ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Privacy and Auditability in Decentralized Payment Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΕΩΡΓΙΟΥ ΠΑΠΑΔΟΥΛΗ

Επιβλέπων
Αριστείδης Παγουρτζής
Καθηγητής Ε.Μ.Π

Αθήνα, Ιούνιος 2024

Contents

Chapter 1

Introduction

Privacy vs Auditability: Addressing this dilemma becomes urgent, as blockchain technology advances and decentralized digital payment systems (DPS) evolve and gain in popularity. This prominence of DPS brings about integration with the heavily regulated traditional financial systems. A major question that must be answered is to what extent this can be achieved without sacrificing privacy and decentralization.

All flavors of DPS have some built-in support for privacy and regulation, even if it is rudimentary. Starting with Bitcoin [19] all DPS share the common feature of depending on a globally distributed, append-only, public ledger to document monetary transactions in a transparent, verifiable and immutable manner. The underlying consensus mechanism used to settle exchange history and introduce new transactions, along with the security properties of the cryptographic primitives employed, makes sure that these systems adhere to some (simple) rules. Further auditing can be achieved by merely inspecting this ledger, as everything is in the clear. To protect their privacy, users rely on the use of renewable pseudonyms to obscure their identities (but not the amounts exchanged). It has been shown, though, that by combining publicly available data from the blockchain in a smart way [18], anyone could link the pseudoidentities of the users and even uncover their real-world identities.

To overcome this problem, privacy-enhanced cryptocurrencies (e.g. Zerocash [5], Monero [21], Quisquis [11]) arose. These systems hide transaction identities and amounts exchanged, thus providing privacy in a provable cryptographic manner. At the same time, however, they allow malicious users to conduct illegal activities (e.g. money laundering, unauthorized money transition, tax evasion). This misuse of privacy has led to the need for a compromise, i.e. the creation of protocols that combine user privacy and auditability. Such auditable privacy solutions [12, 9, 14, 15, 20] aim to guarantee that both the system and its participants comply with financial regulations and laws, preventing them from engaging in illicit activities without being accountable to the authorities. Financial regulations that are usually supported in such schemes are KYC (Know-Your-Customer), Anti Money Laundering (AML) as well as restrictions to the number or the value of transactions a single user can make, or the total value that can be exchanged in a single transaction.

Our proposal. We propose AQQUA: a system to equip DPS with auditability, without changing its decentralized, permissionless, and trustless nature. AQQUA extends the well-known Quisquis [11] DPS with mechanisms to allow the auditing of transactions and to enforce policies on the users. We achieve this by introducing two more entities to the system: A Registration Authority (`RA`) and an Audit Authority (`AA`). These authorities serve specific auditing-related goals, and do not interfere with day-to-day transactions.

In order to transact in AQQUA, users must first register to the `RA` and provide their real-world credentials, thus fulfilling KYC. They acquire a cryptographic pseudonym, a unique initial public key, which can be used to create new accounts within AQQUA. The `RA` maintains a mapping between the total number of accounts that belong to a particular user and their initial public key, but does not further interfere with monetary interchanges, i.e. the `RA` cannot censor users after they have enrolled to the system. To protect user privacy, the total number of accounts for each registered user is maintained in committed form.

In order to ensure anonymous participation, each user can subsequently create new accounts that are provably unlinkable to their registered public key. This is achieved by utilizing *updatable public keys*, introduced in [11], which allow the creation of new, provably indistinguishable and independent public keys, from an initial key pair, without changing the underlying secret counterpart. While each user can create accounts on their own, they must always update the number of accounts that they use with the `RA`. AQQUA accounts consist of commitments (for confidentiality) for the balance, the total amount of coins spent and the total amount of coins received in the corresponding updatable public key. In AQQUA, transactions can be thought of as 'wealth redistribution' between inputs and outputs, an idea originating from Quisquis [11]. Input accounts include the senders, the recipients as well as an anonymity set. Output accounts are new, updated but unlikable accounts for the senders, recipients, and decoys. To counter theft prevention, the sender proves in zero-knowledge that they have correctly updated the accounts and have not taken coins away from anyone except themselves.

