P-DART: A Tokenization Framework for Settlement on Polymesh Blockchain

Amirreza Sarencheh, Robert Gabriel Jakabosky, and Lovesh Harchandani

[1] The University of Edinburgh, and Polymesh Labs, UK
amirreza.sarencheh@ed.ac.uk, amirreza@polymesh.network
[2] Polymesh Labs
robert@polymesh.network
[3] Polymesh Labs
lovesh@polymesh.network

Abstract. This work builds on *DART* [38], a fully anonymous, account-based tokenization system that achieves regulatory compliance with multi-asset support, asset-specific auditing, and fully privacy-preserving transfers of constant transaction size. While DART establishes the foundations of **D**ecentralized, **A**nonymous, and **R**egulation-friendly **T**okenization, we introduce **P**-DART as its evolution, extending functionality to meet the **P**olymesh blockchain's settlement and compliance requirements. In addition to supporting tokenization and full anonymity, P-DART introduces a *settlement creation mechanism*: a settlement creator defines the sender, receiver, asset type, and value of each leg in a proposed settlement. Settlements in P-DART require explicit affirmations from all counterparties, including the sender. Furthermore, while DART employs *asset-specific auditors* as hidden, passive compliance entities capable of decrypting transactions for their associated asset type, P-DART extends this model with *mediators*. Mediators retain anonymity and decryption authority but act as active participants in the transaction lifecycle—if mediation is required, settlement remains pending until the mediator affirms or rejects it. This dual model supports both retrospective auditability (via auditors) and prospective regulatory control (via mediators). Additionally, P-DART supports account freezing and forced transfers.

Keywords: Polymesh · DART · Digital Asset · Settlement · Privacy · Regulatory Compliance · Tokenization · Anonymity

Table of Contents

1 Introduction

As P-DART builds upon DART [38], this introduction closely follows that of the DART paper. The adaptations and extensions that distinguish P-DART are presented in the subsequent sections.

A fully anonymous payment system ensures that users remain hidden within the entire set of system participants. Specifically, a payment transcript provides the counterparties of any transaction with an anonymity set as large as the total number of users in the system. Several existing blockchain systems, such as Zerocash [40] and Ouroboros Crypsinous [22], achieve this level of anonymity under varying assumptions. A common characteristic of these systems, however, is that they utilize coin-based bookkeeping, sometimes referred to as UTXO-based bookkeeping[4]. Intuitively, this means that during a payment, users spend existing coins and create new coins (of specified amounts) to facilitate value transfer. An inherent consequence of this coin-based operation is that the token holdings of any individual user are dispersed across the various coins they control.

In contrast to the coin-based approach discussed above, the account-based approach employed by systems like Ethereum [8] offers a different method of managing transactions. In an account-based system, each account has a unique identifier which is publicly known, and senders transfer funds directly from their accounts by reducing their balances and crediting the recipient's account. A key advantage of this approach is that the system state has a straightforward representation, such as a table displaying each account's balance. Consequently, various account-related operations can be significantly simpler to perform compared to the UTXO model, where a user's balance is distributed.

The high-level idea behind confidential and anonymous payment schemes, whether UTXO-based or account-based, is for the sender to generate a Zero-Knowledge Proof (ZKP) that demonstrates the well-formedness of the transaction. The transaction conceals the transferred value within a commitment or encryption. In the ZKP, the sender proves the knowledge of secret values required for payment authorization, and proves the transferred value is positive and lies within the appropriate range as dictated by the input and output balances of the coins or accounts involved. Finally, double-spending is prevented by recording transaction-specific cryptographic objects, e.g., nullifiers, or by (homomorphically) reducing the sender's balance as soon as the transaction is received.

From a privacy perspective, the account-based approach poses significant challenges in achieving anonymity. Systems like QuisQuis [16][5], Zether [7], Anonymous Zether [14], and PriDe CT [19] restrict the sender's anonymity set to k out of n, where $k \ll n$ is a security parameter, and their transaction size is $\mathcal{O}(k)$.[6] The primary challenge arises from the requirement that, for full anonymity, every transaction must interact with the complete ledger state. If certain accounts

[4] UTXO stands for "unspent transaction output".

[5] Note that QuisQuis is an account-based cryptocurrency that also uses UTXOs.

[6] For instance, k is 64 for QuisQuis and 256 for anonymous Zether. For an anonymity set of 16, the transaction size for QuisQuis is 26 KB, whereas for anonymous Zether, it is 6 KB. Note that, the maximum block size is assumed to be 100 KB [25].

are excluded from being updated during a transaction, the anonymity of that transaction is compromised, as a small k opens up the possibility of privacy attacks (e.g., see [24,31]). The limitation on the anonymity set is inherent to these systems, as the value of k is constrained by the maximum transaction size that a block can hold. Another challenge is the need to carefully choose the accounts that are part of the anonymity set as the level of anonymity can be greatly diminished depending on the choice of these accounts.

Standard digital currency transactions (e.g., in Bitcoin [32]) allow the receiver to obtain the funds immediately upon blockchain validators processing the transaction. Moving beyond simple payments, the concept of *receiver affirmation* is a well-known regulatory requirement in broader financial transactions such as securities transfers, emphasized by the US Securities and Exchange Commission (SEC) [44], and others [41], [35], [42], [36], [6], [43]. This mechanism grants receivers the authority to accept/reject incoming payments before their accounts are updated. Receiver affirmation enables receivers to manage their obligations proactively and is particularly important for transactions with legal or tax implications, where receivers assume specific responsibilities upon receiving an asset.

Accommodating receiver affirmation introduces additional challenges, most notably in achieving *proof of balance* (PoB). PoB allows any verifier to request the balance of a specific asset under a specific public key or address. A related concept is proof of assets (PoA) [12], which enables a prover to convince a verifier that they possess *at least* a certain amount of assets. Unlike PoB, which requires revealing the *exact* balance in the user's account, PoA focuses solely on proving sufficiency. With receiver affirmation, PoB becomes non-trivial as the sender's balance in their account is split while waiting for the receiver to come online. Depending on the receiver's decision (affirming or rejecting), the funds can either be reversed (added back to the sender's balance) or transferred to the receiver's account balance. Also, receiver affirmation introduces another challenge that addresses scenarios in which the receiver does not affirm the transaction. The system should support *reversibility*, ensuring that if a receiver does not affirm, the sender can reclaim the funds. By allowing the reclamation of unaffirmed funds, reversibility prevents assets from being lost. Moreover, maintaining *simultaneous transactions* allows senders to engage in multiple transactions simultaneously with different parties, without waiting for pending transactions to be affirmed or reversed. This ensures high throughput in the payment system.

In the anonymous payment literature, specifically account-based models, a well-known issue referred to as *concurrency issues* may arise, given that the ZKP of users depends on their current account state [1, 2, 7, 25]. To illustrate the challenge consider that proving a sender's sufficient balance for an outgoing transfer requires referencing a valid account state. However, if the account state changes (for instance, the account is a recipient of another user's transaction which modifies its cryptographic form, e.g., by rerandomizing it with zero, a step necessary to achieve anonymity in some schemes), the sender's ZKP becomes inadmissible. Consequently, the sender's transaction has to be rejected, and the user may also lose the associated transaction fees. We call a system that circumvents this problem as satisfying concurrency of incoming/outgoing transactions. To address this problem, a periodic *pending transfers* mechanism can be introduced, where incoming transfers are held in a pending state and then rolled into accounts periodically at the end of fixed length epochs (time periods consisting of a set number of blocks). By requiring users to publish transactions at the beginning of an epoch and ensuring that pending transfers are processed only at epoch boundaries, the design preserves ZKP validity and prevents conflicts caused by account state changes during transaction processing. At the beginning of each epoch, the sender must query the blockchain to obtain their most updated account state before generating their ZKP. To mitigate double spending, each user is restricted to generating at most *one* transaction per epoch (note that the identity of the sender is hidden inside the anonymity set). While longer epochs slow down transaction generation, shorter epochs increase the likelihood of ZKP invalidation if another user's transaction rerandomizes the sender's account state.

For a payment system, it is important to support *non-interactivity*, allowing senders to initiate payments without requiring the receiver to be online at the same time. This ensures that transactions are not hindered by the receiver's immediate presence. Additionally, the ability of a sender to send to multiple receivers and a receiver to receive (in our context affirm) from multiple senders in a single transaction (referred to as a multi-party transaction (MPT), or batchability [19]) becomes particularly important in privacy-preserving payments. Since the cost of each transaction is generally higher than in a non-privacy-preserving one, supporting MPT increases efficiency. Furthermore, multiple assets support refers to the capability of a protocol to manage, process, and facilitate transactions involving various types of digital assets within a system. Ensuring privacy for the asset type being transferred is also an important consideration.

Regulatory compliance issues arise with full privacy, as auditors may lack access to necessary information when all data is encrypted. While ZKPs can enforce compliance on-chain without the need for encrypting information under an auditor's public key, decisions involving human discretion, such as judicial rulings, are subjective and challenging (or may be even impossible in some cases) to formalize mathematically. Thus, auditors may require direct access to the underlying transaction data. Furthermore, verifying certain regulatory policies that rely heavily on off-chain data can be costly and impractical to facilitate entirely on-chain. In a system with multiple asset types (potentially spanning different jurisdictions), auditing necessitates careful considerations, such as addressing *asset-specific auditing* mechanisms to ensure that only the designated auditors associated with each asset are granted access to perform their audits.

Despite significant advances in anonymous payment systems [4], current solutions face critical challenges in achieving a practical balance between full anonymity, addressing real-world considerations, and regulatory compliance. Reconciling the need for receiver affirmation, PoB, non-interactivity, MPT, support for multiple asset types, circumventing concurrency issues, and asset-specific auditing in a privacy-preserving manner, preferably in a permissionless setting remains an open area of investigation. Addressing these limitations altogether requires a novel approach that draws from the best features of existing systems while overcoming their inherent drawbacks, leading to the design of a versatile, anonymous, and permissionless payment system as we do in this paper.

2 DART and P-DART

2.1 DART vs. P-DART

We design **P-DART** as an evolution of the DART framework [38], building on its principles of **D**ecentralization, **A**nonymity, and **R**egulation-friendly **T**okenization while capturing additional settlement needs. In addition to supporting tokenization and privacy-preserving transfers, P-DART introduces a *settlement creation mechanism*: a settlement creator can define the sender, receiver, asset type, and value of each leg in a proposed settlement. Transaction execution requires all counterparties to submit explicit affirmations or rejections, ensuring that settlement occurs only when all designated participants have actively consented. In DART, issuers designate asset-specific auditors who remain hidden while being authorized to decrypt transactions for a given asset type, serving as passive compliance entities without affecting settlement. P-DART extends this by introducing *mediators*, who, like auditors, have hidden identities and decryption capabilities but act as active participants in the transaction lifecycle. If an asset requires mediation, transactions remain pending until the mediator explicitly affirms or rejects them, ensuring that settlement only occurs under active oversight. This distinction allows P-DART to capture more stringent real-world compliance needs: auditors provide retrospective visibility, while mediators provide prospective control, enabling Polymesh to support both auditability and active, privacy-preserving regulatory intervention depending on the asset type. P-DART also supports account freezing and forced transfers.

2.2 Details of DART

DART builds upon the *account-based* model of PEReDi [23] and Platypus [45] as well as the *unlinkability* techniques of Zerocash [40] while overcoming their inherent drawbacks in this setting. In PEReDi and Platypus each user holds an account and for each transaction, they update their account and efficiently prove in zero-knowledge that the update was performed correctly. However, these account state transitions are authorized by (possibly distributed) validator using a blind re-randomizable signature. Importantly, the public key of the validator must be *fixed* and known to all network participants. In Zerocash, on the other hand, the idea for achieving anonymity is to generate an anonymity set that includes all coins ever introduced into the system and efficiently prove knowledge of a (fresh) UTXO as a member of this anonymity set, without revealing it. While this aligns well with permissionless operation it appears to be inherently dependent on a coin/UTXO oriented mode of operation.

A natural crossover of the two approaches in DART is to generate an anonymity set comprising all *account* states with transactions of constant size. Whenever a user wants to perform an account state transition, they prove knowledge of an unused (old) account state in the ledger and submit their new account state. The validators can verify that the old account is not being reused (e.g., preventing double-spending) without seeing the old account itself, allowing the user's new account to be added to the ledger. Importantly, the old account remains hidden from validators to ensure unlinkability. To prevent double-spending, a deterministic *nullifier* (also known as a tag) is derived and revealed from the account state. This nullifier is recorded on the ledger, ensuring that each account *state* can be used only once. The nullifier set on-chain grows linearly with respect to the number of transactions. Note, however, that this account update process requires both the sender and receiver to be *online* to update their respective accounts at the same time, as only the users themselves have the necessary ZKP witness to make the account updates.

In DART, this crossover approach is extended to obviate the need for interactivity. This is achieved as follows: when the sender submits a transaction, an ephemeral *account-update-record* (AUR) is generated for the (possibly offline) receiver, who can claim it anytime later when they will update their account. The AUR remains pending (as "funds in transit") until the receiver affirms it or the sender reclaims it using the *reversibility* mechanism. Importantly, the receiver affirmation step (which claims an AUR) does not reveal any details about the receiver. It is worth highlighting that the notion of *ephemeral* AUR in DART crucially differs from a coin or UTXO based approach. Particularly, an AUR itself is *not* spendable by the receiver and may be reclaimed by the sender at any time, prior to the receiver affirming and incorporating it into their account.

To prevent double-spending, the sender's balance is debited so that the sender can initiate other transactions while any previous ones are pending. Additionally, the sender can initiate an *efficient* AUR reversion protocol *anytime* before the receiver affirms AUR, thereby preventing indefinite resource locking. Note that once the sender reclaims the funds the receiver becomes permanently unable to claim them as their own. Additionally, a malicious sender cannot reclaim an AUR more than once. Note that, *by definition*, to achieve receiver affirmation there is a need for the receiver to submit a transaction/affirmation on-chain. Hence, for a DART transaction to be considered finalized, two transactions are necessary: one from the sender and one from the receiver. Importantly, in DART, these two transactions do not need to be concurrent.

In order to address regulatory compliance, DART lets asset issuers designate *asset-specific* auditors for their assets. When transacting with a specific asset type, users must include the designated auditor (if any) in the transaction, ensuring that the auditor can decrypt the required data. Only the auditor registered for that asset type can decrypt the transaction data, while other auditors remain oblivious to the transaction details. Achieving asset-type privacy in such a regulated framework presents challenges. Crucially, to maintain asset type privacy, it is necessary to also conceal the public key of the auditor involved in the transaction, as it could reveal the underlying asset type due to the existence of a publicly known mapping between asset types and their associated auditors. Thus, the challenge is twofold: (i) to hide both the asset type and the corresponding auditor's public key[7], (ii) to enable the sender to *efficiently* prove that the hidden public key belongs to the correct auditor for the asset being transferred.

DART notes that the sender and receiver could, in principle, agree *off-chain* by privately communicating to update their account states and subsequently submitting a proof on-chain to prove that the state transition was executed correctly. However, DART intentionally avoids this approach for several reasons: (i) it does not assume the existence of an off-chain communication channel between the sender and receiver at the time of value transfer, although such an assumption is common in the literature (see e.g., [20]). Note that, with this assumption, the receiver in DART can operate more efficiently, as they no longer need to scan the ledger to identify the receipt of AUR. (ii) This approach would necessitate interactive payments to avoid concurrency issues (as discussed before) as there is no way for on-chain validators to verify account state transitions anonymously. Therefore, DART adopts the approach of requiring two on-chain submissions for each payment. This is crucial for maintaining asynchrony between sender and receiver as well as avoiding concurrency issues.

Supporting multiple assets in users' accounts requires careful consideration, particularly in maintaining the hiding properties. The subtlety lies in minimizing the overhead on transaction generation for the sender's ZKP. For instance, if all asset types were to be included within a single account, the witness for the sender's proof in every account state transition would need to encompass all asset types associated with that account. This approach would result in witnesses proportional to the total number of assets supported by the system. Although SNARKs can provide succinct proof sizes, the proving time would remain considerably high since the proof generation complexity scales linearly with the witness size. To address this, DART adopts the use of asset-specific accounts. This approach confines the witness for transaction generation to only the relevant asset type, thereby reducing the computational overhead involved in proof generation.[8]

DART efficiently facilitates *multi-party transactions* (MPT), namely, transfer of a specific asset to multiple receivers in a single transaction, and further receipt/affirm from multiple senders within a single transaction, thereby enhancing efficiency (also known as batchability [19]). Moreover, DART supports a proof of balance (PoB) mechanism, which allows any party to request the

[7] DART requires *key-privacy* [3], as CPA-security alone is insufficient for the problem at hand. Without key-privacy, the encryptions under auditors' public keys could reveal the public keys, thereby compromising privacy.

[8] Having asset-specific accounts leads to linear on-chain storage growth proportional to the number of assets linked to each key. In contrast, consolidating all assets within a single account can reduce storage requirements. However, DART prioritizes sender proof efficiency over on-chain storage minimization in this trade-off.

balance from some user given an account identifier. Users register asset-specific accounts, and the ledger maintains mappings between account identifiers and their associated asset types. A user can prove their balance for a particular asset without revealing any additional details. It is crucial to ensure that a malicious user cannot bypass the PoB protocol. For instance, a malicious sender could initiate a transaction reducing their *spendable* finalized balance, then engage with the PoB verifier and subsequently reverse the pending transaction without the PoB verifier being aware of the pending transaction. To prevent such an exploit, the system enforces the sender to update their (non-spendable) pending balance within their account (hidden) data. This ensures that information about the pending balance is also available to the PoB verifier, allowing them to account for the possibility of potential reversion in the future. Moreover, the reversibility functionality provides an incentive for the receiver to claim their AUR promptly, as the sender retains the ability to reverse the transaction at any time. Not only does the sender have financial incentives to do so, but it is also preferable for the sender to have no pending transactions for privacy reasons (elaborated on later). This creates another incentive for the sender to reverse any pending transactions, which in turn results in the receiver losing access to the funds.

Summary of DART. DART makes the following contributions:

- *Full anonymity and confidentiality in an account-based model with constant transaction size.* DART formalizes the notion of a payment system that ensures privacy of balances, transaction values, and the identities/public keys of both senders and receivers, while also guaranteeing unlinkability between transactions. Moreover, payments in DART obscure the underlying asset types and the identities of auditors. DART operates under an account-based bookkeeping model that differs from other account-based systems with respect to anonymity guarantees in that they are: (i) *partially anonymous*, such as QuisQuis [16], Zether [7], Anonymous Zether [14], and PriDe CT [19]. (ii) *permissioned*, like PEReDi [23] and PARScoin [39]. (iii) *centralized* like Platypus [45]. (iv) *inherently inefficient* where the validator has complexity proportional to the account set for each transaction and have *concurrency issues* like PriFHEte [25].
- *Support for receiver affirmation, proof of balance, reversibility, and simultaneous transactions.* DART formalizes the concept of *receiver affirmation* in digital currencies, enabling receivers to affirm incoming payments before account updates, thus aligning blockchain-based payments with well-known regulatory requirements and empowering receivers to manage obligations, particularly in scenarios involving legal or tax implications. DART addresses the challenges introduced by receiver affirmation, specifically by: (i) introducing *proof of balance* (PoB) protocol to enable verifiers to check exact balances, while accommodating the complexities of split balances during receiver affirmation. (ii) ensuring *reversibility*, allowing senders to reclaim unaffirmed funds to prevent indefinite asset locking. (iii) supporting *simultaneous transactions*, enabling senders to engage in multiple transactions concurrently, while waiting for previous transactions' receiver affirmations.

- *Support for non-interactivity, multi-party transaction, multiple asset-types, and asset-specific auditing while avoiding concurrency issues.* DART supports *non-interactivity* for offline receivers, *multi-party transactions* (MPT) to enable batch transactions for improved efficiency in privacy-preserving settings, and *multiple asset-types* to handle a diverse set of digital assets while ensuring privacy. DART addresses regulatory compliance challenges in privacy-preserving systems by incorporating encrypted transaction data with auditor access when necessary, enabling *asset-specific auditing* to ensure jurisdictional alignment and granting designated auditors access. Finally, DART's design does not suffer from the *concurrency issues* common in account-based anonymous payments as discussed above.
- *Universal composition (UC) security and performance evaluation.* DART formalizes decentralized and fully anonymous payments with the properties described above in a UC framework [10] via its ideal functionality $\mathcal{F}_{\mathsf{DART}}$ and provides an efficient realization of $\mathcal{F}_{\mathsf{DART}}$ via its construction Π_{DART}. This ensures that Π_{DART} can be securely integrated with other decentralized finance (DeFi) systems or traditional finance (TradFi) legacy systems. Π_{DART} employs well-established cryptographic schemes and utilizes two non-interactive zero-knowledge (NIZK) proof systems: Σ-protocols and Bulletproofs. Its efficiency is demonstrated through implementation and performance evaluation.

3 Related works

Overview of the related works. Early research highlighted that Bitcoin [32] provides only pseudonymity rather than anonymity [26, 27]. This limitation spurred significant academic and practical efforts to address the issue, leading to the development of a rich body of privacy-preserving payment systems each operating under distinct assumptions, settings, and properties. These systems can be broadly categorized into two classes: UTXO-based systems and account-based systems. In the UTXO-based category, notable examples include Zerocoin [30], Zerocash [40], and Monero [34]. On the other hand, examples of account-based systems include Zether [7], Anonymous Zether [14], Platypus [45], PEReDi [23], PriDe CT [19], PARScoin [39], and PriFHEte [25]. We present an overview of these systems as well as how they compare to *DART* in Table 1. To maintain clarity and conciseness, additional properties such as **permissionlessness**, **multiple assets support**, and **asset-specific auditing** are not included in this table. For a more detailed comparison, see the following paragraphs.

Among the first pioneering approaches to privacy enhanced payments was Zerocash [40], which builds upon the earlier protocol Zerocoin [30]. These works introduced the concept of *decentralized anonymous payments*, proposing a payment system that guarantees strong confidentiality, anonymity, and public verifiability. Zerocash, a UTXO-based fully anonymous payment system, employs commitments and zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) to conceal transaction details, including the transaction value and the identities or public keys of both the sender and the receiver. Each transaction

Table 1: Overview of privacy-preserving payment systems and their properties.
Full anonymity (✓*: The system only supports a small number of users.)
Proof of balance (△: The scheme does not offer the property, but it can be achieved with reasonable modifications to the construction.)
Concurrency: concurrency of incoming/outgoing transactions
Receiver affirmation and reversibility (✓*: The system has interactive transactions, hence receiver affirmation is trivially addressed.)

	Full anonymity	Proof of balance	Concurrency	Receiver affirmation	Non-interactivity	Multi party transaction	Transaction size
Zerocash [40]	✓	✗	✓	✗	✓	✗	$\mathcal{O}(1)$
Monero [34]	✗	✗	✓	✗	✓	✓	$\mathcal{O}(k)$
zkLedger [33]	✓*	✓	✓	✗	✓	✓	$\mathcal{O}(k)$
QuisQuis [16]	✗	✗	✗	✗	✓	✓	$\mathcal{O}(k)$
Zether [7]	✗	△	✗	✗	✓	✗	$\mathcal{O}(k)$
Ano. Zether [14]	✗	△	✗	✗	✓	✓	$\mathcal{O}(k)$
Veksel [9]	✗	✗	✓	✗	✓	✗	$\mathcal{O}(1)$
Platypus [45]	✓	△	✓	✓*	✗	✗	$\mathcal{O}(1)$
PEReDi [23]	✓	△	✓	✓*	✗	✗	$\mathcal{O}(1)$
PriDe CT [19]	✗	△	✗	✗	✓	✓	$\mathcal{O}(k)$
PARScoin [39]	✓	✗	✓	✗	✓	✗	$\mathcal{O}(1)$
Ocash [20]	✓	✗	✓	✗	✓	✓	$\mathcal{O}(1)$
PriFHEte [25]	✓	△	✗	✗	✓	✗	$\mathcal{O}(1)$
DART	✓	✓	✓	✓	✓	✓	$\mathcal{O}(1)$

in Zerocash includes the value of a cryptographic accumulator that aggregates all coins ever introduced into the system, thereby forming the anonymity set. The sender proves ownership of a coin within this set by generating a zk-SNARK proof, without explicitly revealing the specific coin being spent. Notably, the size of each transaction remains constant independent of the size of the anonymity set, unlike the majority of (partially) anonymous account-based systems. To prevent double-spending, each coin is associated with a unique nullifier, which is publicly revealed upon spending as in Chaum's original blind signature-based e-cash [13].

