates audit authorization from audit capability using a tracing-token manager and an auditor and uses direction-specific tracing tokens to support forward and backward flow tracing without releasing the base tracing token.
- We add epoch-scoped audit authorization using epoch keys derived from a binary key-derivation tree, enabling compact disclosure for audit intervals.
- We analyze security through ledger indistinguishability, transaction nonmalleability, balance, audit correctness, and restricted audit authorization in an honest authorization model under the stated cryptographic assumptions, assuming that the tracing token manager and the auditor do not collude beyond explicit audit authorizations.

- We implement the transfer and registration circuits and the smart contract functions and compare the overhead over Azeroth in terms of circuit constraints, gas cost, transaction size, and client-side running time.

2 | Related Work

Auditability and accountability in distributed payment systems have been systematized as regulatory requirements that must be balanced with transaction privacy [15]. In this section, we distinguish accountability based on disclosure from auditability by an authorized auditor. Accountability based on disclosure means that a user voluntarily reveals payment information to a third party. In contrast, auditability by an authorized auditor refers to a mechanism that allows an auditor or authority to obtain audit information without relying on voluntary disclosure.

2.1 | Private Payment Systems

Zerocash [5] provides anonymous payments in the UTXO model by hiding coin ownership and transaction values with commitments and zero-knowledge proofs. Zeth [6] adapts Zerocash to Ethereum by implementing a smart contract mixer without changing the base ledger. Zether [3] adapts confidential payments to the account-based smart contract setting using additively homomorphic ElGamal encryption and zero-knowledge proofs [16]. It hides balances and transferred amounts, but it does not fully hide the identities of the transaction parties, and its unlinkability support is limited by the size of the anonymity set. BlockMaze [7] uses a dual balance model and zk-SNARKs to hide balances, transferred values, and sender-receiver links. However, it is not designed as a deployable smart contract layer construction.

Some private payment systems also provide disclosure or viewing features that can support accountability. QuisQuis [17] proposes an account-based anonymous cryptocurrency design based on updatable public keys and zero-knowledge arguments. Omniring [18] formalizes ring confidential transactions. It discusses trackability and viewability extensions in which a user delegates tracking or viewing keys for incoming transactions. Lelantus Spark [19] provides private transactions with flexible transaction structures and supports opt-in visibility through incoming and full viewing keys. These systems [17–19] provide anonymity set-based privacy. Zcash [20] defines viewing keys that separate the viewing capability from the spending capability.

Solidus [21] supports bank-intermediated confidential ledger transactions using publicly verifiable ORAM. It hides transaction values and user-level account links while the transacting banks remain visible. In Solidus, a bank is both a ledger participant that maintains customer accounts and a party that can enable transaction-level audit visibility by disclosing decrypted logs or sharing decryption keys. In contrast, our tracing token manager is not a ledger participant and cannot audit transactions by itself.

2.2 | Auditable Systems With Authorized Auditors

zkLedger [8] supports privacy-preserving auditing for ledgers maintained by organizations. It hides transaction values and

relationships, while its auditor checks aggregate ledger queries and accounting properties with cryptographic proofs. Thus, zkLedger provides confidentiality and unlinkability, but its audit scope is aggregate rather than transaction level, and its cost grows with the number of organizations. FabZK [9] extends Hyperledger Fabric with privacy-preserving and auditable smart contracts by storing encrypted transaction data, anonymizing payer-payee relationships, using commitments and zero-knowledge proofs for validation, and supporting automated audit over encrypted data. Its audit validates encrypted smart contract execution in a permissioned setting rather than giving an auditor unrestricted plaintext access to the transaction contents. Auditable token payments [11] provide a UTXO-based privacy-preserving token management system for permissioned blockchains. They conceal values and sender-receiver relationships while encrypting transfer information for the auditors assigned to the relevant users. Because each user is assigned an auditor at registration, the disclosed transaction information is scoped to the users whose auditors receive the corresponding ciphertexts. If all users are assigned to the same auditor, this deployment can give that auditor ledger-wide visibility.

PGC [10] provides confidential payments with auditability using additively homomorphic public-key encryption. Its audit mechanism lets an external party specify a set of transactions and ask the participant to prove policy compliance without revealing the underlying transaction values. Its unlinkability support is limited in practice because anonymity-set-based extensions increase the transaction cost with the set size. ASOZ [12] is based on the Zerocash framework and combines strong transaction privacy with full transaction auditability in a global mixing setting, where an auditor can open transaction-level information. PARscoin [13] studies privacy-preserving stablecoin payments with auditability, but its audit target is reserve sufficiency: the auditor checks whether the off-chain reserve covers the total stablecoin supply rather than opening individual transfer amounts.

Azeroth [14] provides auditable zero-knowledge transfers in a public smart contract setting. Its audit model is practical because an authorized auditor can open transaction-level auditor ciphertexts without user cooperation or a new authorization step for each transaction. However, the auditor receives ledger-wide audit visibility once it holds the auditor secret key. Our construction keeps the same deployment target and transaction-level audit capability but changes the granularity of audit authorization.

2.3 | On/Off-Chain Cooperation

Several smart-contract systems outside payment settings use the ledger for auditable coordination while keeping sensitive computation off the chain. SPDS [22] records data-usage policies and auditable usage logs on a blockchain for smart-grid data sharing, while executing confidential data processing in an off-chain trusted execution environment. SEAL [23] uses smart contracts and hashchain-based micropayments for fair exchange in UAV computation offloading, together with an off-chain auction protocol for bid privacy.

TABLE 1 | Comparison of related systems.

System			Properties			
Name	Model	Deployment	Unlink.	Conf.	Audit	Audit granularity
Zerocash/Zcash [5, 20]	UTXO	Platform	✓	✓	×	—
Lelantus Spark [19]	UTXO	Platform	△	✓	×	—
Omniring [18]	UTXO	Platform	△	✓	×	—
ASOZ [12]	UTXO	Platform	✓	✓	✓	Ledger-wide tx-level data open
Auditable token payments [11]	UTXO	Permissioned	✓	✓	✓	Assigned-user tx-level data open
FabZK [9]	Account	Permissioned	✓	✓	✓	Ledger-wide tx-level validation check
QuisQuis [17]	Account	Platform	△	✓	×	—
Solidus [21]	Account	Platform	△	✓	△	Bank mediated tx-level data open
BlockMaze [7]	Account	Platform	✓	✓	×	—
PGC [10]	Account	Platform	△	✓	✓	Policy compliance check
PARScoin [13]	Account	Platform	✓	✓	✓	Reserve sufficiency check
zkLedger [8]	Account	Platform	✓	✓	✓	Aggregated value check
Zeth [6]	Account	Smart contract	✓	✓	×	—
Zether [3]	Account	Smart contract	△	✓	×	—
Azeroth [14]	Account	Smart contract	✓	✓	✓	Ledger-wide tx-level data open
Ours	Account	Smart contract	✓	✓	✓	Token-guided flow tracing with epoch-scoped tx-level data open

Note: “Unlink.” denotes ledger-level unlinkability of the private transfer relation; for ours, it refers to hidden private commitment flow, not sender-address anonymity against the contract. △ denotes anonymity set based or limited link hiding. “Audit” denotes auditability by an auditor or authority, excluding voluntary viewing-key disclosure; △ denotes indirect or institution-mediated audit support. “Audit granularity” summarizes what information or compliance conditions can be checked through the audit mechanism.

These works address different application domains from ours, but they illustrate a common design pattern in which the ledger provides accountability or settlement while sensitive computation remains off-chain. Our work uses a related separation in a different setting. Transfer validation remains on the chain, whereas audit authorization and tracing are performed off the chain using cryptographic material scoped to a transaction flow, tracing direction, and epoch range.

Table 1 summarizes the comparison among the private payment and auditable transfer systems. Since the compared systems provide transaction confidentiality, the main distinctions lie in the degree of unlinkability and audit granularity.

3 | Preliminaries

The security parameter is denoted by λ . A function $\text{negl}(\lambda)$ is negligible if it is smaller than the inverse of every polynomial for sufficiently large λ . An algorithm is PPT if it runs in probabilistic polynomial time in λ . For a finite set S , $x \xleftarrow{\$} S$ means that x is sampled uniformly from S . The notation $y \leftarrow \text{Alg}(x)$ means that algorithm Alg is run on input x and returns y . The symbol $\parallel$ denotes concatenation. We write $\mathcal{A}^{\mathcal{O}}$ for an adversary with oracle access to $\mathcal{O}$, and use st to denote local state passed between stages of an adversary. We use $\mathbb{F}$ for the scalar field and let G denote a generator of the underlying group. For $d \geq 1$ and $j \in [0, 2^d - 1]$,

$\text{bin}_d(j)$ denotes the d -bit zero-padded binary representation of j with the most significant bit first.

3.1 | zk-SNARK

Let $\mathcal{R}$ be a relation for an NP language $\mathcal{L}_{\mathcal{R}} = \{x: \exists w \text{ such that } (x, w) \in \mathcal{R}\}$. A zk-SNARK for $\mathcal{R}$ is a tuple of PPT algorithms $\text{SNARK} = (\text{Setup}, \text{Prove}, \text{Verify})$ with the following syntax.

- $\text{pp} \leftarrow \text{SNARK.Setup}(\lambda, \mathcal{R})$: on input the security parameter λ and the relation $\mathcal{R}$, the setup algorithm outputs public parameters pp .
- $\pi \leftarrow \text{SNARK.Prove}(\text{pp}, x, w)$: on input public parameters pp , a statement x , and a witness w , the proving algorithm outputs a proof π .
- $b \leftarrow \text{SNARK.Verify}(\text{pp}, x, \pi)$: on input public parameters pp , a statement x , and a proof π , the verification algorithm outputs $b = 1$ if π is accepted for x , and outputs $b = 0$ otherwise.

3.1.1 | Completeness

A zk-SNARK satisfies completeness if, for every $(x, w) \in \mathcal{R}$,

$$\Pr[\text{SNARK.Verify}(\text{pp}, x, \pi) = 1 \mid \text{pp} \leftarrow \text{SNARK.Setup}(\lambda, \mathcal{R}), \pi \leftarrow \text{SNARK.Prove}(\text{pp}, x, w)] = 1.$$

3.1.2 | Knowledge Soundness

A zk-SNARK satisfies knowledge soundness if, for every PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}$ such that

$$\Pr \left[\begin{array}{l} \text{SNARK.Verify}(\text{pp}, x, \pi) = 1 \wedge \\ (x, w) \notin \mathcal{R} \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{SNARK.Setup}(\lambda, \mathcal{R}), \\ ((x, \pi), w) \leftarrow (\mathcal{A} \parallel \mathcal{E})(\text{pp}) \end{array} \right] \leq \text{negl}(\lambda).$$

Here, $(\mathcal{A} \parallel \mathcal{E})(\text{pp})$ denotes the execution of $\mathcal{A}$ with extractor $\mathcal{E}$, in which $\mathcal{A}$ outputs (x, π) and $\mathcal{E}$ outputs w given access to $\mathcal{A}$'s execution.

3.1.3 | Zero-Knowledge

A zk-SNARK satisfies zero-knowledge if there exists a PPT simulator $S = (S_{\text{Setup}}, S_{\text{Prove}})$ such that, for every PPT distinguisher $\mathcal{D}$ and every $(x, w) \in \mathcal{R}$,

$$\left| \Pr \left[\begin{array}{l} \mathcal{D}(\mathcal{R}, \text{pp}, x, \pi) = 1 \\ \pi \leftarrow \text{SNARK.Prove}(\text{pp}, x, w) \end{array} \right] - \Pr \left[\begin{array}{l} \mathcal{D}(\mathcal{R}, \widetilde{\text{pp}}, x, \widetilde{\pi}) = 1 \\ (\widetilde{\text{pp}}, \text{td}) \leftarrow S_{\text{Setup}}(\lambda, \mathcal{R}), \\ \widetilde{\pi} \leftarrow S_{\text{Prove}}(\widetilde{\text{pp}}, \text{td}, x) \end{array} \right] \right| \leq \text{negl}(\lambda).$$

The simulator receives each statement but not its witness. Here, td denotes simulator only auxiliary information.

3.1.4 | Succinctness

A zk-SNARK satisfies succinctness if the proof size and verification time are polynomial in λ and polylogarithmic in the witness size.

3.2 | Collision-Resistant Hash

Let $\mathcal{M}_\lambda$ and $\mathcal{Y}_\lambda$ denote the message space and output space, respectively. A hash function CRH is a deterministic polynomial-time function $\text{CRH}: \mathcal{M}_\lambda \rightarrow \mathcal{Y}_\lambda$. We write $y = \text{CRH}(m)$. When CRH is applied to a tuple, the tuple is encoded injectively; for notational simplicity, we omit the encoding and the public hash-function description. We require CRH to satisfy collision resistance and preimage resistance.

3.2.1 | Collision Resistance

For every PPT adversary $\mathcal{A}$,

$$\Pr \left[\begin{array}{l} m \neq m' \wedge \\ \text{CRH}(m) = \text{CRH}(m') \end{array} \middle| (m, m') \leftarrow \mathcal{A}(1^\lambda) \right] \leq \text{negl}(\lambda).$$

3.2.2 | Preimage Resistance

For every PPT adversary $\mathcal{A}$,

$$\Pr \left[\begin{array}{l} \text{CRH}(m') = y \\ m' \leftarrow \mathcal{A}(1^\lambda, y) \end{array} \middle| \begin{array}{l} m \xleftarrow{\$} \mathcal{M}_\lambda, \\ y \leftarrow \text{CRH}(m) \end{array} \right] \leq \text{negl}(\lambda).$$

3.3 | PRF

We write $\text{PRF}_k(m)$ for a PRF evaluated on input m under key k . Pseudorandomness means that no PPT adversary with oracle access can distinguish $\text{PRF}_k(\cdot)$, for a uniformly sampled key k , from a truly random function with the same domain and range, except with negligible advantage. When PRF outputs are used as public identifiers, we additionally assume collision resistance over key-input pairs: no PPT adversary can find distinct pairs $(k, m) \neq (k', m')$ such that $\text{PRF}_k(m) = \text{PRF}_{k'}(m')$, as in Zerocash [5]. We additionally assume that PRF_k is preimage resistant with respect to its input even when k is known.

3.4 | Commitment Scheme

A commitment scheme $\text{CM} = (\text{Setup}, \text{Com}, \text{Open})$ consists of the following PPT algorithms.

- $\text{pp} \leftarrow \text{CM.Setup}(\lambda)$: on input the security parameter λ , the setup algorithm outputs public parameters pp .
- $\text{cm} \leftarrow \text{CM.Com}(\text{pp}, m; o)$: on input public parameters pp , a message m , and opening randomness o , the commitment algorithm outputs a commitment cm .
- $b \leftarrow \text{CM.Open}(\text{pp}, \text{cm}, m, o)$: on input public parameters pp , a commitment cm , a message m , and opening randomness o , the opening algorithm outputs $b = 1$ if cm is a valid commitment to m with opening randomness o , and outputs $b = 0$ otherwise.

We require CM to satisfy binding and hiding. Binding means that no PPT adversary can output a commitment together with valid openings to two distinct messages, except with negligible probability. Hiding means that commitments to any two messages are computationally indistinguishable.

3.5 | Accumulator

An accumulator $\text{ACC} = (\text{Setup}, \text{Add}, \text{CreateMem}, \text{VerifyMem})$ represents a set of elements by a succinct accumulator state acc . It consists of the following PPT algorithms.

- $(\text{pp}, \text{acc}) \leftarrow \text{ACC.Setup}(\lambda)$: on input the security parameter λ , the setup algorithm outputs public parameters pp and an empty accumulator state acc .
- $\text{acc}' \leftarrow \text{ACC.Add}(\text{pp}, \text{acc}, u)$: on input public parameters pp , an accumulator state acc , and an element u , the update

algorithm inserts u and outputs the updated accumulator state acc' .

- $\omega \leftarrow \text{ACC.CreateMem}(\text{pp}, \text{acc}, \text{List}, u)$: on input public parameters pp , an accumulator state acc , an element list List , and an element u , the membership proving algorithm outputs a membership proof ω .
- $b \leftarrow \text{ACC.VerifyMem}(\text{pp}, \omega, \text{acc}, u)$: on input public parameters pp , a membership proof ω , an accumulator state acc , and an element u , the verification algorithm outputs $b = 1$ if ω is a valid membership proof for u with respect to acc , and outputs $b = 0$ otherwise.

Correctness requires that honestly generated membership proofs for accumulated elements verify. Soundness requires that no PPT adversary can produce a valid membership proof for an element that is not contained in the accumulated set, except with negligible probability.

3.6 | Symmetric-Key Encryption

A symmetric-key encryption scheme SE consists of three PPT algorithms $\text{SE} = (\text{Gen}, \text{Enc}, \text{Dec})$ with the following syntax.

- $k \leftarrow \text{SE.Gen}(\lambda)$: on input the security parameter λ , the key generation algorithm outputs a secret key k .
- $\text{sct} \leftarrow \text{SE.Enc}_k(m)$: on input a key k and a message m , the encryption algorithm outputs a ciphertext sct .
- $m \leftarrow \text{SE.Dec}_k(\text{sct})$: on input a key k and a ciphertext sct , the decryption algorithm outputs a message m or $\perp$.

3.6.1 | Correctness

For every message m in the message space,

$$\Pr \left[\text{SE.Dec}_k(\text{sct}) = m \mid \begin{array}{l} k \leftarrow \text{SE.Gen}(\lambda), \\ \text{sct} \leftarrow \text{SE.Enc}_k(m) \end{array} \right] = 1.$$

3.6.2 | IND-CPA Security

SE satisfies indistinguishability under chosen-plaintext attacks if, for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$\Pr \left[b' = b \mid \begin{array}{l} k \leftarrow \text{SE.Gen}(\lambda), \\ (m_0, m_1, \text{st}) \leftarrow \mathcal{A}_1^{\text{SE.Enc}_k(\cdot)}(1^\lambda), \\ b \xleftarrow{\$} \{0, 1\}, \\ \text{sct}^* \leftarrow \text{SE.Enc}_k(m_b), \\ b' \leftarrow \mathcal{A}_2^{\text{SE.Enc}_k(\cdot)}(\text{st}, \text{sct}^*) \end{array} \right] - \frac{1}{2} \leq \text{negl}(\lambda),$$

where m_0 and m_1 have the same length.

3.6.3 | IK-CPA Security

SE satisfies key indistinguishability under chosen-plaintext attacks if, for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$\Pr \left[b' = b \mid \begin{array}{l} k_0 \leftarrow \text{SE.Gen}(\lambda), k_1 \leftarrow \text{SE.Gen}(\lambda), \\ (m, \text{st}) \leftarrow \mathcal{A}_1^{\text{SE.Enc}_{k_0}(\cdot), \text{SE.Enc}_{k_1}(\cdot)}(1^\lambda), \\ b \xleftarrow{\$} \{0, 1\}, \\ \text{sct}^* \leftarrow \text{SE.Enc}_{k_b}(m), \\ b' \leftarrow \mathcal{A}_2^{\text{SE.Enc}_{k_0}(\cdot), \text{SE.Enc}_{k_1}(\cdot)}(\text{st}, \text{sct}^*) \end{array} \right] - \frac{1}{2} \leq \text{negl}(\lambda).$$

3.7 | Key Encapsulation Mechanism

A key encapsulation mechanism KEM consists of four PPT algorithms $\text{KEM} = (\text{Gen}, \text{PubDerive}, \text{Encap}, \text{Decap})$ with the following syntax.