Finally, the audit is executed by the `AA` asynchronously on the initial public key of each user. During auditing, each user should prove in zero-knowledge that for a specified period of time all of the their accounts are compliant to the system's policies using data that are only stored on-chain. Penalties for non-compliance can then be enforced to the users.

Chapter 2

Background

2.1 Cryptographic Preliminaries

This section describes some necessary cryptographic tools used in the following sections [1].

2.1.1 DLOG and DDH assumptions

In this section two important problem used extensively in cryptography, Discrete Logarithm Problem and Decisional Diffie-Hellman Problem are defined.

Definition 1. *Discrete Logarithm Problem (DLP)*
Let $\mathbb{G}$ be a finite cyclic group and g one of its generators. Given $h \in \mathbb{G}$ find $x < |\mathbb{G}|$ such that $g^x = h$.

Definition 2. *Decisional Diffie-Hellman Problem (DDHP)*
Let $\mathbb{G}$ be a finite cyclic group and g one of its generators. Given $g^\alpha, g^\beta, g^\gamma \in \mathbb{G}$ find if $\gamma = \alpha \cdot \beta$.

If it holds that $(G, \cdot) < (\mathbb{Z}_p^*, \cdot)$ and $|G| = q$ where p, q large prime numbers (at least 1024 bits) then the problems DLP, DDHP are considered computational difficult.

Definition 3. *Discrete Logarithm Assumption (DLOG)*
The discrete logarithm assumption holds in a group $\mathbb{G}$ if for all PPT algorithms $\mathcal{A}$ there exists a negligible function $negl_{\mathcal{A}}$ such that:

$$Pr[x \leftarrow \mathcal{A}(1^\lambda, \mathbb{G}, h, g)] \leq negl_{\mathcal{A}}(\lambda)$$

2.1.2 Public Key Encryption

The components of public key cryptography will be strictly defined in this section.

A cryptosystem is a tuple of the following three algorithms $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ such that if m a message then:

- $k \leftarrow \mathsf{KGen}()$
- $c \leftarrow \mathsf{Enc}(k, m)$

- $m \leftarrow \mathsf{Dec}(k, c)$
- Also it holds that $\mathsf{Dec}(\mathsf{Enc}(m)) = m$.

A public key encryption cryptosystem has the following properties:

The KGen algorithm produces a pair of keys, denoted as $(\mathsf{pk}, \mathsf{sk})$. sk is the private key and must remain hidden, while pk is the public key and is known to everyone using the system. When someone wants to send a message m to a particular recipient, the sender encrypts m using the Enc algorithm under the recipient's pk. Only the owner of the corresponding sk can use the Dec algorithm and decrypt the encrypted message to the original plaintext m.

The positive aspect of public key cryptography is that it does not require an exchange of keys between each user of the system; as soon as a new person joins the network, he only needs to publish his public key and can receive messages from any other user.

An example of public key encryption scheme is the ElGamal cryptosystem.

ElGamal

ElGamal is based on DLOG assumptions. It consists of the following algorithms:

- **Key-pair Generation** - $\mathsf{KGen}()$:
  1. Choose p, q large primes such that $q|(p-1)$ and a generator g of subgroup $\mathbb{G}$ with order q of group $\mathbb{Z}_p^*$.
  2. Choose random $x \in \mathbb{Z}_q$
  3. Calculate $y = g^x \bmod p$
  4. Return key-pair $(\mathsf{sk}, \mathsf{pk}) = (x, y)$
- **Encryption** - $\mathsf{Enc}(\mathsf{pk}, m)$:
  1. Sample $r \leftarrow\!\!\$\, \mathbb{Z}_q$
  2. Calculate $G = g^r \bmod p$
  3. Calculate $M = my^r \bmod p$
  4. Return ciphertext $c = (G, M)$
- **Decrypt** - $\mathsf{Dec}(\mathsf{sk}, c)$:
  1. Given ciphertext $c = (G, M)$ and the owner of $\mathsf{sk} = x$ can calculate $m = \frac{M}{G^x}$.