One of the main distinctions between DART and Zerocash [40] is that PoB is not supported by Zerocash, as the user's balance in Zerocash is distributed across (potentially many) UTXOs, which a malicious receiver may selectively omit during a PoB procedure. Note that once a sender submits their transaction, the ownership of the funds is transferred to the receiver. In contrast, in DART, ownership is *not* transferred to the receiver until they claim AUR and update their account balance accordingly (i.e., AURs are not spendable). Until that moment, the sender retains the ability to reclaim AUR at any time via the reversibility operation. Furthermore, the PoB prover has to disclose any pending funds. The authors of Zerocash assume two inputs and two outputs for simplicity, but their scheme can be extended to n inputs and m outputs. As such, Zerocash can in principle support MPT. However, for n UTXOs consumed as inputs, the user must provide n accumulator membership proofs, which constitute the computationally costly part of the user's ZKP. In contrast, DART allows a receiver to claim multiple AURs within a single account state transition, requiring only *one* accumulator membership proof via a *single* transaction. Similarly, a sender can provide a single accumulator membership proof while simultaneously creating multiple AURs intended for different receivers within a single transaction. Note that the number of AURs is upper-bounded by the block size.

Monero [34] employs a UTXO-based model to ensure transaction anonymity. It leverages ring signatures, where the anonymity set consists of a group of public keys included in the signature's ring and a transaction grows proportionally to that. In QuisQuis [16], each user maintains an account, and transactions utilize UTXO inputs rather than direct account debits. To preserve anonymity, QuisQuis incorporates anonymity sets with updatable keys that provide unlinkability.

To support auditing functionalities, zkLedger [33] introduced a ledger representation using a table-based structure, where each row corresponds to a transaction with entries for all participants in the system. Each transfer is recorded as a transaction containing commitments to values, depending on whether the amount is being debited (commitment to a negative value) or credited (commitment to a positive value). This design allows auditors to efficiently verify the balance of each user. However, zkLedger has a significant limitation: it is designed to support anonymity for only a small number of users as the transaction size grows linearly in the number of users, unlike DART where transaction size is of constant size. Moreover, zkLedger operates in a permissioned setting and requires participants to have out-of-band communications.

Zether [7] and Anonymous Zether [14] are anonymous payment system designs that hide the user balance, transaction value, and sender and receiver identities. These systems are account-based, where each public key holder is associated with an ElGamal encryption of their balance under their own public key. To generate a transaction, the sender encrypts the transaction value under the receiver's public key and the negative transaction value under their own public key. Additionally, the sender encrypts zero under some randomly chosen (dummy) public keys. These dummy public keys, along with the sender's and receiver's public keys, form an anonymity set (a ring), concealing the identities of both the sender and the receiver. The sender then proves in zero knowledge that all encryptions are well-formed and that the transaction has been constructed correctly. The system restricts each sender to at most *one* transaction per epoch, where an epoch consists of k consecutive blocks. At the beginning of each epoch, the sender must query the blockchain to obtain their most updated state before generating their ZKP. This update is necessary because, at the end of each epoch, the blockchain rolls over all pending states to permanent ones. If other users include the sender's public key in their anonymity sets, the sender's state will be updated accordingly at the end of the epoch. Anonymous Zether makes ZKPs of Zether more efficient. These works only support k-anonymity and have concurrency issues, with a transaction size of $\mathcal{O}(k)$. PriFHEte [25], while achieving full anonymity in an account-based model with a constant transaction size of $\mathcal{O}(1)$, similarly suffers from concurrency issues (further discussion on PriFHEte follows). In contrast, in DART, no one updates another user's account state except the user themselves, allowing the sender to submit transactions at any time and avoiding such concurrency issues. Moreover, receiver affirmation, multiple assets, and asset-type-specific auditing are not supported in [7, 14, 25]. It is worth noting, however, that unlike DART, the nullifier set in Zether, Anonymous Zether, and PriFHEte does not grow, as it is reset to empty at the end of each epoch. PriDe CT [19] is a follow-up work on Anonymous Zether, inheriting some of its properties such as being account-based, achieving k-anonymity, and having a transaction size of $\mathcal{O}(k)$, while also introducing support for MPT.

Veksel [9] is an anonymous account-based payment system that shares similarities with Zerocash [40] in that it supports an arbitrarily large anonymity set. Payments in Veksel are facilitated by having the sender generate a coin commitment along with its encryption under the receiver's public key. During the collection process, a random spending tag (i.e., nullifier) is revealed. While the system ensures unlinkability for each transaction to its sender and receiver and obscures the transferred value, it does not hide the identities of the involved parties (e.g., balance updates are known), thus lacking anonymity.

PEReDi [23] and Platypus [45] are fully anonymous, account-based Central Bank Digital Currency (CBDC) payment systems. In Platypus, the central bank maintains the state of the system. PEReDi employs a distributed architecture, where cryptographically thresholdized maintainers manage the state of the system in a distributed (also referred to as *permissioned*) and asynchronous manner. In both schemes, the sender interacts with the receiver (hence, these are *interactive* systems[9]) to generate a transaction, proving in zero-knowledge that their account state transitions are correct. Each account state transition leaves a unique nullifier as a footprint, enabling the state maintainer(s) to detect double-spending. Additionally, for each state transition, users receive a signature from the state maintainer(s), confirming the validity of the operation. Payments in PEReDi's distributed setting do not require Byzantine Agreement or Byzantine Broadcast during the optimistic path of a transaction. This leads to efficient state transitions whenever transaction counterparties behave honestly. Maintainers independently sign each user's account state transition using a threshold blind and randomizable signature without communicating with one another. Users then collect signature shares from maintainers to consolidate a full signature on their updated account state. PEReDi introduces an approach for tracing potentially malicious users through the collaboration of a threshold number of

[9] A limitation of interactive systems is that a sender cannot send funds if the receiver is not online and engaged in the transaction submission.

maintainers. The revealed double-spending prevention unique tags also serve as a means to trace users. Multiple assets and asset-type-specific auditing are not supported in PEReDi and Platypus. In Platypus, the entire system operates under a centralized auditor, and in PEReDi, it operates under a cryptographically thresholdized set of auditors (the auditors are not asset-specific). These systems do not have concurrency issues and have a transaction size of $\mathcal{O}(1)$. Due to their interactive nature, they trivially support receiver affirmation by default.

PARScoin [39] is a fiat currency tokenization follow-up work on PEReDi that similarly has a fully anonymous, asynchronous, account-based and distributed model with no concurrency issues and transaction size of $\mathcal{O}(1)$. Note that fiat-backed stablecoins are a specific use case of DART's general paradigm. In the case of PARScoin, the real-world asset is specifically a fiat currency, and as a result of tokenization, the on-chain asset is referred to as a stablecoin, representing the tokenized form of the fiat currency[10]. PARScoin extends the interactive payment model of PEReDi to a non-interactive one. It supports privacy-preserving auditability using homomorphic encryptions ensuring that the stablecoin in circulation is backed by sufficient off-chain funds, which is important for price stability. In PARScoin, if the receiver is unable to claim the funds for any reason (e.g., unwillingness to affirm the transaction due to tax liabilities, loss of access to their wallet, etc.), the sender's funds are locked forever. In contrast, a DART sender is able to reclaim funds if the receiver does not affirm the transaction. DART's reversibility solution incentivizes the receiver to affirm the transaction and update their account balance as soon as possible to guarantee the ownership transfer. This differs from PARScoin's setting, in which ownership is immediately transferred to the receiver as soon as the sender submits their transaction. In PARScoin, the receiver must update their account balance by claiming the funds sent by the sender to be able to spend them. However, the receiver can perform this action at any time. For example, consider a balance upper-bound requirement, which is a common regulatory requirement [23,39,45]. A potentially malicious receiver could delay claiming the incoming funds until after sending funds to others (reducing their balance), ensuring that their balance remains within the permissible range to accept new funds.

Recently, OCash [20] introduced a novel approach to anonymous payment systems in the account-based model, specifically designed to accommodate light clients. At a high level, the payer conducts payments by placing coins on the ledger, which can later be collected by the payees in a privacy-preserving manner. The system offers two levels of anonymity. In the weaker variant, the payer is able to observe when a payee claims the coin. However, the proposed design is limited by two assumptions that constrain its practical usability. First, it assumes the existence of a private channel between the payer and the payee. Second, the system requires an anonymizer service that maintains a private state and regularly posts messages on the ledger. The anonymity guarantees of OCash heavily depend on this second assumption, as the anonymizer functions

[10] Since DART supports multiple asset types, one of these asset types can be a stablecoin.

as the core of an Oblivious RAM-like structure [17], with the ledger acting as the database containing the committed or encrypted coins.

PriFHEte [25] is a novel approach to ensuring anonymity and confidentiality in account-based cryptocurrency transactions. The proposed protocol achieves full anonymity while having $\mathcal{O}(1)$ transaction size. This work leverages a property known as wrong-key decryption (WKE). The authors combine this property with fully homomorphic encryption (FHE) to design a payment system capable of facilitating "compact" transactions, which, with the assistance of chain validators, simultaneously update all accounts in the system for each transaction. The protocol employs a key-private public key encryption scheme with the WKE property to encrypt transactions, and an FHE scheme to encrypt user balances stored on-chain. When Alice sends an amount v to Bob, she creates a transaction that encrypts v using both her and Bob's WKE public keys. She also generates a corresponding NIZK proof to verify the transaction's validity. User balances are stored on-chain as ciphertexts, encrypted under their respective FHE public keys. Chain validators, equipped with the WKE public key, the FHE public key, and an encryption of the WKE secret key under the user's FHE public key, facilitate account state transitions. Validators must maintain this information for all users (accounts). After validating the sender's proof, validators execute the following steps for each account commitment recorded on-chain: Using the ciphertext of the WKE secret key encrypted under the user's FHE public key and the sender's transaction (which contains the encryption of v under both the sender's and receiver's WKE public keys), they generate encryptions with a plaintext value of zero for all non-counterparty users. For the actual sender and receiver, the plaintext value is v. Next, the FHE scheme is used to homomorphically deduct the amount v from the sender's balance and add it to the receiver's balance. The balances of all other users remain unchanged, as their old balances are updated by zero. The approach leverages the WKE property to avoid requiring the sender to generate zero-value encryptions for dummy public keys. Instead, validators utilize the WKE property to blindly generate these zero-value encryptions for all non-counterparty users. In addition to concurrency issues (discussed above), PriFHEte [25] needs homomorphic updates to *all* account commitments recorded on-chain for *each* transaction by blockchain validators (the validator complexity is $\mathcal{O}(n)$ for each transaction), raising concerns about its practicality in terms of both *transaction fees* and *computational efficiency*[11]. Biccer and Tschudin [5] demonstrated a double-spending attack on PriFHEte. In response, the authors proposed a mitigation technique involving two commitments to address this vulnerability.

[11] As the number of accounts in the system grows, the fees associated with each transaction would increase, since validators must update more on-chain entries, including the sender's. This dynamic seems counterintuitive, as network effects typically reduce user costs.

4 Ideal functionalities

Our construction employs the following ideal functionalities: the key generation $\mathcal{F}_{\mathsf{KeyReg}}$ (Section 4.1), which securely manages parties' key pairs by generating, storing, and returning public/secret keys on request, and by mapping public keys back to their owning party identifiers; the non-interactive zero-knowledge (NIZK) functionality $\mathcal{F}_{\mathsf{NIZK}}$ (Section 4.2), which provides a way to generate and verify non-interactive zero-knowledge proofs such that any party can check validity without learning the hidden witness, ensuring soundness and zero-knowledge; its variant $\mathcal{F}^{\dagger}_{\mathsf{NIZK}}$, which strengthens this by preventing leakage of the statement to the adversary; the ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ (Section 4.3), which models a global public ledger where parties can read the current state or append validated transactions, subject to adversarial control and validation rules; and the communication channel functionality $\mathcal{F}_{\mathsf{ch}}$ (Section 4.4), which provides a secure channel that delivers messages from a sender to a receiver.

Remark 1. This section is identical to the ideal functionalities section of the DART paper [38].

4.1 Key generation functionality

The ideal functionality $\mathcal{F}_{\mathsf{KeyReg}}$ specifies a public key encryption (PKE) and account key generations, parameterized by a PKE key generation algorithm $(\mathsf{pk}_{\mathsf{en}}, \mathsf{sk}_{\mathsf{en}}) \xleftarrow{\$} \mathsf{PKE.KeyGen}(1^{\lambda})$ (Definition 1) and an account key pair $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}})$ derived from $\mathsf{Acc.KeyGen}(1^{\lambda})$, where $\mathsf{sk}_{\mathsf{acct}} \xleftarrow{\$} \mathbb{Z}_q^*$ and $\mathsf{pk}_{\mathsf{acct}} \leftarrow g^{\mathsf{sk}_{\mathsf{acct}}}$ for a generator g of a cyclic group $\mathbb{G}$ of prime order q. The functionality supports three operations: (i) *key generation:* upon receiving a key generation request from a party P, it generates new PKE and account key pairs if none already exist for P, otherwise output already exist ones. Upon generation $\mathcal{F}_{\mathsf{KeyReg}}$ stores them as $\mathsf{addr}_{\mathsf{pk}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$ and $\mathsf{addr}_{\mathsf{sk}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$ under P, and outputs them to P. (ii) *key retrieval:* upon receiving P^* from any network entities, $\mathcal{F}_{\mathsf{KeyReg}}$ returns the public key pair $\mathsf{addr}_{\mathsf{pk}}$ corresponding to P^* if it exists. (iii) *identifier retrieval:* upon receiving pk from any network entity P, $\mathcal{F}_{\mathsf{KeyReg}}$ identifies and returns the associated party P^* if pk matches either the PKE or account public key recorded for P^*.

Functionality $\mathcal{F}_{\mathsf{KeyReg}}$

– **Key generation.** Upon input $(\mathtt{Gen.Key}, \mathsf{sid})$ from some party P, if there already exists $(\mathsf{P}, \mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$, output $(\mathtt{Gen.Key}, \mathsf{sid}, \mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ to P and abort. Else, call $\mathsf{PKE.KeyGen}$, and $\mathsf{Acc.KeyGen}$ to generate $(\mathsf{pk}_{\mathsf{en}}, \mathsf{sk}_{\mathsf{en}})$ and $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}})$ respectively (re-call if they are not fresh). Set $\mathsf{addr}_{\mathsf{pk}} \leftarrow (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$, and $\mathsf{addr}_{\mathsf{sk}} \leftarrow (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$. Record $(\mathsf{P},$

$\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$. Output $(\texttt{Gen.Key}, \mathsf{sid}, \mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ to P via private-delayed output.

- **Key retrieval.** Upon input $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P}^*)$ from some party P, ignore if $(\mathsf{P}^*, \mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ is not recorded. Else, output $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P}^*, \mathsf{addr_{pk}})$ to P.
- **Identifier retrieval.** Upon input $(\texttt{Rtrv.id}, \mathsf{sid}, \mathsf{pk})$ from some party P, ignore if $(\mathsf{P}^*, (\mathsf{pk}, \cdot), \cdot)$ or $(\mathsf{P}^*, (\cdot, \mathsf{pk}), \cdot)$ is not recorded for some P^*. Else, output $(\texttt{Rtrved.id}, \mathsf{sid}, \mathsf{P}^*)$ to P.

4.2 Non-interactive zero-knowledge functionality

The functionality $\mathcal{F}_{\mathsf{NIZK}}$ defines a *non-interactive zero-knowledge* (NIZK) proof system parameterized by a relation $\mathcal{R}(x, w)$, where x is a statement and w is a corresponding witness introduced by Groth et al. [18]. $\mathcal{F}_{\mathsf{NIZK}}$ consists of two main operations: *proof generation* and *proof verification*. In contrast to interactive zero-knowledge proofs, NIZK does not require the prior specification of the verifier. Consequently, the resulting proof can undergo verification by any party.

In the *proof generation* phase, when a party U requests to generate a proof by sending (x, w), $\mathcal{F}_{\mathsf{NIZK}}$ first checks if w satisfies the relation $\mathcal{R}(x, w) = 1$. If the relation holds, it notifies the adversary $\mathcal{A}$ with x but not w. The adversary then sends back a proof π, which $\mathcal{F}_{\mathsf{NIZK}}$ stores together with x and returns π to U.

In the *proof verification* phase, when a party U sends a request (x, π) to verify a proof, $\mathcal{F}_{\mathsf{NIZK}}$ checks if the pair (x, π) has already been stored. If so, it outputs a verification success. If not, it forwards the request to the adversary $\mathcal{A}$ and waits for a response in the form of a witness w. If $\mathcal{R}(x, w) = 1$, the proof is considered valid, and $\mathcal{F}_{\mathsf{NIZK}}$ stores the pair (x, π) and outputs verification success. This ensures that proofs can only be verified if valid witnesses exist, while maintaining zero-knowledge by hiding the witness during proof generation.

Functionality $\mathcal{F}_{\mathsf{NIZK}}$

$\mathcal{F}_{\mathsf{NIZK}}$ is parameterized by a relation $\mathcal{R}$.

- **Proof generation:** On receiving $(\texttt{Prove}, \mathsf{sid}, x, w)$ from some party P, ignore if $\mathcal{R}(x, w) = 0$. Else, send $(\texttt{Prove}, \mathsf{sid}, x)$ to $\mathcal{A}$. Upon receiving $(\texttt{Proof}, \mathsf{sid}, \pi)$ from $\mathcal{A}$, store (x, π) and send $(\texttt{Proof}, \mathsf{sid}, \pi)$ to P.
- **Proof verification:** Upon receiving $(\texttt{Verify}, \mathsf{sid}, x, \pi)$ from some party P, check whether (x, π) is stored. If not send $(\texttt{Verify}, \mathsf{sid}, x, \pi)$ to $\mathcal{A}$. Upon receiving the answer $(\texttt{Witness}, \mathsf{sid}, w)$ from $\mathcal{A}$, check $\mathcal{R}(x, w) = 1$ and if so, store (x, π). If (x, π) has been stored, output $(\texttt{Vrfed}, \mathsf{sid}, 1)$ to P, else output $(\texttt{Vrfed}, \mathsf{sid}, 0)$.

A concept closely related to NIZK is the notion of *signatures of knowledge* (SoK). SoK combines digital signatures with zero-knowledge proofs and allows a signer to authenticate a message not only by proving possession of a secret key but also by demonstrating knowledge of a witness w for an NP statement $x \in L$. Such a signature ensures that, upon verification, it is confirmed that the signer possesses a valid witness for the truth of the statement. The concept of SoK was formally defined by Chase et al. [11] via an ideal functionality $\mathcal{F}_{\mathsf{SoK}}$. Similar to $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{SoK}}$ is for $(x, w) \in \mathcal{R}$. It consists of three primary phases: *setup*, *signature generation*, and *signature verification*. Unlike $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{SoK}}$ receives $(\mathtt{Verify}, \mathtt{Sign}, \mathsf{SimSign}, \mathsf{Extract})$ from the adversary $\mathcal{A}$. Upon receiving a request $(\mathtt{Sign}, \mathsf{sid}, m, x, w)$ from a party P, $\mathcal{F}_{\mathsf{SoK}}$ checks whether $(x, w) \in \mathcal{R}$. If the check passes, $\mathcal{F}_{\mathsf{SoK}}$ computes $\sigma \leftarrow \mathsf{SimSign}(m, x)$. Unlike $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{SoK}}$ does not leak the statement x to the adversary. To avoid leaking the statement to the adversary in the proof generation of proof of balance, we utilize $\mathcal{F}_{\mathsf{SoK}}$, which inherently prevents such leakage. Although $\mathcal{F}_{\mathsf{SoK}}$ introduces a message m as part of the signature, this component is unnecessary in our setting. Therefore, we simplify $\mathcal{F}_{\mathsf{SoK}}$ by fixing m to a predefined dummy message for all sessions. This transforms $\mathcal{F}_{\mathsf{SoK}}$ into a form that functions as $\mathcal{F}_{\mathsf{NIZK}}$ without the undesired statement leakage. Therefore, in the following we present the formal definition of $\mathcal{F}_{\mathsf{SoK}}$ from [11] under a simplifying assumption: the message m in all interfaces and algorithms is a predefined (dummy) message, hence, we avoid addressing it. Without loss of generality, we denote the following functionality as $\mathcal{F}^{\dagger}_{\mathsf{NIZK}}$.

Functionality $\mathcal{F}^{\dagger}_{\mathsf{NIZK}}$

The functionality is parameterized by a relation $\mathcal{R}$. Prove, $\mathsf{SimProve}$, and $\mathsf{Extract}$ are descriptions of PPT TMs, and Verify is a description of a deterministic polytime TM.

- **Setup:** Upon receiving $(\mathtt{Setup}, \mathsf{sid})$ from some party P, if this is the first time that $(\mathtt{Setup}, \mathsf{sid})$ is received, send $(\mathtt{Setup}, \mathsf{sid})$ to $\mathcal{A}$. Upon receiving $(\mathtt{Algorithms}, \mathsf{sid}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{SimProve}, \mathsf{Extract})$ from $\mathcal{A}$, store these algorithms. Output $(\mathtt{Algorithms}, \mathsf{sid}, \mathsf{Prove}, \mathsf{Verify})$ to P.
- **Proof generation:** Upon receiving $(\mathtt{Prove}, \mathsf{sid}, x, w)$ from P, if $(x, w) \notin \mathcal{R}$, ignore. Else, compute $\pi \leftarrow \mathsf{SimProve}(x)$, and check that $\mathsf{Verify}(x, \pi) = 1$. If so, output $(\mathtt{Proof}, \mathsf{sid}, \pi)$ to P and record the entry (x, π). Else, output an error message (*Completeness error*) to P and halt.
- **Proof verification:** Upon receiving $(\mathtt{Verify}, \mathsf{sid}, x, \pi)$ from P, check whether (x, π) is stored. If stored, output $(\mathtt{Vrfed}, \mathsf{sid}, 1)$ to P. Else, let $w \leftarrow \mathsf{Extract}(x, \pi)$. If $(x, w) \in \mathcal{R}$, output $(\mathtt{Vrfed}, \mathsf{sid}, 1)$ to P. Else, if $\mathsf{Verify}(x, \pi) = 0$, output $(\mathtt{Vrfed}, \mathsf{sid}, 0)$ to P. Otherwise, output an error message (*Verification error*) to P and halt.

4.3 Ledger functionality

The ledger functionality $\mathcal{F}_{\mathsf{Ledger}}$ abstracts a public ledger and provides a global access to the ledger to all parties, parameterized by a Validate predicate (to check transaction validity), an Update function (to update the ledger state), and an initial state state (which is empty at the start). The functionality supports two main operations: *read* and *append*. In the *read* operation, a party U requests the current state of the ledger. Upon receiving this request, $\mathcal{F}_{\mathsf{Ledger}}$ informs the adversary $\mathcal{A}$ that a request for read is submitted. If the adversary permits the read, the current ledger state is returned to the requester. Otherwise, the request is ignored, ensuring that reads occur only when permitted by $\mathcal{A}$.

In the *append* operation, a party submits a transaction tnx to be added to the ledger. The functionality forwards this tnx to the adversary. If the adversary approves, the Validate function checks whether the transaction tnx is valid according to the current state state. If valid, the Update function updates the state by appending tnx to the ledger. Otherwise, the transaction is discarded. This model captures both transparency and adversarial control typical in blockchain systems, where reads and appends are subject to validation rules and external influence while ensuring consistent state updates. Depending on the type of transaction tnx, the functionality $\mathcal{F}_{\mathsf{Ledger}}$ invokes a sub-Validate and sub-Update algorithms tailored to different transaction types described in Section 6.