- $(\text{sk}, \text{pk}) \leftarrow \text{KEM.Gen}(\lambda)$: on input the security parameter λ , the key generation algorithm outputs a secret key sk and a public key pk .
- $\text{pk} \leftarrow \text{KEM.PubDerive}(\text{sk})$: on input a secret key sk , the public-key derivation algorithm outputs the corresponding public key pk .
- $(\text{ek}, \text{epk}, \text{aux}) \leftarrow \text{KEM.Encap}(\text{pk})$: on input a public key pk , the encapsulation algorithm outputs an encapsulated key ek , an ephemeral public key epk , and auxiliary witness data aux . The value epk is the public encapsulation value sent to the decapsulator.
- $\text{ek} \leftarrow \text{KEM.Decap}(\text{sk}, \text{epk})$: on input a secret key sk and an ephemeral public key epk , the decapsulation algorithm outputs an encapsulated key ek or $\perp$.

The value aux denotes encapsulation randomness or auxiliary witness data that can be used to prove correct encapsulation.

3.7.1 | Correctness

For every key pair (sk, pk) generated by KEM.Gen ,

$$\Pr[\text{KEM.Decap}(\text{sk}, \text{epk}) = \text{ek} \mid (\text{ek}, \text{epk}, \text{aux}) \leftarrow \text{KEM.Encap}(\text{pk})] = 1.$$

3.7.2 | Key Indistinguishability

Let $\mathcal{K}_\lambda$ denote the encapsulated-key space. KEM satisfies key indistinguishability if, for every PPT adversary $\mathcal{A}$,

$$\Pr \left[b' = b \mid \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KEM.Gen}(\lambda), \\ (\text{ek}_0, \text{epk}, \text{aux}) \leftarrow \text{KEM.Encap}(\text{pk}), \\ \text{ek}_1 \xleftarrow{\$} \mathcal{K}_\lambda, \\ b \xleftarrow{\$} \{0, 1\}, \\ b' \leftarrow \mathcal{A}(1^\lambda, \text{pk}, \text{epk}, \text{ek}_b) \end{array} \right] - \frac{1}{2} \leq \text{negl}(\lambda).$$

Thus, the encapsulated key is computationally indistinguishable from a uniformly random key given the public key and the ephemeral public key.

When pk is a tuple of public keys and decapsulation requires the corresponding tuple of secret keys, we require the key

indistinguishability property even against an adversary that knows any strict subset, but not all, of the corresponding secret keys.

3.8 | Public-Key Encryption

A public-key encryption scheme PE consists of three PPT algorithms $PE = (\text{Gen}, \text{Enc}, \text{Dec})$ with the following syntax.

- $(\text{sk}, \text{pk}) \leftarrow \text{PE.Gen}(\lambda)$: on input the security parameter λ , the key generation algorithm outputs a secret key sk and a public key pk .
- $(\text{pct}, \text{aux}) \leftarrow \text{PE.Enc}_{\text{pk}}(m)$: on input a public key pk and a message m , the encryption algorithm outputs a ciphertext pct and auxiliary witness data aux .
- $m \leftarrow \text{PE.Dec}_{\text{sk}}(\text{pct})$: on input a secret key sk and a ciphertext pct , the decryption algorithm outputs a message m or $\perp$.

The value aux denotes encryption randomness or auxiliary witness data that can be used to prove correct encryption.

3.8.1 | Correctness

For every message m in the message space,

$$\Pr \left[\text{PE.Dec}_{\text{sk}}(\text{pct}) = m \mid \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{PE.Gen}(\lambda), \\ (\text{pct}, \text{aux}) \leftarrow \text{PE.Enc}_{\text{pk}}(m) \end{array} \right] = 1.$$

3.8.2 | IND-CPA Security

PE satisfies indistinguishability under chosen-plaintext attacks if, for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$\left| \Pr \left[b' = b \mid \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{PE.Gen}(\lambda), \\ (m_0, m_1, \text{st}) \leftarrow \mathcal{A}_1(\text{pk}), \\ b \xleftarrow{\$} \{0, 1\}, \\ (\text{pct}^*, \text{aux}^*) \leftarrow \text{PE.Enc}_{\text{pk}}(m_b), \\ b' \leftarrow \mathcal{A}_2(\text{st}, \text{pct}^*) \end{array} \right] - \frac{1}{2} \right| \leq \text{negl}(\lambda),$$

where m_0 and m_1 have the same length.

3.8.3 | IK-CPA Security

PE satisfies key indistinguishability under chosen-plaintext attacks if, for every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$,

$$\left| \Pr \left[b' = b \mid \begin{array}{l} (\text{sk}_0, \text{pk}_0) \leftarrow \text{PE.Gen}(\lambda), \\ (\text{sk}_1, \text{pk}_1) \leftarrow \text{PE.Gen}(\lambda), \\ (m, \text{st}) \leftarrow \mathcal{A}_1(\text{pk}_0, \text{pk}_1), \\ b \xleftarrow{\$} \{0, 1\}, \\ (\text{pct}^*, \text{aux}^*) \leftarrow \text{PE.Enc}_{\text{pk}_0}(m), \\ b' \leftarrow \mathcal{A}_2(\text{st}, \text{pct}^*) \end{array} \right] - \frac{1}{2} \right| \leq \text{negl}(\lambda).$$

3.9 | Digital Signatures

A digital signature scheme $\text{SIG} = (\text{Gen}, \text{Sign}, \text{Verify})$ consists of the following PPT algorithms.

- $(\text{sk}, \text{vk}) \leftarrow \text{SIG.Gen}(\lambda)$: on input the security parameter λ , the key generation algorithm outputs a signing key sk and a verification key vk .
- $\sigma \leftarrow \text{SIG.Sign}(\text{sk}, m)$: on input a signing key sk and a message m , the signing algorithm outputs a signature σ .
- $b \leftarrow \text{SIG.Verify}(\text{vk}, m, \sigma)$: on input a verification key vk , a message m , and a signature σ , the verification algorithm outputs $b = 1$ if σ is valid for m under vk , and outputs $b = 0$ otherwise.

Correctness requires that honestly generated signatures verify under the corresponding verification key. We require SIG to satisfy existential unforgeability under chosen-message attacks (EUF-CMA). For any PPT adversary with signing-oracle access, the probability that it outputs a valid signature under vk on a message not previously submitted to the signing oracle is negligible.

4 | Construction

This section presents our construction for token-guided flow tracing in auditable zero-knowledge smart contract transfers. It preserves Azeroth's core transfer workflow while adding tracing tokens, a tracing token manager, and epoch keys for restricted audit authorization, incorporating them into the account state, commitments, ciphertexts, and transfer proof used by the transfer protocol. The goal is to replace ledger-wide auditor visibility with restricted audit capability: according to the intended authorization policy, the manager authorizes an audit interval by releasing tracing tokens and epoch secret key material, and the auditor uses the released direction-specific tracing tokens and derived epoch secret keys to perform the authorized audit and follow only the corresponding transaction flow. This guarantee relies on role separation and assumes that the manager and auditor do not collude beyond explicit audit authorizations.

4.1 | Overview

Following Azeroth, our construction keeps the unified transfer procedure zkTransfer for deposit, withdrawal, public transfer, private transfer, and their mixed forms. Rather than introducing separate contract functions for different transfer types, we extend the unified procedure with the components needed for token-guided flow tracing and epoch-scoped audit authorization. This keeps the design compatible with a ledger-based smart contract setting while adding moderate on-chain overhead, mainly from additional ciphertexts, public inputs, and state updates.

To reduce the authority of the auditor, we introduce an additional manager. A simple way to restrict the auditor would be to require it to request permission from the manager for each

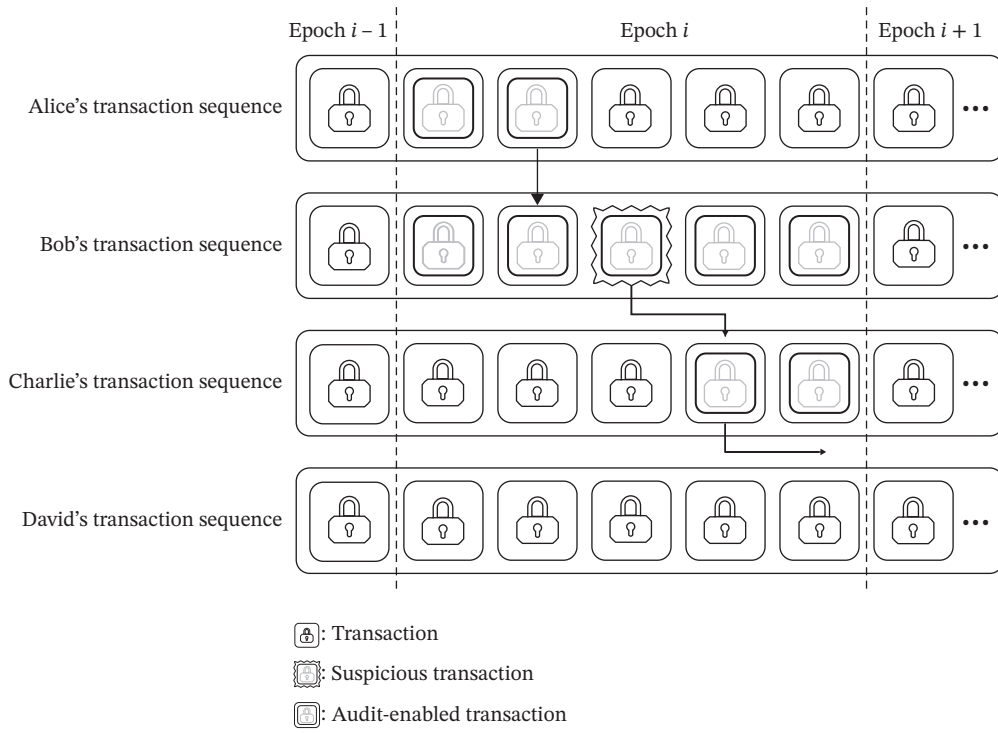


FIGURE 1 | Overview of token-guided flow tracing with epoch-scoped authorization.

transaction to be inspected. However, this approach is unsuitable for tracing the transaction flows. Once an investigation starts from a suspicious transaction, the auditor may need to inspect several connected transactions, and asking the manager for every adjacent transaction would require repeated interactions. Moreover, the original audit mechanism of Azeroth does not directly support flow tracing from a selected transaction. Even after an auditor decrypts a transaction, the decrypted information does not identify the spent commitment. Thus, reconstructing a transaction flow requires decrypting and correlating transaction information across the ledger.

To address these issues, we attach tracing tokens to encrypted notes and propagate them through transfer transactions. Given an authorized starting token, the auditor can follow the corresponding transaction flow without requesting a new authorization for every step. However, using only one tracing token is too coarse. If a single token was used for all directions, an auditor who is allowed to trace backward from a selected transaction could also obtain information that should be reachable only in the forward direction. For instance, after reaching the transaction that created the spent commitment, the auditor could continue to trace later transactions derived from that point, although the intended permission is to inspect only the preceding flow. Therefore, our construction derives a forward token and a backward token from one base tracing token. The forward token enables forward tracing, and the backward token enables backward tracing. The transfer ciphertexts were then arranged so that these direction-specific tokens were updated consistently across token transitions.

We additionally use epoch keys to bind the tracing capability to explicitly authorized epochs. Even if the auditor obtains a valid tracing token, it can open or trace a transaction only for epochs for which it has the corresponding epoch secret key. If the

manager generated and disclosed epoch keys independently for each epoch, the amount of communication would be linear in the length of the audit interval. To reduce this cost, the manager derives epoch secret keys from a binary key-derivation tree instantiated with a PRF. For an authorized audit interval, the manager discloses a compact set of tree nodes, and the auditor derives the required leaf epoch secret keys locally.

Figure 1 shows an example of token-guided flow tracing with epoch-scoped authorization across epochs $i-1$, i , and $i+1$. Arrows show note-spending links from a transaction that creates a commitment to a later transaction that spends it, while transactions aligned in the same sequence illustrate same-user token updates. Starting from a suspicious transaction, the auditor follows this transaction flow only through transactions that can be opened with a disclosed direction-specific tracing token and the authorized epoch secret key.

4.2 | System Model and Data Types

4.2.1 | Account Model

We follow Azeroth's dual account model. Each user has an externally owned account EOA and an encrypted account ENA. The EOA is the ordinary public account managed by the underlying blockchain and is used for public token transfers to and from the smart contract. The ENA is maintained by the smart contract as an encrypted balance $\text{ENA}[\text{addr}]$, and only the user holding k_{ENA} can decrypt its value.

4.2.2 | Entities and Contract State

The system consists of users, a smart contract, a tracing-token manager M , and an auditor A . A user owns a public key $\text{upk} = (\text{addr}, \text{pk}_{\text{own}}, \text{pk}_{\text{U}})$ and a secret key $\text{usk} = (k_{\text{ENA}}, \text{sk}_{\text{own}}, \text{sk}_{\text{U}})$,

T, o^T). The address $addr$ is the public account identifier used for the encrypted balance introduced above. The pair (sk_U, pk_U) is a public-key encryption key pair used to deliver notes. Following Azeroth, a note denotes the private data required to spend a commitment, including the transferred value, the commitment opening, and the associated on-chain commitment. The value sk_{own} is used to derive nullifiers, which prevent double-spending. It also determines pk_{own} and authorizes spending. In order to spend a note associated with $addr$, the prover must show knowledge of the corresponding sk_{own} . In our construction, to support tracing, each user maintains only its current tracing token T together with the corresponding commitment opening o^T . Whenever the token is updated, the new token is sampled independently and uniformly rather than derived from its predecessor and replaces the previous token.

The smart contract maintains the accumulator state, the set of registered addresses, used nullifiers, created commitments, encrypted account balances, and the public parameters required for verification and auditing. Following Azeroth, a commitment denotes an on-chain hidden transfer output. The accumulator compactly represents the set of all created commitments, and a transfer proof uses it to prove the membership of the spent commitment without revealing which commitment is spent. Compared with Azeroth, the additional contract state consists of the tracing-token manager public key PK_M , the active epoch public key PK_e , the current epoch index $EPOCH$, the epoch-key verification key VK_E , a token commitment map $CM_T[addr]$, and a set $List_{tag}$ of used tracing-token tags. For each tracing token T , its public tag is defined as $tag_T = PRF_T("tag")$. Because this tag is deterministic, the contract can detect reuse of T for a new token state by rejecting a registration or transfer if the tag already appears in $List_{tag}$; otherwise, it records the tag. The token commitment map associates each registered sender address with a commitment to the current tracing token stored in the sender's secret state. During a transfer, the zero-knowledge proof shows that the old token commitment opens to the sender's old token and that the new token commitment is formed correctly.

4.2.3 | Audit Tracing Tokens and Epoch Keys

To support tracing in both directions of a transaction flow, the audit mechanism uses direction-specific tracing tokens. Instead of storing separate forward and backward tracing tokens, a user stores a single token T and derives a forward token FT and a backward token BT from it:

$$FT = PRF_T("fwd"), BT = PRF_T("bwd").$$

Each recipient receives the base tracing token T as part of the private note delivered in a transfer. During a transfer, the sender uses both the token carried by the spent note and the sender's account-level token and generates a new account-level token associated with that transfer.

An auditor's tracing capability is determined by the disclosed direction-specific token. In this paper, the transaction flow includes both same-user token-chain transitions and note-spending transitions through output commitments. Given a forward token FT , the auditor can trace one step forward, either along the

same user's transaction chain or to a later transaction that spends the output commitment. Given a backward token BT , the auditor can trace one step backward, either along the same user's transaction chain or to the transaction that created the input commitment. For bidirectional authorization, the tracing-token manager discloses both direction-specific tokens FT and BT rather than the base token T , allowing the auditor to start tracing in either direction without learning the base token itself.

Epoch keys restrict audit decryption to an authorized range of epoch indices in the accepted ledger. The tracing-token manager derives epoch secret keys from a binary key-derivation tree and publishes signed epoch public keys with their eligibility times. Let $N \geq 2$ be the public maximum number of epochs supported by one tree, and let $d = \lceil \log_2 N \rceil$. The binary key-derivation tree has its root at depth 0 and 2^d leaves at depth d . Left and right edges are labeled 0 and 1, respectively. A node is identified by its path $p \in \{0, 1\}^{\leq d}$. Let ϵ denote the empty bit string identifying the root, and let τ_p denote the secret of node p . We set $\tau_\epsilon = msk_e$ and, for $b \in \{0, 1\}$, derive each child secret as

$$\tau_{p||b} = PRF_{\tau_p}(b).$$

Epoch $j \in [0, N - 1]$ is assigned to the leaf labeled $bin_d(j)$; equivalently, epoch indices are mapped to leaves in increasing order from left to right. The secret of this leaf is used as the epoch secret key, and its corresponding public key is

$$sk_e[j] = \tau_{bin_d(j)}, pk_e[j] = KEM.PubDerive(sk_e[j]).$$

If $N < 2^d$, leaves $N, \dots, 2^d - 1$ are unused. The smart contract accepts an epoch public key only when its eligibility time has passed and the signature is valid.

For a valid inclusive authorization interval $I = [j_0, j_1]$, the manager computes its cover recursively. For a node with path p , let $h = |p|$ and let k be the integer value of p , with the empty root path $p = \epsilon$ interpreted as $k = 0$. This node covers the leaf interval

$$[k \cdot 2^{d-h}, (k + 1) \cdot 2^{d-h} - 1].$$

Starting from the root, the manager discards a node if its leaf interval is disjoint from I . If the node's leaf interval is contained in I , the manager selects the node and does not visit its descendants. Otherwise, the manager applies the same rule to its two children. The disclosed set S_E contains the path p and PRF-derived secret τ_p of each selected node. Since a node is selected as soon as its complete leaf interval is contained in I , the selected nodes are the maximal contained subtrees and form a minimum-size set of tree nodes whose descendant leaves are exactly $j_0, \dots, j_1$. The auditor derives the covered epoch secret keys from these disclosed node secrets. The cover contains at most $2d$ nodes.

4.2.4 | Transaction Ciphertexts

- User ciphertext pct_U . It is intended only for the receiver and encrypts $(v, addr, T, o)$ to pk_U , where v is the private value, $addr$ is the receiver address, T is the received tracing token, and o is the commitment opening. Together with the public output commitment cm included in the transaction, the

private value, tracing token, and opening define the spendable note (v , T , o , cm).

- Auditor ciphertext pct_A . It is generated with respect to pk_A and PK_e , which are the auditor and active-epoch public keys for KEM. It contains encrypted audit information for the transaction, such as the private output value, receiver address, and output commitment, but it does not reveal the base tracing token T directly. It can be opened only by an auditor holding the auditor secret key sk_A , the required epoch secret key, and the appropriate direction-specific token, either FT or BT.
- Tracing-token manager ciphertext pct_M . It is encrypted under the tracing-token manager public key pk_M and contains the tracing token T associated with the transaction. Since pct_M is included in the on-chain transaction, the manager need not maintain a separate token database and can recover the token from the ledger when authorization is required. From pct_M , the manager recovers only T ; without sk_A , it cannot recover the confidential audit information protected by pct_A . When granting authorization, the manager derives the authorized subset of $\{FT, BT\}$ from the recovered base token and discloses only those direction-specific tokens to the auditor.

4.2.5 | Unified Transfer and Value Types

Our construction inherits Azeroth's unified transfer interface, in which a transfer contains both public and private value components. We use v_{in}^{pub} and v_{out}^{pub} for values entering from the sender's public account and leaving to a receiver public account, respectively. We use v_{in}^{priv} for the value obtained from a previously created commitment and v_{out}^{priv} for the value hidden in the newly created output commitment. The encrypted account value v_{ENA} is updated so that the total input and output values are balanced:

$$v_{ENA}^{new} = v_{ENA}^{old} + v_{in}^{priv} - v_{out}^{priv} + v_{in}^{pub} - v_{out}^{pub}.$$

A unified transfer may take value from the sender's public account EOA, the sender's encrypted account ENA, and an input private note, and it may output value to a receiver public account, the sender's updated encrypted account, and a new private note. Depending on the transfer type, some of these value components may be set to zero. The new private note data is delivered to the receiver through pct_U . Hence, deposits, withdrawals, public transfers, private transfers, and mixed transfers are expressed by the same balance equation.