  It is obvious that $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(m)) = m$ since:

$$\frac{M}{G^x} = \frac{my^r}{(g^r)^x} = \frac{m(g)^r}{g^{xr}} = m$$

There is also a variation of ElGamal encryption that returns as message g^m instead of m that is called exponential ElGamal. This variation is offers also additive homomorphism.

2.1.3 Commitments

A commitment scheme is a tool that allows an entity to select a value without revealing it and without having the ability to change it. More specifically, a commitment scheme consists of the following algorithms:

- $ck \leftarrow \mathsf{KGen}(1^\lambda)$: Creates the public commitment key ck.
- $(c, o) \leftarrow \mathsf{Commit}_{ck}(m)$: Binds message m to c and produces an oppening o.
- $0/1 \leftarrow \mathsf{OpenCom}_{ck}(c, o, m)$: Checks if the commitment c corresponds to the message m.

A commitment scheme should have the following properties:

- **Hiding**: This property ensures that the commitment hides the value from the receiver until the committer chooses to reveal it. The receiver cannot derive any information about the committed value from the commitment alone.
- **Binding**: This property ensures that the committer cannot change the value after the commit. This property protects the receiver.

2.1.4 Σ-protocols

Let $\mathcal{R}$ be a binary relation for instances x and witnesses w, and let $\mathcal{L}$ be its corresponding language, i.e. $\mathcal{L} = \{x | \exists w : (x, w) \in \mathcal{R}\}$. A Σ-protocol for $\mathcal{R}$ is a three-move public-coin protocol between two PPT algorithms P, V, whose transcript consists of the following phases:

1. **Commit**: P commits to an initial message a and sends it to V.
2. **Challenge**: V sends a challenge c to P.
3. **Response**: P responds to the challenge with message z.

A Σ-protocol must satisfy the following properties:

- **Completeness**: if $x \in \mathcal{L}$, V always accepts the transcript.
- **Special Soundness**: given two transcripts with the same commitment and different challenges $(a, c, z), (a, c', z')$ one can efficient compute w such that $(x, w) \in \mathcal{R}$.
- **Special honest-verifier zero-knowledge (SHVZK)**: there exists a PPT simulator Sim that on input $x \in L$ and a honestly generated verifier's challenge c, outputs an accepting transcript of the form (a, c, z) with the same probability distribution as a transcript between honest P, V on input x.

2.1.5 Hash Functions

Hash Functions play fundamental role in modern cryptography. They map elements of a set with a large number of elements to another set with a smaller number of elements. Therefore, these functions are of the form of $H : X \rightarrow Y, |X| > |Y|$, where it is possible that $|X| = \infty$, while $|Y|$ can be a finite set. It is obvious, that exists some elements pf X that will be mapped tp the same elements of Y.

More specifically, a hash function is a function that has the following properties:

- **Compression**. The value $H(x)$ has a specific length for any input x.
- **Computationally efficient**. There is a deterministic polynomial-time algorithm A such that $H(x) = A(H, x)\ \forall x$.

A Hash Function need to several additional properties in order to be useful:

- **Pre-image Resistance**: Given a hash value y, it is computational hard to find x such that $H(x) = y$.
- **Second Pre-image Resistance**: Given an element x_1 and its hash value $H(x_1)$ it is computational difficult to find an element $x_2 \neq x_1$ such that $H(x_2) = H(x_1)$.
- **Collision Resistance**: It is computationally infeasible to find any two different inputs x_1, x_2 such that $H(x_1) = H(x_2)$.

2.1.6 Merkle Trees

The Merkle tree is a structure that can be represented graphically in the form of a regular binary tree. Its characteristic is that all information is stored in its leaves and each non-leaf node stores the hash of its children's values. Inductively, only one value is stored at the root of the tree, which is obtained in the way described above. It is obvious that the value of the root hash is influenced by all the data in the leaves and is somehow representative of all the information.

Because of the collision resistant property of hash function, if two merkle trees have the same root then with very high probability (approximately 1) they will contain the same data. If the hashes of the roots are not identical, then again there is a high probability that the information in the two trees is not the same, and a binary search (following the different hashes) can efficiently find - in logarithmic time to the size of the data - where the two data blocks differ.