Functionality $\mathcal{F}_{\mathsf{Ledger}}$

The functionality is globally available to all parties and is parameterized by a predicate Validate, and Update function and a variable state where initially $\mathsf{state} \leftarrow \emptyset$.

- **Read:** Upon receiving $(\mathtt{Read}, \mathsf{sid})$ from a party P, generate a fresh γ and set $L(\gamma) \leftarrow \mathsf{P}$. Send $(\mathtt{Read}, \mathsf{sid}, \gamma)$ to $\mathcal{A}$. Upon receiving $(\mathtt{Read.ok}, \mathsf{sid}, \gamma)$ from $\mathcal{A}$, ignore if $L(\gamma) = \perp$. Else, return $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$ to P where $L(\gamma) = \mathsf{P}$. Set $L(\gamma) \leftarrow \perp$.
- **Append:** Upon receiving $(\mathtt{Append}, \mathsf{sid}, \mathsf{tnx})$ from a party P, send $(\mathtt{Append.req}, \mathsf{sid}, \mathsf{tnx})$ to $\mathcal{A}$. Upon receiving $(\mathtt{Append.req.Ok}, \mathsf{sid}, \mathsf{tnx})$ from $\mathcal{A}$, invoke $\textsc{Validate}(\mathsf{state}, \mathsf{tnx})$, and if it outputs 1, update state via calling $\mathsf{state} \leftarrow \textsc{Update}(\mathsf{state}, \mathsf{tnx})$. Otherwise, ignore.

4.4 Communication channel functionality

Despite robust cryptographic protections at the protocol level, "network leakage" can still reveal information about the communicating parties. Therefore, in our UC framework, maintaining a degree of sender anonymity is essential to preserve (UC) anonymity. This issue is not unique to Π_{DART} but arises in any

UC-based formalization of privacy-preserving mechanisms. For example, Zerocash [40] would provide no anonymity without a sender-anonymous channel. Practical deployments of Π_{DART} could use a VPN, the Tor network, temporary network identities (e.g., public terminals), or JAP [21].

The functionality $\mathcal{F}_{\mathsf{ch}}$ defines a secure and anonymous communication channel between a sender P^{s} and a receiver P^{r}. When the sender P^{s} inputs a message m along with the receiver's identifier P^{r}, $\mathcal{F}_{\mathsf{ch}}$ records the sender and receiver identifiers, and message in an internal mapping associated with a randomly generated message ID, mid. It notifies the adversary $\mathcal{A}$ of mid and the message length. When the adversary approves, $\mathcal{F}_{\mathsf{ch}}$ retrieves the corresponding message and delivers it to the receiver, indicating the sender's identifier and the full message contents.

Functionality $\mathcal{F}_{\mathsf{ch}}$

Let P^{s} be a sender and P^{r} be a receiver. The message is denoted by m.

- Upon input $(\texttt{Send}, \mathsf{sid}, \mathsf{P}^{\mathsf{r}}, m)$ from P^{s}, record a mapping $\mathsf{P}(\mathsf{mid}) \leftarrow (\mathsf{P}^{\mathsf{s}}, \mathsf{P}^{\mathsf{r}}, m)$ where mid is chosen at random. Output $(\texttt{Send}, \mathsf{sid}, \mathsf{mid}, |m|)$ to $\mathcal{A}$.
- Upon receiving $(\texttt{Ok}, \mathsf{sid}, \mathsf{mid})$ from $\mathcal{A}$, retrieve $\mathsf{P}(\mathsf{mid}) = (\mathsf{P}^{\mathsf{s}}, \mathsf{P}^{\mathsf{r}}, m)$ and send $(\texttt{Received}, \mathsf{sid}, \mathsf{P}^{\mathsf{s}}, m)$ to P^{r} via private-delayed output.

5 Cryptographic schemes

Remark 2. This section is identical to the cryptographic schemes section of the DART paper [38].

5.1 Public key encryption (PKE) schemes

Definition 1 (Public key encryption (PKE) scheme). *Let* $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *denote a public key encryption (PKE) scheme. A PKE scheme is defined by the following components:*

- $\mathsf{PKE.KeyGen}$ ***(Key generation):*** *This algorithm takes a security parameter* λ *as input and outputs a pair* $(\mathsf{pk}, \mathsf{sk})$*, where* pk *is the public key and* sk *is the secret (private) key. The public key* pk *also (implicitly) defines a corresponding message space* $\mathsf{MessageSpace}(\mathsf{pk})$ *which specifies the set of valid plaintexts for encryption.*
- $\mathsf{PKE.Enc}$ ***(Encryption):*** *This algorithm takes the public key* pk *and a plaintext message* m *(where* $m \in \mathsf{MessageSpace}(\mathsf{pk})$*) as input and produces a ciphertext* C*.*

- $\mathsf{PKE.Dec}$ ***(Decryption):*** *The decryption algorithm takes the private key* sk *and a ciphertext* C *as input and outputs either the original plaintext message* m *or a special failure symbol* $\perp$ *if decryption fails.*

Definition 2 (PKE correctness). *A public key encryption (PKE) scheme* $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *is said to be correct if, for all* $\lambda \in \mathbb{N}$*, for all key pairs* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$*, and for all messages* $m \in \mathsf{MessageSpace}(\mathsf{pk})$*,* $\mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{PKE.Enc}(\mathsf{pk}, m)) = m$ *holds with overwhelming probability.*

Definition 3 (PKE CPA security). *A public key encryption (PKE) scheme* $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ *is* $\mathsf{IND\text{-}CPA}$*-secure if, for all PPT adversaries* $\mathcal{A}$*, the advantage* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}(\lambda)$ *in the following experiment is negligible in* λ*:* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}(\lambda) \leq \mathsf{negl}(\lambda)$.

The experiment, denoted $\mathsf{IND\text{-}CPA}^b_{\mathsf{PKE}}(\mathcal{A}, \lambda)$*, is defined between a probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$ *and a challenger, with a security parameter* λ *and a bit* $b \in \{0, 1\}$*:*

1. *The challenger generates a key pair* $(\mathsf{pk}, \mathsf{sk})$ *by running* $\mathsf{PKE.KeyGen}(1^\lambda)$ *and sends the public key* pk *to the adversary* $\mathcal{A}$*.*
2. *The adversary* $\mathcal{A}$ *submits two plaintext messages* (m_0, m_1)*, ensuring that* $|m_0| = |m_1|$ *and* $m_0, m_1 \in \mathsf{MessageSpace}(\mathsf{pk})$*.*
3. *The challenger selects a random bit* $b \xleftarrow{\$} \{0, 1\}$*, chooses one of the two plaintexts based on* b*, and computes the ciphertext* $C_b = \mathsf{PKE.Enc}(\mathsf{pk}, m_b)$*.*
4. *The challenger sends the ciphertext* C_b *to the adversary* $\mathcal{A}$*.*
5. *The adversary* $\mathcal{A}$ *outputs a bit* b' *as its guess for* b*. If* $\mathcal{A}$ *does not output any value, the guess* b' *is set to 0 by default.*

The adversary's advantage $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}(\lambda)$ *in this experiment is defined as:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{IND\text{-}CPA}}(\lambda) = \big| \Pr[\mathsf{IND\text{-}CPA}^1_{\mathsf{PKE}}(\mathcal{A}, \lambda) = 1] - \Pr[\mathsf{IND\text{-}CPA}^0_{\mathsf{PKE}}(\mathcal{A}, \lambda) = 1] \big|.$$

Definition 4 (PKE key privacy). *Key privacy ensures that the public key used to generate a ciphertext remains hidden [3]. The* $\mathsf{KeyPriv}$ *security experiment is conducted between a probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$ *and a challenger. A public key encryption scheme* PKE *is* $\mathsf{KeyPriv}$*-secure if, for all PPT adversaries* $\mathcal{A}$*, the advantage* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{KeyPriv}}(\lambda)$ *is negligible in the security parameter* λ*:* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{KeyPriv}}(\lambda) \leq \mathsf{negl}(\lambda)$.

This experiment, denoted $\mathsf{KeyPriv}^b_{\mathsf{PKE}}(\mathcal{A}, \lambda)$*, is defined with a bit* $b \in \{0, 1\}$ *and a security parameter* λ*:*

1. *The adversary is given access to two encryption-decryption oracles associated with different public keys.*
2. *For* $k \in \{0, 1\}$*:* $(\mathsf{pk}_k, \mathsf{sk}_k) \xleftarrow{\$} \mathsf{PKE.KeyGen}(1^\lambda)$*, where* pk_k *and* sk_k *are the public and secret keys for the* k*-th oracle, respectively.*
3. *The adversary submits a message* m*, where:* $m \leftarrow \mathcal{A}^{O_{\mathsf{sk}_0, \mathsf{pk}_0}, O_{\mathsf{sk}_1, \mathsf{pk}_1}}(\mathsf{pk}_0, \mathsf{pk}_1)$.

4. *The challenger selects a random bit* $b \xleftarrow{\$} \{0,1\}$.
5. *A ciphertext* C *is generated by encrypting* m *under the public key* pk_b*:* $C \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_b, m)$.
6. *The adversary receives* C *and outputs a bit* b' *as its guess for* b*:* $b' \leftarrow \mathcal{A}^{O_{\mathsf{sk}_0,\mathsf{pk}_0}, O_{\mathsf{sk}_1,\mathsf{pk}_1}}(\mathsf{pk}_0, \mathsf{pk}_1, C)$.
7. *The adversary wins if* $b = b'$*. The oracles reject decryption queries for the challenge ciphertext* C*.*

The adversary's advantage $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{KeyPriv}}(\lambda)$ *in this experiment is defined as:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{KeyPriv}}(\lambda) = \left|\Pr[\mathsf{KeyPriv}_{\mathsf{PKE}}^{1}(\mathcal{A}, \lambda) = 1] - \Pr[\mathsf{KeyPriv}_{\mathsf{PKE}}^{0}(\mathcal{A}, \lambda) = 1]\right|.$$

Definition 5 (ElGamal encryption scheme). *The ElGamal encryption scheme [15], denoted* EG*, is a public-key encryption (PKE) scheme that operates over a prime-order cyclic group* $\mathbb{G}_q$ *with generator* g*. The scheme consists of three main algorithms* $\mathsf{EG} = (\mathsf{EG.KeyGen}, \mathsf{EG.Enc}, \mathsf{EG.Dec})$ *defined as follows:*

- $\mathsf{EG.KeyGen}$ ***(Key generation):*** *Let* $\mathcal{G}$ *be a prime-order group generator that outputs* (q, g)*, where* q *is a prime and* g *is a generator of the cyclic group* $\mathbb{G}_q$*. The key generation algorithm proceeds as follows:*
  - *Take the security parameter* 1^λ *as input.*
  - *Sample the secret key* $\mathsf{sk} = x \xleftarrow{\$} \mathbb{Z}_q$ *uniformly at random.*
  - *Compute the public key component* $\mathsf{P} = g^x$*.*
  - *The public key is* $\mathsf{pk} = (q, g, \mathsf{P})$*, and the secret key is* $\mathsf{sk} = x$*.*
- $\mathsf{EG.Enc}$ ***(Encryption):*** *To encrypt a message* $M \in \mathbb{G}_q$*, the encryption process is as follows:*
  - *Sample a random value* $r \xleftarrow{\$} \mathbb{Z}_q$*.*
  - *Compute:* $\mathsf{C}_1 = g^r, \quad \mathsf{B} = \mathsf{P}^r$*.*
  - *Compute the ciphertext components:* $\psi = (\mathsf{C}_1, \mathsf{C}_2), \quad \text{where } \mathsf{C}_2 = \mathsf{B} \cdot M$*.*
- $\mathsf{EG.Dec}$ ***(Decryption):*** *Given a ciphertext* $\psi = (\mathsf{C}_1, \mathsf{C}_2)$*, the decryption process is as follows:*
  - *Compute:* $\mathsf{B} = \mathsf{C}_1^x$
  - *Recover the original message* M *by computing:* $M = \mathsf{C}_2 \cdot \mathsf{B}^{-1}$*.*

The security of the ElGamal encryption scheme under chosen-plaintext attack relies on the hardness of the decisional Diffie-Hellman (DDH) problem in the group $\mathbb{G}_q$*.*

5.2 Commitment schemes

Definition 6 (Commitment scheme). *Let* $\mathsf{COM} = (\mathsf{COM.KeyGen}, \mathsf{COM}, \mathsf{Com.Verify})$ *denote a commitment scheme. A commitment scheme is defined by the following components:*

- $\mathsf{COM.KeyGen}$ ***(Key generation):*** *This probabilistic algorithm takes a security parameter* λ *as input and outputs a pair of keys* $(\mathsf{ck}, \mathsf{vk})$*, where* ck *is the commitment key used by the committer and* vk *is the verification key used by the verifier. In the case of a publicly verifiable scheme,* $\mathsf{ck} = \mathsf{vk}$*.*

– COM ***(Commitment phase):*** *This probabilistic algorithm takes the commitment key* ck *and a message* m *(where* $m \in \mathsf{MessageSpace}(\mathsf{ck})$*) as input. It outputs a commitment value* c *and an opening value* d*:* $(c, d) \leftarrow \mathsf{COM}(\mathsf{ck}, m)$*. The pair* (c, d) *allows the committer to convince the verifier that the committed message is* m *during the verification phase.*
– $\mathsf{Com.Verify}$ ***(Verification phase):*** *This deterministic algorithm takes the verification key* vk*, a message* m*, and an opening value* d *as input. It outputs* accept *if* (c, d) *correctly reconstructs the message* m *and satisfies the commitment's consistency conditions; otherwise, it outputs* reject*. More formally:* $\mathsf{Com.Verify}(\mathsf{vk}, c, m, d) = \begin{cases} \mathsf{accept} & \textit{if } (c,d) \textit{ is a valid opening for } m, \\ \mathsf{reject} & \textit{otherwise.} \end{cases}$

Definition 7 (Commitment correctness). *The correctness property ensures that, assuming honest behavior by both the committer and the verifier, a valid commitment for any (valid) message* m *is always accepted by the verifier with the probability 1.*

Definition 8 (Commitment hiding). *Let* $\mathsf{COM} = (\mathsf{COM.KeyGen}, \mathsf{COM}, \mathsf{Com.Verify})$ *denote a (publicly verifiable) commitment scheme.* ***Hiding*** *security, for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$*, is defined as follows. The hiding experiment* $\mathsf{Hide}^b_{\mathsf{COM}}(\mathcal{A}, \lambda)$ *between a PPT adversary* $\mathcal{A}$ *and a challenger proceeds as follows:*

1. *The challenger generates a key* $\mathsf{ck} = \mathsf{vk}$ *by running* $\mathsf{COM.KeyGen}(1^\lambda)$ *and sends the key* ck *to the adversary* $\mathcal{A}$*.*
2. *The adversary* $\mathcal{A}$ *submits two messages* (m_0, m_1)*, ensuring that* $|m_0| = |m_1|$ *and* $m_0, m_1 \in \mathsf{MessageSpace}(\mathsf{ck})$*.*
3. *The challenger selects a random bit* $b \xleftarrow{\$} \{0, 1\}$*, computes the commitment* $(c, d) \leftarrow \mathsf{COM}(\mathsf{ck}, m_b)$*, and sends the commitment* c *to the adversary* $\mathcal{A}$*.*
4. *The adversary* $\mathcal{A}$ *outputs a bit* b' *as its guess for* b*.*

The adversary's advantage $\mathsf{Adv}^{\mathsf{Hide}}_{\mathcal{A}}(\lambda)$ *in this experiment is defined as:*

$$\mathsf{Adv}^{\mathsf{Hide}}_{\mathcal{A}}(\lambda) = \left| \Pr[\mathsf{Hide}^1_{\mathsf{COM}}(\mathcal{A}, \lambda) = 1] - \Pr[\mathsf{Hide}^0_{\mathsf{COM}}(\mathcal{A}, \lambda) = 1] \right|.$$

A commitment scheme satisfies the ***hiding security*** *property if:*

– ***Perfect hiding:*** *For all adversaries* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathsf{Hide}}_{\mathcal{A}}(\lambda) = 0$
– ***Computational hiding:*** *For all PPT adversaries* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathsf{Hide}}_{\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$

Intuitively, this guarantees that no adversary can distinguish the commitment of m_0 *from* m_1 *with probability significantly better than a random guess. Hence, the commitment hides the underlying message.*

Definition 9 (Commitment binding). *Let* $\mathsf{COM} = (\mathsf{COM.KeyGen}, \mathsf{COM}, \mathsf{Com.Verify})$ *denote a (publicly verifiable) commitment scheme. The* ***binding*** *security for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$ *is defined as follows. The binding experiment* $\mathsf{Bind}_{\mathsf{COM}}(\mathcal{A}, \lambda)$ *between a PPT adversary* $\mathcal{A}$ *and a challenger proceeds as follows:*

1. *The challenger generates a key* $\mathsf{ck} = \mathsf{vk}$ *by running* $\mathsf{COM.KeyGen}(1^\lambda)$ *and sends the key* ck *to the adversary* $\mathcal{A}$.
2. *The adversary* $\mathcal{A}$ *outputs a tuple* (c, m, d, m', d')*, where* $m, m' \in \mathsf{MessageSpace}(\mathsf{ck})$.
3. *The challenger checks:*

$$\mathsf{Com.Verify}(\mathsf{vk}, m, c, d) = 1 \quad \textit{and} \quad \mathsf{Com.Verify}(\mathsf{vk}, m', c, d') = 1.$$

4. *If both verifications succeed and* $m \neq m'$*, the adversary* $\mathcal{A}$ *wins the binding game. Otherwise, the adversary fails.*

The adversary's advantage $\mathsf{Adv}^{\mathsf{Bind}}_{\mathcal{A}}(\lambda)$ *in this experiment is defined as:*

$$\mathsf{Adv}^{\mathsf{Bind}}_{\mathcal{A}}(\lambda) = \Pr[\mathsf{Bind}_{\mathsf{COM}}(\mathcal{A}, \lambda) = 1].$$

A commitment scheme satisfies the ***binding security*** *property if:*

- ***Perfect binding:*** *For all adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{Bind}}_{\mathcal{A}}(\lambda) = 0$.
- ***Computational binding:*** *For all PPT adversaries* $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{Bind}}_{\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$.

Intuitively, this guarantees that no adversary can produce a valid commitment that can be opened to two different messages with probability significantly better than negligible. Hence, the commitment binds the committer to a single value.

Definition 10 (Pedersen commitment scheme). *Let* $\mathsf{PCom} = (\mathsf{Com.KeyGen}, \mathsf{Com}, \mathsf{Com.Verify})$ *denote the Pedersen commitment scheme [37]. The Pedersen commitment scheme is defined by the following components:*

- $\mathsf{Com.KeyGen}$ ***(Key generation):*** *This probabilistic algorithm takes a security parameter* λ *as input and outputs the cyclic group parameters* (G, q, g)*, where* G *is a cyclic group of prime order* q*, and* g *is a generator of* G*. Additionally, it selects a random group element* $h \xleftarrow{\$} G$ *and outputs* $\mathsf{ck} = \mathsf{vk} = (G, q, g, h)$.
- Com ***(Commitment):*** *This probabilistic algorithm takes the commitment key* $\mathsf{ck} = (G, q, g, h)$ *and a message* $m \in \mathbb{Z}_q$ *as input. It generates a random opening value* $d \xleftarrow{\$} \mathbb{Z}_q$ *and computes the commitment value* $c = g^d \cdot h^m$*. The output is the pair* (c, d)*:* $(c, d) \leftarrow \mathsf{Com}(\mathsf{ck}, m)$.
- $\mathsf{Com.Verify}$ ***(Verification):*** *This deterministic algorithm takes the verification key* $\mathsf{vk} = (G, q, g, h)$*, a message* $m \in \mathbb{Z}_q$*, an opening value* $d \in \mathbb{Z}_q$*, and a commitment value* $c \in G$ *as input. The algorithm outputs* accept *if the equality* $g^d \cdot h^m = c$ *holds, indicating a valid commitment to* m*, and* reject *otherwise:* $\mathsf{Com.Verify}(\mathsf{vk}, c, m, d) = \begin{cases} \mathsf{accept} & \textit{if } g^d h^m = c, \\ \mathsf{reject} & \textit{otherwise.} \end{cases}$

The Pedersen commitment scheme PCom *is perfectly hiding and computationally binding [28, 29].*

5.3 Pseudorandom function (PRF) schemes

Definition 11 (Pseudorandom function (PRF) scheme). *Let* $\mathsf{PRF} = (\mathsf{F.KeyGen}, \mathsf{F.Eval})$ *denote a pseudorandom function (PRF) scheme. A PRF scheme is defined by the following components:*

- $\mathsf{F.KeyGen}$ ***(Key generation):*** *This algorithm takes a security parameter* λ *as input and outputs a secret key* $k \in K$*, where* K *denotes the key space.*
- $\mathsf{F.Eval}$ ***(Function evaluation):*** *This algorithm takes the secret key* k *and an input* $x \in X$*, where* X *is the domain, and outputs a value* $y \in Y$*, where* Y *is the range. Formally, the function is defined as* $\mathsf{PRF}_k : X \to Y$*, where* $\mathsf{PRF}_k(x) = y$*.*

Definition 12 (PRF pseudorandomness). *A pseudorandom function (PRF) scheme* $\mathsf{PRF} = (\mathsf{F.KeyGen}, \mathsf{F.Eval})$ *is pseudorandom if, for all probabilistic polynomial-time (PPT) adversaries* $\mathcal{A}$*, the advantage* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF.Pseudo}}(\lambda)$ *in the following experiment is negligible in* λ*:* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF.Pseudo}}(\lambda) \leq \mathsf{negl}(\lambda)$*.*

The security experiment, denoted $\mathsf{Pseudo}_{\mathsf{PRF}}^{b}(\mathcal{A}, \lambda)$ *for* $b \in \{0,1\}$*, is defined between a probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$ *and a challenger as follows:*

1. *The challenger takes the security parameter* λ *and generates a secret key* $k \in K$ *by running* $\mathsf{F.KeyGen}(1^\lambda)$*.*
2. *For* $b = 0$*: The challenger defines the function* $\mathsf{PRF}_k : X \to Y$ *as* $\mathsf{PRF}_k(x) = \mathsf{F.Eval}(k, x)$*, where* X *is the domain and* Y *is the range.*
3. *For* $b = 1$*: The challenger chooses a truly random function* $f : X \to Y$ *uniformly at random.*
4. *The challenger selects a random bit* $b \xleftarrow{\$} \{0,1\}$*.*
5. *The adversary* $\mathcal{A}$ *submits* q *queries* $x_1, \ldots, x_q \in X$*. For each query* x_i*, the challenger responds with* $f(x_i)$ *(if* $b = 1$*) or* $\mathsf{PRF}_k(x_i)$ *(if* $b = 0$*).*
6. *The adversary* $\mathcal{A}$ *outputs a bit* $b' \in \{0,1\}$*, indicating its guess of whether the function is pseudorandom (*$b = 0$*) or truly random (*$b = 1$*).*

The adversary's advantage $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF.Pseudo}}(\lambda)$ *is defined as:*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{PRF.Pseudo}}(\lambda) = \left| \Pr[\mathsf{Pseudo}_{\mathsf{PRF}}^{1}(\mathcal{A}, \lambda) = 1] - \Pr[\mathsf{Pseudo}_{\mathsf{PRF}}^{0}(\mathcal{A}, \lambda) = 1] \right|,$$

5.4 Accumulator schemes

An accumulator scheme enables the compact representation of an arbitrarily large set S of elements. Given a membership witness for an accumulated element x and the current accumulator value $\mathsf{v}_{\mathsf{ACCU}}$, it is possible to efficiently verify whether x is a valid member of the accumulated set. Importantly, the accumulated set can be dynamically updated as elements are added.