4.3 | Algorithms

We now describe the construction algorithms. Algorithm 1 gives the on-chain procedures, while Algorithms 2 and 3 list the client-side procedures, split for presentation clarity. Algorithm 2 covers setup, epoch-key update preparation, and user registration; Algorithm 3 covers transfer construction, auditor audit and tracing, and user note retrieval. Algorithm 4 specifies the zero-knowledge relation verified by the contract.

ALGORITHM 1 | Smart contract algorithms.

Setup_{SC}(Tx_{SET})

```

parse TxSET as (ppACC, ppKGU, ppZKT, acc, pkM, pkA, vkE,
pke)
Listacc.Append(acc)
PPKGU ← ppKGU
PPZKT ← ppZKT
PPACC ← ppACC
PKM ← pkM
PKA ← pkA
PKe ← pke
VKE ← vkE
EPOCH ← 0

```

UpdateEpochKey_{SC}(Tx_{UEK})

```

parse TxUEK as (j, pke, t, σj)
assert block.timestamp ≥ t
assert j = EPOCH + 1
assert SIG.Verify(VKE, addrSC || chainid || j || pke || t, σj) = 1
PKe ← pke
EPOCH ← j

```

RegisterUser_{SC}(Tx_{KGU})

```

parse TxKGU as (π, addr, cmT, tagT)
assert addr ∉ Listaddr
assert tagT ∉ Listtag
x := (addr, cmT, tagT)
assert SNARK.Verify(PPKGU, x, π) = 1
Listaddr.Append(addr)
Listtag.Append(tagT)
CMT[addr] ← cmT
ENA[addr] ← 0

```

zkTransfer_{SC}(Tx_{ZKT})

```

parse TxZKT as (π, acc, nf, upksend, cmnew, sctnew, vinpub, voutpub,
pctU, EOArecv, cmnewT, tagT, pctA, pctM, epoch)
parse upksend as (addrsend, _, _)
assert epoch = EPOCH
assert addrsend ∈ Listaddr
assert acc ∈ Listacc
assert nf ∉ Listnf
assert cmnew ∉ Listcm

```

(continued)

ALGORITHM 1 | (Continued)

```

assert tagT ∉ Listtag
x := (PKM, PKA, PKE, acc, nf, upksend, cmnew,
      ENA[addrsend], sctnewENA,
      vinpub, voutpub, pctU, CMT[addrsend], cmnewT, tagT, pctA, pctM,
      EOArecv)

assert SNARK.Verify(PPZKT, x, π) = 1
CMT[addrsend] ← cmnewT
ENA[addrsend] ← sctnewENA
Listcm.Append(cmnew)
EOAsend ← msg.sender
accnew ← ACC.Add(PPACC, Listacc.Top, cmnew)
Listacc.Append(accnew)
Listnf.Append(nf)
Listtag.Append(tagT)
if vinpub > 0 then TransferFrom(EOAsend, this, vinpub)
if voutpub > 0 then Transfer(EOArecv, voutpub).

```

ALGORITHM 2 | Client algorithms (Part 1).

```

SetupClient(λ, RKGU, RZKT, N, addrSC, chainid)
ppCM ← CM.Setup(λ)
(ppACC, acc) ← ACC.Setup(λ)
ppKGU ← SNARK.Setup(λ, RKGU)
ppZKT ← SNARK.Setup(λ, RZKT)
pp := (ppCM, ppACC, ppKGU, ppZKT)
(skM, pkM) ← PE.Gen(λ)
(skA, pkA) ← KEM.Gen(λ)
(skE, vkE) ← SIG.Gen(λ)
mske  $\xleftarrow{\$}$  F
d ← ⌈log2 N⌉
τe ← mske
for each j ∈ [0, N − 1] do
    let bind(j) = b1 ... bd
    p ← ε
    for t = 1 to d do
        τp||bt ← PRFτp(bt)
        p ← p||bt
    ske[j] ← τp
    pke[j] ← KEM.PubDerive(ske[j])
    let tj denote the eligibility time of epoch j

```

(continued)

ALGORITHM 2 | (Continued)

```

σj ← SIG.Sign(skE, addrSC || chainid || j || pke[j] || tj)
BBE[j] := (j, pke[j], tj, σj)
TxSET := (ppACC, ppKGU, ppZKT, acc, pkM, pkA, vkE, pke[0])
return pp, (skM, pkM, mske), (skA, pkA), BBE, TxSET

UpdateEpochKeyClient(BBE, j)
    parse BBE[j] as (j, pke, t, σj)
    TxUEK := (j, pke, t, σj)
    return TxUEK

KeyGenUsersClient(λ, pp)
    parse pp as (ppCM, _, ppKGU, _)
    (skU, pkU) ← PE.Gen(λ)
    kENA ← SE.Gen(λ)
    skown  $\xleftarrow{\$}$  F
    T  $\xleftarrow{\$}$  F
    oT  $\xleftarrow{\$}$  F
    cmT ← CM.Com(ppCM, T; oT)
    tagT ← PRFT("tag")
    pkown ← CRH(skown)
    addr ← CRH(pkown || pkU)
    usk := (kENA, skown, skU, T, oT)
    upk := (addr, pkown, pkU)
    x := (addr, cmT, tagT)
    w := (T, oT)
    π ← SNARK.Prove(ppKGU, x, w)
    TxKGU := (π, addr, cmT, tagT)
    return (usk, upk), TxKGU

```

Algorithm 1 maintains the public ledger state. The procedure Setup_{SC} parses the setup transaction and initializes the empty accumulator state, zk-SNARKs verification parameters, tracing-token manager public key, auditor public key, epoch-key verification key, and initial epoch public key. The procedure RegisterUser_{SC} registers a user address together with an initial token commitment and a token tag. It verifies that the tag is fresh and that the registration proof binds the tag and the commitment to the same initial tracing token. If verification succeeds, the contract stores the token commitment and records the tag in List_{tag}. The procedure UpdateEpochKey_{SC} is permissionless: any party may submit an epoch-key update transaction obtained from the public bulletin board. The contract accepts the update only if the submitted epoch index j is sequential, the epoch is eligible, and the signature under VK_E verifies over addr_{SC}||chainid||j||pk_e[j]||t. Here, addr_{SC} denotes the deployed contract address and chainid denotes the chain identifier; including them in the signed message binds the epoch-key update to a

$\text{zkTransfer}_{\text{Client}}(\lambda, \text{pp}, \text{note}, \text{pk}_M, \text{pk}_A, \text{usk}^{\text{send}}, \text{upk}^{\text{send}}, \text{upk}^{\text{recv}}, v_{\text{out}}^{\text{priv}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{EOA}^{\text{recv}})$
 parse pp as $(\text{pp}_{\text{CM}}, \text{pp}_{\text{ACC}}, _, \text{pp}_{\text{ZKT}})$
 parse usk^{send} as $(k_{\text{ENA}}^{\text{send}}, \text{sk}_{\text{own}}^{\text{send}}, \text{sk}_{\text{U}}^{\text{send}}, T_{\text{old}}, o_{\text{old}}^T)$
 parse upk^{send} as $(\text{addr}^{\text{send}}, \text{pk}_{\text{own}}^{\text{send}}, \text{pk}_{\text{U}}^{\text{send}})$
 parse upk^{recv} as $(\text{addr}^{\text{recv}}, \text{pk}_{\text{own}}^{\text{recv}}, \text{pk}_{\text{U}}^{\text{recv}})$
 read $\text{CM}_T[\text{addr}^{\text{send}}]$, $\text{ENA}[\text{addr}^{\text{send}}]$, PK_e ,
 EPOCH, List_{acc} , and List_{cm} from the current smart contract state
 $\text{acc} \leftarrow \text{List}_{\text{acc}}.\text{Top}$
if note $\neq \perp$ **then**
 parse note as $(v_{\text{in}}^{\text{priv}}, T^{\text{note}}, o_{\text{old}}, \text{cm}_{\text{old}})$
 $b_{\text{in}} \leftarrow 1$
 $\omega \leftarrow \text{ACC}.\text{CreateMem}(\text{pp}_{\text{ACC}}, \text{acc}, \text{List}_{\text{cm}}, \text{cm}_{\text{old}})$
else
 $v_{\text{in}}^{\text{priv}} \leftarrow 0$
 $T^{\text{note}} \leftarrow \text{PRF}_{\text{sk}_{\text{own}}^{\text{send}}}(\text{"dummy"} \parallel T_{\text{old}})$
 $o_{\text{old}} \xleftarrow{\$} \mathbb{F}$
 $\text{cm}_{\text{old}} \leftarrow \text{CM}.\text{Com}(\text{pp}_{\text{CM}}, (v_{\text{in}}^{\text{priv}}, \text{addr}^{\text{send}}, T^{\text{note}}); o_{\text{old}})$
 $b_{\text{in}} \leftarrow 0$
 $\omega \leftarrow \perp$
 $\text{nf} \leftarrow \text{PRF}_{\text{sk}_{\text{own}}^{\text{send}}}(\text{cm}_{\text{old}})$
 $T_{\text{new}} \xleftarrow{\$} \mathbb{F}$
 $o_{\text{new}}^T \xleftarrow{\$} \mathbb{F}$
 $\text{cm}_{\text{new}}^T \leftarrow \text{CM}.\text{Com}(\text{pp}_{\text{CM}}, T_{\text{new}}; o_{\text{new}}^T)$
 $\text{tag}_T \leftarrow \text{PRF}_{T_{\text{new}}}(\text{"tag"})$
 $o_{\text{new}} \xleftarrow{\$} \mathbb{F}$
 $\text{cm}_{\text{new}} \leftarrow \text{CM}.\text{Com}(\text{pp}_{\text{CM}}, (v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, T_{\text{new}}); o_{\text{new}})$
 $(\text{pct}_{\text{U}}, \text{aux}_{\text{U}}) \leftarrow \text{PE}.\text{Enc}_{\text{pk}_{\text{U}}^{\text{recv}}}(v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, T_{\text{new}}, o_{\text{new}})$
 $\text{FT}_{\text{old}} \leftarrow \text{PRF}_{T_{\text{old}}}(\text{"fwd"})$
 $\text{BT}_{\text{old}} \leftarrow \text{PRF}_{T_{\text{old}}}(\text{"bwd"})$
 $\text{FT}^{\text{note}} \leftarrow \text{PRF}_{T^{\text{note}}}(\text{"fwd"})$
 $\text{BT}^{\text{note}} \leftarrow \text{PRF}_{T^{\text{note}}}(\text{"bwd"})$
 $\text{FT}_{\text{new}} \leftarrow \text{PRF}_{T_{\text{new}}}(\text{"fwd"})$
 $\text{BT}_{\text{new}} \leftarrow \text{PRF}_{T_{\text{new}}}(\text{"bwd"})$
 $(\text{ek}, \text{epk}, \text{aux}_A) \leftarrow \text{KEM}.\text{Encap}((\text{pk}_A, \text{PK}_e))$
 $\text{ct}_{\text{self}}^{\text{fwd}} \leftarrow \text{FT}_{\text{new}} + \text{PRF}_{\text{ek}}(\text{"sfwd"} \parallel \text{FT}_{\text{old}})$
 $\text{ct}_{\text{ext}}^{\text{fwd}} \leftarrow \text{FT}_{\text{new}} + \text{PRF}_{\text{ek}}(\text{"efwd"} \parallel \text{FT}^{\text{note}})$
 $\text{ct}_{\text{self}}^{\text{bwd}} \leftarrow \text{BT}_{\text{old}} + \text{PRF}_{\text{ek}}(\text{"sbwd"} \parallel \text{BT}_{\text{new}})$
 $\text{ct}_{\text{ext}}^{\text{bwd}} \leftarrow \text{BT}_{\text{new}} + \text{PRF}_{\text{ek}}(\text{"ebwd"} \parallel \text{BT}_{\text{new}})$

(continued)

$k_A \xleftarrow{\$} \mathbb{F}$
 $\text{wk}_A^{\text{FT}} \leftarrow k_A + \text{PRF}_{\text{ek}}(\text{"wrap-fwd"} \parallel \text{FT}_{\text{new}})$
 $\text{wk}_A^{\text{BT}} \leftarrow k_A + \text{PRF}_{\text{ek}}(\text{"wrap-bwd"} \parallel \text{BT}_{\text{new}})$
 $\text{ct}_A^{\text{msg}} \leftarrow \text{SE}.\text{Enc}_{k_A}(v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, \text{cm}_{\text{new}})$
 $\text{pct}_A := (\text{epk}, \text{ct}_{\text{self}}^{\text{fwd}}, \text{ct}_{\text{ext}}^{\text{fwd}}, \text{ct}_{\text{self}}^{\text{bwd}}, \text{ct}_{\text{ext}}^{\text{bwd}}, \text{wk}_A^{\text{FT}}, \text{wk}_A^{\text{BT}}, \text{ct}_A^{\text{msg}})$
 $(\text{pct}_M, \text{aux}_M) \leftarrow \text{PE}.\text{Enc}_{\text{pk}_M}(T_{\text{new}})$
 $\text{sct}_{\text{old}}^{\text{ENA}} \leftarrow \text{ENA}[\text{addr}^{\text{send}}]$
if $\text{sct}_{\text{old}}^{\text{ENA}} \neq 0$ **then**
 $v_{\text{old}}^{\text{ENA}} \leftarrow \text{SE}.\text{Dec}_{k_{\text{ENA}}^{\text{send}}}(\text{sct}_{\text{old}}^{\text{ENA}})$
else
 $v_{\text{old}}^{\text{ENA}} \leftarrow 0$
 $v_{\text{new}}^{\text{ENA}} \leftarrow v_{\text{old}}^{\text{ENA}} + v_{\text{in}}^{\text{priv}} - v_{\text{out}}^{\text{priv}} + v_{\text{in}}^{\text{pub}} - v_{\text{out}}^{\text{pub}}$
 $\text{sct}_{\text{new}}^{\text{ENA}} \leftarrow \text{SE}.\text{Enc}_{k_{\text{ENA}}^{\text{send}}}(v_{\text{new}}^{\text{ENA}})$
 $x := (\text{pk}_M, \text{pk}_A, \text{PK}_e, \text{acc}, \text{nf}, \text{upk}^{\text{send}}, \text{cm}_{\text{new}}, \text{sct}_{\text{old}}^{\text{ENA}}, \text{sct}_{\text{new}}^{\text{ENA}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{pct}_{\text{U}}, \text{CM}_T[\text{addr}^{\text{send}}], \text{cm}_{\text{new}}^T, \text{tag}_T, \text{pct}_A, \text{pct}_M, \text{EOA}^{\text{recv}})$
 $w := (\text{usk}^{\text{send}}, \text{cm}_{\text{old}}, o_{\text{old}}, v_{\text{in}}^{\text{priv}}, \text{upk}^{\text{recv}}, o_{\text{new}}, v_{\text{out}}^{\text{priv}}, \text{aux}_{\text{U}}, \omega, T_{\text{new}}, o_{\text{new}}^T, T^{\text{note}}, \text{ek}, k_A, \text{aux}_A, \text{aux}_M, b_{\text{in}})$
 $\pi \leftarrow \text{SNARK}.\text{Prove}(\text{pp}_{\text{ZKT}}, x, w)$
 $\text{Tx}_{\text{ZKT}} := (\pi, \text{acc}, \text{nf}, \text{upk}^{\text{send}}, \text{cm}_{\text{new}}, \text{sct}_{\text{new}}^{\text{ENA}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{pct}_{\text{U}}, \text{EOA}^{\text{recv}}, \text{cm}_{\text{new}}^T, \text{tag}_T, \text{pct}_A, \text{pct}_M, \text{EPOCH})$
 $\text{st}_{\text{pend}} := (T_{\text{new}}, o_{\text{new}}^T)$
 return $\text{Tx}_{\text{ZKT}}, \text{st}_{\text{pend}}$
 $\text{Trace}_{\text{Client}}(L, \text{Tx}_{\text{ZKT}}, \text{sk}_A, T_{\text{input}}, \text{mode}, \text{Table}_{\text{sk}_e})$
 parse Tx_{ZKT} as $(\dots, \text{pct}_A, \dots, \text{epoch})$
if mode = "Backward Self" or mode = "Backward External" **then**
 if $\text{Table}_{\text{sk}_e}[\text{epoch}]$ is undefined **then**
 return $\perp$
 parse pct_A as $(\text{epk}, _, _, \text{ct}_{\text{self}}^{\text{bwd}}, \text{ct}_{\text{ext}}^{\text{bwd}}, _, _, _)$
 $\text{ek} \leftarrow \text{KEM}.\text{Decap}((\text{sk}_A, \text{Table}_{\text{sk}_e}[\text{epoch}]), \text{epk})$
 for each candidate Tx_{ZKT}^* in the mode-specific range of L
 parse Tx_{ZKT}^* as $(\dots, \text{cm}_{\text{new}}^*, \dots, \text{pct}_A^*, \dots, \text{epoch}^*)$
 if $\text{Table}_{\text{sk}_e}[\text{epoch}^*]$ is undefined **then**
 continue
 parse pct_A^* as $(\text{epk}^*, \text{ct}_{\text{self}}^{\text{fwd}}, \text{ct}_{\text{ext}}^{\text{fwd}}, _, _, \text{wk}_A^{\text{FT}}, \text{wk}_A^{\text{BT}}, \text{ct}_A^{\text{msg}})$
 $\text{ek}^* \leftarrow \text{KEM}.\text{Decap}((\text{sk}_A, \text{Table}_{\text{sk}_e}[\text{epoch}^*]), \text{epk}^*)$
 if mode = "Forward Self" **then**
 type $\leftarrow$ "Forward"

(continued)

```

     $T \leftarrow \text{ct}_{\text{self}}^{\text{fwd}} - \text{PRF}_{\text{ek}}^*(\text{"sfwd"} \parallel T_{\text{input}})$ 
     $k_A \leftarrow \text{wk}_A^{\text{FT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-fwd"} \parallel T)$ 
else if mode = "Forward External" then
    type  $\leftarrow$  "Forward"
     $T \leftarrow \text{ct}_{\text{ext}}^{\text{fwd}} - \text{PRF}_{\text{ek}}^*(\text{"efwd"} \parallel T_{\text{input}})$ 
     $k_A \leftarrow \text{wk}_A^{\text{FT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-fwd"} \parallel T)$ 
else if mode = "Backward Self" then
    type  $\leftarrow$  "Backward"
     $T \leftarrow \text{ct}_{\text{self}}^{\text{bwd}} - \text{PRF}_{\text{ek}}^*(\text{"sbwd"} \parallel T_{\text{input}})$ 
     $k_A \leftarrow \text{wk}_A^{\text{BT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-bwd"} \parallel T)$ 
else if mode = "Backward External" then
    type  $\leftarrow$  "Backward"
     $T \leftarrow \text{ct}_{\text{ext}}^{\text{bwd}} - \text{PRF}_{\text{ek}}^*(\text{"ebwd"} \parallel T_{\text{input}})$ 
     $k_A \leftarrow \text{wk}_A^{\text{BT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-bwd"} \parallel T)$ 
else
    return  $\perp$ 
     $(v, \text{addr}, \text{cm}) \leftarrow \text{SE.Dec}_{k_A}(\text{ct}_A^{\text{msg}})$ 
    if  $\text{cm} = \text{cm}_{\text{new}}^*$  then return  $T, \text{Tx}_{\text{ZKT}}^*, \text{type}$ 
return  $\perp$ 

```

Audit_{Client}($\text{Tx}_{\text{ZKT}}, \text{sk}_A, \text{Table}_{\text{sk}_e}, T, \text{type}$)

```

    parse  $\text{Tx}_{\text{ZKT}}$  as  $(\dots, \text{cm}_{\text{new}}, \dots, \text{pct}_A, \dots, \text{epoch})$ 
    if  $\text{Table}_{\text{sk}_e}[\text{epoch}]$  is undefined then return  $\perp$ 
    parse  $\text{pct}_A$  as  $(\text{epk}, \_, \_, \_, \_, \text{wk}_A^{\text{FT}}, \text{wk}_A^{\text{BT}}, \text{ct}_A^{\text{msg}})$ 
     $\text{ek} \leftarrow \text{KEM.Decap}((\text{sk}_A, \text{Table}_{\text{sk}_e}[\text{epoch}]), \text{epk})$ 
    if type = "Forward" then
         $k_A \leftarrow \text{wk}_A^{\text{FT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-fwd"} \parallel T)$ 
    else if type = "Backward" then
         $k_A \leftarrow \text{wk}_A^{\text{BT}} - \text{PRF}_{\text{ek}}^*(\text{"wrap-bwd"} \parallel T)$ 
    else
        return  $\perp$ 
     $(v, \text{addr}, \text{cm}) \leftarrow \text{SE.Dec}_{k_A}(\text{ct}_A^{\text{msg}})$ 
    if  $\text{cm} \neq \text{cm}_{\text{new}}$  then return  $\perp$ 
     $\text{msg} := (v, \text{addr})$ 
    return msg

```

RetrieveNote_{Client}($L, \text{pp}, \text{usk}, \text{upk}$)

```

    parse pp as  $(\text{pp}_{\text{CM}}, \_, \_, \_)$ 
    parse usk as  $(\_, \_, \text{sk}_U, \_, \_)$ 
    parse upk as  $(\text{addr}, \_, \_)$ 
    notes  $\leftarrow \emptyset$ 
    for each  $\text{Tx}_{\text{ZKT}} \in L$ 

```

(continued)

```

    parse  $\text{Tx}_{\text{ZKT}}$  as  $(\dots, \text{cm}_{\text{new}}, \dots, \text{pct}_U, \dots)$ 
     $(v, \text{addr}^*, T, o) \leftarrow \text{PE.Dec}_{\text{sk}_U}(\text{pct}_U)$ 
    if  $\text{addr} = \text{addr}^*$  and  $v \in [0, 2^{64})$  then
        if  $\text{CM.Open}(\text{pp}_{\text{CM}}, \text{cm}_{\text{new}}, (v, \text{addr}, T), o) = 1$  then
            notes.Append( $(v, T, o, \text{cm}_{\text{new}})$ )
    return notes

```

specific contract deployment and blockchain, preventing cross-contract and cross-chain replay.