Merkle trees can be used in order nodes to maintain a concise representation of shared data.

2.2 Blockchain

The introduction of blockchain technology with the launch of Bitcoin [19] in 2008 has revolutionized the way digital transactions are conducted and is the backbone of cryptocurrencies. Basically, blockchain is a distributed, immutable, append-only ledger that contains transactions across a peer-to-peer network.

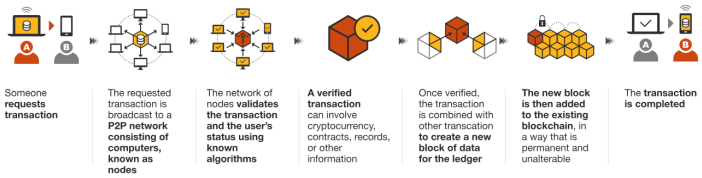


Figure 2.1: How blockchain works [16]

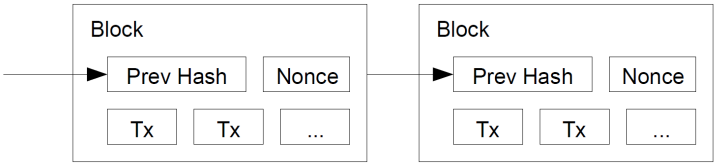


Figure 2.2: Blocks in Blockchain [19]

Before blockchain, traditional payment systems relied on a centralized authority, such as banks, to ensure the validation of transactions. Blockchain technology allows two willing parties to transact directly with each other without the need for a trusted third party. This is achieved by using cryptographic techniques and a consensus mechanism to ensure that once a transaction is added to the Blockchain, it cannot be altered or deleted. All transaction are publicly available, as well as publicly verifiable, and each participant, or node, maintains a copy of the entire blockchain. This eliminates the need for a central authority and reduces the risk of single points of failure Figure 2.1.

In payment systems, blockchain data contains the transactions of users. Transactions are stored in blocks, and each block is linked to the previous block by hashing its contents. In that way, a chain of blocks is created Figure 2.2.

Permissioned vs Permissionless

The consensus mechanism ensures that all participants agree on the same ledger. There are two categories of Blockchain, depending on whether those involved are known or not known to the system.

- **Permissioned**: Permissioned blockchains are private networks where access to the blockchain is restricted. In this type only authorized participants can join the network. This ensures that all participants are known and trusted.

- **Permissionless**: In a permissionless setting, anyone can join the network, participate in the consensus process, and view all transactions on the blockchain. However, in a permissionless blockchain, a mechanism to prevent sybil identities is required to ensure the validity of the consensus algorithm, such us Proof of Work (PoW) or Proof of Stake (PoS).

How to track ownership in Blockchain

In blockchains, there are basically two ways to do ownership tracking:

- **UTXO model**: In UTXO (Unspent Transaction Output) model transactions contains inputs and outputs. Transaction inputs are references to previous unspent transaction outputs, meaning that they are constructed from a transaction but have not been used as inputs in any previous transaction. Each transaction consumes its transaction inputs and generates new UTXOs, specifying the amount of the cryptocurrency and the recipient address. participants need to maintain all unspent transactions and balance is the sum of UTXOs destined to specific address.

- **Account-based model**: In the account-based model, each address has an account on the blockchain associated with a balance that is updated based on the currency transfer transactions that account issues or receives.

Chapter 3

Privacy in Payment Systems

3.1 Privacy

Privacy is a very useful and beneficial feature in payment systems. First and foremost, payment activities involve some sensitive personal information. Users usually do not want this information to be disclosed in order to protect their personal information, as it can be used to discriminate against individuals based on their financial history or purchasing habits.

Apart from the protection of personal data, the lack of privacy raises concerns in other areas as well [2]. It also has a negative impact on fairness by enabling front-running attacks [10]. In these attacks, a malicious individual monitors the transactions of other users and races to issue his own transaction, aiming to have it confirmed first. An example could be racing to win an auction by exploiting this advantage. It can also create "tainted" currency. That is, coins that no one wants to own because they are associated with an undesirable coin history, such as being part of an illegal trade.