Definition 13 (Accumulator scheme). *Let* $\mathsf{ACCU} = (\mathsf{ACCU.Setup}, \mathsf{Accu}, \mathsf{Add}, \mathsf{ACCU.PrvMem}, \mathsf{ACCU.VfyMem})$ *denote an accumulator scheme over the universe* $\mathcal{U}$*. An accumulator scheme is defined by the following components:*

– $\mathsf{ACCU.Setup}$ ***(Setup):*** *This algorithm takes the security parameter λ as input and outputs the public parameters $\mathsf{pp_{ACCU}}$: $\mathsf{pp_{ACCU}} \leftarrow \mathsf{ACCU.Setup}(1^\lambda)$.*
– Accu ***(Accumulate):*** *This deterministic algorithm takes the public parameters $\mathsf{pp_{ACCU}}$ and a set $S \subseteq \mathcal{U}$ as input and computes an accumulator value $\mathsf{v_{ACCU}}$ for the set S: $\mathsf{v_{ACCU}} \leftarrow \mathsf{Accu}(\mathsf{pp_{ACCU}}, S)$.*
– Add ***(Add element):*** *This algorithm takes the public parameters $\mathsf{pp_{ACCU}}$, an accumulator value $\mathsf{v_{ACCU}}$, and an element $x \in \mathcal{U}$ as input and outputs an updated accumulator value $\mathsf{v'_{ACCU}}$: $\mathsf{v'_{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, x)$.*
– $\mathsf{ACCU.PrvMem}$ ***(Prove membership):*** *This algorithm takes the public parameters $\mathsf{pp_{ACCU}}$, a set S, and an element x as input and outputs a membership proof π_ACCU showing that x is in the accumulated set S: $\pi_\mathsf{ACCU} \leftarrow \mathsf{ACCU.PrvMem}(\mathsf{pp_{ACCU}}, S, x)$.*
– $\mathsf{ACCU.VfyMem}$ ***(Verify membership):*** *This algorithm takes the public parameters $\mathsf{pp_{ACCU}}$, an accumulator value $\mathsf{v_{ACCU}}$, an element x, and a membership proof π_ACCU as input and outputs a bit $\{0,1\}$ indicating whether x belongs to the set accumulated in $\mathsf{v_{ACCU}}$: $\{0,1\} \leftarrow \mathsf{ACCU.VfyMem}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, x, \pi_\mathsf{ACCU})$.*

Definition 14 (Accumulator correctness). *Let $\mathsf{ACCU} = (\mathsf{ACCU.Setup}, \mathsf{Accu}, \mathsf{Add}, \mathsf{ACCU.PrvMem}, \mathsf{ACCU.VfyMem})$ denote an accumulator scheme over the universe $\mathcal{U}$. An accumulator scheme is said to be correct if the following properties hold, for all security parameters $\lambda \in \mathbb{N}$, all elements $x \in \mathcal{U}$, and all sets $S \subseteq \mathcal{U}$:*

– *let $\mathsf{pp_{ACCU}} \leftarrow \mathsf{ACCU.Setup}(1^\lambda)$ and $\mathsf{v_{ACCU}} \leftarrow \mathsf{Accu}(\mathsf{pp_{ACCU}}, S)$. Then, for the updated accumulator $\mathsf{v'_{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, x)$, we have: $\mathsf{v'_{ACCU}} = \mathsf{Accu}(\mathsf{pp_{ACCU}}, S \cup \{x\})$.*
– *for $x \in S$, let $\mathsf{pp_{ACCU}} \leftarrow \mathsf{ACCU.Setup}(1^\lambda)$, $\mathsf{v_{ACCU}} \leftarrow \mathsf{Accu}(\mathsf{pp_{ACCU}}, S)$, and $\pi_\mathsf{ACCU} \leftarrow \mathsf{ACCU.PrvMem}(\mathsf{pp_{ACCU}}, S, x)$. Then the membership proof π_ACCU is valid, i.e., $\mathsf{ACCU.VfyMem}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, x, \pi_\mathsf{ACCU}) = 1$.*

Definition 15 (Accumulator soundness). *Soundness guarantees that no efficient adversary can choose a set S and generate a proof that verifies against $\mathsf{Accu}(\mathsf{pp_{ACCU}}, S)$ for an element $x \notin S$. More formally, an accumulator ACCU is said to be sound if the following property holds for all security parameters $\lambda \in \mathbb{N}$, and PPT adversaries $\mathcal{A}$:*

$$\Pr\left[\begin{matrix}\mathsf{pp_{ACCU}} \leftarrow \mathsf{ACCU.Setup}(1^\lambda) \\ (S, x, \pi_\mathsf{ACCU}) \leftarrow \mathcal{A}(\mathsf{pp_{ACCU}})\end{matrix} : \begin{matrix} x \notin S \ \wedge \ \mathsf{v_{ACCU}} \leftarrow \mathsf{Accu}(\mathsf{pp_{ACCU}}, S) \ \wedge \\ \mathsf{ACCU.VfyMem}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, x, \pi_\mathsf{ACCU}) = 1\end{matrix}\right] \leq \mathsf{negl}(\lambda)$$

5.5 Assumptions

Definition 16 (d-strong decisional Diffie-Hellman (d-sDDH) assumption). *Let $\mathbb{G}$ be a cyclic group of prime order p, where p is a prime, and let g be a generator of $\mathbb{G}$. Suppose $(\mathbb{G}, p, g) \leftarrow \mathcal{G}(1^\lambda)$, where $\mathcal{G}$ is a group generation algorithm parameterized by the security parameter λ. For $x, x_1, \ldots, x_d \xleftarrow{\$} \mathbb{Z}_p$, the*

d-strong decisional Diffie-Hellman (d-sDDH) assumption asserts that the following holds relative to $\mathcal{G}$. For any probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\lambda)$ *such that:*

$$\begin{aligned}\mathsf{Adv}_{\mathcal{A}}^{d\text{-}\mathsf{sDDH}}(\lambda) = \Big| &\Pr\left[\mathcal{A}(\mathbb{G}, p, g, g^{x}, g^{x^2}, \ldots, g^{x^d}) = 1\right] \\ &- \Pr\left[\mathcal{A}(\mathbb{G}, p, g, g^{x_1}, g^{x_2}, \ldots, g^{x_d}) = 1\right]\Big| \leq \mathsf{negl}(\lambda)\end{aligned}$$

6 P-DART

Reading Note. For completeness and ease of understanding, we recommend that readers first familiarize themselves with the DART protocol [38], as P-DART builds directly upon its design principles and extends it to the Polymesh setting. Reading DART beforehand will make it easier to follow the rationale and construction choices in P-DART. Nevertheless, this section is self-contained: all necessary concepts are revisited or summarized here, ensuring that readers can fully understand the P-DART design even without prior exposure to DART. At the core of P-DART, each user controls an account that evolves over time through transactions. When a payment occurs, the old account $\mathsf{acct}_{\mathsf{old}}$ transitions into a new one $\mathsf{acct}_{\mathsf{new}}$. To preserve anonymity, the old account is never revealed on the ledger; instead, only a commitment to the new account, $\mathsf{acct}_{\mathsf{new}}^{\mathsf{cm}}$, is published. A NIZK proof allows the user to demonstrate that the old account was a valid (and unspent) member of the blockchain state and that it is correctly transformed into the new account, without exposing any sensitive details. To prevent double spending, each account has a unique *nullifier* nul, derived from the user's secret randomness. When spending, the nullifier is revealed and proved to be well-formed with respect to the committed $\mathsf{acct}_{\mathsf{old}}^{\mathsf{cm}}$ (for which a membership proof is provided). The ledger records this nullifier as "spent" and simultaneously adds the commitment for the new account to the global state.

6.1 Building blocks

We incorporate several cryptographic primitives: public-key encryption schemes (Definition 1) with algorithms $(\mathsf{Enc}, \mathsf{Dec})$, commitment schemes (Definition 6), defined by algorithm Com, and an accumulator scheme (Definition 13) with algorithms $(\mathsf{Accu}, \mathsf{PrvMem}, \mathsf{VfyMem})$. We use the accumulator to compactly represent large sets (e.g., the set of account commitments). For a set S, Accu computes an accumulator value $\mathsf{v}_{\mathsf{ACCU}}$. Given an element x, a prover can run PrvMem to compute a membership proof π_{ACCU} showing that x is contained in S. Conversely, given $\mathsf{v}_{\mathsf{ACCU}}$, an element x, and its claimed membership proof π_{ACCU}, one can run VfyMem to check if x indeed belongs to the set represented by $\mathsf{v}_{\mathsf{ACCU}}$. See Section 5 for formal definitions and security properties.

6.2 Core components

Accounts. A user account is defined as a tuple $\mathsf{acct} := (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t}, \rho, \rho^{(n+2)}, s^{(2^n)})$, where $\mathsf{sk}_{\mathsf{acct}}$ is the account secret key, $\mathsf{f.b}$ is the finalized (spendable) balance, cnt is the pending transaction counter, $\mathsf{a.t}$ is the asset type, (ρ, s) are randomness values used, respectively, to derive the nullifier nul and to generate the hiding account commitment $\mathsf{acct}^{\mathsf{cm}}$, and n is transaction counter that is initially zero. We define a key-generation algorithm KeyGen tied to accounts; when we write $\mathsf{Acc.KeyGen}$ this refers to the account-specific key-generation procedure. The system is parameterized by a bound $\mathsf{v}_{\mathsf{max}}$ on transaction value.

Ledger. The ledger state is $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, where state' contains all submitted transactions, $\mathsf{v}_{\mathsf{ACCU}}$ is an accumulator value accumulating all the account commitments (that are part of submitted transactions) in state', and $\mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}$ are initially empty lists defined below.

- The *Issuer–Asset–Auditor* list $\mathbb{L}_{\mathsf{IAA}}$ is updated with a tuple $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$ whenever an issuer with public key $\mathsf{pk}_{\mathsf{acct}}$ issues an asset $\mathsf{a.t}$ with auditor $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$.
- The *Key–Asset* list $\mathbb{L}_{\mathsf{KA}}$ is updated with $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t})$ when a party registers an asset-specific account acct with public key $\mathsf{pk}_{\mathsf{acct}}$ and asset type $\mathsf{a.t}$. Each $\mathsf{pk}_{\mathsf{acct}}$ can register at most one account per asset type.
- The list $\mathbb{L}_{\mathsf{NUL}}$ stores all spent nullifiers.

We assume the ledger provides two operations: (i) verifying whether a submitted transaction is valid, and (ii) updating the state upon acceptance. The consensus protocol is left abstract, as it is not the focus of this work.

$\mathcal{F}_{\mathsf{Ledger}}$ maintains the state of the ledger $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$. state parses into all transactions submitted to the chain state', the most recent accumulator value $\mathsf{v}_{\mathsf{ACCU}}$, and *most recent* state of three lists $(\mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}})$. $\mathcal{F}_{\mathsf{Ledger}}$ is parameterized by two functions: (i) (0 or 1) $\leftarrow$ $\textsc{Validate}(\mathsf{state}, \mathsf{tnx})$, and (ii) $\mathsf{state}^{\mathsf{new}} \leftarrow \textsc{Update}(\mathsf{state}^{\mathsf{old}}, \mathsf{tnx})$, both taking as input blockchain state and a transaction tnx. Each of these functions includes several sub-algorithms tailored to different types of transactions as we will see later. Each consensus member (whether part of a permissioned or permissionless blockchain) reads the blockchain state state and verify the transaction by executing $\textsc{Validate}(\mathsf{state}, \mathsf{tnx})$. If the transaction is verified, state is updated accordingly by executing $\textsc{Update}(\mathsf{state}, \mathsf{tnx})$.

6.3 Algorithms

6.3.1 Address generation Each user calls $\mathcal{F}_{\mathsf{KeyReg}}$ to receive two pairs of cryptographic keys: one pair $(\mathsf{pk}_{\mathsf{en}}, \mathsf{sk}_{\mathsf{en}})$ for PKE and another pair $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}})$ for their account. The algorithm is provided in Figure 1.

▷ Upon receiving $(\mathsf{address}, \mathsf{sid})$ from $\mathcal{Z}$ call $\mathcal{F}_{\mathsf{KeyReg}}$ with $(\mathtt{Gen.Key}, \mathsf{sid})$.

▷ Upon receiving $(\texttt{Gen.Key}, \mathsf{sid}, \mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ from $\mathcal{F}_{\mathsf{KeyReg}}$, record $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$.
▷ Output $(\mathsf{address.gened}, \mathsf{sid}, \mathsf{addr}_{\mathsf{pk}})$ to $\mathcal{Z}$.

Fig. 1: Address generation

6.3.2 Asset issuance There is a public list called the Issuer-Asset-Auditor list, denoted by $\mathbb{L}_{\mathsf{IAA}}$, which is maintained by consensus (on-chain) and initially set to empty. When a party P issues an asset $\mathsf{a.t}$, $\mathbb{L}_{\mathsf{IAA}}$ is updated accordingly. $\mathbb{L}_{\mathsf{IAA}}$ contains tuples of the form $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$, where $\mathsf{pk}_{\mathsf{acct}}$ is issuer's account public key and $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$ is the auditor/mediator's public key encryption[12]. In our NIZK relations, we may need a concise representation of $\mathbb{L}_{\mathsf{IAA}}$, which only contains pairs of $(\mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$. We will therefore abuse the notation when describing proof relations so that for an index i, we write $(\mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}) = \mathbb{L}_{\mathsf{IAA}}[i]$ to indicate that $(\mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$ is the i-th entry in $\mathbb{L}_{\mathsf{IAA}}$, providing the association of $\mathsf{a.t}$ and $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$. In Section 7.1, we discuss how to address situations where it becomes necessary to revoke an auditor/mediator. The algorithm is provided in Figure 2. The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{issue}} = \mathsf{sk}_{\mathsf{acct}}$
- $x_{\mathsf{issue}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$
- $\mathcal{R}_{\mathsf{issue}} = \{\, (\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \,\}$.

$\mathsf{tnx}_{\mathsf{issue}}$ Generation

▷ Upon receiving $(\mathsf{issue}, \mathsf{sid}, \mathsf{a.t}, \mathsf{A})$ from $\mathcal{Z}$: ignore if $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ is not recorded.
▷ Else, call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\texttt{Read}, \mathsf{sid})$. Upon receiving $(\texttt{Read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, parse $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.
▷ Ignore if $(\cdot, \mathsf{a.t}, \cdot) \in \mathbb{L}_{\mathsf{IAA}}$.
▷ Else, parse $\mathsf{addr}_{\mathsf{pk}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$, and $\mathsf{addr}_{\mathsf{sk}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$.
▷ Call $\mathcal{F}_{\mathsf{KeyReg}}$ with $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{A})$ and upon receiving $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{A}, \mathsf{addr}^{\mathsf{A}}_{\mathsf{pk}})$ retrieve $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$ from $\mathsf{addr}^{\mathsf{A}}_{\mathsf{pk}}$.
▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with $(\texttt{Prove}, \mathsf{sid}, x_{\mathsf{issue}}, w_{\mathsf{issue}})$ and receive $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{issue}})$.
▷ Set $\mathsf{tnx}_{\mathsf{issue}} \leftarrow (x_{\mathsf{issue}}, \pi_{\mathsf{issue}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\texttt{Append}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{issue}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\texttt{Read}, \mathsf{sid})$; upon receiving back $(\texttt{Read}, \mathsf{sid}, \mathsf{state})$, if $\mathsf{tnx}_{\mathsf{issue}} \in \mathsf{state}'$ for $\mathsf{state}' \in \mathsf{state}$ output $(\mathsf{issued}, \mathsf{sid}, \mathsf{a.t})$ to $\mathcal{Z}$.

$\mathsf{tnx}_{\mathsf{issue}}$ Verification

▷ Execute $\textsc{Validate}_{\mathsf{issue}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{issue}})$:

[12] We assume a single auditor/mediator A here for simplicity; this can be straightforwardly extended to accommodate a group of auditors and mediators.

– parse $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{issue}} = (x_{\mathsf{issue}}, \pi_{\mathsf{issue}})$, and $x_{\mathsf{issue}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$.
– ignore if any of the following conditions hold:
  - $(\cdot, \mathsf{a.t}, \cdot) \in \mathbb{L}_{\mathsf{IAA}}$
  - upon calling $\mathcal{F}_{\mathsf{NIZK}}$ with $(\mathtt{Verify}, \mathsf{sid}, x_{\mathsf{issue}}, \pi_{\mathsf{issue}})$, $(\mathtt{Vrfed}, \mathsf{sid}, 0)$ is returned.
– else, call $\mathcal{F}_{\mathsf{KeyReg}}$ with $(\mathtt{Rtrv.id}, \mathsf{sid}, \mathsf{pk})$ for $\mathsf{pk} = \mathsf{pk}_{\mathsf{acct}}$, and $\mathsf{pk} = \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$.
– upon receiving $(\mathtt{Rtrved.id}, \mathsf{sid}, \mathsf{P})$ from $\mathcal{F}_{\mathsf{KeyReg}}$ for $\mathsf{P} = \mathsf{P}^*$ and $\mathsf{P} = \mathsf{A}$ output 1.

▷ Execute $\textsc{Update}_{\mathsf{issue}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{issue}})$:
– parse $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{issue}} = (x_{\mathsf{issue}}, \pi_{\mathsf{issue}})$, and $x_{\mathsf{issue}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$.
– set $\mathbb{L}_{\mathsf{IAA}} \leftarrow \mathbb{L}_{\mathsf{IAA}} \cup \{(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})\}$.
– set $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\cdot, \mathbb{L}_{\mathsf{IAA}}, \cdot, \cdot, \cdot)\}$.
– output state.

Fig. 2: Asset issuance transaction $\mathsf{tnx}_{\mathsf{issue}}$

6.3.3 Account registration

There is a public list called the Key-Asset list, denoted by $\mathbb{L}_{\mathsf{KA}}$, which is maintained by consensus. The list is initially set to empty and is updated when a party P registers an asset-specific account acct. The $\mathbb{L}_{\mathsf{KA}}$ list stores tuples of the form $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t})$, where each $\mathsf{pk}_{\mathsf{acct}}$ can register one account for a given $\mathsf{a.t}$. P proves that account commitment $\mathsf{acct}^{\mathsf{cm}}$ is well-formed, and that they possess knowledge of the associated secret key of $\mathsf{pk}_{\mathsf{acct}}$. The algorithm is provided in Figure 3. The NIZK witness, statement, and relation are defined as follows.

– $w_{\mathsf{register}} = (\mathsf{sk}_{\mathsf{acct}}, \rho, s, \text{randomness of } \Omega)$
– $x_{\mathsf{register}} = (\mathsf{acct}^{\mathsf{cm}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, N, \Omega, \mathsf{pk}_{\mathsf{T}})$
– The relation $\mathcal{R}_{\mathsf{register}}(x_{\mathsf{register}}, w_{\mathsf{register}})$ is defined as:

$$\mathcal{R}_{\mathsf{register}} = \Big\{ \mathsf{acct}^{\mathsf{cm}} = \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, 0, 0, \mathsf{a.t}, \rho, \rho^2; s) \\ \wedge\ \rho = f(\mathsf{sk}_{\mathsf{acct}}, \mathsf{a.t} \mid\mid \mathsf{counter}) \\ \wedge\ \Omega = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{T}}}(s) \\ \wedge\ N = g_5^{\rho} \\ \wedge\ (\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \Big\}.$$

$\mathsf{tnx}_{\mathsf{register}}$ Generation

▷ Upon receiving $(\mathsf{register}, \mathsf{sid}, \mathsf{a.t})$ from $\mathcal{Z}$ proceed as follows.
▷ Ignore if $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ is not recorded.
▷ Otherwise, parse: $\mathsf{addr}_{\mathsf{pk}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.

▷ Ignore if:
- $(\cdot, \mathsf{a.t}, \cdot) \notin \mathbb{L}_{\mathsf{IAA}}$; or
- $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}) \in \mathbb{L}_{\mathsf{KA}}$.

▷ Parse: $\mathsf{addr}_{\mathsf{sk}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$.

▷ Sample and compute:

$$s \xleftarrow{\$} \mathbb{Z}_q, \quad \rho \leftarrow f(\mathsf{sk}_{\mathsf{acct}}, \mathsf{a.t} \mid\mid \mathsf{counter}), \quad N \leftarrow g_5^{\rho}$$

// f can be a zk-friendly keyed hash function.

The purpose of including the counter $\mathsf{counter}$ in the ρ-generation procedure is to allow the user to derive a new ρ value while retaining the same secret key and asset type.

▷ Retrieve pk_{T} and compute $\Omega \leftarrow \mathsf{Enc}_{\mathsf{pk}_{\mathsf{T}}}(s)$.

// Ω is generated so that, if account freezing is later required, the holder of sk_{T} can decrypt Ω to recover s and thereby freeze the user's account. The entity holding sk_{T} is designated by the issuer, and the mapping between the issuer, asset type, and pk_{T} is recorded on-chain. We omit further details here.

▷ Set the account parameters: $\mathsf{acct} \leftarrow (\mathsf{sk}_{\mathsf{acct}}, 0, 0, \mathsf{a.t}, \rho, \rho^2, s)$.

▷ Compute the account commitment: $\mathsf{acct}^{\mathsf{cm}} \leftarrow \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, 0, 0, \mathsf{a.t}, \rho, \rho^2; s)$.

▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Prove}, \mathsf{sid}, x_{\mathsf{register}}, w_{\mathsf{register}})$. and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{register}})$.

▷ Set the registration transaction: $\mathsf{tnx}_{\mathsf{register}} \leftarrow (x_{\mathsf{register}}, \pi_{\mathsf{register}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Append}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{register}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$, if $\mathsf{tnx}_{\mathsf{register}} \in \mathsf{state}$, record: $(\mathsf{acct}^{\mathsf{cm}}, \mathsf{acct})$.

▷ Output: $(\mathsf{registered}, \mathsf{sid}, \mathsf{a.t})$ to $\mathcal{Z}$.

$\mathsf{tnx}_{\mathsf{register}}$ Verification

▷ Execute $\textsc{Validate}_{\mathsf{register}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{register}})$:
- Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{register}} = (x_{\mathsf{register}}, \pi_{\mathsf{register}})$, $x_{\mathsf{register}} = (\mathsf{acct}^{\mathsf{cm}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, N, \Omega, \mathsf{pk}_{\mathsf{T}})$.
- Ignore if any of the following conditions hold:
  - $(\cdot, \mathsf{a.t}, \cdot) \notin \mathbb{L}_{\mathsf{IAA}}$.
  - $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}) \in \mathbb{L}_{\mathsf{KA}}$.
  - $N \in \mathbb{L}_{\mathsf{NUL}}$.
  - If $\exists\, (\mathsf{pk}'_{\mathsf{acct}}, \mathsf{a.t}')$ with $\mathsf{Store}(\mathsf{pk}'_{\mathsf{acct}}, \mathsf{a.t}') = (N', \Omega') \wedge N = N'$.
  - Upon calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Verify}, \mathsf{sid}, x_{\mathsf{register}}, \pi_{\mathsf{register}})$, the output is: $(\mathtt{Vrfed}, \mathsf{sid}, 0)$.
  - Given $\mathsf{a.t}$, pk_{T} is not valid.
- Otherwise, call $\mathcal{F}_{\mathsf{KeyReg}}$ with: $(\mathtt{Rtrv.id}, \mathsf{sid}, \mathsf{pk}_{\mathsf{acct}})$. Upon receiving: $(\mathtt{Rtrved.id}, \mathsf{sid}, \mathsf{P})$, output 1.

▷ Execute $\textsc{Update}_{\mathsf{register}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{register}})$:
- Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{register}} = (x_{\mathsf{register}}, \pi_{\mathsf{register}})$, $x_{\mathsf{register}} = (\mathsf{acct}^{\mathsf{cm}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, N, \Omega, \mathsf{pk}_{\mathsf{T}})$.
- Call: $\mathsf{v}_{\mathsf{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm}})$.
- Update the key-asset list: $\mathbb{L}_{\mathsf{KA}} \leftarrow \mathbb{L}_{\mathsf{KA}} \cup \{(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t})\}$.
- Record $\mathsf{Store}(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}) = (N, \Omega)$ in state'.
- Update the nullifier set: $\mathbb{L}_{\mathsf{NUL}} \leftarrow \mathbb{L}_{\mathsf{NUL}} \cup \{N\}$.
- Update the state: $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\mathsf{state}', \cdot, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})\}$.
- Output state.