Finally, $\text{zkTransfer}_{\text{SC}}$ is the on-chain transfer procedure. It checks that the submitted accumulator state is known, the nullifier is unused, the token tag of the newly generated tracing token is unused, the sender is registered, the output commitment is fresh, and the transaction epoch is current. It then verifies a zk-SNARK proof for the transfer relation. If the proof verifies, the contract records the nullifier and token tag, inserts the new output commitment into the accumulator, updates the sender's encrypted account balance, and replaces the sender's stored token commitment with the new token commitment. It also executes the public value movements: $v_{\text{in}}^{\text{pub}}$ is transferred from the sender's public account to the contract, and $v_{\text{out}}^{\text{pub}}$ is transferred from the contract to EOA^{recv} . Here, acc is the accumulator state used to verify membership of the spent commitment, while List_{acc} . Top is the current accumulator state used for appending the new commitment. EOA^{send} denotes the blockchain account that directly signs and submits the contract call, i.e., msg.sender . The contract passes the parsed EOA^{recv} as a public input to the verifier, so a valid proof is bound to the public withdrawal recipient.

In Algorithm 2, the procedure $\text{Setup}_{\text{Client}}$ is written as a single initialization procedure for presentation, but its secret key generation steps are performed by the corresponding parties. The public parameters for the commitment scheme, accumulator, and zk-SNARKs were generated during setup. The tracing-token manager generates its public-key encryption key pair $(\text{sk}_M, \text{pk}_M)$ and samples a master epoch secret msk_e . From msk_e , it derives epoch secret keys using a PRF-based binary key-derivation tree. It then computes the corresponding epoch public keys. For each epoch, after the deployment domain is fixed, the manager signs the epoch index, epoch public key, and eligibility time together with addr_{SC} and chainid , and publishes the signed tuple on the public bulletin board BB_E . In practice, the epoch-key signatures are generated only after the deployment domain is fixed; equivalently, addr_{SC} must be known or otherwise fixed before signing. The auditor independently generates a KEM key pair $(\text{sk}_A, \text{pk}_A)$ and provides only pk_A for the setup transaction; the tracing-token manager never learns sk_A . pk_A and the epoch public keys are used for the two-public-key encapsulation in pct_A . The procedure outputs Tx_{SET} , which contains the public parameters, pk_M , pk_A , the epoch-key verification key, and the initial epoch public key. The procedure $\text{UpdateEpochKey}_{\text{Client}}$ reads the signed epoch public key for epoch j from the public bulletin board BB_E and forms the epoch-key update transaction Tx_{UEK} .

$\mathcal{R}_{\text{KGU}}(x, w)$
 parse x as $(\text{addr}, \text{cm}^T, \text{tag}_T)$
 parse w as (T, o^T)
 enforce a binding constraint on addr , e.g., $\text{addr} = 1 \cdot \text{addr}$
 $\text{CM.Open}(\text{pp}_{\text{CM}}, \text{cm}^T, T; \text{o}^T) = 1$
 $\text{tag}_T = \text{PRF}_T(\text{"tag"})$
 $\mathcal{R}_{\text{ZKT}}(x, w)$
 parse x as $(\text{pk}_M, \text{pk}_A, \text{pk}_e, \text{acc}, \text{nf}, \text{upk}^{\text{send}}, \text{cm}_{\text{new}}, \text{sct}_{\text{old}}^{\text{ENA}}, \text{sct}_{\text{new}}^{\text{ENA}},$
 $v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{pct}_U, \text{cm}_{\text{old}}^T, \text{cm}_{\text{new}}^T, \text{tag}_T, \text{pct}_A, \text{pct}_M, \text{EOA}^{\text{recv}})$
 parse w as $(\text{usk}^{\text{send}}, \text{cm}_{\text{old}}, \text{o}_{\text{old}}, v_{\text{in}}^{\text{priv}}, \text{upk}^{\text{recv}}, \text{o}_{\text{new}}, v_{\text{out}}^{\text{priv}}, \text{aux}_U,$
 $\omega,$
 $T_{\text{new}}, \text{o}_{\text{new}}^T, T_{\text{note}}, \text{ek}, k_A, \text{aux}_A, \text{aux}_M, b_{\text{in}})$
 parse usk^{send} as $(k_{\text{ENA}}^{\text{send}}, \text{sk}_{\text{own}}^{\text{send}}, \text{sk}_U^{\text{send}}, T_{\text{old}}, \text{o}_{\text{old}}^T)$
 parse upk^{send} as $(\text{addr}^{\text{send}}, \text{pk}_{\text{own}}^{\text{send}}, \text{pk}_U^{\text{send}})$
 parse upk^{recv} as $(\text{addr}^{\text{recv}}, \text{pk}_{\text{own}}^{\text{recv}}, \text{pk}_U^{\text{recv}})$
 parse pct_A as $(\text{epk}, \text{ct}_{\text{self}}^{\text{fwd}}, \text{ct}_{\text{ext}}^{\text{fwd}}, \text{ct}_{\text{self}}^{\text{bwd}}, \text{ct}_{\text{ext}}^{\text{bwd}}, \text{wk}_A^{\text{FT}}, \text{wk}_A^{\text{BT}}, \text{ct}_A^{\text{msg}})$
 enforce a binding constraint on EOA^{recv} , e.g., $\text{EOA}^{\text{recv}} = 1 \cdot \text{EOA}^{\text{recv}}$
 $\text{pk}_{\text{own}}^{\text{send}} = \text{CRH}(\text{sk}_{\text{own}}^{\text{send}})$
 $\text{addr}^{\text{send}} = \text{CRH}(\text{pk}_{\text{own}}^{\text{send}} \parallel \text{pk}_U^{\text{send}})$
 $\text{addr}^{\text{recv}} = \text{CRH}(\text{pk}_{\text{own}}^{\text{recv}} \parallel \text{pk}_U^{\text{recv}})$
 $\text{CM.Open}(\text{pp}_{\text{CM}}, \text{cm}_{\text{old}}^T, T_{\text{old}}; \text{o}_{\text{old}}^T) = 1$
 $\text{CM.Open}(\text{pp}_{\text{CM}}, \text{cm}_{\text{old}}, (v_{\text{in}}^{\text{priv}}, \text{addr}^{\text{send}}, T_{\text{note}}); \text{o}_{\text{old}}) = 1$
 $b_{\text{in}} \in \{0, 1\}$
if $b_{\text{in}} = 1$ **then**
 $\text{ACC.VerifyMem}(\text{pp}_{\text{ACC}}, \omega, \text{acc}, \text{cm}_{\text{old}}) = 1$
else
 $v_{\text{in}}^{\text{priv}} = 0$
 $T_{\text{note}} = \text{PRF}_{\text{sk}_{\text{own}}^{\text{send}}}(\text{"dummy"} \parallel T_{\text{old}})$
 $\text{nf} = \text{PRF}_{\text{sk}_{\text{own}}^{\text{send}}}(\text{cm}_{\text{old}})$
 $\text{cm}_{\text{new}}^T = \text{CM.Com}(\text{pp}_{\text{CM}}, T_{\text{new}}; \text{o}_{\text{new}}^T)$
 $\text{tag}_T = \text{PRF}_T(\text{"tag"})$
 $\text{cm}_{\text{new}} = \text{CM.Com}(\text{pp}_{\text{CM}}, (v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, T_{\text{new}}); \text{o}_{\text{new}})$
 $(\text{pct}_U, \text{aux}_U) = \text{PE.Enc}_{\text{pk}_U^{\text{recv}}}(v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, T_{\text{new}}, \text{o}_{\text{new}})$
 $\text{FT}_{\text{old}} \leftarrow \text{PRF}_{T_{\text{old}}}(\text{"fwd"})$
 $\text{BT}_{\text{old}} \leftarrow \text{PRF}_{T_{\text{old}}}(\text{"bwd"})$
 $\text{FT}_{\text{note}} \leftarrow \text{PRF}_{T_{\text{note}}}(\text{"fwd"})$
 $\text{BT}_{\text{note}} \leftarrow \text{PRF}_{T_{\text{note}}}(\text{"bwd"})$
 $\text{FT}_{\text{new}} \leftarrow \text{PRF}_{T_{\text{new}}}(\text{"fwd"})$
 $\text{BT}_{\text{new}} \leftarrow \text{PRF}_{T_{\text{new}}}(\text{"bwd"})$
 $(\text{ek}, \text{epk}, \text{aux}_A) = \text{KEM.Encap}((\text{pk}_A, \text{pk}_e))$

(continued)

$\text{ct}_{\text{self}}^{\text{fwd}} = \text{FT}_{\text{new}} + \text{PRF}_{\text{ek}}(\text{"sfwd"} \parallel \text{FT}_{\text{old}})$
 $\text{ct}_{\text{ext}}^{\text{fwd}} = \text{FT}_{\text{new}} + \text{PRF}_{\text{ek}}(\text{"efwd"} \parallel \text{FT}_{\text{note}})$
 $\text{ct}_{\text{self}}^{\text{bwd}} = \text{BT}_{\text{old}} + \text{PRF}_{\text{ek}}(\text{"sbwd"} \parallel \text{BT}_{\text{new}})$
 $\text{ct}_{\text{ext}}^{\text{bwd}} = \text{BT}_{\text{note}} + \text{PRF}_{\text{ek}}(\text{"ebwd"} \parallel \text{BT}_{\text{new}})$
 $\text{wk}_A^{\text{FT}} = k_A + \text{PRF}_{\text{ek}}(\text{"wrap-fwd"} \parallel \text{FT}_{\text{new}})$
 $\text{wk}_A^{\text{BT}} = k_A + \text{PRF}_{\text{ek}}(\text{"wrap-bwd"} \parallel \text{BT}_{\text{new}})$
 $\text{ct}_A^{\text{msg}} = \text{SE.Enc}_{k_A}(v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, \text{cm}_{\text{new}})$
 $(\text{pct}_M, \text{aux}_M) = \text{PE.Enc}_{\text{pk}_M}(T_{\text{new}})$
if $\text{sct}_{\text{old}}^{\text{ENA}} = 0$ **then**
 $v_{\text{old}}^{\text{ENA}} \leftarrow 0$
else
 $v_{\text{old}}^{\text{ENA}} \leftarrow \text{SE.Dec}_{k_{\text{ENA}}^{\text{send}}}(\text{sct}_{\text{old}}^{\text{ENA}})$
 $v_{\text{new}}^{\text{ENA}} \leftarrow \text{SE.Dec}_{k_{\text{ENA}}^{\text{send}}}(\text{sct}_{\text{new}}^{\text{ENA}})$
 $v_{\text{new}}^{\text{ENA}} = v_{\text{old}}^{\text{ENA}} + v_{\text{in}}^{\text{priv}} - v_{\text{out}}^{\text{priv}} + v_{\text{in}}^{\text{pub}} - v_{\text{out}}^{\text{pub}}$
 $v_{\text{new}}^{\text{ENA}}, v_{\text{old}}^{\text{ENA}}, v_{\text{in}}^{\text{priv}}, v_{\text{out}}^{\text{priv}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}} \in [0, 2^{64})$

The procedure $\text{KeyGenUsers}_{\text{Client}}$ generates the user's encryption key pair $(\text{sk}_U, \text{pk}_U)$, encrypted-balance key k_{ENA} , and ownership secret sk_{own} . It derives $\text{pk}_{\text{own}} = \text{CRH}(\text{sk}_{\text{own}})$ and $\text{addr} = \text{CRH}(\text{pk}_{\text{own}} \parallel \text{pk}_U)$, and forms the user secret key usk and user public key upk . It also samples an initial tracing token T and commitment opening o^T , computes the initial token commitment cm^T and tag $\text{tag}_T = \text{PRF}_T(\text{"tag"})$, and generates a proof for $\mathcal{R}_{\text{KGU}}$ showing that the commitment and tag are derived from the same token, with addr constrained as a public input. The registration transaction contains π , addr , cm^T , and tag_T .

The left part of Algorithm 3 gives $\text{zkTransfer}_{\text{Client}}$, which constructs a private transfer transaction from the inputs shown in the algorithm and the current contract state. If an input note is provided, the procedure parses it as a real private input and computes an accumulator membership proof; otherwise, it creates a dummy zero-value input with $b_{\text{in}} = 0$, $v_{\text{in}}^{\text{priv}} = 0$, and $T_{\text{note}} = \text{PRF}_{\text{sk}_{\text{own}}^{\text{send}}}(\text{"dummy"} \parallel T_{\text{old}})$. The transfer relation requires accumulator membership exactly when $b_{\text{in}} = 1$, so zero-valued real notes are still checked, while dummy inputs cannot point to an arbitrary token from a real note. The sender then derives the nullifier from the ownership secret, samples a fresh account-level tracing token T_{new} , computes $\text{tag}_T = \text{PRF}_T(\text{"tag"})$, forms $\text{cm}_{\text{new}}^T = \text{CM.Com}(\text{pp}_{\text{CM}}, T_{\text{new}}; \text{o}_{\text{new}}^T)$, derives the direction-specific forward and backward tokens from T_{old} , T_{note} , and T_{new} , and computes $\text{cm}_{\text{new}} = \text{CM.Com}(\text{pp}_{\text{CM}}, (v_{\text{out}}^{\text{priv}}, \text{addr}^{\text{recv}}, T_{\text{new}}); \text{o}_{\text{new}})$.

To create pct_A , the procedure runs $\text{KEM.Encap}((\text{pk}_A, \text{pk}_e))$ and obtains ek and epk . The procedure uses ek to form the four tracing components $\text{ct}_{\text{self}}^{\text{fwd}}$, $\text{ct}_{\text{ext}}^{\text{fwd}}$, $\text{ct}_{\text{self}}^{\text{bwd}}$, and $\text{ct}_{\text{ext}}^{\text{bwd}}$. These components correspond to four tracing modes: forward self, forward external, backward self, and backward external. For each mode, the corresponding tracing component allows an auditor with a valid input tracing token to derive the adjacent tracing token. In the forward self mode, FT_{old} leads to FT_{new} , while in the

forward external mode, FT^{note} leads to FT_{new} . Conversely, in the backward self mode, BT_{new} leads back to BT_{old} , and in the backward external mode, BT_{new} leads back to BT^{note} .

To complete pct_A , the procedure samples an audit key k_A and computes two wrapped keys, wk_A^{FT} and wk_A^{BT} . The key wk_A^{FT} wraps k_A using ek and FT_{new} , while wk_A^{BT} wraps k_A using ek and BT_{new} . The audit-message ciphertext ct_A^{msg} encrypts the private output value, receiver address, and output commitment under k_A ; the output commitment lets audit and tracing procedures check consistency with the public output commitment of the transaction. Thus, $pct_A = (epk, ct_{self}^{fwd}, ct_{ext}^{fwd}, ct_{self}^{bwd}, ct_{ext}^{bwd}, wk_A^{FT}, wk_A^{BT}, ct_A^{msg})$.

The procedure also creates pct_U and pct_M . pct_U encrypts the private output value, receiver address, fresh tracing token, and output opening to pk_U^{ecv} . pct_M encrypts only T_{new} to pk_M . Finally, the sender updates the encrypted account balance, forms the public statement and witness, generates the zero-knowledge proof, and outputs a transfer transaction Tx_{ZKT} containing the proof, public values, ciphertexts, new output commitment, new token commitment, public token tag, and epoch index. After creating Tx_{ZKT} , the sender stores the tentative state $st_{pend} := (T_{new}, o_{new}^T)$. The confirmed user secret state is updated to st_{pend} only after Tx_{ZKT} is accepted on-chain.

The right part of Algorithm 3 gives the auditor procedures $Trace_{Client}$ and $Audit_{Client}$, and the user procedure $RetrieveNote_{Client}$. The tracing and audit procedures use the epoch secret keys authorized by the tracing token manager. For an authorized interval I , the tracing-token manager discloses the compact set of PRF tree nodes S_E . The auditor expands each disclosed node by recursively applying the child-derivation rule in Section 4.2.3 until reaching its descendant leaves, thereby obtaining $sk_e[j]$ for every epoch $j \in I$. It stores these keys in $Table_{sk_e}$, leaving entries outside I undefined. We write $Table_{sk_e} \leftarrow \text{DeriveEpochKeys}(S_E)$ for this expansion. The resulting table is given to $Trace_{Client}$ and $Audit_{Client}$.

The procedure $Trace_{Client}$ takes as input the ledger L , the current transaction Tx_{ZKT} , the auditor secret key sk_A , a direction-specific tracing token T_{input} , a tracing mode $mode \in \{\text{"Forward Self," "Forward External," "Backward Self," "Backward External"}\}$, and the epoch-key table $Table_{sk_e}$. It searches for an adjacent transaction according to the tracing mode: forward modes consider transactions after Tx_{ZKT} in ledger order, while backward modes consider earlier transactions. Candidates are visited in increasing ledger order in forward modes and in decreasing ledger order in backward modes, so the first successful candidate is the adjacent transaction in the selected mode. The procedure returns the matching transaction, if any, in this mode-specific range together with the information needed to continue tracing. Because the contract records token tags both at registration and at transfer, each accepted account-level token state uses a fresh tracing token except through a tag collisions. Therefore, for a fixed tracing mode, the matching transaction is serialized in accepted ledgers: same-sender transitions are serialized by the sender token-commitment update, while note-spending transitions are serialized by the nullifier check on the spent commitment.

Forward tracing searches for a later transaction whose forward component matches the current tracing token T_{input} . Thus, it tests the forward components of candidate transactions and only requires the epoch keys of those candidates. Backward tracing proceeds from the current transaction toward the appropriate predecessor. Thus, it first uses the corresponding backward component inside the current transaction, together with the current transaction's epoch key, to derive the predecessor token and then scans earlier candidate transactions using the derived token.