The first level of privacy used by Bitcoin and most existing cryptocurrencies is pseudonymity. In this approach, transactions hide neither the participants nor the values transferred. Privacy relies on pseudonymous addressing, which aims to break the link between the addresses of the system and the identities of real users. Payment systems that use the permissionless setting allow and encourage users to have more than one such pseudonym. However, even this provides a very weak anonymity guarantee. Various de-anonymization attacks have been proposed in the literature [17], [22], [24], [25], based on clustering by analyzing either the inputs and outputs of the transactions or the behavior of the users [3].

Stronger privacy guarantees include the following properties: confidentiality and anonymity. Confidentiality hides the transaction amounts, while anonymity hides the pseudonyms of the real participants in a transaction. There are two levels of anonymity, set anonymity and full anonymity. In set anonymity, the user's identity is either one of n possible identities, where n is the size of the set. Full anonymity is provided when the participants can be any user of the system.

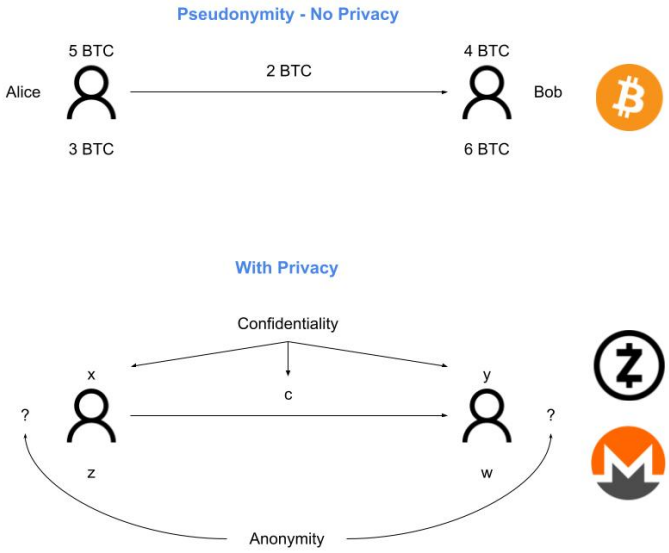


Figure 3.1: Privacy in Payment Systems

To implement privacy properties, private payment systems use the following basic idea:

- confidentiality is achieved through homomorphic commitments, which aim to hide the amounts transacted as well as the values of users' accounts.
- Anonymity is achieved through anonymity sets (e.g., shielded pools, ring signatures) that hide the pseudonyms of the real participants in a transaction.

In addition, a necessary tool for private payment systems is zero-knowledge proof, which aims to ensure the validity of transactions without revealing the above information.

3.2 Private Schemes

Some typical examples of private payment schemes (Zerocash [5], Monero [21] and Quisquis [11]) are presented below:

3.2.1 Zcash

Zerocash [5] is a novel proposal for a cryptographic protocol that addresses the inherent privacy limitations of bitcoin. It introduces the concept of decentralized anonymous payment (DAP) scheme, meaning a digital currency system where all transactions are guaranteed to be completely anonymous. That is, the origin, destination, and amount are all hidden from the public ledger.

Zerocash achieves privacy through the use of Zero-Knowledge Succinct, Non-interactive Arguments of Knowledge (zk-SNARKs). This cryptographic tool is

essential to prove that transactions are valid, without revealing any information about the parties involved in a transaction and the amounts involved.

More specifically, Zerocash functions on top of any ledger-based currency (e.g. Bitcoin). First, each user generates an arbitrary number of address key pairs. These contain a public key pk that allows others to send payments to the user, and a secret key sk that is used to receive payments sent to the corresponding public key. Then Zerocash gives the ability to users to convert regular coins into anonymous coins through the mint functionality. Afterwards, users can spend (move) this coins in transactions without revealing any information.