Fig. 3: Account registration transaction $\mathsf{tnx}_{\mathsf{register}}$

6.3.4 Increase Asset Supply The asset issuer can increase the supply of an asset they have previously issued. The ledger verifies the legitimacy of the issuer for the specified asset based on $\mathbb{L}_{\mathsf{IAA}}$. The issuer's finalized balance increases as: $\mathsf{f.b}_{\mathsf{new}} = \mathsf{f.b}_{\mathsf{old}} + v$. In this algorithm, no privacy-preserving operations are performed, and $\mathsf{pk}_{\mathsf{acct}}$ is publicly visible. However, $\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}$ remains private to prevent linking the issuer's previous transactions (e.g., a send transaction) to this one, thereby preserving transaction privacy.
The issuer publicly reveals their new account commitment $\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}$ along with the nullifier $\mathsf{nul}_{\mathsf{old}}$ of $\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}$, for which an accumulator membership proof is computed. The issuer also proves knowledge of $\mathsf{sk}_{\mathsf{acct}}$ while disclosing $\mathsf{a.t}$ and v. Additionally, the well-formedness of $\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}$ is verified. The algorithm is provided in Figure 4.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{increase}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}}, \rho, s, n, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \pi_{\mathsf{ACCU}})$
- $x_{\mathsf{increase}} = (\mathsf{nul}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, v)$
- The relation $\mathcal{R}_{\mathsf{increase}}(x_{\mathsf{increase}}, w_{\mathsf{increase}})$ is defined as:

$$\mathcal{R}_{\mathsf{increase}} = \Big\{ \mathsf{ACCU.VfyMem}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \pi_{\mathsf{ACCU}}) = 1$$
$$\wedge\ \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}} = \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}} + v, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho, \rho^{(n+3)}; s^{(2^{n+1})})$$
$$\wedge\ \mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}} = \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho, \rho^{(n+2)}; s^{(2^{n})})$$
$$\wedge\ \mathsf{nul}_{\mathsf{old}} = g^{\rho^{(n+2)}}$$
$$\wedge\ (\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \Big\}.$$

$\mathsf{tnx}_{\mathsf{increase}}$ **Generation**

▷ Upon receiving $(\mathsf{increase}, \mathsf{sid}, \mathsf{a.t}, v)$ from $\mathcal{Z}$ proceed as follows.
▷ Ignore if:
- $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ is not recorded; or

– $v \leq 0$.

▷ Otherwise, parse: $\mathsf{addr}_{\mathsf{pk}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$.

▷ Retrieve the recorded entry $(\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \mathsf{acct}_{\mathsf{old}})$ where: $\mathsf{acct}_{\mathsf{old}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho, \rho^{(n+2)}, s^{(2^n)})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.

▷ Ignore if: $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \cdot) \notin \mathbb{L}_{\mathsf{IAA}}$.

▷ Set $\mathsf{nul}_{\mathsf{old}} \leftarrow g^{\rho^{(n+2)}}$.

▷ Set the new account parameters:

$$\mathsf{acct}_{\mathsf{new}} \leftarrow (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}} + v, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho, \rho^{(n+3)}, s^{(2^{n+1})}).$$

▷ Compute the new account commitment:

$$\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}} \leftarrow \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}} + v, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho, \rho^{(n+3)}; s^{(2^{n+1})}).$$

▷ Compute membership proof: $\pi_{\mathsf{ACCU}} \leftarrow \mathsf{ACCU.PrvMem}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{state}', \mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}})$.

▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Prove}, \mathsf{sid}, x_{\mathsf{increase}}, w_{\mathsf{increase}})$. and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{increase}})$.

▷ Set the transaction: $\mathsf{tnx}_{\mathsf{increase}} \leftarrow (x_{\mathsf{increase}}, \pi_{\mathsf{increase}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Append}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{increase}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$, if $\mathsf{tnx}_{\mathsf{increase}} \in \mathsf{state}$, update: $(\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \mathsf{acct}_{\mathsf{old}}) \leftarrow (\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{acct}_{\mathsf{new}})$.

▷ Output: $(\mathtt{increased}, \mathsf{sid}, \mathsf{a.t}, v)$ to $\mathcal{Z}$.

$\mathsf{tnx}_{\mathsf{increase}}$ Verification

▷ Execute $\textsc{Validate}_{\mathsf{increase}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{increase}})$:
- Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{increase}} = (x_{\mathsf{increase}}, \pi_{\mathsf{increase}})$, $x_{\mathsf{increase}} = (\mathsf{nul}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, v)$.
- Ignore if any of the following conditions hold:
  - $v \leq 0$.
  - $\mathsf{nul}_{\mathsf{old}} \in \mathbb{L}_{\mathsf{NUL}}$.
  - $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \cdot) \notin \mathbb{L}_{\mathsf{IAA}}$.
  - Calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Verify}, \mathsf{sid}, x_{\mathsf{increase}}, \pi_{\mathsf{increase}})$, returns: $(\mathtt{Vrfed}, \mathsf{sid}, 0)$.
- Otherwise, output 1.

▷ Execute $\textsc{Update}_{\mathsf{increase}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{increase}})$:
- Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{increase}} = (x_{\mathsf{increase}}, \pi_{\mathsf{increase}})$, $x_{\mathsf{increase}} = (\mathsf{nul}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, v)$.
- Call: $\mathsf{v}_{\mathsf{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}})$.
- Update the nullifier set: $\mathbb{L}_{\mathsf{NUL}} \leftarrow \mathbb{L}_{\mathsf{NUL}} \cup \{\mathsf{nul}_{\mathsf{old}}\}$.
- Update the state: $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\cdot, \cdot, \cdot, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})\}$.
- Output state.

Fig. 4: Increase asset supply transaction $\mathsf{tnx}_{\mathsf{increase}}$

6.3.5 Settlement/Leg Creation [13]

A settlement may consist of multiple legs, where each leg specifies a sender P^{s} and a receiver P^{r}, along with an amount v of asset type $\mathsf{a.t}$ to be transferred from the sender's account $\mathsf{acct}^{\mathsf{s}}$ to the receiver's account $\mathsf{acct}^{\mathsf{r}}$. The transfer is executed once affirmations have been received from the sender, receiver, and the designated mediator of $\mathsf{a.t}$. Upon receiving a valid settlement transaction $\mathsf{tnx}_{\mathsf{settl}}$, the consensus assigns a unique identifier $\mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}$ and the initial status $\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}}$ that is $\emptyset$. The algorithm is described in Figure 7.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{settl}} = (\mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}}, v, \mathsf{a.t}, \mathsf{pk}^{\mathsf{s}}_{\mathsf{en}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{en}}, \mathsf{pk}^{\mathsf{A}_1}_{\mathsf{en}}, \ldots, \mathsf{pk}^{\mathsf{A}_n}_{\mathsf{en}}, r_1, \ldots, r_4, \pi_{\tilde{\mathsf{ACCU}}})$
- $x_{\mathsf{settl}} = (\mathsf{v}_{\tilde{\mathsf{ACCU}}}, C^{\mathsf{s}}_r, C^{\mathsf{r}}_r, C^{\mathsf{A}_1}_{r_1}, \ldots, C^{\mathsf{A}_1}_{r_4}, \ldots, C^{\mathsf{A}_n}_{r_1}, \ldots, C^{\mathsf{A}_n}_{r_4}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{\mathsf{r}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}})$
- The relation $\mathcal{R}^{\mathsf{A}}_{\mathsf{settl}}(x_{\mathsf{settl}}, w_{\mathsf{settl}})$ is defined as:

$$\mathcal{R}^{\mathsf{A}}_{\mathsf{settl}} = \Big\{ C^{\mathsf{A}_1}_{r_1} = (\mathsf{pk}^{\mathsf{A}_1}_{\mathsf{en}})^{r_1} \wedge \ldots \wedge C^{\mathsf{A}_1}_{r_4} = (\mathsf{pk}^{\mathsf{A}_1}_{\mathsf{en}})^{r_4} \wedge$$
$$\vdots$$
$$C^{\mathsf{A}_n}_{r_1} = (\mathsf{pk}^{\mathsf{A}_n}_{\mathsf{en}})^{r_1} \wedge \ldots \wedge C^{\mathsf{A}_n}_{r_4} = (\mathsf{pk}^{\mathsf{A}_n}_{\mathsf{en}})^{r_4} \wedge$$
$$C^{\mathsf{s}}_{\mathsf{leg}} = g^{r_1} \cdot \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}} \wedge$$
$$C^{\mathsf{r}}_{\mathsf{leg}} = g^{r_2} \cdot \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}} \wedge$$
$$C^{v}_{\mathsf{leg}} = g^{r_3} \cdot h^{v} \wedge$$
$$C^{\mathsf{a.t}}_{\mathsf{leg}} = g^{r_4} \cdot h^{\mathsf{a.t}} \wedge$$
$$0 < v \leq \mathsf{V}_{\mathsf{max}} \wedge$$
$$\mathsf{ACCU}.\tilde{\mathsf{V}}\mathsf{fyMem}(\mathsf{pp}_{\tilde{\mathsf{ACCU}}}, \mathsf{v}_{\tilde{\mathsf{ACCU}}}, (\mathsf{a.t}, \mathsf{pk}^{\mathsf{A}_1}_{\mathsf{en}}, \ldots, \mathsf{pk}^{\mathsf{A}_n}_{\mathsf{en}}), \pi_{\tilde{\mathsf{ACCU}}}) = 1 \quad ^{14} \Big\}.$$

Informally, the settlement creator proves the well-formedness of leg. Both the public key of the auditor(s)/mediator(s), $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$, and the asset type, $\mathsf{a.t}$, are included in the witness, and their association is expressed as: $\mathbb{L}_{\mathsf{IAA}}[i]$, where the index i is also part of the witness.

[13] In this paper, the terms *settlement* and *leg* are used interchangeably. Once there exists a settlement with multiple legs, the settlement creator is responsible for defining and connecting these legs.

[14] In the accumulator, there is a distinction between the roles of the auditor and the mediator. In the case of a mediator, the chain waits for the mediator's affirmation. However, in the case of an auditor, there is no such affirmation. So, while the accumulator distinguishes between these roles, we do not discuss these implementation details here.

$\mathsf{tnx_{settl}}$ Generation

▷ Upon receiving $(\texttt{Create.Settlement}, \mathsf{sid}, \mathsf{P^s}, \mathsf{P^r}, v, \mathsf{a.t})$ from $\mathcal{Z}$ proceed as follows.[a]

▷ Ignore the request if $v \leq 0$.

▷ Otherwise, call $\mathcal{F}_{\mathsf{KeyReg}}$ with: $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P^s})$ and $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P^r})$.

▷ Upon receiving: $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P^s}, \mathsf{addr^s_{pk}})$ and $(\texttt{Rtrv.Key}, \mathsf{sid}, \mathsf{P^r}, \mathsf{addr^r_{pk}})$, parse: $\mathsf{addr^s_{pk}} = (\mathsf{pk^s_{acct}}, \mathsf{pk^s_{en}}), \quad \mathsf{addr^r_{pk}} = (\mathsf{pk^r_{acct}}, \mathsf{pk^r_{en}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Retrieve}, \mathsf{sid})$. Upon receiving: $(\texttt{Retrieved}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v_{ACCU}})$.

▷ Using $\mathsf{a.t}$, retrieve: $(\mathsf{a.t}, \mathsf{pk^{A_1}_{en}}, \ldots, \mathsf{pk^{A_n}_{en}}) = \mathbb{L}_{\mathsf{IAA}}[i]$.

▷ Compute leg information encryption leg (optimized for multiple auditors):

- Sample $y \xleftarrow{\$} \mathbb{Z}_p$ and compute:

$$C^{\mathsf{s}}_r = (\mathsf{pk^s_{en}})^y, \quad C^{\mathsf{r}}_r = (\mathsf{pk^r_{en}})^y$$
$$r_i = \mathtt{Hash}(g^y \parallel i), \quad i \in \{1,2,3,4\}$$

- Given the auditor/mediator public keys $\mathsf{pk^{A_1}_{en}}, \ldots, \mathsf{pk^{A_n}_{en}}$ compute the following:
  - $C^{\mathsf{A_1}}_{r_1} = (\mathsf{pk^{A_1}_{en}})^{r_1}, \ldots, C^{\mathsf{A_1}}_{r_4} = (\mathsf{pk^{A_1}_{en}})^{r_4}$

    $\vdots$
  - $C^{\mathsf{A}_n}_{r_1} = (\mathsf{pk^{A_n}_{en}})^{r_1}, \ldots, C^{\mathsf{A}_n}_{r_4} = (\mathsf{pk^{A_n}_{en}})^{r_4}$
- $C^{\mathsf{s}}_{\mathsf{leg}} = g^{r_1} \cdot \mathsf{pk^s_{acct}}$
- $C^{\mathsf{r}}_{\mathsf{leg}} = g^{r_2} \cdot \mathsf{pk^r_{acct}}$
- $C^{v}_{\mathsf{leg}} = g^{r_3} \cdot h^v$
- $C^{\mathsf{a.t}}_{\mathsf{leg}} = g^{r_4} \cdot h^{\mathsf{a.t}}$

Set the leg encryption $\mathsf{leg} \leftarrow (C^{\mathsf{s}}_r, C^{\mathsf{r}}_r, C^{\mathsf{A_1}}_{r_1}, \ldots, C^{\mathsf{A_1}}_{r_4}, \ldots, C^{\mathsf{A}_n}_{r_1}, \ldots, C^{\mathsf{A}_n}_{r_4}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{\mathsf{r}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}})$.

▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\texttt{Prove}, \mathsf{sid}, x_{\mathsf{settl}}, w_{\mathsf{settl}})$ and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{settl}})$.

▷ Set the settlement transaction: $\mathsf{tnx_{settl}} \leftarrow (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Append}, \mathsf{sid}, \mathsf{tnx_{settl}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Retrieve}, \mathsf{sid})$. Upon receiving: $(\texttt{Retrieved}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v_{ACCU}})$, if $\mathsf{tnx_{settl}} \in \mathsf{TNX}$, $\mathsf{TNX} \in \mathsf{state}'$ output: $(\texttt{Settlement.Created}, \mathsf{P^s}, \mathsf{P^r}, v, \mathsf{a.t}, \mathsf{leg})$ to $\mathcal{Z}$.

$\mathsf{tnx_{settl}}$ Verification

▷ Run $\textsc{Validate}_{\mathsf{settl}}(\mathsf{state}, \mathsf{tnx_{settl}})$:

- Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v_{ACCU}})$, $\mathsf{tnx_{settl}} = (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$, $x_{\mathsf{settl}} = (\mathsf{v_{\tilde{ACCU}}}, \mathsf{leg})$.
- Retrieve the locally stored list accumulator value $\mathsf{v_{\tilde{ACCU}}}^*$ and ignore if:[b] $\mathsf{v_{\tilde{ACCU}}}^* \neq \mathsf{v_{\tilde{ACCU}}}$.

– Otherwise, ignore if calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\texttt{Verify}, \mathsf{sid}, x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$ returns: $(\texttt{Vrfed}, \mathsf{sid}, 0)$.
– Otherwise, output 1.

▷ Run $\textsc{Update}_{\mathsf{settl}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{settl}})$:
– Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{settl}} = (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$, $x_{\mathsf{settl}} = (\mathsf{v}_{\tilde{\mathsf{ACCU}}}, \mathsf{leg})$.
– Sample a fresh identifier $\mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}$ and set initial status: $\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}} \leftarrow \emptyset$.
– Define: $\mathsf{TNX} \leftarrow (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$, update: $\mathsf{state}' \leftarrow \mathsf{state}' \cup \{\mathsf{TNX}\}$.
– Update the global state: $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\mathsf{state}', \cdot, \cdot, \cdot, \cdot)\}$.
– Output state.

[a] Without loss of generality, the algorithm is described for a single leg $(\mathsf{P}^{\mathsf{s}}, \mathsf{P}^{\mathsf{r}}, v, \mathsf{a.t})$.

[b] We assume that chain validators periodically accumulate the elements of $\mathbb{L}_{\mathsf{IAA}}$ (see Section 7.1) and generate a single accumulated value $\mathsf{v}_{\tilde{\mathsf{ACCU}}}$. Instead of reading the entire list $\mathbb{L}_{\mathsf{IAA}}$, they verify the NIZK proof of the settlement creator against this accumulated value $\mathsf{v}_{\tilde{\mathsf{ACCU}}}$.

Fig. 5: Settlement Creation Transaction $\mathsf{tnx}_{\mathsf{settl}}$

6.3.6 Ledger Scan

This algorithm can be executed by a sender P^{s}, a receiver P^{r} or an auditor/mediator A.

▷ Upon receiving $(\mathsf{Scan}, \mathsf{sid}, \mathsf{addr}_{\mathsf{pk}}, \texttt{role})$ from a party proceed as follows.

▷ Ignore the request if $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ is not recorded. Otherwise, parse: $\mathsf{addr}_{\mathsf{sk}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$.

▷ Invoke $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Retrieve}, \mathsf{sid})$. Upon receiving: $(\texttt{Retrieved}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.

▷ Parse each:[a] $\mathsf{TNX}^i = (\mathsf{tnx}^i_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}})$ in state'.

▷ For each j such that $f(\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}) = 1$, proceed as follows:[b]
– Parse $\mathsf{tnx}^j_{\mathsf{settl}} = (x^j_{\mathsf{settl}}, \pi^j_{\mathsf{settl}})$ and further parse: $x^j_{\mathsf{settl}} = (\mathsf{v}_{\tilde{\mathsf{ACCU}}}{}^j, \mathsf{leg}^j)$. and parse:

$$\mathsf{leg}^j = \begin{pmatrix} C_r^{\mathsf{s},j},\ C_r^{\mathsf{r},j},\ C_{r_1}^{\mathsf{A}_1,j},\ \ldots,\ C_{r_4}^{\mathsf{A}_1,j},\ \ldots,\ C_{r_1}^{\mathsf{A}_n,j},\ \ldots,\ C_{r_4}^{\mathsf{A}_n,j}, \\ C_{\mathsf{leg}}^{\mathsf{s},j},\ C_{\mathsf{leg}}^{\mathsf{r},j},\ C_{\mathsf{leg}}^{v,j},\ C_{\mathsf{leg}}^{\mathsf{a.t},j} \end{pmatrix}$$

– Set α^j based on $\texttt{role}$:
  • If $\texttt{role} = \mathsf{P}^{\mathsf{s}}$, set $\alpha^j \leftarrow (C_r^{\mathsf{s},j}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{\mathsf{r}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}})$.
  • If $\texttt{role} = \mathsf{P}^{\mathsf{r}}$, set $\alpha^j \leftarrow (C_r^{\mathsf{r},j}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{\mathsf{r}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}})$.
  • For $\texttt{role} = \mathsf{A}_k$ with $k = 1, \ldots, N$:
    ∗ If $\texttt{role} = \mathsf{A}_1$, set $\alpha^{j,1} \leftarrow (C_{r_1}^{\mathsf{A}_1}, \ldots, C_{r_4}^{\mathsf{A}_1}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{\mathsf{r}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}})$.

∗ If $\texttt{role} = \mathsf{A}_2$, set $\alpha^{j,2} \leftarrow (C_{r_1}^{\mathsf{A}_2}, \ldots, C_{r_4}^{\mathsf{A}_2}, C_{\mathsf{leg}}^{\mathsf{s}}, C_{\mathsf{leg}}^{\mathsf{r}}, C_{\mathsf{leg}}^{v}, C_{\mathsf{leg}}^{\mathsf{a.t}})$.

⋮

∗ If $\texttt{role} = \mathsf{A}_n$, set $\alpha^{j,n} \leftarrow (C_{r_1}^{\mathsf{A}_n}, \ldots, C_{r_4}^{\mathsf{A}_n}, C_{\mathsf{leg}}^{\mathsf{s}}, C_{\mathsf{leg}}^{\mathsf{r}}, C_{\mathsf{leg}}^{v}, C_{\mathsf{leg}}^{\mathsf{a.t}})$.

– Compute the decryption: $o_1^j \leftarrow (r_1^j, \cdots, r_4^j)$ (if $\mathsf{P^s}$ / $\mathsf{P^r}$) or $(g^{r_1^j}, \cdots, g^{r_4^j})$ if A. $o_2^j \leftarrow (\mathsf{pk}_{\mathsf{acct}}^{\mathsf{s}_j}, \mathsf{pk}_{\mathsf{acct}}^{\mathsf{r}_j}, v^j, \mathsf{a.t}^j)$.

▷ Output: o_1^j, o_2^j, and TNX^j.

[a] For simplicity, we omit implementation details related to efficiency, specifically how the wallet maintains state by recording the last moment it scanned the ledger. In practice, the wallet could be stateful, ensuring that future scans resume from the last recorded point.

[b] The function $f(\cdot)$ may consist of multiple sub-functions depending on the caller's role. For example, if the caller is the sender, the function takes a specific form, whereas for a receiver or auditor, it may differ. The function enforces the transaction process's intended logic. For instance, if the receiver has rejected a leg but the sender has not yet reclaimed it, we could determine whether the receiver should be allowed to come online again, change their decision, and accept the leg. We intentionally leave $f(\cdot)$ undefined to allow for future specification.

Fig. 6: The `Ledger.Scan` algorithm

6.3.7 Sender Transaction

The sender scans the ledger to identify settlement transactions $\mathsf{tnx}_{\mathsf{settl}}$ and then extracts leg in which they are the associated sender. The sender proves ownership of a fresh old account (one that has not been previously used and is an element of the accumulator) by providing a proof of membership in the accumulator π_{ACCU}. The sender also reveals the nullifier for their old account $\mathsf{nul}_{\mathsf{old}}^{\mathsf{s}}$ and demonstrates the well-formedness of their new account commitment $\mathsf{acct}_{\mathsf{new}}^{\mathsf{cm,s}}$.
Depending on the sender's affirmation bit $b_{\mathsf{s}} \in \{0, 1\}$, there exist two distinct NIZK relations, each corresponding to a specific affirmation choice.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{send}}^1 = (r_1, r_3, r_4, \mathsf{sk}_{\mathsf{acct}}^{\mathsf{s}}, \mathsf{f.b}_{\mathsf{old}}^{\mathsf{s}}, \mathsf{cnt}_{\mathsf{old}}^{\mathsf{s}}, \rho_{\mathsf{old}}^{\mathsf{s}}, s_{\mathsf{old}}^{\mathsf{s}}, \mathsf{acct}_{\mathsf{old}}^{\mathsf{cm,s}}, v, \mathsf{a.t}, \pi_{\mathsf{ACCU}}, n)$
- $x_{\mathsf{send}}^1 = (\mathsf{v}_{\mathsf{ACCU}}, \mathsf{nul}_{\mathsf{old}}^{\mathsf{s}}, \mathsf{acct}_{\mathsf{new}}^{\mathsf{cm,s}}, C_{\mathsf{leg}}^{\mathsf{s}}, C_{\mathsf{leg}}^{v}, C_{\mathsf{leg}}^{\mathsf{a.t}}, \mathsf{tnx}_{\mathsf{settl}}^{\mathsf{id}})$
- The relation $\mathcal{R}_{\mathsf{send}}^1(x_{\mathsf{send}}^1, w_{\mathsf{send}}^1)$ is defined as:

$$\mathcal{R}^1_{\mathsf{send}} = \Big\{ \mathsf{ACCU.VfyMem}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{old}}, \pi_{\mathsf{ACCU}}) = 1$$
$$\wedge\ \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}} = \mathsf{Com}(\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}} - v, \mathsf{cnt}^{\mathsf{s}}_{\mathsf{old}} + 1, \mathsf{a.t}, \rho_{\mathsf{s}}, \rho_{\mathsf{s}}^{(n+3)}; s_{\mathsf{s}}^{(2^{n+1})})$$
$$\wedge\ \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{old}} = \mathsf{Com}(\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{cnt}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{a.t}, \rho_{\mathsf{s}}, \rho_{\mathsf{s}}^{(n+2)}; s_{\mathsf{s}}^{(2^{n})})$$
$$\wedge\ \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}} \geq v$$
$$\wedge\ \mathsf{nul}_{\mathsf{old}} = g^{\rho_{\mathsf{s}}^{(n+2)}}$$
$$\wedge\ C^{\mathsf{s}}_{\mathsf{leg}} = g^{r_1} \cdot \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}$$
$$\wedge\ C^{v}_{\mathsf{leg}} = g^{r_3} \cdot h^{v}$$
$$\wedge\ C^{\mathsf{a.t}}_{\mathsf{leg}} = g^{r_4} \cdot h^{\mathsf{a.t}}$$
$$\wedge\ (\mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \Big\}.$$

The NIZK witness, statement, and relation are defined as follows.