For backward tracing, it reads the epoch of the current transaction and parses its auditor ciphertext pct_A to obtain epk , ct_{self}^{bwd} , and ct_{ext}^{bwd} . If $Table_{sk_e}[\text{epoch}]$ is not available, the procedure returns $\perp$; otherwise, it decapsulates epk using sk_A and $Table_{sk_e}[\text{epoch}]$ to recover the current encapsulated key ek . For forward tracing, this current-transaction decapsulation step is not needed.

The procedure then scans candidate transactions Tx_{ZKT}^* in the ledger. For each candidate, it parses Tx_{ZKT}^* to obtain the candidate output commitment cm_{new}^* , auditor ciphertext pct_A^* , and epoch epoch^* . If $Table_{sk_e}[\text{epoch}^*]$ is unavailable, the candidate is skipped. Otherwise, the procedure parses pct_A^* to obtain epk^* , ct_{self}^{fwd} , ct_{ext}^{fwd} , wk_A^{FT} , wk_A^{BT} , and ct_A^{msg} , and decapsulates epk^* to recover ek^* .

In the forward self mode, the candidate token is derived from ct_{self}^{fwd} using $PRF_{ek^*}(\text{"sfwd"} || T_{input})$; in the forward external mode, it is derived from ct_{ext}^{fwd} using $PRF_{ek^*}(\text{"efwd"} || T_{input})$. In the backward self mode, the candidate token is derived from ct_{self}^{bwd} using $PRF_{ek}(\text{"sbwd"} || T_{input})$; in the backward external mode, it is derived from ct_{ext}^{bwd} using $PRF_{ek}(\text{"ebwd"} || T_{input})$. The derived token is then used to recover the candidate audit key from the corresponding wrapped key: wk_A^{FT} for forward tracing and wk_A^{BT} for backward tracing. The procedure decrypts ct_A^{msg} and checks consistency by testing whether the decrypted output commitment equals the candidate output commitment cm_{new}^* . If the commitment check succeeds, it returns the derived tracing token, the matching transaction, and the corresponding token type.

The procedure $Audit_{Client}$ opens the audit information of a single transaction. It first parses the transaction epoch and reads from pct_A the encapsulation value epk , the wrapped audit keys wk_A^{FT} and wk_A^{BT} , and the encrypted audit message ct_A^{msg} . If the epoch secret key $Table_{sk_e}[\text{epoch}]$ is not available, the procedure returns $\perp$; otherwise, it decapsulates epk using both the auditor secret key sk_A and $Table_{sk_e}[\text{epoch}]$ to recover ek . The audit input includes a token type $\text{type} \in \{\text{"Forward," "Backward"}\}$, indicating whether T is a forward or backward tracing token. If type is "Forward," the procedure uses wk_A^{FT} and $PRF_{ek}(\text{"wrap-fwd"} || T)$ to recover k_A . If type is "Backward," it uses wk_A^{BT} and $PRF_{ek}(\text{"wrap-bwd"} || T)$. The recovered audit key k_A decrypts ct_A^{msg} ; the procedure outputs the message only if the decrypted commitment equals cm_{new} .

The procedure $RetrieveNote_{Client}$ scans transfer transactions in the ledger. For each transaction, it decrypts pct_U using sk_U and checks whether the decrypted receiver address matches the user's address. It also checks that the recovered value is in the valid range and that the decrypted value, address, tracing token, and

opening correctly open the public output commitment cm_{new} . If these checks succeed, it appends the recovered note (v, T, o, cm_{new}) to **notes**. After scanning the ledger, it returns the list **notes**.

Algorithm 4 defines the registration and transfer relations. $\mathcal{R}_{KGU}$ includes the registered address as a constrained public input and binds the initial token commitment and token tag to the same tracing token. $\mathcal{R}_{ZKT}$ checks well-formed sender and receiver addresses, real-note membership and dummy-input handling, nullifier derivation, encrypted-balance update with range constraints, output commitment formation, and public withdrawal-recipient binding through EOA^{rev} . It checks the token update, including the old token commitment opening, new token commitment, and token tag, as well as the receiver ciphertext, manager ciphertext, and auditor ciphertext under the active epoch public key. The auditor-ciphertext checks cover the direction-specific tracing components, wrapped audit keys, and encrypted audit messages. Thus, an accepting proof binds the output commitment, old and new token commitments, token tag, receiver ciphertext, auditor ciphertext, and manager ciphertext to one consistent transfer witness. The parameters pp_{CM} and pp_{ACC} are fixed when the transfer circuit is generated.

4.4 | Design and Deployment Considerations

4.4.1 | Sender-Address Visibility

In our account-based smart contract construction, the sender address $addr^{send}$ is public because the contract must identify the encrypted account balance $ENA[addr^{send}]$ and the token commitment $CM_T[addr^{send}]$ to be updated. Consequently, an observer can identify and link transfers carrying the same sender address, even though the private transfer value, the private-output receiver, and the relation between a created commitment and its later spend remain hidden. The present construction therefore does not provide sender-address anonymity. Accordingly, the sender address is treated as public information in the ledger indistinguishability definition.

Within the present construction, this linkage can be reduced, although not eliminated, by using fresh registered addresses. Specifically, a user can generate a fresh key pair, register the resulting address, transfer value to it through a private output commitment, and later spend the corresponding note under that address. Because the private receiver and the relation between the output commitment and its later spend are hidden from public observers, the ledger provides no direct cryptographic link between the old and new sender addresses, while an authorized auditor can still follow this transition using external tracing. This approach does not hide the sender address appearing in an individual transaction, and repeated use of the same registered address remains publicly linkable. Public caller metadata, timing, and other auxiliary information may nevertheless allow correlation across fresh addresses. Providing sender-address anonymity against the contract state for each individual transaction would require a commitment-based private account state with zero-knowledge state updates rather than the address-indexed state used here. Such a design would constitute a different construction and require a separate security and performance analysis.

4.4.2 | Trust and Collusion Boundary

The restricted audit guarantee is conditional on separation between the tracing token manager and the auditor; it does not provide security against unrestricted collusion between them. Under the prescribed authorization procedure, the manager recovers T internally and releases only the requested subset of $\{FT, BT\}$ together with the tree nodes covering the approved epoch interval. If the manager instead discloses T , the auditor can derive both FT and BT , and the direction restriction is lost. The epoch restriction remains effective only while the auditor does not obtain secret keys for epochs outside the approved interval. Under unrestricted collusion, the manager can use sk_M to recover the base tokens from pct_M and msk_e to derive the epoch secret keys. Combined with sk_A , this gives the colluding parties audit visibility beyond the authorized scope. Preventing or detecting unrestricted collusion between the tracing token manager and the auditor is outside the scope of the present construction.

In a practical deployment, the two roles should be assigned to administratively independent organizations with separate operators and key custody. For example, an authorization authority may operate the tracing token manager, while an external regulator or independent audit organization holds sk_A . The manager secret key material can be kept in protected key storage or a hardware security module that decrypts pct_M internally and releases only the approved direction-specific tokens and the tree nodes covering the approved epoch interval. Each release can require approval by multiple operators and be recorded in a signed, access-controlled append-only log. These measures provide operational separation and accountability, but they do not provide cryptographic security against unrestricted collusion or compromise of both roles. Threshold authorization or secure multiparty computation could increase the number of parties that must collude, but such extensions are outside the scope of the present construction.

4.4.3 | Epoch-Key Transitions

A transaction is constructed with respect to the currently active epoch public key, so an epoch-key update before on-chain confirmation may invalidate a proof generated under the previous epoch key. A practical deployment can avoid submissions near epoch transitions or allow a short grace window in which both the current and immediately previous epoch public keys are accepted; the latter requires the contract to retain the previous epoch public key and verify the proof against the key for the transaction epoch. The epoch-key update mechanism prevents activation before the eligibility condition is satisfied; with timely updates, the contract advances one epoch at a time, while delayed updates may make multiple overdue epoch keys eligible in rapid succession.

4.4.4 | Epoch-Tree Parameters

The tree depth is a deployment capacity parameter rather than a cryptographic security parameter. For a planned service lifetime H and epoch duration Δ , a deployment sets

$$N = \left\lceil \frac{H}{\Delta} \right\rceil, d = \lceil \log_2 N \rceil,$$

and may set the eligibility schedule as $t_i = t_0 + j\Delta$, where t_0 is the eligibility time of epoch 0. As a concrete example, a deployment may use 24-h epochs with $N = 2^{16}$ and $d = 16$. This configuration supports 65,536 epochs, or approximately 179 years, and an arbitrary interval requires at most 32 disclosed tree nodes. Reducing the duration of each epoch allows audit authorization to be limited to narrower time intervals, but requires more frequent updates of the active epoch public key.

4.4.5 | Per-Sender Transfer Serialization and Parallel State Lanes

The present construction serializes transfers from the same sender because each transfer reads and updates both $\text{ENA}[\text{addr}]$ and $\text{CM}_T[\text{addr}]$. Pending transfers may be chained using tentative states, but they remain valid only if accepted in the assumed order; otherwise, later-dependent transactions must be regenerated.

A configurable multi-lane extension could maintain independent state pairs $\text{ENA}[\text{addr}, \ell]$ and $\text{CM}_T[\text{addr}, \ell]$ for lanes $\ell \in \{0, \dots, K-1\}$, where K is a deployment parameter. All lanes would share the same registered address and ownership key. Each transaction would include a public lane identifier ℓ , and its proof would bind and update only the state pair selected by (addr, ℓ) . Since a received private-note commitment is bound to addr rather than to a lane, the user could spend that note through any selected lane. The global nullifier set would prevent the same note from being spent through multiple lanes. Consequently, up to K otherwise nonconflicting transactions assigned to distinct lanes could be pending and accepted in any order, while transactions within one lane would remain serialized. This extension would require $O(K)$ state per address, balance management across lanes, and corresponding changes to the algorithms; it is not included in the present implementation or evaluation.

4.4.6 | Tracing-Token State Recovery

The base construction assumes that users securely retain their secret wallet state. If either T or o^T is lost, the sender cannot open $\text{CM}_T[\text{addr}]$ and therefore cannot generate another valid transfer. The state is not recoverable from the ledger because the commitment is hiding and pct_M does not contain o^T ; previously recorded audit information, however, remains valid. An optional recovery extension could define

$$k_R := \text{PRF}_{k_{\text{ENA}}}(\text{"token-recovery"}), \text{ sct}_R \leftarrow \text{SE.Enc}_{k_R}(T_{\text{new}}, o_{\text{new}}^T).$$

Registration and each transfer would include the corresponding sct_R , with the proof binding it to the token commitment. A user retaining k_{ENA} could decrypt the latest accepted sct_R and verify the recovered pair against the current $\text{CM}_T[\text{addr}]$, thereby recovering the state without breaking tracing continuity. This option adds one symmetric ciphertext and its consistency check per state update. If k_{ENA} is also lost, recovery remains impossible.

4.4.7 | Unspent Notes and Voluntary Token Disclosure

If a suspicious note has not been spent, there is no later note-spending successor to reveal. Although the base token alone does not allow the user to perform tracing, the user may voluntarily disclose it to an auditor, who may then derive the corresponding direction-specific tokens and trace the corresponding flow only for epochs for which it has the required epoch-secret keys. Similarly, if a participant deviates from the prescribed uniform sampling procedure and selects a predictable base token, an observer may recover it. These are inherent limitations of token-guided flow tracing and are treated as deployment and policy considerations.

4.4.8 | Retrieval and Tracing Complexity

Let N_L be the number of accepted transfer transactions in the ledger. Since the intended receiver is not publicly identifiable, $\text{RetrieveNote}_{\text{Client}}$ must test pct_U in every accepted transfer, and a complete retrieval takes $O(N_L)$ time. After the initial scan, a client can retain its last processed ledger position and examine only newly appended transfers; the cumulative work over the complete ledger remains $O(N_L)$.

Consider one tracing session authorized for a fixed epoch interval I , and let N_I be the number of accepted transfer transactions whose epochs lie in I . The public epoch value, sender address, and ledger position permit a local off-chain index. A single pass over the N_L accepted transfers builds an array ordered by epoch and ledger position and, for each transfer position, direct links to the immediately preceding and following accepted transfers of the same public sender. The initial construction processes each accepted transfer once and takes $O(N_L)$ time and $O(N_L)$ storage. Each newly appended transfer adds one index record and updates at most two same sender links in $O(1)$ time and $O(1)$ additional storage. If the index has not already been maintained, its $O(N_L)$ construction cost is paid once before the first tracing session.

Because the index is ordered by epoch and ledger position, two binary searches locate the lower and upper boundaries of I in $O(\log N_L)$ time. A complete scan of the N_I transactions between these boundaries takes $O(N_I)$ time. After preprocessing, a direct link identifies the preceding or following transaction from the same sender, and checking this candidate takes $O(1)$ time. Let ℓ be the number of $\text{Trace}_{\text{Client}}$ calls in the session, and let n_r be the number of epoch-authorized candidates cryptographically tested by the r th call. Then $n_r \leq 1$ for a self mode call, while $n_r \leq N_I$ for an external-mode call. For a fixed security parameter, each candidate test uses a constant number of KEM decapsulations, PRF evaluations, symmetric key decryptions, and consistency checks; backward tracing additionally decapsulates the current transaction once. Including the initial interval lookup, the online tracing cost is

$$O\left(\log N_L + \ell + \sum_{r=1}^{\ell} n_r\right),$$

with the conservative bound $O(\log N_L + \ell(N_I + 1))$. For a single tracing path that proceeds only forward or only backward and uses each returned transaction as the starting point of the next

call, retaining the current ledger position prevents previously examined candidate ranges from being scanned again, giving $O(\log N_L + \ell + N_I)$ total online work. A tracing path that follows only successive transactions from the same sender requires $O(\ell)$ online work after preprocessing. The index uses only public metadata and requires no change to the contract or transaction format.

External note-spending edges cannot be indexed in the same way because the creating commitment and the later nullifier are deliberately not publicly linkable, and pct_A contains no stable public lookup key. An index constructed solely from public transaction fields cannot identify such an edge without revealing the hidden flow relation. Even after epoch filtering, locating a previously unknown external edge retains $O(N_I)$ worst-case work. Achieving a faster lookup without exposing the hidden flow relation is beyond the scope of the present construction.

Let $|I| = j_1 - j_0 + 1$ denote the number of epoch indices in the authorized interval. The disclosed set S_E contains at most $2d = O(\log N)$ nodes. Constructing $\text{Table}_{\text{sk}_e}$ by expanding these nodes to obtain $\text{sk}_e[j]$ for every $j \in I$ takes $O(|I|)$ time and $O(|I|)$ storage. This is a one-time authorization preprocessing cost and is separate from the online tracing costs reported above.

5 | Security

This section defines the security model and shows that our construction satisfies ledger indistinguishability, transaction non-malleability, balance, audit correctness, and restricted audit authorization. Since our construction refines the auditor's capability from ledger-wide decryption to flow-restricted and epoch-scoped authorization, we explicitly model the restricted audit scope.

5.1 | Security Model and Oracles

Let Π denote our construction. At the beginning of each experiment, the challenger runs the system initialization algorithms and obtains the public parameters pp , including the public keys of the tracing-token manager and the auditor. The challenger keeps the tracing token manager secret key, including the secret key material used for audit authorization and epoch-key generation. Unless explicitly stated, the auditor secret key is not given to the adversary.

The challenger maintains a public ledger L , an account table **Acct**, a note table **Note**, a token table **Tok**, and the epoch key public bulletin board BB_E . The tables are indexed by public identifiers: **Acct**[addr] stores the user key pairs associated with address addr , **Note**[cm] stores the note needed to spend commitment cm , and **Tok**[addr] stores the private tracing token state needed to open and update the token commitment of address addr . The ledger contains accepted registration, epoch update, and transfer transactions. The adversary can adaptively issue the following oracle queries.

Oracle $\mathcal{O}_{\text{KGU}}$. On a $\mathcal{O}_{\text{KGU}}$ query, the challenger runs $\text{KeyGenUsers}_{\text{Client}}$ to obtain a user key pair (usk, upk) and a registration transaction Tx_{KGU} . It stores the user state in **Acct**,

initializes the user's tracing-token state in **Tok**, appends Tx_{KGU} to L , initializes the user's encrypted account balance, and returns the public output $(\text{upk}, \text{Tx}_{\text{KGU}})$ to the adversary.

Oracle $\mathcal{O}_{\text{UEK}}$. On input an epoch index j , the challenger reads the signed epoch-key tuple from BB_E , forms Tx_{UEK} , and submits it to the ledger. If it satisfies the smart contract checks, the challenger appends it to L and returns Tx_{UEK} ; otherwise, it returns $\perp$.

Oracle $\mathcal{O}_{\text{ZKT}}$. On input an input commitment cm_{old} or $\perp$, a sender public key, a receiver public key, private output value, public input and output values, and receiver external account, the challenger first retrieves the sender secret keys from **Acct** and the sender's tracing-token state from **Tok**. If $\text{cm}_{\text{old}} \neq \perp$, it also retrieves the corresponding input note from **Note**[cm_{old}]; otherwise, it uses the dummy zero-value input note. The challenger computes the accumulator state from the commitments appearing in the accepted transfer transactions in L . The challenger then runs $\text{zkTransfer}_{\text{Client}}$. If the resulting transaction passes the smart contract checks, the challenger appends the transaction to L , stores the newly generated output note in **Note**[cm_{new}], updates **Tok** for the sender to the new tracing-token state, and returns the public transfer transaction. If any check fails, it returns $\perp$.

Oracle $\mathcal{O}_{\text{INS}}$. On input a transaction generated by the adversary, the challenger accepts it only if it satisfies the smart contract checks with respect to the current ledger state. If the transaction is accepted, the challenger appends it to L and returns the transaction; otherwise, it returns $\perp$. This oracle captures adversarially generated valid transactions. The challenger updates its private tables only for states that it can recover from the transaction.

Oracle $\mathcal{O}_{\text{AUTH}}$. This oracle is available only in audit experiments, does not modify the ledger, and represents explicit audit authorization. Authorization is direction-scoped rather than mode-scoped: a forward (resp. backward) authorization permits both self and external tracing steps in that direction. The self and external cases are invocation modes of $\text{Trace}_{\text{Client}}$, not separate authorization scopes. Accordingly, a forward authorization permits both "Forward Self" and "Forward External," a backward authorization permits both "Backward Self" and "Backward External," and a bidirectional authorization permits all four modes. The auditor may invoke any permitted mode as needed within the authorized epoch interval without requesting separate authorization for the self and external edges. On input an initial target transaction Tx_{start} , an audit interval $[j_0, j_1]$, and a tracing scope in $\{\text{"Forward"}, \text{"Backward"}, \text{"Bidirectional"}\}$, the challenger first checks that $0 \leq j_0 \leq j_1 < N$ and returns $\perp$ otherwise. The challenger also checks that Tx_{start} appears as an accepted transfer transaction in the current ledger L and returns $\perp$ otherwise. It then decrypts the corresponding manager ciphertext pct_M from Tx_{start} and internally recovers the base tracing token T_0 associated with the output commitment created by Tx_{start} . It then derives the direction-specific start tokens $\text{FT}_0 = \text{PRF}_{T_0}(\text{"fwd"})$ and $\text{BT}_0 = \text{PRF}_{T_0}(\text{"bwd"})$ as needed. It returns an authorization transcript $\theta = (\text{Tx}_{\text{start}}, j_0, j_1, \text{TokSet}, S_E)$, where S_E is the compact PRF tree node set needed to derive epoch secret keys for $[j_0, j_1]$ and is computed by the recursive procedure described in Section 4.2.3, and TokSet is the set of authorized direction-specific token tuples. For forward authorization, $\text{TokSet} = \{(\text{FT}_0,$

“Forward”)). For backward authorization, $\text{TokSet} = \{(\text{BT}_0, \text{“Backward”})\}$. For bidirectional authorization, $\text{TokSet} = \{(\text{FT}_0, \text{“Forward”}), (\text{BT}_0, \text{“Backward”})\}$. The base tracing token T_0 is used only internally by the challenger and is not released in the authorization transcript. The audit and tracing algorithms are invoked only with direction-specific token types, namely, “Forward” or “Backward.”