Mint coins

When minting a coin, a user generates a secret trapdoor ρ, used to create a random serial number unique for each coin (through a PRF) and two randomnesses r, s. Then they create a commitment to the value of the coin, the recipient's address and the serial number in the two following steps. First, user computes an intermediate commitment k using randomness r for the concatenation of the recipient address and the trapdoor ρ, $k = \mathtt{com}(\mathsf{pk}||\rho; r)$. In the second step user creates a commitment using randomness s for the concatenation of the coin's value and the intermediate commitment k, $cm = \mathtt{com}(\mathtt{v}||k; s)$. Finally the user publish on the blockchain the mint transaction $\mathtt{tx}_{MINT} = (\mathtt{v}, k, s, cm)$.

Anyone can verify that cm is a commitment to a coin of value v by checking whether $\mathtt{com}(\mathtt{v}||k; s)$ is equal to cm. However, they will not be able to distinguish between the owner pk and the serial number, since these values are hidden inside the commitment k.

All of the minted coins are stored in a "shielded pool" and are not deleted when spend in order to provide anonymity.

Pour coins (Spend functionality)

In order to spend a coin anonymously, the user produces a zero-knowledge proof (using zk-SNARK) for the statement: "I know the one opening (ρ, r) of a coin commitment of the "shielded pool" and the corresponding secret key sk such that $k = \mathtt{com}(\mathsf{pk}||\rho; r)$ and $sn = PRF(\mathsf{sk}, \rho)$.

After that user reveal the serial number sn. The serial number is used in order to handle double spending. Zerocash allocates a unique sequence number to each coin, which is published on-chain when the coin is spent. Therefore, an attempt at double spending is indicated by a new transaction that produces a sequence number(s) that has already been published.

Finally, user creates a new coin for the new recipient address with a new serial number.

After confirming and recording the pour transaction in the ledger, the new coins can be used for future transactions. Throughout this process, the use of zk-SNARKs and commitments guarantees that all details of the transaction, including the coin spent and the recipient of the new coin, remain private and secure. This process maintains user anonymity while preserving the integrity and validity of transactions within the Zerocash system.

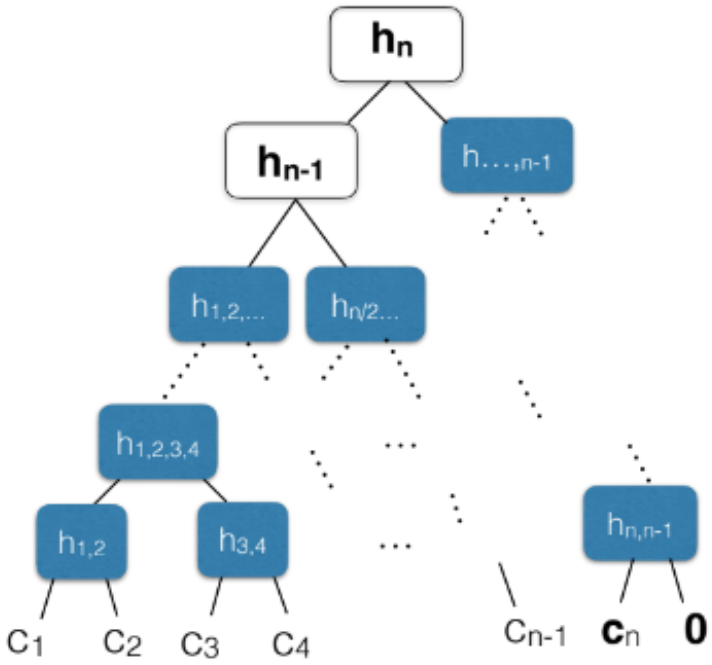


Figure 3.2: Zerocash Merkle Tree of Coin Commitments

Merkle Tree

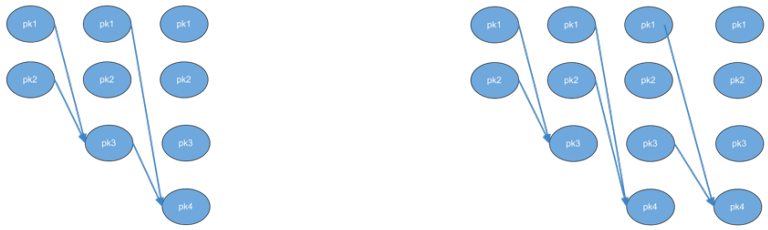
As mentioned above, to ensure anonymity, no one can tell which coin was used in each transaction. Thus, the spent coins cannot be removed from the "shielded pool". As a result, its size increases with each transaction. Therefore, the naive approach to implement it as a list of coin commitments will lead to a linear growth for the blockchain size.