- $w^0_{\mathsf{send}} = (r_1, \mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}})$
- $x^0_{\mathsf{send}} = (C^{\mathsf{s}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}})$
- The relation $\mathcal{R}^0_{\mathsf{send}}(x^0_{\mathsf{send}}, w^0_{\mathsf{send}})$ is defined as:

$$\mathcal{R}^0_{\mathsf{send}} = \Big\{ C^{\mathsf{s}}_{\mathsf{leg}} = g^{r_1} \cdot \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}$$
$$\wedge\ (\mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \Big\}.$$

$\mathsf{tnx}_{\mathsf{send}}$ Generation

▷ Upon receiving $(\mathsf{send}, \mathsf{sid}, \mathsf{addr}^{\mathsf{s}}_{\mathsf{pk}})$ from $\mathcal{Z}$ proceed as follows.
▷ Ignore if $(\mathsf{addr}^{\mathsf{s}}_{\mathsf{pk}}, \mathsf{addr}^{\mathsf{s}}_{\mathsf{sk}})$ is not recorded.
▷ Otherwise, parse: $\mathsf{addr}^{\mathsf{s}}_{\mathsf{sk}} = (\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{sk}^{\mathsf{s}}_{\mathsf{en}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Retrieve}, \mathsf{sid})$.
▷ Upon receiving: $(\mathtt{Retrieved}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.
▷ Call $\mathtt{Ledger.Scan}$ with: $(\mathsf{Scan}, \mathsf{sid}, \mathsf{addr}^{\mathsf{s}}_{\mathsf{pk}}, \mathsf{P}^{\mathsf{s}})$. Upon receiving (o_1, o_2, TNX),[a] proceed as follows.
▷ Parse: $o_1 = (r_1, \cdots, r_4)$, $o_2 = (\mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}}, v, \mathsf{a.t})$, $\mathsf{TNX} = (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$, $\mathsf{tnx}_{\mathsf{settl}} = (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$, $x_{\mathsf{settl}} = (\tilde{\mathsf{v}_{\mathsf{ACCU}}}, \mathsf{leg})$.
▷ Choose sender affirmation bit: $b_{\mathsf{s}} \leftarrow \{0, 1\}$.
▷ If $b_{\mathsf{s}} = 1$, perform the following:
  - Retrieve the recorded entry $(\mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{s}}_{\mathsf{old}})$, where: $\mathsf{acct}^{\mathsf{s}}_{\mathsf{old}} = (\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{cnt}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{a.t}, \rho_{\mathsf{s}}, \rho_{\mathsf{s}}^{n}, s_{\mathsf{s}}^{(2^n)})$.
  - Compute nullifier: $\mathsf{nul}^{\mathsf{s}}_{\mathsf{old}} = g^{\rho_{\mathsf{s}}^{n}}$.
  - Set new sender account:

$$\mathsf{acct}^{\mathsf{s}}_{\mathsf{new}} \leftarrow (\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}} - v, \mathsf{cnt}^{\mathsf{s}}_{\mathsf{old}} + 1, \mathsf{a.t}, \rho_{\mathsf{s}}, \rho_{\mathsf{s}}^{(n+3)}, s_{\mathsf{s}}^{2^{(n+1)}}).$$

– Compute new sender account commitment:

$$\mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}} \leftarrow \mathsf{Com}(\mathsf{sk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{s}}_{\mathsf{old}} - v, \mathsf{cnt}^{\mathsf{s}}_{\mathsf{old}} + 1, \mathsf{a.t}, \rho_{\mathsf{s}}, \rho_{\mathsf{s}}^{(n+3)}; s_{\mathsf{s}}^{2^{(n+1)}}).$$

– Compute membership proof: $\pi_{\mathsf{ACCU}} \leftarrow \mathsf{ACCU.PrvMem}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{state}', \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{old}})$.

– Set NIZK statement x^1_{send}, and NIZK witness w^1_{send} as described above.

▷ If $b_{\mathsf{s}} = 0$, set NIZK statement x^0_{send}, and NIZK witness w^0_{send} as described above.

▷ For $i = 0$ or 1, call $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\texttt{Prove}, \mathsf{sid}, x^i_{\mathsf{send}}, w^i_{\mathsf{send}})$. and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi^i_{\mathsf{send}})$.

▷ Set the send transaction: $\mathsf{tnx}^i_{\mathsf{send}} \leftarrow (x^i_{\mathsf{send}}, \pi^i_{\mathsf{send}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Append}, \mathsf{sid}, \mathsf{tnx}^i_{\mathsf{send}})$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\texttt{Retrieve}, \mathsf{sid})$. Upon receiving: $(\texttt{Retrieved}, \mathsf{sid}, \mathsf{state})$, parse $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$ if $\mathsf{tnx}^i_{\mathsf{send}} \in \mathsf{state}'$, update: $(\mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{s}}_{\mathsf{old}}) \leftarrow (\mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}}, \mathsf{acct}^{\mathsf{s}}_{\mathsf{new}})$, and update: $\mathsf{LT} \leftarrow \mathsf{LT} \cup \{(o_1, o_2, \mathsf{TNX})\}$. Output: $(\mathsf{sent}, \mathsf{sid}, \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}}, v, \mathsf{a.t}, \mathsf{TNX})$ to $\mathcal{Z}$.

$\mathsf{tnx}_{\mathsf{send}}$ Verification

▷ Run $\text{VALIDATE}_{\mathsf{send}}(\mathsf{state}, \mathsf{tnx}^i_{\mathsf{send}})$:

– Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}^i_{\mathsf{send}} = (x^i_{\mathsf{send}}, \pi^i_{\mathsf{send}})$.

– If $i = 1$, parse: $x^1_{\mathsf{send}} = (\mathsf{v}_{\mathsf{ACCU}}, \mathsf{nul}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}})$ If $i = 0$, parse: $x^0_{\mathsf{send}} = (C^{\mathsf{s}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}})$

– Given $\mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}} \in x^i_{\mathsf{send}}$, retrieve: $\mathsf{TNX} = (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$ where $\mathsf{TNX} \in \mathsf{state}'$. parse: $\mathsf{tnx}_{\mathsf{settl}} = (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$, $x_{\mathsf{settl}} = (\mathsf{v}_{\tilde{\mathsf{ACCU}}}, \mathsf{leg}^*)$

– Ignore if any of the following conditions hold:

- for $i = 1$: $(C^{*\mathsf{s}}_{\mathsf{leg}}, C^{*v}_{\mathsf{leg}}, C^{*\mathsf{a.t}}_{\mathsf{leg}}) \in \mathsf{leg}^* \neq (C^{\mathsf{s}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}}) \in x^1_{\mathsf{send}}$; for $i = 0$: $C^{*\mathsf{s}}_{\mathsf{leg}} \in \mathsf{leg}^* \neq C^{\mathsf{s}}_{\mathsf{leg}} \in x^0_{\mathsf{send}}$
- $\mathsf{nul}^{\mathsf{s}}_{\mathsf{old}} \in \mathbb{L}_{\mathsf{NUL}}$.
- $f(\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}}) \neq 1$.
- Calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\texttt{Verify}, \mathsf{sid}, x^i_{\mathsf{send}}, \pi^i_{\mathsf{send}})$ returns: $(\texttt{Vrfed}, \mathsf{sid}, 0)$.

Otherwise, output 1.

▷ Run $\text{UPDATE}_{\mathsf{send}}(\mathsf{state}, \mathsf{tnx}^i_{\mathsf{send}})$:

– Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}^i_{\mathsf{send}} = (x^i_{\mathsf{send}}, \pi^i_{\mathsf{send}})$.

– If $i = 1$, parse: $x^1_{\mathsf{send}} = (\mathsf{v}_{\mathsf{ACCU}}, \mathsf{nul}^{\mathsf{s}}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}}, C^{\mathsf{s}}_{\mathsf{leg}}, C^{v}_{\mathsf{leg}}, C^{\mathsf{a.t}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}})$ If $i = 0$, parse: $x^0_{\mathsf{send}} = (C^{\mathsf{s}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}})$.

– Given $\mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}} \in x^i_{\mathsf{send}}$, retrieve: $\mathsf{TNX} = (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$ where $\mathsf{TNX} \in \mathsf{state}'$.

– Update the transaction status:[b] $\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}} \leftarrow \mathsf{update}(\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$.

– Update TNX and state': $\mathsf{TNX} \leftarrow (\cdot, \cdot, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$, $\mathsf{state}' \leftarrow \mathsf{state}' \cup \{\mathsf{TNX}\}$.

– Update the accumulator value: $\mathsf{v}_{\mathsf{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm,s}}_{\mathsf{new}})$.

– Update the nullifier set: $\mathbb{L}_{\mathsf{NUL}} \leftarrow \mathbb{L}_{\mathsf{NUL}} \cup \{\mathsf{nul}^{\mathsf{s}}_{\mathsf{old}}\}$.
– Update the global state: $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\mathsf{state}', \cdot, \cdot, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})\}$.
– Output state.

[a] The `Ledger.Scan` algorithm outputs multiple decryption values. For simplicity, we assume the output consists of a single tuple (o_1, o_2, TNX). However, the algorithm may output multiple such tuples, and the following steps apply to each. The wallet may be assumed to be stateful (see Figure 6).
[b] For the description of update see this note.

Fig. 7: Sender transaction $\mathsf{tnx}_{\mathsf{send}}$

6.3.8 Receiver Transaction

The receiver scans the ledger to identify settlement transactions $\mathsf{tnx}_{\mathsf{settl}}$ in which they are the designated recipient. This algorithm closely follows the sender's algorithm, with the key distinction that the new receiver account is set as:

$$\mathsf{acct}^{\mathsf{r}}_{\mathsf{new}} = (\mathsf{sk}^{\mathsf{r}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{r}}_{\mathsf{old}}, \mathsf{cnt}^{\mathsf{r}}_{\mathsf{old}} + 1, \mathsf{a.t}, \rho_{\mathsf{r}}, \rho_{\mathsf{r}}^{(n+3)}, s_{\mathsf{r}}^{(2^{n+1})}).$$

The corresponding zero-knowledge proof relation is adapted to accommodate this modification, ensuring correctness while preserving the integrity of the transaction logic. The remainder of the algorithm adheres to this change. Whenever all affirmations of the settlement have been received, the receiver can submit another account state transition in which:

$$\mathsf{acct}^{\mathsf{r}}_{\mathsf{new}} = (\mathsf{sk}^{\mathsf{r}}_{\mathsf{acct}}, \mathsf{f.b}^{\mathsf{r}}_{\mathsf{old}} + v, \mathsf{cnt}^{\mathsf{r}}_{\mathsf{old}} - 1, \mathsf{a.t}, \rho_{\mathsf{r}}, \rho_{\mathsf{r}}^{(n+4)}, s_{\mathsf{r}}^{(2^{n+2})}).$$

6.3.9 Reversion

Remark 3. Another difference between P-DART and DART [38] lies in the reversion algorithm. In DART, the sender creates transactions that immediately reduce their spendable finalized balance to prevent double spending. If the receiver does not affirm the transaction, the sender can reverse it by restoring their finalized balance. In contrast, in P-DART, the leg is created by a leg creator without affecting the sender's finalized balance. As a result, the concept of reversion slightly differs from that described in the DART paper. Here, reversion refers to cases where an entity (e.g., the sender or receiver of a leg) that has previously affirmed a transaction may later choose to reverse their decision based on conditions defined by $f(\cdot)$. For example, a receiver who has affirmed a transaction may later decide to reverse their affirmation. In such cases, the receiver changes their stance from affirming to rejecting the transaction. Since this reversion mechanism follows a straightforward process similar to that of sender transactions, we do not elaborate on it further. Specifically, if the sender reverses, their spendable finalized balance increases while the counter decreases by one; if the receiver reverses, only their counter decreases by one.

6.3.10 Proof of balance. We present two approaches for proof of balance (PoB). In the first approach, the party P interacts with the associated auditor A (for the asset type $\mathsf{a.t}$ whose balance is requested). A outputs balance. In the second approach, outlined as a generic solution in Section 6.3.11, P proves their balance directly to any PoB verifier, without relying on A. The first approach is provided in Figure 8, where P reveals their $\mathsf{f.b}$ and cnt from their associated account and proves the correctness of the revealed values. There is no account state transition in PoB; however, P must reveal the nullifier nul to enable A to verify that P is disclosing $\mathsf{f.b}$ and cnt corresponding to their most *recent* account. P also provides pointers to all finalized and pending transactions to justify the value of cnt. The values $\mathsf{f.b}$ and $\mathsf{pending\ balance}$ are output.
The NIZK witness, statement, and relation are defined as follows.

- $w^{\mathsf{A}}_{\mathsf{PoB}} = (\mathsf{sk}_{\mathsf{acct}}, \rho, s)$
- $x^{\mathsf{A}}_{\mathsf{PoB}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{nul}, \mathsf{acct}^{\mathsf{cm}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t})$
- The relation $\mathcal{R}_{\mathsf{PoB}}(x^{\mathsf{A}}_{\mathsf{PoB}}, w^{\mathsf{A}}_{\mathsf{PoB}})$ is defined as:

$$\mathcal{R}_{\mathsf{PoB}} = \Big\{ \mathsf{acct}^{\mathsf{cm}} = \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t}, \rho, \rho^{(n+2)}; s^{(2^n)}) \\ \wedge (\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda}) \\ \wedge \mathsf{nul} = g^{\rho^{(n+2)}} \Big\}.$$

$\mathsf{tnx}_{\mathsf{PoB}}$ Generation executed by P

▷ Upon receiving $(\mathsf{balance}, \mathsf{sid}, \mathsf{a.t})$ from $\mathcal{Z}$, parse: $\mathsf{addr}_{\mathsf{pk}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}})$, $\mathsf{addr}_{\mathsf{sk}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{en}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.
▷ Retrieve the recorded entry $(\mathsf{acct}^{\mathsf{cm}}, \mathsf{acct})$, where: $\mathsf{acct} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t}, \rho, \rho^{(n+2)}, s^{(2^n)})$.
▷ Compute the nullifier: $\mathsf{nul} = g^{\rho^{(n+2)}}$.
▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Prove}, \mathsf{sid}, x^{\mathsf{A}}_{\mathsf{PoB}}, w^{\mathsf{A}}_{\mathsf{PoB}})$, and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi^{\mathsf{A}}_{\mathsf{PoB}})$.
▷ Retrieve all entries $(o^i_1, o^i_2, \mathsf{TNX}^i)$ in LT where, upon parsing: $o^i_2 = (\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}}, v^i, \mathsf{a.t}^i)$, $\mathsf{TNX}^i = (\mathsf{tnx}^i_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}})$, the conditions $\mathsf{a.t}^i = \mathsf{a.t}$ and $(\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}} = \mathsf{pk}_{\mathsf{acct}}$ or $\mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}} = \mathsf{pk}_{\mathsf{acct}})$ hold.
▷ Define two lists, both initially empty: $\overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(1)} \leftarrow \varnothing$, $\overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(0)} \leftarrow \varnothing$.
▷ Retrieve all $\mathsf{TNX}^j \in \mathsf{state}'$ where: $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$.
▷ For each $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}$ retrieved from LT:
  - Add $(\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathtt{role}^i)$ to $\overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(1)}$ if: $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} = \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$ and $\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}$: finalized, for some: $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$.
  - Add $(\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathtt{role}^i)$ to $\overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(0)}$ if: $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} = \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$ and $\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}$: pending, for some: $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$.

▷ Using $\mathsf{a.t}$, retrieve $\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}}$ from $\mathbb{L}_{\mathsf{IAA}}$.

▷ Call $\mathcal{F}_{\mathsf{KeyReg}}$ with: $(\mathtt{Rtrv.id}, \mathsf{sid}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})$. Upon receiving: $(\mathtt{Rtrved.id}, \mathsf{sid}, \mathsf{A})$, proceed.
▷ Set: $\mathsf{tnx}^{\mathsf{A}}_{\mathsf{PoB}} \leftarrow (x^{\mathsf{A}}_{\mathsf{PoB}}, \pi^{\mathsf{A}}_{\mathsf{PoB}}, \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(1)}, \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(0)})$, and call $\mathcal{F}_{\mathsf{ch}}$ with: $(\mathtt{Send}, \mathsf{sid}, \mathsf{A}, \mathsf{tnx}^{\mathsf{A}}_{\mathsf{PoB}})$.

PoB Output Executed by A

▷ Upon receiving: $(\mathtt{Received}, \mathsf{sid}, \mathsf{P}, \mathsf{tnx}^{\mathsf{A}}_{\mathsf{PoB}})$ from $\mathcal{F}_{\mathsf{ch}}$, parse: $\mathsf{tnx}^{\mathsf{A}}_{\mathsf{PoB}} = (x^{\mathsf{A}}_{\mathsf{PoB}}, \pi^{\mathsf{A}}_{\mathsf{PoB}}, \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(1)}, \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(0)})$, $x^{\mathsf{A}}_{\mathsf{PoB}} = (\mathsf{pk}_{\mathsf{acct}}, \mathsf{nul}, \mathsf{acct}^{\mathsf{cm}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t})$.
▷ Call $\mathcal{F}_{\mathsf{KeyReg}}$ with: $(\mathtt{Rtrv.Key}, \mathsf{sid}, \mathsf{P})$. Ignore if the response is: $(\mathtt{Rtrv.Key}, \mathsf{sid}, \mathsf{P}, (\mathsf{pk}'_{\mathsf{acct}}, \cdot))$ and $\mathsf{pk}'_{\mathsf{acct}} \neq \mathsf{pk}_{\mathsf{acct}}$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with: $(\mathtt{Read}, \mathsf{sid})$. Upon receiving: $(\mathtt{Read}, \mathsf{sid}, \mathsf{state})$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.
▷ Ignore if any of the following conditions hold:
- $\mathsf{nul} \in \mathbb{L}_{\mathsf{NUL}}$,
- $\mathsf{acct}^{\mathsf{cm}} \notin \mathsf{state}'$,
- Upon calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Verify}, \mathsf{sid}, x^{\mathsf{A}}_{\mathsf{PoB}}, \pi^{\mathsf{A}}_{\mathsf{PoB}})$, the response is: $(\mathtt{Vrfed}, \mathsf{sid}, 0)$.

▷ For all $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} \in \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(1)}$:
- Ignore if no $\mathsf{TNX}^i \in \mathsf{state}'$ exists where: $\mathsf{TNX}^i = (\mathsf{tnx}^i_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}})$ with $\mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}}$: finalized.
- Parse: $\mathsf{tnx}^i_{\mathsf{settl}} = (x^i_{\mathsf{settl}}, \pi^i_{\mathsf{settl}})$, $x^i_{\mathsf{settl}} = (\tilde{\mathsf{v}_{\mathsf{ACCU}}}^i, \mathsf{leg}^i)$.
- Compute decryption using $\mathsf{sk}^{\mathsf{A}}_{\mathsf{en}}$: o^i_1, o^i_2.
- Ignore if $o^i_2 = \perp$. Otherwise, parse: $o^i_2 = (\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}}, v^i, \mathsf{a.t}^i)$.
- If:
  - $\mathtt{role}^i = \mathsf{P}^{\mathsf{s}}$, ignore if $\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}} \neq \mathsf{pk}_{\mathsf{acct}}$ or $\mathsf{a.t}^i \neq \mathsf{a.t}$.
  - $\mathtt{role}^i = \mathsf{P}^{\mathsf{r}}$, ignore if $\mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}} \neq \mathsf{pk}_{\mathsf{acct}}$ or $\mathsf{a.t}^i \neq \mathsf{a.t}$.

  Otherwise, increment: $\mathsf{cnt}_{(1)} \leftarrow \mathsf{cnt}_{(1)} + 1$, where initially $\mathsf{cnt}_{(1)} = 0$.

▷ For all $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} \in \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{(0)}$:
- Ignore if no $\mathsf{TNX}^i \in \mathsf{state}'$ exists where: $\mathsf{TNX}^i = (\mathsf{tnx}^i_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}})$ with $\mathsf{tnx}^{\mathsf{st}_i}_{\mathsf{settl}}$: pending.
- Parse: $\mathsf{tnx}^i_{\mathsf{settl}} = (x^i_{\mathsf{settl}}, \pi^i_{\mathsf{settl}})$, $x^i_{\mathsf{settl}} = (\tilde{\mathsf{v}_{\mathsf{ACCU}}}^i, \mathsf{leg}^i)$.
- Compute decryption using $\mathsf{sk}^{\mathsf{A}}_{\mathsf{en}}$: o^i_1, o^i_2.
- Ignore if $o^i_2 = \perp$. Otherwise, parse: $o^i_2 = (\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}}, v^i, \mathsf{a.t}^i)$.
- If:
  - $\mathtt{role}^i = \mathsf{P}^{\mathsf{s}}$, ignore if $\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}} \neq \mathsf{pk}_{\mathsf{acct}}$ or $\mathsf{a.t}^i \neq \mathsf{a.t}$.
  - $\mathtt{role}^i = \mathsf{P}^{\mathsf{r}}$, ignore if $\mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}} \neq \mathsf{pk}_{\mathsf{acct}}$ or $\mathsf{a.t}^i \neq \mathsf{a.t}$.

  Otherwise, increment: $\mathsf{cnt}_{(0)} \leftarrow \mathsf{cnt}_{(0)} + 1$, where initially $\mathsf{cnt}_{(0)} = 0$.
- For $\mathtt{role}^i = \mathsf{P}^{\mathsf{s}}$ (sender), update: $V^{\mathtt{sender}}_{(0)} \leftarrow V^{\mathtt{sender}}_{(0)} + v^i$, where initially $V^{\mathtt{sender}}_{(0)} = 0$.
- For $\mathtt{role}^i = \mathsf{P}^{\mathsf{r}}$ (receiver), update: $V^{\mathtt{receiver}}_{(0)} \leftarrow V^{\mathtt{receiver}}_{(0)} + v^i$, where initially $V^{\mathtt{receiver}}_{(0)} = 0$.

▷ Ignore if: $\mathsf{cnt}_{(1)} + \mathsf{cnt}_{(0)} \neq \mathsf{cnt}$.
▷ Otherwise, output: $(\mathsf{balance}, \mathsf{sid}, \mathsf{P}, \mathsf{a.t}, \mathsf{f.b}, V^{\mathtt{sender}}_{(0)}, V^{\mathtt{receiver}}_{(0)})$ to $\mathcal{Z}$.

Fig. 8: Proof of balance transaction $\mathsf{tnx}_{\mathsf{PoB}}$ – 1st Approach

6.3.11 Proof of balance: a generic solution In addition to the finalized balance $\mathsf{f.b}$, the user must also reveal their pending balance counter cnt to the Proof of Balance (PoB) verifier. To better understand the trade-offs, let us make two simplifying assumptions *for now*: (i) the mediator (if any) has already affirmed the transaction, and (ii) the settlement contains only a single leg.

If $\mathsf{cnt} > 0$, this implies that there exists at least one transaction that is either pending or finalized. In this case, the user needs to take further action by pointing to the transactions and justifying the value of cnt. This is crucial from the balance verifying perspective because the transaction status determines whether the funds can still be reversed and, therefore, whether they could be potentially part of the user's balance or not. (i) if pending, reversion is still possible, and (ii) if finalized, reversion is not possible. Therefore, in this case, the sender should point to transactions and prove that cnt equals the sum of all legs in which user is involved either as a sender or receiver. This allows the PoB verifier to understand how many of legs out of cnt is irreversible and how much is reversible.