5.2 | Public Consistency

A pair of oracle query sequences is publicly consistent if the two sequences contain the same number and order of public operations, and every corresponding query pair (Q_0, Q_1) has the same oracle type and satisfies the public conditions specified below. For paired $\mathcal{O}_{\text{KGU}}$ queries, the two oracle instances must use the same user key pair and tracing-token state and return identical public user keys and registration transactions. For paired $\mathcal{O}_{\text{UEK}}$ queries, the same public epoch-key update transactions are returned, since epoch-key updates are read from the public bulletin board. For paired $\mathcal{O}_{\text{INS}}$ queries, the inserted transactions must have identical public fields and the same acceptance behavior.

For $\mathcal{O}_{\text{ZKT}}$ queries, write the oracle input as

$$Q_{\text{ZKT}} = (\text{cm}_{\text{old}}, \text{upk}^{\text{send}}, \text{upk}^{\text{recv}}, v_{\text{out}}^{\text{priv}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{EOA}^{\text{recv}}),$$

where cm_{old} may be $\perp$ for the dummy input. The resulting public transaction is written as

$$\text{T}_{\text{ZKT}} = (\pi, \text{acc}, \text{nf}, \text{upk}^{\text{send}}, \text{cm}_{\text{new}}, \text{sct}_{\text{new}}^{\text{ENA}}, v_{\text{in}}^{\text{pub}}, v_{\text{out}}^{\text{pub}}, \text{pct}_{\text{U}}, \text{EOA}^{\text{recv}}, \text{cm}_{\text{new}}^{\text{T}}, \text{tag}_{\text{T}}, \text{pct}_{\text{A}}, \text{pct}_{\text{M}}, \text{EPOCH}).$$

The caller account EOA^{send} of the submitted contract transaction is also public ledger metadata. For two corresponding $\mathcal{O}_{\text{ZKT}}$ queries, the sender public keys upk^{send} must be corresponding public keys generated by matching $\mathcal{O}_{\text{KGU}}$ queries, and the caller accounts EOA^{send} must be identical. The receiver external account EOA^{recv} , the public input and output values $v_{\text{in}}^{\text{pub}}$ and $v_{\text{out}}^{\text{pub}}$, and the epoch index EPOCH must be identical. The remaining public transaction components need not be identical. Specifically, π , acc , nf , cm_{new} , $\text{sct}_{\text{new}}^{\text{ENA}}$, pct_{U} , $\text{cm}_{\text{new}}^{\text{T}}$, tag_{T} , pct_{A} , and pct_{M} may differ. The hidden query inputs cm_{old} , upk^{recv} , and $v_{\text{out}}^{\text{priv}}$ may also differ, subject to the conditions below.

Each query must be individually well formed, including value conservation and the required range conditions, and the two queries must either both use a real input commitment or both use a dummy input. The nullifier and token tag generated in each query must not already appear in either oracle ledger. If either receiver public key is not registered in **Acct**, then the receiver public keys and the private output values in the two corresponding queries must be identical. If one input commitment refers to a note generated by an adversarially inserted transaction through the $\mathcal{O}_{\text{INS}}$ oracle, then the other input commitment must also refer to a note generated by an adversarially inserted transaction, and the private input values contained in the two notes must be identical. These requirements rule out trivial distinctions caused by different rejection behaviors, receiver outputs outside the

challenger-managed account table, or different sources of spent notes.

5.3 | Ledger Indistinguishability

Ledger indistinguishability is the privacy notion for adversaries that observe the public ledger. Informally, if two processes for generating a ledger satisfy the public consistency conditions, then an adversary should not be able to determine which process generated the observed ledger. This notion does not provide sender-address anonymity: upk^{send} identifies the registered sender account and EOA^{send} identifies the contract caller, and both are public. An observer can, therefore, link transfers carrying the same sender key and observe their timing and frequency. Its unlinkability claim is limited to the private commitment flow. The adversary cannot determine which commitment is spent by a transfer, except for information fixed by the public consistency conditions.

In Algorithm 5, the adversary receives the two public ledgers as an ordered pair whose order is determined by the challenge bit. When $b = 0$, the pair given to the adversary is (L_0, L_1) , and when $b = 1$, the pair is (L_1, L_0) . The adversary must distinguish the ordering (L_0, L_1) from the ordering (L_1, L_0) . At each iteration, the adversary either submits a query pair (Q_0, Q_1) or outputs its final guess b' . Each query is typed as one of $\mathcal{O}_{\text{KGU}}$, $\mathcal{O}_{\text{UEK}}$, $\mathcal{O}_{\text{INS}}$, or $\mathcal{O}_{\text{ZKT}}$, together with the corresponding input. The challenger checks public consistency and submits Q_0 to $\mathcal{O}_0^H$ and Q_1 to $\mathcal{O}_1^H$, independently of b . Each oracle returns a public answer ans_i and updates its corresponding ledger L_i . The adversary receives the answers and the updated ledgers. We say that $\mathcal{A}$ satisfies ledger indistinguishability if, for every PPT adversary $\mathcal{A}$, the probability that Algorithm 5 returns 1 is at most $1/2 + \text{negl}(\lambda)$.

Theorem 1. *If SNARK is zero-knowledge, CM is hiding, PRF is pseudorandom, the nullifier PRF and token tag PRF are collision resistant, PE and SE satisfy IND-CPA and IK-CPA security, and KEM satisfies key indistinguishability for the two-public-key*

ALGORITHM 5 | Ledger indistinguishability experiment.

```

Run setup algorithms to obtain pp
Initialize  $\mathcal{O}_0^H, \mathcal{O}_1^H$  with pp
Initialize ledgers  $L_0, L_1$  as empty ledgers
 $b \xleftarrow{\$} \{0, 1\}$ 
 $(\text{ans}_0, \text{ans}_1) \leftarrow (\perp, \perp)$ 
while  $\mathcal{A}$  has not output a bit  $b'$  do
     $(Q_0, Q_1) \leftarrow \mathcal{A}(\text{pp}, L_b, L_{1-b}, \text{ans}_b, \text{ans}_{1-b})$ 
    if  $(Q_0, Q_1)$  is not publicly consistent then
        return 0
     $\text{ans}_0 \leftarrow \text{Query}_{\mathcal{O}_0^H}(Q_0)$ 
     $\text{ans}_1 \leftarrow \text{Query}_{\mathcal{O}_1^H}(Q_1)$ 
return  $(b' = b)$ 

```

encapsulation, then Π satisfies ledger indistinguishability as defined in Algorithm 5.

Proof. Let L_0 and L_1 be two publicly consistent ledgers in Algorithm 5. Let q be the total number of oracle queries made by $\mathcal{A}$; since $\mathcal{A}$ is PPT, q is polynomial in λ . We use $\text{Adv}_{\text{SNARK}}^{\text{ZK}}$, $\text{Adv}_{\text{CM}}^{\text{hide}}$, Adv_{PRF} , Adv_{PE} , Adv_{SE} , and Adv_{KEM} to denote the standard advantages against zero-knowledge, commitment hiding, PRF pseudorandomness, public-key encryption security, symmetric-key encryption security, and KEM key indistinguishability, respectively, where Adv_{PE} and Adv_{SE} denote the combined advantages for their IND-CPA and IK-CPA security notions.

If a submitted query pair is of type $\mathcal{O}_{\text{KGU}}$, then the public answers are generated by the same key generation and registration procedures and are independent of the challenge bit. If the query pair is of type $\mathcal{O}_{\text{UEK}}$, then the answers are determined by the public epoch-key bulletin board and are identical in both ledgers. If the query pair is of type $\mathcal{O}_{\text{INS}}$, then public consistency requires the inserted transactions to have identical public fields and the same acceptance behavior. Therefore, these cases give no distinguishing advantage, and it remains to consider query pairs of type $\mathcal{O}_{\text{ZKT}}$.

By public consistency, corresponding $\mathcal{O}_{\text{ZKT}}$ query pairs use corresponding sender public keys generated by matching $\mathcal{O}_{\text{KGU}}$ queries, and have the same receiver external account, public input and output values, and epoch index. We show that each component that may differ is indistinguishable.

We define a sequence of hybrids. Let G_0 be the real experiment in Algorithm 5. In G_1 , the zk-SNARK public parameters are generated by the simulator, and all zk-SNARK proofs π generated by the challenger in $\mathcal{O}_{\text{ZKT}}$ queries are replaced by simulated proofs. By the zero-knowledge property of SNARK,

$$|\Pr[G_0 = 1] - \Pr[G_1 = 1]| \leq q \cdot \text{Adv}_{\text{SNARK}}^{\text{ZK}}.$$

After this replacement, the proofs reveal no information about the witnesses. In all subsequent hybrids, whenever a public statement is modified, the corresponding simulated proof is regenerated for the modified statement.

In G_2 , the public-key ciphertexts pct_U and pct_M are replaced by encryptions of challenge-independent messages. If either receiver public key is not registered in $\mathbf{Acct}$, pct_U is left unchanged; in this case, public consistency requires identical receiver public keys and private output values, while T_{new} and o_{new} are sampled independently and uniformly, so the honestly generated receiver ciphertexts have the same distribution in the two oracle instances. When it is replaced, the receiver ciphertext pct_U hides the private output value, receiver address, tracing token, and output opening, while pct_M hides the tracing token for the tracing-token manager. Since at most $2q$ public-key ciphertexts are replaced, the IND-CPA and IK-CPA security of PE gives

$$|\Pr[G_1 = 1] - \Pr[G_2 = 1]| \leq 2q \cdot \text{Adv}_{\text{PE}}.$$

In G_3 , the output commitments cm_{new} for query pairs in which both receiver public keys are registered in $\mathbf{Acct}$ and token commitments cm_{new}^T are replaced by commitments to challenge-independent values. If either receiver public key is not registered in $\mathbf{Acct}$, cm_{new} is left unchanged; the unchanged tuple

$(\text{pct}_U, \text{cm}_{\text{new}}, \text{tag}_T)$ has the same joint distribution in the two oracle instances by public consistency and the fresh independent randomness. Since at most $2q$ commitments are replaced, the hiding property of CM gives

$$|\Pr[G_2 = 1] - \Pr[G_3 = 1]| \leq 2q \cdot \text{Adv}_{\text{CM}}^{\text{hide}}.$$

The accumulator state acc is derived from the corresponding sequence of commitments by public accumulator updates; hence, the accumulator states remain indistinguishable after the commitment replacement.

In G_4 , the encapsulated keys ek used in auditor ciphertexts are replaced by uniformly random keys, while the public encapsulation values epk remain unchanged. Since at most q auditor ciphertexts are replaced, the key indistinguishability of KEM gives

$$|\Pr[G_3 = 1] - \Pr[G_4 = 1]| \leq q \cdot \text{Adv}_{\text{KEM}}.$$

In G_5 , PRF-derived values that may carry challenge-dependent information are replaced by fresh pseudorandom values. These values include the nullifier nf and the tracing components and wrapped audit keys derived from ek in pct_A . By public consistency, the nullifiers in the paired queries are fresh with respect to both Oracle ledgers. The public token tags tag_T are left unchanged. If either receiver public key is not registered in $\mathbf{Acct}$, the adversary may recover T_{new} from pct_U , but the relation $\text{tag}_T = \text{PRF}_{T_{\text{new}}}(\text{"tag"})$ and the joint distribution identified in G_3 remain unchanged. Otherwise, because the input "tag" is fixed and public, an observer can test a guessed token T' by comparing $\text{PRF}_{T'}(\text{"tag"})$ with tag_T . Nevertheless, each fresh T is sampled independently and uniformly from $\mathbb{F}$, so polynomially many guesses succeed only with negligible probability. Because the underlying T values are sampled independently and uniformly, the tags have a challenge-independent joint distribution. After replacing nf , the only way this step changes rejection behavior is an accidental collision with a previously used nullifier, which occurs with negligible probability by the collision resistance of the nullifier PRF. Since this hybrid replaces at most 7 public PRF-derived values per transfer, the pseudorandomness of PRF gives

$$|\Pr[G_4 = 1] - \Pr[G_5 = 1]| \leq 7q \cdot \text{Adv}_{\text{PRF}}.$$

In G_6 , the symmetric ciphertexts are replaced by encryptions of challenge-independent messages. These ciphertexts include the encrypted account ciphertext $\text{sc}_{\text{new}}^{\text{ENA}}$ and the audit message ciphertext inside pct_A . Since at most $2q$ symmetric ciphertexts are replaced, the IND-CPA and IK-CPA security of SE gives

$$|\Pr[G_5 = 1] - \Pr[G_6 = 1]| \leq 2q \cdot \text{Adv}_{\text{SE}}.$$

Combining the above bounds gives

$$|\Pr[G_0 = 1] - \Pr[G_6 = 1]| \leq q \cdot \text{Adv}_{\text{SNARK}}^{\text{ZK}} + 2q \cdot \text{Adv}_{\text{CM}}^{\text{hide}} + 2q \cdot \text{Adv}_{\text{PE}} + q \cdot \text{Adv}_{\text{KEM}} + 7q \cdot \text{Adv}_{\text{PRF}} + 2q \cdot \text{Adv}_{\text{SE}}.$$

Since q is polynomial and all primitive advantages are negligible, the right-hand side is negligible. In G_6 , every field of the public view that may differ between the two publicly consistent ledgers has the same distribution for $b=0$ and $b=1$. Thus, the adversary's public view is independent of b , so $\Pr[G_6 = 1] = 1/2$.

Hence the probability that Algorithm 5 returns 1 is at most $1/2 + \text{negl}(\lambda)$. $\square$

5.4 | Transaction Nonmalleability

Transaction nonmalleability prevents an adversary from producing a different valid submitted transaction that spends the same hidden commitment as an existing transfer. The spent commitment itself is hidden, so the public way to identify the same hidden spend is equality of nullifiers. For this notion only, let $\text{pub}(\text{Tx})$ denote the submitted transfer transaction with its proof component removed. This treats proof-only variants as the same transfer, since they do not change the public transaction fields or the resulting ledger effect. Thus, Algorithm 6 regards a new transaction as a successful attack if it has the same nullifier as an oracle-generated transfer, has different submitted transaction fields excluding the proof, and would be valid with respect to the ledger state before that transaction was accepted. A transaction with a fresh nullifier is treated as an independently valid transfer.

We say that Π satisfies transaction nonmalleability if, for every PPT adversary $\mathcal{A}$, Algorithm 6 returns 1 with negligible probability. Here, $\mathcal{L}_{\text{ZKT}}^{\mathcal{O}}$ is the set of transfer transactions returned by $\mathcal{O}_{\text{ZKT}}$, L' is the ledger snapshot immediately before Tx was accepted, and $\text{VerifyTx}(\text{Tx}^*, L')$ denotes smart contract verification of Tx^* with respect to that snapshot. The adversary is given both sk_M and sk_A so that the experiment covers adversaries that act as, or obtain the secret keys of, the tracing-token manager or the auditor. In this experiment, oracle responses contain only public outputs. The user secret state maintained by the challenger, including ownership secrets in **Acct**, is not returned to $\mathcal{A}$.

Theorem 2. *Under the oracle interface specified above, if SNARK satisfies knowledge soundness and zero-knowledge, the nullifier PRF is collision resistant and pseudorandom, and CRH is preimage resistant, then Π satisfies transaction nonmalleability as defined in Algorithm 6.*

Proof. Assume that a PPT adversary $\mathcal{A}$ wins Algorithm 6 with non-negligible probability. Then there exist transfer transactions Tx and Tx^* such that Tx was returned by the $\mathcal{O}_{\text{ZKT}}$ oracle, $\text{pub}(\text{Tx}^*) \neq \text{pub}(\text{Tx})$, Tx^* is valid with respect to the ledger snapshot L' before Tx was accepted, and $\text{Tx}^*.\text{nf} = \text{Tx}.\text{nf}$. Since $\text{VerifyTx}(\text{Tx}^*, L') = 1$, the transaction Tx^* contains an accepting zk-SNARK proof. By knowledge soundness, except with negligible probability, there exists an extractable witness for Tx^* . In particular, the witness contains a spent commitment cm_{old}^* and an ownership secret sk_{own}^* such that $\text{Tx}^*.\text{nf} = \text{PRF}_{\text{sk}_{\text{own}}^*}(\text{cm}_{\text{old}}^*)$.

ALGORITHM 6 | Transaction nonmalleability experiment.

```

Run setup algorithms to obtain pp, skM, and skA
Initialize the oracle instance  $\mathcal{O}^{\Pi}$  with pp
 $L \leftarrow \mathcal{A}^{\mathcal{O}^{\Pi}}(\text{pp}, \text{sk}_M, \text{sk}_A)$ 
 $\text{Tx}^* \leftarrow \mathcal{A}^{\mathcal{O}^{\Pi}}(L)$ 

return  $\exists \text{Tx} \in \mathcal{L}_{\text{ZKT}}^{\mathcal{O}} : \text{pub}(\text{Tx}^*) \neq \text{pub}(\text{Tx}) \wedge \text{Tx}^*.\text{nf}$ 
 $= \text{Tx}.\text{nf} \wedge \text{VerifyTx}(\text{Tx}^*, L')$ 

```

The transaction Tx was generated by the $\mathcal{O}_{\text{ZKT}}$ oracle with a valid witness containing cm_{old} and sk_{own} , so $\text{Tx}.\text{nf} = \text{PRF}_{\text{sk}_{\text{own}}}(\text{cm}_{\text{old}})$. Since the two nullifiers are equal, their defining equations give

$$\text{PRF}_{\text{sk}_{\text{own}}^*}(\text{cm}_{\text{old}}^*) = \text{PRF}_{\text{sk}_{\text{own}}}(\text{cm}_{\text{old}}).$$

If $(\text{sk}_{\text{own}}^*, \text{cm}_{\text{old}}^*) \neq (\text{sk}_{\text{own}}, \text{cm}_{\text{old}})$, then two distinct key-input pairs for the nullifier PRF lead to the same output. This contradicts the collision resistance assumption for the nullifier PRF.

It remains to consider the case $(\text{sk}_{\text{own}}^*, \text{cm}_{\text{old}}^*) = (\text{sk}_{\text{own}}, \text{cm}_{\text{old}})$. In this case, the extracted witness for Tx^* contains the same ownership secret sk_{own} used in the oracle-generated transaction Tx . However, sk_{own} is sampled uniformly by the challenger and is not returned by the oracle interface. The public values derived from sk_{own} are $\text{pk}_{\text{own}} = \text{CRH}(\text{sk}_{\text{own}})$ and nullifier values of the form $\text{PRF}_{\text{sk}_{\text{own}}}(\cdot)$. The preimage resistance of CRH prevents recovering sk_{own} from pk_{own} , while the pseudorandomness of the nullifier PRF prevents its public outputs from revealing the PRF key sk_{own} . The zero-knowledge property of SNARK ensures that the oracle-generated proof transcript does not reveal the witness value sk_{own} beyond the public statement. Therefore, producing such a Tx^* with non-negligible probability would imply that $\mathcal{A}$ obtains an unrevealed ownership secret sk_{own} from public transaction data, contradicting the preimage resistance of CRH together with the pseudorandomness of the nullifier PRF and the zero-knowledge property of the proof system.

Thus, a non-negligible winning probability in Algorithm 6 would imply a violation of zk-SNARK knowledge soundness, a collision in the nullifier PRF, or recovery of an unrevealed ownership secret from public values that are assumed not to reveal it. These events occur only with a negligible probability under the stated assumptions. Hence, Algorithm 6 returns 1 only with negligible probability. $\square$

5.5 | Balance

The balance means that an adversary cannot create more value than it consumes through accepted transactions. The public and private parts of a transfer are linked by the transfer relation, while nullifiers prevent the same private note from contributing value more than once.

Here, $\text{List}_{\text{note}}^{\mathcal{A}}$ is treated as a set of notes indexed by distinct commitments; it is not the challenger's internal **Note** table. The procedure **Compute** parses the accepted ledger L and the sets submitted by the adversary, and computes five totals. Only unspent notes that validly open distinct output commitments appearing in L are counted, where a note is considered unspent if its corresponding nullifier does not appear in the ledger nullifier set. The value v^{ENA} is the total remaining value in the selected encrypted accounts, $v_{\text{out}}^{\text{pub}}$ is the total value withdrawn to public accounts, $v_{\text{out}}^{\text{priv}}$ is the total value contained in the selected private output notes, $v_{\text{in}}^{\text{pub}}$ is the total value deposited from public accounts, and $v_{\text{in}}^{\text{priv}}$ is the total value of private input notes that were already present before the challenged ledger computation. We say that Π satisfies balance if, for every PPT adversary $\mathcal{A}$, Algorithm 7 returns 1 with negligible probability.

```

Run setup algorithms to obtain pp
Initialize the oracle instance  $\mathcal{O}^\Pi$  with pp
 $L \leftarrow \mathcal{A}^{\mathcal{O}^\Pi}(\text{pp})$ 
 $(\text{List}_{\text{note}}^{\mathcal{A}}, \text{ENA}, \text{EOA}) \leftarrow \mathcal{A}^{\mathcal{O}^\Pi}(L)$ 
 $(v_{\text{out}}^{\text{ENA}}, v_{\text{out}}^{\text{pub}}, v_{\text{out}}^{\text{priv}}, v_{\text{in}}^{\text{pub}}, v_{\text{in}}^{\text{priv}}) \leftarrow \text{Compute}(L, \text{List}_{\text{note}}^{\mathcal{A}}, \text{ENA}, \text{EOA})$ 
return  $(v_{\text{out}}^{\text{ENA}} + v_{\text{out}}^{\text{pub}} + v_{\text{out}}^{\text{priv}} > v_{\text{in}}^{\text{pub}} + v_{\text{in}}^{\text{priv}})$ 

```

Theorem 3. *If SNARK satisfies knowledge soundness, CM is binding, CRH is collision resistant and preimage resistant, no PPT adversary can produce a valid accumulator membership proof for an element outside the accumulated set, and the nullifier PRF is collision resistant, then Π satisfies balance as defined in Algorithm 7.*

Proof. Assume that a PPT adversary $\mathcal{A}$ wins Algorithm 7 with non-negligible probability. We consider the accepted ledger L together with witnesses for its accepted transfer transactions. For every transfer transaction generated by the oracle, the challenger knows the corresponding witness because it generated the transaction by running $\text{zkTransfer}_{\text{Client}}$.

It remains to account for the accepted transfer transactions inserted by $\mathcal{A}$ through $\mathcal{O}_{\text{INS}}$. Let Bad_{ext} be the event that the knowledge extractor fails to produce a witness satisfying $\mathcal{R}_{\text{ZKT}}$ for at least one such transaction. Since $\mathcal{A}$ is PPT, the number of these transactions is polynomially bounded. Knowledge soundness guarantees that extraction fails for each accepting proof only with a negligible probability. Therefore, by a union bound over all such transactions, $\Pr[\text{Bad}_{\text{ext}}]$ is negligible.

Conditioned on $\neg\text{Bad}_{\text{ext}}$, every accepted transfer transaction inserted by $\mathcal{A}$ has a valid extracted witness. Let W consist of these extracted witnesses together with the witnesses already known to the challenger for transfers produced by $\mathcal{O}_{\text{ZKT}}$, and consider the augmented ledger (L, W) .

We say that (L, W) is balanced if the following conditions hold for every accepted transfer transaction.

1. Each output commitment cm_{new} has a unique opening to the private output value, receiver address, and tracing token contained in the corresponding witness.
2. If an input commitment cm_{old} refers to a previously created output commitment, then the value, receiver address, and tracing token opened by cm_{old} are equal to those of the corresponding previous output commitment.
3. The transfer witness satisfies the value equation $v_{\text{new}}^{\text{ENA}} = v_{\text{old}}^{\text{ENA}} + v_{\text{in}}^{\text{priv}} - v_{\text{out}}^{\text{priv}} + v_{\text{in}}^{\text{pub}} - v_{\text{out}}^{\text{pub}}$, together with the required range conditions.
4. Each private input commitment with $b_{\text{in}} = 1$ is contained in the accumulated commitment set.
5. The same private note cannot be used twice because two spends of the same note produce the same nullifier.

6. An inserted transfer cannot spend a previous output note assigned to a registered account unless it uses the corresponding registered user's spending key.

We now argue that each condition can fail only with a negligible probability. If either Condition 1 or Condition 2 fails, then the same commitment admits two different openings, which contradicts the binding property of CM. If Condition 3 fails while the transaction is accepted, then an accepting zk-SNARK proof exists for a false transfer statement, contradicting knowledge soundness. If Condition 4 fails, then an accepted proof contains a valid accumulator membership proof for a commitment not contained in the accumulated set, contradicting the assumption that no PPT adversary can produce such a proof. If Condition 5 fails, then either the smart contract accepted a repeated nullifier or two distinct nullifier inputs produced the same nullifier. The former contradicts the nullifier check in the smart contract and the latter contradicts the collision resistance of the nullifier PRF. If Condition 6 fails, then an inserted transfer spends an output commitment whose receiver address is registered in **Acct**. The inserted transfer must provide a witness that opens this commitment and also derives the sender address from the spending keys. If the witness opens the commitment to an address different from the registered receiver address, then the same commitment has two different openings, which breaks the binding of CM. If the witness uses different spending keys but still derives the registered receiver address, then two different key pairs give the same address under CRH, which breaks collision resistance of CRH. If the witness uses the same registered ownership public key, then producing the corresponding ownership secret amounts to recovering an unrevealed preimage of $\text{pk}_{\text{own}} = \text{CRH}(\text{sk}_{\text{own}})$, which contradicts the preimage resistance of CRH under the oracle interface.

Conditioned on $\neg\text{Bad}_{\text{ext}}$, a union bound over the six failure events shows that the probability that any of the above conditions fails is negligible. Since $\Pr[\text{Bad}_{\text{ext}}]$ is also negligible, the overall failure probability remains negligible. Thus, the augmented ledger (L, W) satisfies the above conditions except with negligible probability.

Under these conditions, every accepted transfer preserves the value according to the transfer relation. The binding of commitments makes each private output value fixed, accumulator membership ensures that every spent private input refers to a previously created commitment, and the nullifier check ensures that the same private input is not counted more than once. Therefore, Compute cannot output values satisfying $v_{\text{out}}^{\text{ENA}} + v_{\text{out}}^{\text{pub}} + v_{\text{out}}^{\text{priv}} > v_{\text{in}}^{\text{pub}} + v_{\text{in}}^{\text{priv}}$ except with negligible probability. Hence, Algorithm 7 cannot return 1 except with negligible probability, and Π satisfies balance. $\square$

5.6 | Audit Correctness

Audit correctness requires that an authorized auditor recover the transaction-level audit information for an accepted transfer. Following the audit correctness goal inherited from Azeroth [14], the recovered message should match the private output value and receiver address committed in the accepted transaction. In our construction, this recovery is not enabled by the auditor secret key alone. The auditor must additionally hold an epoch-secret

ALGORITHM 8 | Audit correctness experiment.

```

Run setup algorithms to obtain pp, skA, skM, and mske
Initialize the oracle instance  $\mathcal{O}^{\Pi}$  with pp
 $L \leftarrow \mathcal{A}^{\mathcal{O}^{\Pi}}(\text{pp})$ 
 $(\text{Tx}, \text{aux}, \theta) \leftarrow \mathcal{A}^{\mathcal{O}^{\Pi}, \mathcal{O}_{\text{AUTH}}}(\text{L})$ 
parse Tx as (... , cm, ..., pctA, ..., epoch)
parse aux as (v, addr, T, o)
parse  $\theta$  as (Txstart, j0, j1, TokSet, SE)
Tableske  $\leftarrow$  DeriveEpochKeys(SE)
(TTx, typeTx)  $\leftarrow$  DeriveAuthToken(Tx,  $\theta$ , L, skA, Tableske)
if (TTx, typeTx) =  $\perp$  then
    return 0
(v', addr')  $\leftarrow$  AuditClient(Tx, skA, Tableske, TTx, typeTx)
return VerifyTx(Tx, L)
     $\wedge$  Authorized(Tx,  $\theta$ , L)
     $\wedge$  CM.Open(ppCM, cm, (v, addr, T); o) = 1
     $\wedge$   $\theta$  is returned by  $\mathcal{O}_{\text{AUTH}}$ 
     $\wedge$  (v', addr')  $\neq$  (v, addr)

```

key for the transaction epoch and a valid tracing token for the target transaction.

In contrast to Azeroth, our auditor ciphertexts do not include commitment openings in the decrypted audit message. Therefore, audit correctness is not expressed by checking whether the decrypted audit message itself opens the on-chain commitment. Instead, the zk-SNARK relation enforces that the auditor ciphertext and the output commitment are generated from the same transfer witness. For an accepted transfer whose output commitment opens to (v, addr, T), authorized audit decryption recovers the same private output value and receiver address. The auditor ciphertext also includes the output commitment as a consistency check against the public commitment in the transaction.

Algorithm 8 uses an auxiliary value aux to check that the adversary's claimed output opening is consistent with the output commitment. In our construction, aux consists of (v, addr, T, o), while the auditor is required to recover only (v, addr). Additionally, DeriveEpochKeys(S_E) derives epoch secret keys from the PRF tree nodes returned by $\mathcal{O}_{\text{AUTH}}$. The function DeriveAuthToken starts from an authorized direction-specific token tuple in TokSet and repeatedly applies Trace_{Client} along an authorized path from Tx_{start} to Tx; it returns $\perp$ outside the authorized epoch range or tracing path.

We say that Π satisfies audit correctness if, for every PPT adversary $\mathcal{A}$, Algorithm 8 returns 1 with negligible probability. Here, Authorized(Tx, θ , L) holds if Tx has epoch in [j₀, j₁] and can be reached from Tx_{start} by zero or more tracing steps using one of the direction-specific token tuples in TokSet and the epoch-key table derived from S_E.

Theorem 4. *If SNARK satisfies knowledge soundness, CM is binding, and KEM and SE satisfy correctness, then Π satisfies audit correctness as defined in Algorithm 8.*

Proof. Assume that $\mathcal{A}$ wins Algorithm 8 with non-negligible probability. Since VerifyTx(Tx, L) = 1, the transaction contains an accepting zk-SNARK proof. By knowledge soundness, except with negligible probability, there exists a witness satisfying $\mathcal{R}_{\text{ZKT}}$.

Let the extracted witness contain the output opening (v_w, addr_w, T_w, o_w). The transfer relation enforces that Tx.cm is a commitment to (v_w, addr_w, T_w) with opening o_w. It also enforces that ct_A^{msg} encrypts (v_w, addr_w, Tx.cm) under k_A, and that the wrapped audit keys are formed from the same k_A using the epoch-specific encapsulated key and the corresponding direction-specific token derived from T_w.

The adversary's auxiliary value aux opens the same commitment Tx.cm to (v, addr, T). By the binding property of CM, we have (v, addr, T) = (v_w, addr_w, T_w), except with negligible probability. Since θ is returned by $\mathcal{O}_{\text{AUTH}}$ and Authorized(Tx, θ , L) holds, the epoch keys derived from S_E cover the epoch of Tx, and DeriveAuthToken returns the required direction-specific token for T_w. Therefore, by correctness of KEM and SE and by the commitment check in Audit_{Client}, Audit_{Client} returns (v_w, addr_w) = (v, addr).

Thus, if (v', addr') $\neq$ (v, addr), then an accepted proof is inconsistent with the transfer relation, the same commitment has two valid openings, or the authorized decryption procedures fail. The first two events contradict knowledge soundness or commitment binding, and the last violates the correctness of the underlying decryption procedures. Hence, the experiment returns 1 only with negligible probability. $\square$

5.7 | Restricted Audit Authorization

Restricted audit authorization captures that the auditor secret key alone does not give ledger-wide audit visibility. Algorithm 9 models an auditor-side adversary that knows sk_A and may adaptively query $\mathcal{O}_{\text{AUTH}}$, but does not obtain user secret keys, sk_M, msk_e, the epoch signing key, or unauthorized epoch-tree node secrets. This is an honest authorization model rather than a model of security against collusion. The authorization oracle releases tracing tokens and epoch-key material only according to the specified authorization policy. If a base tracing token becomes known from another source, the base token and the direction-specific token states derivable from it are included in KnownT below. Operations enabled by this disclosed material are therefore not claimed as violations of restricted audit authorization.

Let Θ be the set of authorization transcripts returned by $\mathcal{O}_{\text{AUTH}}$. We write $\mathcal{O}_{\text{AUTH}}^{\Theta}$ for the authorization oracle that records each returned transcript in Θ . A token state (Tx, T, type) is reachable from Θ if there exists a transcript in Θ whose starting transaction is Tx and whose TokSet contains (T, type), or if the token state is obtained by repeatedly applying authorized Trace_{Client} steps from such a released direction-specific token within the released epoch intervals.

```

Run setup algorithms to obtain  $pp, sk_A, sk_M,$  and  $msk_e$ 
Set  $\Theta \leftarrow \emptyset$  and  $KnownT \leftarrow \emptyset$ , and initialize  $\mathcal{O}^\Pi$  with  $pp$ 
Allow  $KnownT$  updates only through explicit token-leakage events
 $L \leftarrow \mathcal{A}^{\mathcal{O}^\Pi, \mathcal{O}_{AUTH}^\Theta}(pp, sk_A)$ 
 $(op^*, Tx^*, T^*, type^*, mode^*, S_E^*) \leftarrow \mathcal{A}^{\mathcal{O}^\Pi, \mathcal{O}_{AUTH}^\Theta}(L)$ 
if  $op^* \notin \{\text{audit}, \text{trace}\}$  then
    return 0
 $Table_{sk_e}^* \leftarrow \text{DeriveEpochKeys}(S_E^*)$ 
if  $op^* = \text{audit}$  then
     $out^* \leftarrow \text{Audit}_{Client}(Tx^*, sk_A, Table_{sk_e}^*, T^*, type^*)$ 
     $Tx_{out}^* \leftarrow Tx^*$ 
else
     $out^* \leftarrow \text{Trace}_{Client}(L, Tx^*, sk_A, T^*, mode^*, Table_{sk_e}^*)$ 
    if  $out^* = \perp$  then
        return 0
    parse  $out^*$  as  $(T_{out}^*, Tx_{out}^*, type_{out}^*)$ 

return  $\text{VerifyTx}(Tx^*, L)$ 
     $\wedge out^* \neq \perp$ 
     $\wedge \text{OutScope}(op^*, Tx^*, Tx_{out}^*, T^*, type^*, mode^*, S_E^*, \Theta, KnownT)$ 

```

We model token states already known to $\mathcal{A}$ independently of manager authorization by an explicit leakage set $KnownT$. This set is maintained as part of the experiment state and contains token states externally disclosed to $\mathcal{A}$, such as voluntarily disclosed base tracing tokens together with the direction-specific token states derivable from them, or token states of adversarially generated transactions that $\mathcal{A}$ explicitly registers as known. The challenger does not infer hidden token knowledge from arbitrary adversarial transactions; only token states included in this explicit leakage set are treated as known for the purpose of OutScope . The set $KnownT$ also includes token states that $\mathcal{A}$ can derive locally from these tokens, as well as token states obtained by tracing steps enabled by epoch keys already available to $\mathcal{A}$, but it does not itself provide epoch authorization.

Authorization material held by the same auditor is cumulative: epoch-key material released in one authorization may be combined with any direction-specific tracing token that the auditor already possesses. This does not provide ledger-wide visibility because a valid direction-specific token is still required for each reachable flow.

A tuple $(op, Tx_{in}, Tx_{out}, T, type, mode, S_E)$ is outside the cumulative authorization scope of $(\Theta, KnownT)$ if either the epoch secret keys needed by the operation cannot be derived from the epoch-key material in Θ , or the token state required for the operation is neither reachable from Θ nor contained in $KnownT$. For $op = \text{audit}$, we set $Tx_{out} = Tx_{in}$ and check reachability with

$(Tx_{in}, T, type)$. For $op = \text{trace}$, the submitted $type$ is ignored and the required token direction is determined by $mode$: forward modes require a forward token, and backward modes require a backward token. The trace is in scope only if Trace_{Client} can be applied from Tx_{in} with the submitted token T in the requested mode to obtain Tx_{out} through authorized reachable token states. We write $\text{OutScope}(op, Tx_{in}, Tx_{out}, T, type, mode, S_E, \Theta, KnownT)$ for this condition.

We say that Π satisfies restricted audit authorization if, for every PPT adversary $\mathcal{A}$, Algorithm 9 returns 1 with negligible probability. This experiment is separate from audit correctness: audit correctness says that authorized material opens the right message, while restricted audit authorization says that the auditor cannot open or trace transactions outside the flow, direction, and epoch scope authorized by the tracing-token manager.

Theorem 5. *Assume that the tracing-token manager follows the authorization policy and does not collude with the auditor outside the authorization transcripts recorded in Θ . Also assume that every token state treated as externally known to the adversary is included in the explicit leakage set $KnownT$, and that neither the manager secret keys nor unauthorized epoch-key material is disclosed. If SNARK is knowledge sound and zero-knowledge, CM is binding and hiding, PRF is pseudorandom and preimage resistant with respect to its input even when the key is known, the token tag PRF is collision resistant, PE satisfies IND-CPA security, the two-public-key KEM satisfies key indistinguishability against adversaries that*

know at most one of the two secret keys, SE satisfies IND-CPA and IK-CPA security, and SIG satisfies EUF-CMA security, then Π satisfies restricted audit authorization as defined in Algorithm 9.

Proof. Assume that a PPT adversary wins Algorithm 9 with a non-negligible probability. Then $\text{Audit}_{\text{Client}}$ returns an audit message or $\text{Trace}_{\text{Client}}$ returns a next tracing token and matching transaction for Tx^* , although the supplied token, direction, and epoch-key material are outside the cumulative authorization scope of (θ, KnownT) . $\square$

5.7.1 | Case 1: Unauthorized Epoch

Suppose that some epoch secret key needed by the audit or tracing operation cannot be derived from the epoch key material released in θ . The epoch-key material in θ consists of PRF tree nodes whose descendant leaves cover only the authorized intervals. Under PRF security, these nodes do not allow the adversary to derive epoch-secret keys outside those intervals. Moreover, epoch public keys accepted by the contract are authenticated by signatures under VK_E ; activating an unauthorized epoch public key would therefore break the EUF-CMA security of SIG. Hence, except with negligible probability, the adversary does not know one of the required epoch-secret keys. The corresponding auditor ciphertext is generated under its epoch public key through KEM. Without the epoch secret key, the encapsulated key ek is computationally indistinguishable from a random key by the key indistinguishability of KEM. Therefore, the masks used in the wrapped audit keys and tracing components are pseudorandom, and recovering either the correct audit key k_A or a valid tracing component with non-negligible probability would break the KEM assumption, PRF pseudorandomness, or the security of SE.