Therefore, in case $\mathsf{cnt} > 0$, the user needs to point to all finalized and pending transactions. If the user has many finalized transactions, pointing to all transactions becomes inefficient and also introduces information leakage. This is due to the ability of the PoB verifier to link every transaction in which the user has been involved, which is undesirable as the PoB verifier needs to (only) verify the the balance. To address this, the user can scan the ledger for their transactions whose status is finalized, and submit a specific transaction called counter-update (cu) transaction ($\mathsf{tnx}_{\mathsf{cu}}$), to reduce cnt, thereby updating the transaction status. Note that $\mathsf{tnx}_{\mathsf{cu}}$ points to a finalized transaction and any two transactions $\mathsf{tnx}_{\mathsf{cu}}$ and $\mathsf{tnx}^*_{\mathsf{cu}}$ are unlinkable, as they reference different finalized transactions and user's account state transition occurs anonymously via the accumulator membership proof. In other words, the account state transition in $\mathsf{tnx}_{\mathsf{cu}}$ provides unlinkability similar to all other transactions.

Submitting $\mathsf{tnx}_{\mathsf{cu}}$ avoids referencing finalized transactions for the PoB verifier and limits the pointer to only pending transactions.

The counter cnt and $\mathsf{tnx}_{\mathsf{cu}}$ transaction (which reduces cnt) are only relevant for PoB. In other words, if a user knows that a specific asset type they hold will never be balance-verified, there is no need for the user to submit $\mathsf{tnx}_{\mathsf{cu}}$ to reduce cnt. The sole purpose of maintaining cnt in the user's account is to support PoB. Submission of $\mathsf{tnx}_{\mathsf{cu}}$ can be done, in a single transaction, pointing to all finalized transactions and transitioning the account state by submitting a large $\mathsf{tnx}_{\mathsf{cu}}$. This reduces their counter by the sum of number of all legs of all finalized transactions in one step, offering better efficiency. However, this approach introduces linkability, as all finalized transactions are linked to each other through the single $\mathsf{tnx}_{\mathsf{cu}}$. Alternatively, for better privacy, the user can submit separate $\mathsf{tnx}_{\mathsf{cu}}$ transactions for each finalized transaction, reducing their counter sequentially.

While this approach avoids linking finalized transactions, it comes at the cost of efficiency.
Note that pointing to pending transactions still introduces privacy leakage, as the PoB verifier will link all these pending transactions. To mitigate this, the user can reverse all pending transactions using the *Reversion* algorithm, reducing cnt and thus avoiding the need to link pending transactions. Reversing transactions, however, may be inefficient, as it would require reinitiating the transactions after completing the PoB protocol (assuming that the user is still interested in completing these transactions).
The incorporation of an asset-specific auditor in DART allows auditors to construct a decrypted view of the ledger for all transactions involving a specific asset type. Collaboration between the asset-specific auditor and the PoB verifier provides the necessary information to the verifier[15]. In the following, we present our second PoB approach, which operates independently of the auditor.
Upon receiving an instruction to prove the balance of the account associated with an asset type $\mathsf{a.t}$, the user calls $\mathcal{F}_{\mathsf{Ledger}}$ to identify all transactions whose status has changed from pending to finalized. Then, the user submits $\mathsf{tnx_{cu}}$ to the ledger, where the user's account transitions to a new state, with the counter decreased by the number of legs whose status is finalized. The user proves, in ZK, that they are transaction counterparty in those. The submission of $\mathsf{tnx_{cu}}$ is done sequentially to avoid likability.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{cu}} = (\mathsf{sk_{acct}}, \mathsf{f.b_{old}}, \mathsf{cnt_{old}}, \rho_{\mathsf{old}}, s_{\mathsf{old}}, \mathsf{acct^{cm}_{old}}, \rho_{\mathsf{new}}, s_{\mathsf{new}}, \mathsf{pk^{r}_{en}}, v, \mathsf{a.t}, \pi_{\mathsf{ACCU}})$
- $x_{\mathsf{cu}} = (\mathsf{v_{ACCU}}, \mathsf{nul_{old}}, \mathsf{acct^{cm}_{new}}, \mathsf{leg}, \mathsf{tnx^{id}_{settl}})$.
- The relation $\mathcal{R}_{\mathsf{cu}}(x_{\mathsf{cu}}, w_{\mathsf{cu}})$ is defined as:

$$\mathcal{R}_{\mathsf{cu}} = \Big\{ \mathsf{ACCU.VfyMem}(\mathsf{pp_{ACCU}}, \mathsf{v_{ACCU}}, \mathsf{acct^{cm}_{old}}, \pi_{\mathsf{ACCU}}) = 1$$
$$\wedge\ \mathsf{acct^{cm}_{new}} = \mathsf{Com}(\mathsf{sk_{acct}}, \mathsf{f.b_{old}}, \mathsf{cnt_{old}} - 1, \mathsf{a.t}, \rho, \rho^{(n+3)}; s^{(2^{n+1})})$$
$$\wedge\ \mathsf{acct^{cm}_{old}} = \mathsf{Com}(\mathsf{sk_{acct}}, \mathsf{f.b_{old}}, \mathsf{cnt_{old}}, \mathsf{a.t}, \rho, \rho^{(n+2)}; s^{(2^{n})})$$
$$\wedge\ \mathsf{nul_{old}} = g^{\rho^{(n+2)}}$$
$$\wedge\ (\mathsf{pk_{acct}}, \mathsf{sk_{acct}}) \in \mathsf{Acc.KeyGen}(1^{\lambda})$$
$$\wedge\ \mathsf{leg} \Big\}.$$

$\mathsf{tnx_{cu}}$ Generation

▷ Upon receiving $(\mathsf{balance}, \mathsf{sid}, \mathsf{a.t})$ from $\mathcal{Z}$, proceed as follows.
▷ Ignore if $(\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ is not recorded. Otherwise, parse: $\mathsf{addr_{pk}} = (\mathsf{pk_{acct}}, \mathsf{pk_{en}})$, $\mathsf{addr_{sk}} = (\mathsf{sk_{acct}}, \mathsf{sk_{en}})$.

[15] Note that this differs from our first PoB approach (Section 6.3.10), in which the auditor selectively decrypts transactions explicitly pointed to by the user.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Retrieve}, \mathsf{sid})$. Upon receiving $(\mathtt{Retrieved}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.
▷ Retrieve the recorded entry $(\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \mathsf{acct}_{\mathsf{old}})$ where the old account is: $\mathsf{acct}_{\mathsf{old}} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}}, \mathsf{a.t}, \rho_{\mathsf{old}}, s_{\mathsf{old}})$.
▷ Retrieve all entries $(o^i_1, o^i_2, \mathsf{TNX}^i)$ in LT such that, upon parsing: $o^i_2 = (\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}_i}_{\mathsf{acct}}, v^i, \mathsf{a.t}^i)$, the conditions $\mathsf{a.t}^i = \mathsf{a.t}$ and $\mathsf{pk}^{\mathsf{s}_i}_{\mathsf{acct}} = \mathsf{pk}_{\mathsf{acct}}$ hold.
▷ For each such entry $(o^i_1, o^i_2, \mathsf{TNX}^i)$, if there is any transaction $\mathsf{TNX}^j \in \mathsf{state}'$ where $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$, $f(\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}) = 1$, and $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} = \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$, proceed as follows:
  – Parse: $o^j_2 = (\mathsf{pk}_{\mathsf{acct}}, \cdot, v^j, \mathsf{a.t})$.
  – Compute the nullifier: $\mathsf{nul}_{\mathsf{old}} = g^{\rho^{(n+2)}}$.
  – Set the new account: $\mathsf{acct}_{\mathsf{new}} \leftarrow (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}} - 1, \mathsf{a.t}, \rho, \rho^{(n+3)}, s^{(2^{n+1})})$.
  – Compute the new account commitment: $\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}} \leftarrow \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}_{\mathsf{old}}, \mathsf{cnt}_{\mathsf{old}} - 1, \mathsf{a.t}, \rho, \rho^{(n+3)}; s^{(2^{n+1})})$.
  – Compute membership proof: $\pi_{\mathsf{ACCU}} \leftarrow \mathsf{ACCU.PrvMem}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{state}', \mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}})$.
  – Call $\mathcal{F}_{\mathsf{NIZK}}$ with $(\mathtt{Prove}, \mathsf{sid}, x_{\mathsf{cu}}, w_{\mathsf{cu}})$, and receive $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{cu}})$.
  – Set the cu transaction: $\mathsf{tnx}_{\mathsf{cu}} \leftarrow (x_{\mathsf{cu}}, \pi_{\mathsf{cu}})$.
  – Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Append}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{cu}})$.
  – Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Retrieve}, \mathsf{sid})$. Upon receiving $(\mathtt{Retrieved}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, if $\mathsf{tnx}_{\mathsf{cu}} \in \mathsf{state}'$, and $\mathsf{state}' \in \mathsf{state}$ update: $(\mathsf{acct}^{\mathsf{cm}}_{\mathsf{old}}, \mathsf{acct}_{\mathsf{old}}) \leftarrow (\mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{acct}_{\mathsf{new}})$. and update the list: $\mathsf{LT} \leftarrow \mathsf{LT} \setminus \{(o^j_1, o^j_2, \mathsf{TNX}^j)\}$. Otherwise, ignore.
▷ Proceed by executing the $\mathsf{tnx}_{\mathsf{PoB}}$ generation algorithm (Figure 10).
Note: There is no output to $\mathcal{Z}$ yet.

$\mathsf{tnx}_{\mathsf{cu}}$ Verification

▷ Run $\textsc{Validate}_{\mathsf{cu}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{cu}})$:
  1. Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{cu}} = (x_{\mathsf{cu}}, \pi_{\mathsf{cu}})$, $x_{\mathsf{cu}} = (\mathsf{v}_{\mathsf{ACCU}}, \mathsf{nul}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{leg}^j, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}})$. Given $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$, retrieve: $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$ where $\mathsf{TNX}^j \in \mathsf{state}'$. parse: $\mathsf{tnx}^j_{\mathsf{settl}} = (x^j_{\mathsf{settl}}, \pi^j_{\mathsf{settl}})$, $x^j_{\mathsf{settl}} = (\tilde{\mathsf{v}_{\mathsf{ACCU}}}^j, \mathsf{leg}^{*j})$
  2. Ignore if any of the following conditions hold:
     – $\mathsf{leg}^{*j} \neq \mathsf{leg}^j$
     – $\mathsf{nul}_{\mathsf{old}} \in \mathbb{L}_{\mathsf{NUL}}$.
     – $f(\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}) \neq 1$.
     – Upon calling $\mathcal{F}_{\mathsf{NIZK}}$ with: $(\mathtt{Verify}, \mathsf{sid}, x_{\mathsf{cu}}, \pi_{\mathsf{cu}})$, the output is: $(\mathtt{Vrfed}, \mathsf{sid}, 0)$.

     Otherwise, output 1.
▷ Run $\textsc{Update}_{\mathsf{cu}}(\mathsf{state}, \mathsf{tnx}_{\mathsf{cu}})$:

1. Parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$, $\mathsf{tnx}_{\mathsf{cu}} = (x_{\mathsf{cu}}, \pi_{\mathsf{cu}})$, $x_{\mathsf{cu}} = (\mathsf{v}_{\mathsf{ACCU}}, \mathsf{nul}_{\mathsf{old}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}}, \mathsf{leg}^j, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}})$. Given $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$, retrieve: $\mathsf{TNX}^j = (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$ where $\mathsf{TNX}^j \in \mathsf{state}'$.
2. Update: $\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}} \leftarrow \mathsf{update}(\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$, $\mathsf{TNX}^j \leftarrow (\cdot, \cdot, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$, $\mathsf{state}' \leftarrow \mathsf{state}' \cup \{\mathsf{TNX}^j\}$.
3. Call: $\mathsf{v}_{\mathsf{ACCU}} \leftarrow \mathsf{Add}(\mathsf{pp}_{\mathsf{ACCU}}, \mathsf{v}_{\mathsf{ACCU}}, \mathsf{acct}^{\mathsf{cm}}_{\mathsf{new}})$. Set: $\mathbb{L}_{\mathsf{NUL}} \leftarrow \mathbb{L}_{\mathsf{NUL}} \cup \{\mathsf{nul}_{\mathsf{old}}\}$, $\mathsf{state} \leftarrow \mathsf{state} \cup \{(\mathsf{state}', \cdot, \cdot, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})\}$.
4. Output state.

Fig. 9: Counter Update (cu) Transaction $\mathsf{tnx}_{\mathsf{cu}}$

Finally, the user executes PoB generation algorithm described in 10.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{PoB}} = (\mathsf{sk}_{\mathsf{acct}}, \rho, s, \overrightarrow{o_2}, \{\mathsf{pk}^{\mathsf{r}_k}_{\mathsf{acct}}\}_k)$
- $x_{\mathsf{PoB}} = \big(\mathsf{nul},\ \mathsf{acct}^{\mathsf{cm}},\ \mathsf{pk}_{\mathsf{acct}},\ \mathsf{f.b},\ \mathsf{cnt},\ V^{\mathtt{sender}}_{(0)},\ V^{\mathtt{receiver}}_{(0)},\ \mathsf{a.t},\ \overrightarrow{\mathsf{leg}}^{\mathtt{sender}},\ \overrightarrow{\mathsf{leg}}^{\mathtt{receiver}},\ \overrightarrow{\mathsf{tnx}}^{\mathtt{sender},\mathsf{id}}_{\mathsf{settl}},\ \overrightarrow{\mathsf{tnx}}^{\mathtt{receiver},\mathsf{id}}_{\mathsf{settl}}\big)$
- The relation $\mathcal{R}_{\mathsf{PoB}}(x_{\mathsf{PoB}}, w_{\mathsf{PoB}})$ is defined as:

$$
\begin{aligned}
\mathcal{R}_{\mathsf{PoB}} = \Big\{ & \mathsf{acct}^{\mathsf{cm}} = \mathsf{Com}(\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t}, \rho, \rho^{(n+2)}; s^{(2^n)}) \\
& \wedge \mathsf{nul} = g^{\rho^{(n+2)}} \\
& \wedge \mathsf{leg}^k = \mathsf{Enc}(o_2^k) \quad \forall o_2^k \in \overrightarrow{o_2} \\
& \wedge |\overrightarrow{o_2}| = \mathsf{cnt} \\
& \wedge \sum_{k^{\mathtt{sender}}} o_2^{k^{\mathtt{sender}}}.v = V^{\mathtt{sender}}_{(0)} \\
& \wedge \sum_{k^{\mathtt{receiver}}} o_2^{k^{\mathtt{receiver}}}.v = V^{\mathtt{receiver}}_{(0)} \\
& \wedge \sum_{k^{\mathtt{sender}}} o_2^{k^{\mathtt{sender}}} + \sum_{k^{\mathtt{receiver}}} o_2^{k^{\mathtt{receiver}}} = \overrightarrow{o_2} \\
& \wedge \overrightarrow{\mathsf{leg}}^{\mathtt{sender}} \cup \overrightarrow{\mathsf{leg}}^{\mathtt{receiver}} = \overrightarrow{\mathsf{leg}} \\
& \wedge \overrightarrow{\mathsf{tnx}}^{\mathtt{sender},\mathsf{id}}_{\mathsf{settl}} \cup \overrightarrow{\mathsf{tnx}}^{\mathtt{receiver},\mathsf{id}}_{\mathsf{settl}} = \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{\mathsf{settl}} \\
& \wedge (\mathsf{pk}_{\mathsf{acct}}, \mathsf{sk}_{\mathsf{acct}}) \in \mathsf{Acc.KeyGen}(1^\lambda) \Big\}.
\end{aligned}
$$

$\mathsf{tnx}_{\mathsf{PoB}}$ Generation

▷ Execute steps 1 to 4 of the $\mathsf{tnx}_{\mathsf{cu}}$ generation algorithm (Figure 9).
▷ For each $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}}$, retrieve all $\mathsf{TNX}^j \in \mathsf{state}'$ where:
- $\mathsf{TNX}^j = (\mathsf{tnx}^j_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}})$.
- $f(\mathsf{tnx}^{\mathsf{st}_j}_{\mathsf{settl}}) = 1.$[a]

– $\mathsf{tnx}^{\mathsf{id}_i}_{\mathsf{settl}} = \mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$.

▷ Let the aggregation of o_2^j, leg^j, and $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}$ for all j be denoted as: $\overrightarrow{o_2}$, $\overrightarrow{\mathsf{leg}}$, and $\overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{\mathsf{settl}}$, respectively.

▷ Retrieve the recorded entry $(\mathsf{acct}^{\mathsf{cm}}, \mathsf{acct})$ where $\mathsf{acct} = (\mathsf{sk}_{\mathsf{acct}}, \mathsf{f.b}, \mathsf{cnt}, \mathsf{a.t}, \rho, \rho^{(n+2)}; s^{(2^n)})$.

▷ Compute the nullifier: $\mathsf{nul} \leftarrow g^{\rho^{(n+2)}}$.

▷ Compute $V^{\mathtt{sender}}_{(0)}$, $V^{\mathtt{receiver}}_{(0)}$.

▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with $(\mathtt{Prove}, \mathsf{sid}, x_{\mathsf{PoB}}, w_{\mathsf{PoB}})$, and receive: $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{PoB}})$.

▷ Set the PoB transaction: $\mathsf{tnx}_{\mathsf{PoB}} \leftarrow (x_{\mathsf{PoB}}, \pi_{\mathsf{PoB}})$.

▷ Output: $(\mathsf{PoB}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{PoB}})$ to $\mathcal{Z}$.

$\mathsf{tnx}_{\mathsf{PoB}}$ **Verification** (off-chain run by PoB verifier)

▷ Upon receiving $(\mathtt{VerifyPoB}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{PoB}})$ from $\mathcal{Z}$ proceed as follows.

▷ Parse: $\mathsf{tnx}_{\mathsf{PoB}} = (x_{\mathsf{PoB}}, \pi_{\mathsf{PoB}})$, $x_{\mathsf{PoB}} = \big(\mathsf{nul},\ \mathsf{acct}^{\mathsf{cm}},\ \mathsf{pk}_{\mathsf{acct}},\ \mathsf{f.b},\ \mathsf{cnt},\ V^{\mathtt{sender}}_{(0)},\ V^{\mathtt{receiver}}_{(0)},\ \mathsf{a.t},\ \overrightarrow{\mathsf{leg}}^{\mathtt{sender}}, \overrightarrow{\mathsf{leg}}^{\mathtt{receiver}},\ \overrightarrow{\mathsf{tnx}}^{\mathtt{sender},\mathsf{id}}_{\mathsf{settl}},\ \overrightarrow{\mathsf{tnx}}^{\mathtt{receiver},\mathsf{id}}_{\mathsf{settl}}\big)$.

▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Retrieve}, \mathsf{sid})$. Upon receiving $(\mathtt{Retrieved}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, parse: $\mathsf{state} = (\mathsf{state}', \mathbb{L}_{\mathsf{IAA}}, \mathbb{L}_{\mathsf{KA}}, \mathbb{L}_{\mathsf{NUL}}, \mathsf{v}_{\mathsf{ACCU}})$.

▷ For each j, $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}} \in \overrightarrow{\mathsf{tnx}}^{\mathsf{id}}_{\mathsf{settl}}$, ignore if any of the following conditions hold:

- $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}} \notin \mathsf{state}'$.
- $f(\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}}) \neq 1$.
- Given:
  - $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}} \in \overrightarrow{\mathsf{tnx}}^{\mathtt{sender},\mathsf{id}}_{\mathsf{settl}}$ or
  - $\mathsf{tnx}^{\mathsf{id}_j}_{\mathsf{settl}} \in \overrightarrow{\mathsf{tnx}}^{\mathtt{receiver},\mathsf{id}}_{\mathsf{settl}}$

  $\mathsf{leg}^{*j} \neq \mathsf{leg}^j$ where $\mathsf{leg}^{*j} \in \mathsf{state}'$ and:
  - $\mathsf{leg}^j \in \overrightarrow{\mathsf{leg}}^{\mathtt{sender}}$, or
  - $\mathsf{leg}^j \in \overrightarrow{\mathsf{leg}}^{\mathtt{receiver}}$

▷ Otherwise, ignore if any of the following conditions hold:

- $\mathsf{nul} \in \mathbb{L}_{\mathsf{NUL}}$.
- Upon calling $\mathcal{F}_{\mathsf{NIZK}}$ with $(\mathtt{Verify}, \mathsf{sid}, x_{\mathsf{PoB}}, \pi_{\mathsf{PoB}})$, the output is $(\mathtt{Vrfed}, \mathsf{sid}, 0)$.

▷ Otherwise, output: $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{f.b}, \mathsf{cnt}, V^{\mathtt{sender}}_{(0)}, V^{\mathtt{receiver}}_{(0)})$.

[a] After the execution of $\mathsf{tnx}_{\mathsf{cu}}$ generation algorithm (Figure 9), the user counter only shows the number of pending legs.

Fig. 10: Proof of Balance Transaction $\mathsf{tnx}_{\mathsf{PoB}}$ (off-chain operation)

6.3.12 Mediator Operation In this step, the mediator gains access to the concealed details of a specific transaction whose asset type corresponds to their

public key (the same applies to the auditor as well) and submits their affirmation bit. The algorithm is illustrated in Figure 11.
The NIZK witness, statement, and relation are defined as follows.

- $w_{\mathsf{mediator}} = \mathsf{sk}^{\mathsf{A}}_{\mathsf{en}}$.
- $x_{\mathsf{mediator}} = (C^{\mathsf{A}}_{r_1}, C^{\mathsf{s}}_{\mathsf{leg}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, b_{\mathsf{A}})$.
- The relation $\mathcal{R}_{\mathsf{mediator}}(x_{\mathsf{mediator}}, w_{\mathsf{mediator}})$ is defined as:

$$\mathcal{R}_{\mathsf{mediator}} = \Big\{ C^{\mathsf{A}}_{r_1} = (\mathsf{pk}^{\mathsf{A}}_{\mathsf{en}})^{r_1} \wedge\ C^{\mathsf{s}}_{\mathsf{leg}} = g^{r_1} \cdot \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}} \Big\}.$$

▷ Upon receiving $(\mathsf{audit}, \mathsf{sid}, \mathsf{addr}^{\mathsf{A}}_{\mathsf{pk}})$ from $\mathcal{Z}$, proceed as follows.
▷ Ignore if $(\mathsf{addr}^{\mathsf{A}}_{\mathsf{pk}}, \mathsf{addr}^{\mathsf{A}}_{\mathsf{sk}})$ is not recorded. Otherwise, parse: $\mathsf{addr}^{\mathsf{A}}_{\mathsf{sk}} = (\mathsf{sk}^{\mathsf{A}}_{\mathsf{acct}}, \mathsf{sk}^{\mathsf{A}}_{\mathsf{en}})$.
▷ Call the Ledger Scan Algorithm with: $(\mathsf{Scan}, \mathsf{sid}, \mathsf{addr}^{\mathsf{A}}_{\mathsf{pk}})$. Upon receiving (o_1, o_2, TNX), proceed as follows.
▷ Parse: $o_1 = (g^{r_1}, \cdots, g^{r_4})$, $o_2 = (\mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{A}}_{\mathsf{aff}}, v, \mathsf{a.t})$, $\mathsf{TNX} = (\mathsf{tnx}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{id}}_{\mathsf{settl}}, \mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}})$, $\mathsf{tnx}_{\mathsf{settl}} = (x_{\mathsf{settl}}, \pi_{\mathsf{settl}})$, $x_{\mathsf{settl}} = (\mathsf{v}_{\tilde{\mathsf{ACCU}}}, \mathsf{leg})$.
▷ Ignore if $f(\mathsf{tnx}^{\mathsf{st}}_{\mathsf{settl}}) \neq 1$.
▷ Otherwise, choose the affirmation bit: $b_{\mathsf{A}} \leftarrow \{0, 1\}$.
▷ Call $\mathcal{F}_{\mathsf{NIZK}}$ with $(\mathtt{Prove}, \mathsf{sid}, x_{\mathsf{mediator}}, w_{\mathsf{mediator}})$, and receive $(\mathsf{Proof}, \mathsf{sid}, \pi_{\mathsf{mediator}})$.
▷ Set the transaction: $\mathsf{tnx}_{\mathsf{mediator}} \leftarrow (x_{\mathsf{mediator}}, \pi_{\mathsf{mediator}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Append}, \mathsf{sid}, \mathsf{tnx}_{\mathsf{mediator}})$.
▷ Call $\mathcal{F}_{\mathsf{Ledger}}$ with $(\mathtt{Retrieve}, \mathsf{sid})$. Upon receiving $(\mathtt{Retrieved}, \mathsf{sid}, \mathsf{state})$ from $\mathcal{F}_{\mathsf{Ledger}}$, if $\mathsf{tnx}_{\mathsf{mediator}} \in \mathsf{state}'$ for $\mathsf{state}' \in \mathsf{state}$, output: $(\mathsf{audited}, \mathsf{sid}, \mathsf{pk}^{\mathsf{s}}_{\mathsf{acct}}, \mathsf{pk}^{\mathsf{r}}_{\mathsf{acct}}, v, \mathsf{a.t})$ to $\mathcal{Z}$.