5.7.2 | Case 2: Unreachable Token or Unauthorized Direction

It remains to consider the case where the required epochs are covered but the submitted token is neither reachable from the tokens released in θ under the requested audit type or tracing mode nor contained in KnownT . By the definition of OutScope , token states explicitly included in KnownT are not counted as manager-authorization breaks; hence, the remaining case concerns tokens outside both the authorized reachable set and the explicit leakage set. By knowledge soundness of SNARK, every accepted registration or transfer transaction admits an extractable witness satisfying the corresponding relation, except with negligible probability. For registered and transfer-generated token states, the corresponding relation binds the recorded tag to the underlying tracing token, and the contract records each accepted tag in List_{tag} . Thus, any later registration or transfer using the same tracing token is rejected by the token-tag freshness check, and two distinct accepted tokens can share a recorded tag only through a collision in the token-tag function. The only tokens released by the tracing-token manager are the direction-specific token tuples contained in authorization transcripts in θ . In particular, authorization transcripts do not release base tracing tokens; they release only the authorized subset of the corresponding forward and backward tokens. Although each accepted token has a public tag $\text{tag}_T = \text{PRF}_T(\text{"tag"})$, PRF

pseudorandomness on domain-separated inputs implies that this tag does not allow the adversary to compute the corresponding direction-specific tokens $\text{PRF}_T(\text{"fwd"})$ and $\text{PRF}_T(\text{"bwd"})$, except with negligible probability. Apart from this public tag, unreleased base tracing tokens are hidden in public commitments, receiver and manager ciphertexts, and registration or transfer proofs by the hiding property of CM, the IND-CPA security of PE, and the zero-knowledge property of SNARK, respectively. Moreover, forward and backward tokens are derived from the base token using domain-separated PRF labels. Hence a token released for one direction does not reveal the token for the opposite direction, except with negligible probability under PRF pseudorandomness. Even after recovering k_A , the opposite wrapped key reveals only a PRF image of the unreleased token, and recovering that token from the image contradicts the assumed preimage resistance of PRF with respect to its input. The tracing components and wrapped audit keys in pct_A are also masked using PRF under the epoch-specific key ek with separate labels for each tracing mode. Thus, an unauthorized token cannot recover the audit key k_A or enable a valid tracing step for an unrelated flow or for an unauthorized direction, except with negligible probability.

5.7.3 | Case 3: Successful Out-of-Scope Audit or Trace

For $\text{op}^* = \text{audit}$, a successful output requires recovering the audit key k_A that decrypts c_A^{msg} and passes the commitment check. For $\text{op}^* = \text{trace}$, a successful output requires deriving a next tracing token and identifying a matching adjacent transaction whose commitment check succeeds. The preceding cases show that these values are computationally hidden outside the authorized epochs and beyond the reachable or externally known token states. Hence any successful out-of-scope audit or trace would break the hiding property of CM, the zero-knowledge property of SNARK, or the security of PE, KEM, PRF, SE, or SIG, and Algorithm 9 returns 1 only with negligible probability.

6 | Implementation and Experiment

6.1 | Implementation

We implemented both Azeroth [14] and our proposed construction in the same codebase to measure the incremental cost of scoped auditability under a consistent experimental setting.¹ The transfer circuits are written in Rust using arkworks,² and the on-chain layer is written in Solidity. We instantiate the proof system with Groth16 [24] over BN254 [25], relying on a relation-specific trusted setup whose trapdoor is assumed to be discarded. All group elements are validated by curve-membership checks, subgroup checks, and identity-point rejection, and all hash inputs use fixed-length, domain-separated encodings.

6.1.1 | CM, CRH, and PRF

We instantiate the collision-resistant hash function CRH using Poseidon [26], a zk-SNARK-friendly hash algorithm over the scalar field. The commitment scheme has no separate public parameters and is realized as $\text{CM.Com}((v, \text{addr}, T); o) := \text{CRH}(v \parallel \text{addr} \parallel T \parallel o)$. We instantiate PRF by the keyed use of the same hash function, written as $\text{PRF}_k(x) := \text{CRH}(k \parallel x)$, and model this keyed use as a PRF.

6.1.2 | Accumulator

We implement ACC as a Merkle hash tree over commitments. For a commitment cm , the witness ω consists of the sibling hashes from the corresponding leaf to the root, and the transfer circuit checks that this path reconstructs the public root.

6.1.3 | Symmetric-Key Encryption

We instantiate SE as a stream cipher based on a PRF over field elements. For messages consisting of multiple field elements, the mask is generated in a counter (CTR) manner. The i th mask element is $PRF_k(i||r)$.

SE.Enc_k(m) → sct	SE.Dec_k(sct) → m
parse m as $(m_1, \dots, m_\ell) \in \mathbb{F}^\ell$	$(r, ct) \leftarrow sct$
$r \xleftarrow{\$} \mathbb{F}$	parse ct as $(c_1, \dots, c_\ell)$
for $i = 1, \dots, \ell$: $p_i \leftarrow PRF_k(i r)$	for $i = 1, \dots, \ell$: $p_i \leftarrow PRF_k(i r)$
for $i = 1, \dots, \ell$: $c_i \leftarrow m_i + p_i$	for $i = 1, \dots, \ell$: $m_i \leftarrow c_i - p_i$
$ct \leftarrow (c_1, \dots, c_\ell)$	return $m \leftarrow (m_1, \dots, m_\ell)$
return $sct \leftarrow (r, ct)$	

6.1.4 | Public-Key Encapsulation

For auditor ciphertexts, our implementation instantiates the two-public-key KEM using a Diffie–Hellman style construction. The implementation derives two Diffie–Hellman shares from the auditor public key pk_A and the active epoch public key pk_e , and hashes them to obtain the encapsulated key. The resulting encapsulated key can be recovered only using both the auditor secret key sk_A and the corresponding epoch secret key sk_e . The KEM key derivation is domain-separated; in the implementation, the label “audit-kem” is represented by a fixed field constant.

KEM.Encap((pk_A, pk_e)) → (ek, epk, aux)
$r \xleftarrow{\$} \mathbb{F}$
$epk \leftarrow G^r$
$K_A \leftarrow pk'_A$
$K_e \leftarrow pk'_e$
$ek \leftarrow CRH(\text{“audit-kem”} K_A K_e)$
$aux \leftarrow r$
return (ek, epk, aux)
KEM.Decap($(sk_A, sk_e), epk$) → ek
$K_A \leftarrow epk^{sk_A}$
$K_e \leftarrow epk^{sk_e}$
return $ek \leftarrow CRH(\text{“audit-kem”} K_A K_e)$

6.1.5 | Public-Key Encryption

For receiver and tracing-token manager ciphertexts, we use an ElGamal-style public-key encryption scheme [27] with a hash-derived stream mask. The ciphertext contains an ephemeral public key c_0 and masked field elements. The encryption

randomness is included in the witness so that the zk-SNARK relation can check the correct ciphertext generation.

PE.Enc_{pk}(m) → (pct, aux)	PE.Dec_{sk}(pct) → m
parse m as $(m_1, \dots, m_\ell) \in \mathbb{F}^\ell$	parse pct as $(c_0, c_1, \dots, c_\ell)$
$r \xleftarrow{\$} \mathbb{F}$	for $i = 1, \dots, \ell$: $k_i \leftarrow CRH(i c_0^{sk})$
$c_0 \leftarrow G^r$	for $i = 1, \dots, \ell$: $m_i \leftarrow c_i - k_i$
for $i = 1, \dots, \ell$: $k_i \leftarrow CRH(i pk^r)$	return $m \leftarrow (m_1, \dots, m_\ell)$
for $i = 1, \dots, \ell$: $c_i \leftarrow m_i + k_i$	
$pct \leftarrow (c_0, c_1, \dots, c_\ell)$	
$aux \leftarrow r$	
return (pct, aux)	

6.1.6 | Digital Signatures

We use digital signatures only for epoch-key update authorization. In our Solidity implementation, the epoch-key verification key VK_E is represented by an Ethereum address fixed at deployment, and the contract verifies secp256k1 ECDSA signatures through Ethereum’s ecrecover signature-recovery primitive. The signed digest is computed as $Keccak256(addr_{SC} || chainid || j || pk_e || t)$, where $addr_{SC}$ and $chainid$ serve as domain separators. Signature verification occurs only for UpdateEpochKey before replacing the active epoch public key. It is not included in the transfer circuit and therefore does not affect the reported constraint counts.

6.2 | Experiment

We evaluate the incremental overhead of adding token-guided flow tracing and epoch-scoped audit authorization to Azeroth [14]. We report circuit metrics for Merkle tree heights of 16, 32, and 64, and measure gas costs with Foundry³ on an Anvil Ethereum mainnet fork at block 25,062,336. The contracts were compiled with Solidity 0.8.24 with optimizer enabled and 200 runs. For transfer gas, both constructions use the same transfer sequence and report the cost of accepted zkTransfer transactions; deployment and registration costs are reported separately. The circuit setup, proof generation, and proof verification times were measured on a machine running Ubuntu 24.04.4 LTS with an Intel Core i9-13900K CPU and 16 GB of RAM.

Table 2 reports the number of constraints, key sizes, public input size, and proof size for the transfer circuit. Our construction adds 16,349 constraints over Azeroth for all tested tree heights. This increase mainly comes from the additional checks for auditor and manager ciphertexts. Since the Merkle path cost grows with the tree height in both constructions, the relative constraint overhead decreases as the height increases. Our construction increases the transfer public input size from 832 bytes to 1408 bytes. The registration circuit is independent of the Merkle tree height; it has 474 constraints, a proving key of 93,008 bytes, a verification key of 360 bytes, 96 bytes of public inputs, and a Solidity proof of 256 bytes.

Table 3 reports the deployment gas. Our construction deploys one additional registration verifier, which is independent of the

TABLE 2 | Transfer circuit constraints and size metrics by Merkle tree height.

System	Height	Constraints	Size (bytes)			
			PK	VK	Pub. input	Proof
Azeroth	16	21,203	4,115,920	1096	832	256
	32	25,043	4,732,880			
	64	32,723	5,966,800			
Ours	16	37,552	7,780,176	1672	1408	256
	32	41,392	8,397,136			
	64	49,072	9,631,056			

Note: PK, VK, public input, and proof sizes are reported in bytes; PK and VK denote proving and verification keys, respectively.

Merkle tree height and a larger transfer verifier due to the larger public input. The total deployment overhead is about 1.91×10^6 gas across the tested heights.

Table 4 reports the operation gas. Compared with Azeroth, our RegisterUser operation additionally verifies the registration proof and records the initial tracing-token commitment and token tag, while zkTransfer additionally checks and records the transfer token tag. For zkTransfer, our construction adds about 1.78×10^5 gas over Azeroth, and this overhead is almost constant across the tested tree heights. The UpdateEpochKey operation is specific to our construction, costs 62,305 gas, and is independent of the Merkle tree height.

To provide a breakdown of the transfer gas cost, we measure representative internal operations. One on-chain Poseidon hash evaluation costs 33,254 gas in our Solidity implementation. Since each insertion of a transfer output commitment into the Merkle tree performs one leaf hash and one hash per Merkle tree level, the Merkle update cost grows linearly with the tree height. Groth16 verification costs 375,428 gas for the Azeroth transfer proof, 207,921 gas for our registration proof, and 506,618 gas for our transfer proof; the larger transfer verification cost is caused by the larger public input.

Table 5 reports the serialized Ethereum transaction sizes. Our RegisterUser transaction is larger because it includes the registration proof and token tag. Our zkTransfer transaction is 1622

TABLE 4 | Operation gas comparison.

System	Height	RegisterUser	zkTransfer
Azeroth	16	47,637	1,282,429
	32	47,637	1,931,333
	64	47,637	3,229,225
Ours	16	308,940	1,460,233
	32	308,940	2,109,113
	64	308,940	3,407,005

TABLE 5 | Transaction size comparison.

Operation	Azeroth (bytes)	Ours (bytes)
RegisterUser	146	566
UpdateEpochKey	—	405
zkTransfer	1174	1622

bytes, which is 448 bytes larger than Azeroth's 1174 bytes. This increase comes from the additional fields for tracing token ciphertexts, the token commitment update, the token tag, and epoch-related data.

Table 6 reports the Rust running time for the transfer setup, proof generation, and proof verification. Setup and proof generation are slower than Azeroth because our transfer circuit has additional constraints, while the absolute running times remain below 1.9 and 1.5 s, respectively. The registration circuit is much smaller: its setup, proving, and verification times are 0.029 s, 0.023 s, and 0.929 ms on average, respectively.

The evaluation covers the zk-SNARK circuits and smart contract functions. We did not implement or benchmark the off-chain RetrieveNote_{Client} and Trace_{Client} procedures. As analyzed in Section 4.4, complete note retrieval takes $O(N_L)$ time. Constructing the index takes $O(N_L)$ time and $O(N_L)$ storage, while each newly appended transfer can be incorporated in $O(1)$ time with $O(1)$ additional storage. After preprocessing, locating and checking the preceding or following transaction from the same sender takes $O(1)$ time. An external tracing call takes time linear in the number of candidates tested, with $O(N_I)$ work in the worst case for a previously unknown external edge. For a single tracing path

TABLE 3 | Deployment gas comparison.

System	Height	Registration verifier	Transfer verifier	Contract	Total
Azeroth	16	—	1,360,804	2,568,490	3,929,294
	32	—	1,360,804	3,908,206	5,269,010
	64	—	1,361,039	6,587,614	7,948,653
Ours	16	535,992	2,010,435	3,289,837	5,836,264
	32	535,992	2,010,651	4,629,541	7,176,184
	64	535,992	2,010,903	7,308,949	9,855,844

TABLE 6 | Running times for transfer circuit setup, proof generation, and proof verification.

System	Height	Setup (s)	Proving (s)	Verification (ms)
Azeroth	16	0.802	0.617	1.982
	32	0.910	0.715	
	64	1.135	0.901	
Ours	16	1.484	1.142	2.816
	32	1.577	1.224	
	64	1.805	1.404	

Note: Verification time is averaged.

of ℓ calls that proceeds only forward or only backward and uses each returned transaction as the starting point of the next call, the total online work is $O(\log N_L + \ell + N_I)$. These off-chain costs do not affect the on-chain gas, circuit size, or proof generation time.

7 | Conclusion

We proposed token-guided flow tracing for auditable zero-knowledge smart contract transfers, with audit authorization restricted by the transaction flow and epoch scope. By combining a tracing token manager, direction-specific tracing tokens, and epoch keys, the construction lets an auditor follow only an authorized transaction flow within authorized epochs. We analyzed ledger indistinguishability, transaction nonmalleability, balance, audit correctness, and restricted audit authorization and implemented the circuit and smart-contract layers. The implementation adds 16,349 transfer-circuit constraints, about 1.78×10^5 gas per transfer, and 448 bytes to the serialized transfer transaction.

Funding

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2021-II210727, A Study on Cryptographic Primitives for SNARK, 50% and RS-2021-II210528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box, 50%).

Ethics Statement

This article does not contain any studies with human participants or animals performed by any of the authors.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The implementation code that supports the findings of this study is openly available at <https://github.com/zk-Lee/t-auditable-transfer>.

Endnotes

¹<https://github.com/zk-Lee/t-auditable-transfer>

²<https://github.com/arkworks-rs>

³<https://github.com/foundry-rs/foundry>

References

1. S. Meiklejohn and R. Mercer, "Möbius: Trustless Tumbling for Transaction Privacy," *Proceedings on Privacy Enhancing Technologies* 2018, no. 2 (2018): 105–121.
2. S. Noether and A. Mackenzie, "Ring Confidential Transactions," *Ledger* 1 (2016): 1–18.
3. B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards Privacy in a Smart Contract World," in *Financial Cryptography and Data Security*, 12059, (Springer, 2020): 423–443.
4. J. Partala, T. H. Nguyen, and S. Pirttikangas, "Non-Interactive Zero-Knowledge for Blockchain: A Survey," *IEEE Access* 8 (2020): 227945–227961.
5. E. Ben-Sasson, A. Chiesa, C. Garman, et al., "Zerocash: Decentralized Anonymous Payments From Bitcoin," in *Proceedings of the IEEE Symposium on Security and Privacy*, (IEEE, 2014): 459–474.
6. A. Rondelet and M. Zajac, "Zeth: On Integrating Zerocash on Ethereum," ArXiv preprint arXiv:1904.00905 (2019).
7. Z. Guan, Z. Wan, Y. Yang, Y. Zhou, and B. Huang, "BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs," *IEEE Transactions on Dependable and Secure Computing* 19, no. 3 (2022): 1446–1463.
8. N. Narula, W. Vasquez, and M. Virza, "zkLedger: Privacy-Preserving Auditing for Distributed Ledgers," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, (USENIX Association, 2018): 65–80.
9. H. Kang, T. Dai, N. Jean-Louis, S. Tao, and X. Gu, "FabZK: Supporting Privacy-Preserving, Auditable Smart Contracts in Hyperledger Fabric," in *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (IEEE, 2019): 543–555.
10. Y. Chen, X. Ma, C. Tang, and M. H. Au, "PGC: Decentralized Confidential Payment System With Auditability," in *Computer Security—ESORICS*, (ACM, 2020): 591–610.
11. E. Androulaki, J. Camenisch, A. De Caro, M. Dubovitskaya, K. Elkhayaoui, and B. Tackmann, "Privacy-Preserving Auditable Token Payments in a Permissioned Blockchain System," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, (ACM, 2020): 255–267.
12. T. Liu, Y. Liu, D. Zhang, C. Chen, and W. Wang, "ASOZ: Anonymous and Auditable Cryptocurrency With Enhanced Confidential Transaction," in *Security and Privacy in Communication Networks*, (Springer, 2024): 291–311.
13. A. Sarencheh, A. Kiayias, and M. Kohlweiss, "PARScoin: A Privacy-Preserving, Auditable, and Regulation-Friendly Stablecoin," in *Cryptology and Network Security*, (Springer, 2024): 289–313.
14. G. Jeong, N. Lee, J. Kim, and H. Oh, "Azeroth: Auditable Zero-Knowledge Transactions in Smart Contracts," *IEEE Access* 11 (2023): 56463–56480.
15. P. Chatzigiannis, F. Baldimtsi, and K. Chalkias, "SoK: Auditability and Accountability in Distributed Payment Systems," in *Applied Cryptography and Network Security*, 12727, (Springer, 2021): 311–337.
16. B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short Proofs for Confidential Transactions and More," in *Proceedings of the IEEE Symposium on Security and Privacy*, (IEEE, 2018): 315–334.
17. P. M. Fauzi, S. Meiklejohn, R. Mercer, and C. Orlandi, "Quisquis: A New Design for Anonymous Cryptocurrencies," in *Advances in Cryptology—ASIACRYPT*, (Springer, 2019): 649–678.
18. R. W. F. Lai, V. Ronge, T. Ruffing, D. Schröder, S. A. K. Thyagarajan, and J. Wang, "Omniring: Scaling Private Payments Without Trusted Setup," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, (ACM, 2019): 31–48.

19. A. Jivanyan and A. Feickert, "Lelantus Spark: Secure and Flexible Private Transactions," in *Financial Cryptography and Data Security*, (Springer, 2022): 409–447.
20. D. E. Hopwood, S. Bowie, T. Hornby, and N. Wilcox, "Zcash Protocol Specification," 2026, <https://zips.z.cash/protocol/protocol.pdf>.
21. E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, "Solidus: Confidential Distributed Ledger Transactions via PVORM," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, (ACM, 2017): 701–717.
22. Y. Wang, Z. Su, N. Zhang, et al., "SPDS: A Secure and Auditable Private Data Sharing Scheme for Smart Grid Based on Blockchain," *IEEE Transactions on Industrial Informatics* 17, no. 11 (2021): 7688–7699.
23. Y. Wang, Z. Su, T. H. Luan, J. Li, Q. Xu, and R. Li, "SEAL: A Strategy-Proof and Privacy-Preserving UAV Computation Offloading Framework," *IEEE Transactions on Information Forensics and Security* 18 (2023): 5213–5228.
24. J. Groth, "On the Size of Pairing-Based Non-Interactive Arguments," in *Proceedings of the 35th Annual International Conference on Theory and Applications of Cryptographic Techniques*, (ACM, 2016): 305–326.
25. P. S. L. M. Barreto and M. Naehrig, "Pairing-Friendly Elliptic Curves of Prime Order," in *Selected Areas in Cryptography*, (Springer, 2006): 319–331.
26. L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems," in *Proceedings of the 30th USENIX Security Symposium*, (USENIX Association, 2021): 519–535.
27. T. Elgamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory* 31, no. 4 (1985): 469–472.