Fig. 11: Mediator Algorithm (off-chain operation)

6.3.13 Freeze

▷ Given user account public key $\mathsf{pk}_{\mathsf{acct}}$ and asset type $\mathsf{a.t}$, retrieve $N = g_5^{\rho}$ and $\Omega = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{T}}}(s)$ from the local table.
▷ Compute $\mathsf{Dec}_{\mathsf{sk}_{\mathsf{T}}}(\mathsf{Enc}_{\mathsf{pk}_{\mathsf{T}}}(s)) = s$
▷ With the help of the auditor/mediator compute $g_6^{\rho^{(n+3)}}$ where $g_6^{\rho^{(n+3)}}$ is the nullifier for which $g_6^{\rho^{(n+3)}} \notin \mathbb{L}_{\mathsf{NUL}}$ holds.
Hence, the following holds:

$$\mathsf{acct}^{\mathsf{cm}}_{n+1} = \mathsf{pk}_{\mathsf{acct}} \cdot g_2^{\mathsf{f.b}_{n+1}} \cdot g_3^{\mathsf{cnt}_{n+1}} \cdot g_4^{\mathsf{a.t}} \cdot g_5^{\rho}, g_6^{\rho^{(n+3)}} \cdot g_7^{s^{(2^{n+1})}}$$

▷ Given user account public key $\mathsf{pk}_{\mathsf{acct}}$, retrieve $\mathsf{pk}_{\mathsf{en}}$. Compute

$$r_b \xleftarrow{\$} \mathbb{Z}_p, \quad \Omega_b^{\mathsf{freeze}} = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(\mathsf{f.b}_{n+1}) = (X_b, Y_b) = (\mathsf{pk}_{\mathsf{en}}^{r_b},\ g^{r_b} \cdot h^{\mathsf{f.b}_{n+1}})$$

Note: As we will see, r_b must be maintained for the following algorithms, since they rely on it. One way to generate such randomness is to apply a pseudorandom function (PRF) rather than ($r_b \xleftarrow{\$} \mathbb{Z}_p$) – for example, using the algorithm runner's secret key and a counter as input. This approach ensures that the randomness can be deterministically regenerated if it is ever lost.

▷ Set $w_{\mathsf{freeze}} = (r_b, \mathsf{f.b}_{n+1}, s, n, g_5^{\rho})$

▷ Set $x_{\mathsf{freeze}} = (\mathsf{acct}_{n+1}^{\mathsf{cm}}, \mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}}, \mathsf{a.t}, g_5^{\rho^{(n+3)}}, \Omega_b^{\mathsf{freeze}}, \mathsf{cnt}_{n+1})$

▷ The relation $\mathcal{R}_{\mathsf{freeze}}(x_{\mathsf{freeze}}, w_{\mathsf{freeze}})$ is defined as:

$$\mathcal{R}_{\mathsf{freeze}} = \Big\{ \Omega_b^{\mathsf{freeze}} = (\mathsf{pk}_{\mathsf{en}}^{r_b},\ g^{r_b} \cdot h^{\mathsf{f.b}_{n+1}})$$

$$\mathsf{acct}_{n+1}^{\mathsf{cm}} = \mathsf{pk}_{\mathsf{acct}} \cdot g_2^{\mathsf{f.b}_{n+1}} \cdot g_3^{\mathsf{cnt}_{n+1}} \cdot g_4^{\mathsf{a.t}} \cdot g_5^{\rho}, g_6^{\rho^{(n+3)}} \cdot g_7^{s^{(2^{n+1})}}$$

$$g_6^{\rho^{(n+3)}} \Big\}.$$

▷ Asset issuer signs $(\pi_{\mathsf{freeze}}, x_{\mathsf{freeze}})$ with their substrate public key registered for $\mathsf{a.t} \in x_{\mathsf{freeze}}$.

Fig. 12: Freeze

6.3.14 Settlement Clearance

▷ Based on the generated nullifiers at Figure 12 algorithm, identify every settlement in which the user has been a counterparty (either as a sender or a receiver) that still requires further action from the user. This includes settlements that are either:
- Pending, or
- Fully executed, but still require the user to come online to take action (e.g., to claim funds or update their counter, etc.)
- Rejected, but still require the user to come online to take action (e.g., to claim funds or update their counter, etc.)

▷ Once identified, if needed, extract the following encryptions from the settlements[a]:

$$C_{\mathsf{leg}}^{v} = R_3 \cdot h^{v}, \qquad C_{\mathsf{leg}}^{\mathsf{role}} = \begin{cases} R_1 \cdot \mathsf{pk}_{\mathsf{acct}} & \text{(sender)} \\ R_2 \cdot \mathsf{pk}_{\mathsf{acct}} & \text{(receiver)} \end{cases}, \qquad C_{\mathsf{leg}}^{\mathsf{a.t}} = R_4 \cdot h^{\mathsf{a.t}}.$$

– $\alpha \xleftarrow{\$} \mathbb{Z}_p, \quad \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(R_3) = (Q_3, W_3) = (g^{\alpha}, \mathsf{pk}_{\mathsf{en}}^{\alpha} \cdot R_3)$

– $\beta \xleftarrow{\$} \mathbb{Z}_p$, $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(v) = (X_v, Y_v) = (\mathsf{pk}_{\mathsf{en}}^{\beta}, g^{\beta} \cdot h^{v})$
– Set $w_{\mathsf{SettlClr}} = (\alpha, v, \beta)$
– Set $x_{\mathsf{SettlClr}} = (Q_3, W_3, C_{\mathsf{leg}}^{v}, C_{\mathsf{leg}}^{\mathsf{role}}, C_{\mathsf{leg}}^{\mathsf{a.t}}, (R_1 \text{ or } R_2), R_4, \mathsf{a.t}, X_v, Y_v, \mathsf{pk}_{\mathsf{acct}}, \mathsf{pk}_{\mathsf{en}}, \mathsf{tnx}_{\mathsf{settl}}^{\mathsf{id}})$
– The relation $\mathcal{R}_{\mathsf{SettlClr}}(x_{\mathsf{SettlClr}}, w_{\mathsf{SettlClr}})$ is defined as:

$$\mathcal{R}_{\mathsf{SettlClr}} = \Big\{ C_{\mathsf{leg}}^{v} = (W_3/(\mathsf{pk}_{\mathsf{en}}^{\alpha}) \cdot h^{v}) \\ Q_3 = g^{\alpha} \\ X_v = g^{\beta} \\ Y_v = \mathsf{pk}_{\mathsf{en}}^{\beta} \cdot h^{v} \Big\}.$$

– Asset issuer signs $(\pi_{\mathsf{SettlClr}}, x_{\mathsf{SettlClr}})$ with their substrate public key registered for $\mathsf{a.t} \in x_{\mathsf{freeze}}$.
– *Note: Similar to Figure 12, β can be generated by applying a pseudo-random function (PRF) rather than ($\beta \xleftarrow{\$} \mathbb{Z}_p$) – for example, using the algorithm runner's secret key and a counter as input.*

Observation. Because the group elements ($R = R_1$ or R_2) and R_4, the account public key $\mathsf{pk}_{\mathsf{acct}}$, and the asset type $\mathsf{a.t}$ are all revealed, anyone *outside* the zero-knowledge circuit can directly verify

$$C_{\mathsf{leg}}^{\mathsf{role}} \stackrel{?}{=} R \cdot \mathsf{pk}_{\mathsf{acct}} \quad \text{and} \quad C_{\mathsf{leg}}^{\mathsf{a.t}} \stackrel{?}{=} R_4 \cdot h^{\mathsf{a.t}}.$$

Hence no additional proof logic for these commitments is required inside the relation.

Chain can compute:

$$\begin{aligned}(X_b, Y_b) \cdot (X_v, Y_v) &= \big(\mathsf{pk}_{\mathsf{en}}^{r_b},\ g^{r_b}\, h^{\mathsf{f.b}_{n+1}}\big) \ \cdot\ \big(\mathsf{pk}_{\mathsf{en}}^{\beta},\ g^{\beta}\, h^{v}\big) \\ &= \big(\mathsf{pk}_{\mathsf{en}}^{r_b}\, \mathsf{pk}_{\mathsf{en}}^{\beta},\ g^{r_b}\, h^{\mathsf{f.b}_{n+1}}\, g^{\beta}\, h^{v}\big) \\ &= \big(\mathsf{pk}_{\mathsf{en}}^{r_b+\beta},\ g^{r_b+\beta}\, h^{\mathsf{f.b}_{n+1}+v}\big) \\ &= \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(\mathsf{f.b}_{n+1} + v) \\ &= \Omega_b^{\mathsf{SettlClr}}\end{aligned}$$

update counter

$$\mathsf{cnt}_{n+1} \leftarrow \mathsf{cnt}_{n+1} - 1$$

[a] for simplicity and without loss of generality we assume only one such settlement exists with one leg.

Fig. 13: Settlement Clearance

6.3.15 Force transfer

▷ Given

$$\Omega_b^{\mathsf{SettlClr}} = (X_b^{\mathsf{SettlClr}}, Y_b^{\mathsf{SettlClr}}) = \left(\mathsf{pk}_{\mathsf{en}}^{r_b+\beta},\ g^{r_b+\beta}\, h^{\mathsf{f.b}_{n+1}+v}\right)$$

apply force transfer:

$$r_1 \xleftarrow{\$} \mathbb{Z}_p, \quad \Omega_v^{(1)} = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}^{(1)}}(\nu_1) = \left(\left(\mathsf{pk}_{\mathsf{en}}^{(1)}\right)^{r_1},\ g^{r_1}\cdot h^{\nu_1}\right)$$

$$r_2 \xleftarrow{\$} \mathbb{Z}_p, \quad \Omega_v^{(2)} = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(\nu_2) = \left((\mathsf{pk}_{\mathsf{en}})^{r_2},\ g^{r_2}\cdot h^{\mathsf{f.b}_{n+1}+v-\nu_1}\right)$$

Note: In the case of key loss associated with $\mathsf{pk}_{\mathsf{en}}$, we are interested in setting $r_2 = 0$ and $\mathsf{f.b}_{n+1} + v = \nu_1$ in the ciphertext $\Omega_v^{(2)} = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{en}}}(\nu_2)$. This ensures that anyone can publicly verify that $\Omega_v^{(2)}$ encrypts a value of zero, confirming that the entire value $\mathsf{f.b}_{n+1} + v$ has been encrypted in $\Omega_v^{(1)}$.

▷ Set $w_{\mathsf{FrcTrns}} = (r_1, r_2, \nu_1, \beta + r_b, v + \mathsf{f.b}_{n+1})$

▷ Set $x_{\mathsf{FrcTrns}} = (\Omega_b^{\mathsf{SettlClr}}, \Omega_v^{(1)}, \Omega_v^{(2)}, \mathsf{pk}_{\mathsf{en}}^{(1)}, \mathsf{pk}_{\mathsf{en}})$

▷ The relation $\mathcal{R}_{\mathsf{FrcTrns}}(x_{\mathsf{FrcTrns}}, w_{\mathsf{FrcTrns}})$ is defined as:

$$\begin{aligned}\mathcal{R}_{\mathsf{FrcTrns}} = \Big\{ & \Omega_b^{\mathsf{SettlClr}} = \left(\mathsf{pk}_{\mathsf{en}}^{r_b+\beta},\ g^{r_b+\beta}\, h^{\mathsf{f.b}_{n+1}+v}\right) \\ & \Omega_v^{(1)} = \left(\left(\mathsf{pk}_{\mathsf{en}}^{(1)}\right)^{r_1},\ g^{r_1}\cdot h^{\nu_1}\right) \\ & \Omega_v^{(2)} = \left((\mathsf{pk}_{\mathsf{en}})^{r_2},\ g^{r_2}\cdot h^{\mathsf{f.b}_{n+1}+v-\nu_1}\right) \\ & 0 \le \mathsf{f.b}_{n+1} + v - \nu_1 \le \text{known-upper-bound} \\ & 0 \le \nu_1 \le \text{known-upper-bound} \\ \Big\}. & \end{aligned}$$

Fig. 14: Force transfer (issuer-side)

In the event of *Key Loss* or *Clawback*, a user may claim either $\Omega^{(1)}$ or $\Omega^{(2)}$.

Key Loss. Suppose a user has lost their previous secret key. In this case, the user generates a new key pair with public key $\mathsf{pk}_{\mathsf{en}}^{(1)}$. To recover their funds, they reference $\Omega^{(1)}$ and provide a proof of knowledge of the corresponding secret key $\mathsf{sk}_{\mathsf{en}}^{(1)}$. This allows them to claim the value encrypted under $\Omega^{(1)}$.

Clawback. Alternatively, if the user still possesses their secret key, they may claim via $\Omega^{(2)}$. By providing proof of knowledge of the original secret key $\mathsf{sk}_{\mathsf{en}}$, the user can claim the value encrypted under $\Omega^{(2)}$.

$$\Omega_v^{(1)} = \left((\mathsf{pk}_{\mathsf{en}}^{(1)})^{r_1},\ g^{r_1} \cdot h^{\nu_1}\right),$$
$$\Omega_v^{(2)} = \left((\mathsf{pk}_{\mathsf{en}})^{r_2},\ g^{r_2} \cdot h^{\mathsf{f.b}_{n+1}+v-\nu_1}\right).$$

User Account State Transition. The `UserAccountStateTransition` follows the same principles as described previously. In essence, the user demonstrates that, during their account state transition, the balance is updated according to the value v contained within the corresponding Ω encryption.

Fig. 15: Force transfer (user-side)

7 Discussion

7.1 Maintenance of $\mathbb{L}_{\mathsf{IAA}}$

The list $\mathbb{L}_{\mathsf{IAA}}$ with entries of the form $(\mathsf{pk}_{\mathsf{acct}}, \mathsf{a.t}, \mathsf{pk}_{\mathsf{en}}^{\mathsf{A}})$ is publicly maintained on-chain and can be updated arbitrarily. For instance, in practice, an issuer may choose to change the associated auditor/mediator A for their asset due to regulations. This could occur, for example, if the originally assigned auditor/mediator becomes unavailable or is revoked, prompting the issuer to update the their keys $\mathsf{pk}_{\mathsf{en}}^{\mathsf{A}}$ to another one. In such a case, the list will undergo an update. The issuer proves that they are the legitimate associated issuer for the specific $\mathbb{L}_{\mathsf{IAA}}$ entry to authorize this update. Consequently, the list can be updated arbitrarily, and this could invalidate a user's zero-knowledge proof if it was generated based on a previous state of the list that has since changed. To mitigate this issue, we propose introducing structured update intervals; specifically, updates to lists can be restricted to predefined times. Users can generate their zero-knowledge proofs with the assurance that the list will remain stable within a defined period.

7.2 Fee Payment

Fee payment must ensure DoS resistance, user privacy, and efficient proof generation. Since all on-chain actions incur costs, the design must balance simplicity, security, and performance. We evaluate three models for paying fees in POLYX (the native token) each offering different trade-offs between usability, efficiency, and privacy.

Approach 1: Native Currency in Each Asset Account

- **Description:** Each asset account (e.g., Tesla, Acme) holds POLYX (the native currency) to pay fees.
- **Pros:**
  - Fees can be bundled with asset transactions, potentially reducing proving costs.
- **Cons:**
  - Each account requires individual top-ups, creating management overhead.
  - May requires two proofs (fee + asset): to mitigate DoS, fee payment proof could be separated (while bound to the main proof), reducing efficiency.

Approach 2: Separate Fee-Paying Account

- **Description:** A dedicated account holds POLYX exclusively for fee payments.
- **Pros:**
  - Simplifies fee management with clear separation of assets and fees.
- **Cons:**
  - Requires a separate accumulator proof, increasing prover cost.

Approach 3: Third-Party Fee Payers

- **Description:** Intermediaries pay POLYX fees; users reimburse them using assets (e.g., Tesla).
- **Pros:**
  - No need for users to manage native currency balances.
- **Cons:**
  - Risk of abuse with junk assets and reliance on trust/markets for conversion.

Mainline: Approach 2 — Separate Fee Account (POLYX)

Relayer Flow.

1. The user generates a proof that covers both on-chain fees and the relayer commission. Since on-chain fees and the relayer commission are public values, they are not included in the witness of the NIZK proof.
2. The user anonymously submits the proof to the relayer. The relayer, who maintains an up-to-date state of the chain, verifies the proof. The proof consists of two parts: a fee proof, which includes the accumulator membership proof, and an affirmation-related proof. The fee proof is verified first to make DoS prevention more efficient.
3. The relayer signs and broadcasts the transaction.

4. The chain verifies the proof, starting with the first part (the fee proof), followed by the second part. The transaction is included in the block if the fee proof is valid, regardless of the validity of the second part.
5. The chain deducts the network fee from the relayer and reimburses it from the user's fee payment, which includes both the on-chain fee and the relayer commission.

References

1. Almashaqbeh, G., Solomon, R.: Sok: Privacy-preserving computing in the blockchain era. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). pp. 124–139. IEEE (2022)
2. Baldimtsi, F., Chalkias, K.K., Madathil, V., Roy, A.: Sok: Privacy-preserving transactions in blockchains. Cryptology ePrint Archive (2024)
3. Bellare, M., Boldyreva, A., Desai, A., Pointcheval, D.: Key-privacy in public-key encryption. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 566–582. Springer (2001)
4. Benarroch, D., Gillespie, B., Lai, Y.T., Miller, A.: Sok: Programmable privacy in distributed systems. Cryptology ePrint Archive (2024)
5. Biçer, O., Tschudin, C.: Oblivious homomorphic encryption. Cryptology ePrint Archive (2023)
6. BNP Paribas: T+1 trade affirmation and settlement (2024), available at: Link
7. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: International Conference on Financial Cryptography and Data Security. pp. 423–443. Springer (2020)
8. Buterin, V., et al.: Ethereum white paper. GitHub repository **1**(22-23), 5–7 (2013)
9. Campanelli, M., Hall-Andersen, M.: Veksel: Simple, efficient, anonymous payments with large anonymity sets from well-studied assumptions. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. pp. 652–666 (2022)
10. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
11. Chase, M., Lysyanskaya, A.: On signatures of knowledge. In: Advances in Cryptology-CRYPTO 2006: 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006. Proceedings 26. pp. 78–96. Springer (2006)
12. Chatzigiannis, P., Chalkias, K.: Proof of assets in the diem blockchain. In: Applied Cryptography and Network Security Workshops: ACNS 2021 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, and SiMLA, Kamakura, Japan, June 21–24, 2021, Proceedings. pp. 27–41. Springer (2021)
13. Chaum, D.: Blind signatures for untraceable payments. In: Advances in Cryptology: Proceedings of Crypto 82. pp. 199–203. Springer (1983)
14. Diamond, B.E.: Many-out-of-many proofs and applications to anonymous zether. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1800–1817. IEEE (2021)
15. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985)

16. Fauzi, P., Meiklejohn, S., Mercer, R., Orlandi, C.: Quisquis: A new design for anonymous cryptocurrencies. In: Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part I 25. pp. 649–678. Springer (2019)
17. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194 (1987)
18. Groth, J., Ostrovsky, R., Sahai, A.: Perfect non-interactive zero knowledge for np. In: Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25. pp. 339–358. Springer (2006)
19. Guo, Y., Karthikeyan, H., Polychroniadou, A., Huussin, C.: Pride ct: Towards public consensus, private transactions, and forward secrecy in decentralized payments. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 3904–3922. IEEE (2024)
20. Hansen, A.B., Nielsen, J.B., Simkin, M.: Ocash: fully anonymous payments between blockchain light clients. In: IACR International Conference on Public-Key Cryptography. pp. 169–202. Springer (2025)
21. JAP: Jap anonymity & privacy (2009), `http://anon.inf.tu-dresden.de/`
22. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros crypsinous: Privacy-preserving proof-of-stake. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 157–174. IEEE (2019)
23. Kiayias, A., Kohlweiss, M., Sarencheh, A.: Peredi: Privacy-enhanced, regulated and distributed central bank digital currencies. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 1739–1752 (2022)
24. Kumar, A., Fischer, C., Tople, S., Saxena, P.: A traceability analysis of monero's blockchain. In: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22. pp. 153–173. Springer (2017)
25. Madathil, V., Scafuro, A.: Prifhete: Achieving full-privacy in account-based cryptocurrencies is possible. Cryptology ePrint Archive (2023)
26. Meiklejohn, S., Orlandi, C.: Privacy-enhancing overlays in bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 127–141. Springer (2015)
27. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Proceedings of the 2013 conference on Internet measurement conference. pp. 127–140 (2013)
28. Metere, R., Dong, C.: Automated cryptographic analysis of the pedersen commitment scheme. In: Computer Network Security: 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2017, Warsaw, Poland, August 28-30, 2017, Proceedings 7. pp. 275–287. Springer (2017)
29. Micali, S., Rabin, M., Kilian, J.: Zero-knowledge sets. In: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings. pp. 80–91. IEEE (2003)

30. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: 2013 IEEE symposium on security and privacy. pp. 397–411. IEEE (2013)
31. Möser, M., Soska, K., Heilman, E., Lee, K., Heffan, H., Srivastava, S., Hogan, K., Hennessey, J., Miller, A., Narayanan, A., et al.: An empirical analysis of traceability in the monero blockchain. arXiv preprint arXiv:1704.04299 (2017)
32. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
33. Narula, N., Vasquez, W., Virza, M.: {zkLedger}:{Privacy-Preserving} auditing for distributed ledgers. In: 15th USENIX symposium on networked systems design and implementation (NSDI 18). pp. 65–80 (2018)
34. Noether, S., Mackenzie, A., et al.: Ring confidential transactions. Ledger **1**, 1–18 (2016)
35. Norton Rose Fulbright: SEC's Division of Trading and Markets Issues No-Action Letter. Published by Norton Rose Fulbright (October 2020), available at: Link
36. Payment Systems Regulator: Confirmation of payee - preventing app scams (2024), available at: Link
37. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991)
38. Sarencheh, A., Khoshakhlagh, H., Kavousi, A., Kiayias, A.: DART: Decentralized, anonymous, and regulation-friendly tokenization. Cryptology ePrint Archive, Paper 2025/239 (2025), `https://eprint.iacr.org/2025/239`
39. Sarencheh, A., Kiayias, A., Kohlweiss, M.: Parscoin: A privacy-preserving, auditable, and regulation-friendly stablecoin. In: International Conference on Cryptology and Network Security. pp. 289–313. Springer (2024)
40. Sasson, E.B., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE symposium on security and privacy. pp. 459–474. IEEE (2014)
41. The National Law Review: Time to Do the Foxtrot: The Three-Step SEC Establishes for Improved Process in Settlement of Digital Asset Trades. Published in The National Law Review (September 2020), available at: Link
42. The Securities Transfer Association (STA): Guidelines (2023), available at: Link
43. UK Finance: Confirmation of payee guidance (2024), available at: Link
44. U.S. Securities and Exchange Commission: FINRA ATS Role in Settlement of Digital Asset Security Trades (September 2020), available at: Link
45. Wüst, K., Kostiainen, K., Delius, N., Capkun, S.: Platypus: A central bank digital currency with unlinkable transactions and privacy-preserving regulation. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2947–2960 (2022)