A more efficient implementation is the use of Merkle trees. Zerocash maintains an efficiently updateable Merkle tree over the growing coin commitment list Figure 3.2. As a result, the space complexity is reduced to logarithmic.

3.2.2 Monero

Monero [21] is also a privacy-focused cryptocurrency that provides both anonymity and confidentiality. It uses a different approach than Zerocash. Instead of zk-SNARKs, Monero achieves its privacy through the use of ring signatures, stealth addresses, and ring confidential transactions. Ring signatures and stealth addresses provide anonymity and are described in more detail below. Ring confidential transactions provide confidentiality by using pedersen commitments to hide the amount and zero-knowledge proof to ensure the validity.

Users in Monero also own a key-pair of a public and a secret key, while its work model is the UTXO model same as in Bitcoin.



(a) (b)

Figure 3.3: Ring signature example. Figure 3.3a Either one of $\mathsf{pk}_1, \mathsf{pk}_2$ sends to pk_3, so none of them can be removed from utxo. Figure 3.3b There are cases where anonymity is not guaranteed. Both $\mathsf{pk}_1, \mathsf{pk}_2$ is spent after second transaction, so pk_3 is the sender of the third transaction.

Ring signatures

Monero users the ring signatures [23] to hide the sender of the transaction, as well as the transaction graph Figure 3.3. A ring signature is a cryptographic technique that allows a member of a group to anonymously sign a message on behalf of the group, while hiding the identity of the person signing the message. In other words, this primitive allows users to specify a set of possible signers without revealing which member actually created the signature.

In order to create such a signature the signer executes the following steps. First, they choose a set of public keys, including his own, called a 'ring'. Then, they use their own private key to create a signature for the message. However, they combine it with the public keys of others in the ring rather than signing directly with their private key., in order to hide the real signer.

Anyone can publicly verify that the signer is a member of the ring by using the public keys of the ring. However they cannot distinguish which corresponding secret key was used to produce the signature.

So in Monero, when a user wants to create a transaction, he chooses an anonymity set that includes the public keys of other users' utxo inputs, and creates a ring signature with their own utxo input (corresponding to their public address) and those in the anonymity set. Since no one can distinguish the utxo input that was spent in the transaction, it cannot be removed from the utxo set either.

Stealth addresses

In order to improve its anonymity, each Monero transaction generates a one-time destination address, known as a stealth address. This address is derived from the recipient's public key, but is unique for each transaction, ensuring that the recipient's public address remains private.

An example of such addresses is presented bellow implemented using elliptic curves:

Let Alice be the sender and Bob the receiver. Bob owns a key-pair of the form $\mathsf{pk} = (A, B), \mathsf{sk} = (a, b)$ such that $A = aG, B = bG$ where G is the base of the used elliptic curve. Then Alice generates a random r and calculates $P = HrAG + B$, where H is a collision-resistant hash function. Alice publish $P, R = rG$ together with the transaction. Afterwards, Bob can check if $P' = HaRG + B$. If the transaction is destined for Bob then it holds that $P' = P$.

Obviously, no one will know that this address is intended for Bob, but only Bob can prove that this address is associated with him. In this way, Monero hides the recipient address of the transaction.

3.2.3 Quisquis

Quisquis [11] is another approach to the construction of a decentralized private payment system. It's goal is to implement such a system without the disadvantages of the aforementioned systems. These drawbacks relate to the trusted setup of zk-SNARKs and the limited storage scalability of the ever-growing list of UTXO set.

Quisquis working model is a hybrid one.In other words, it combines the account model, since each user has several accounts with its corresponding balance, with the UTXO logic, since transactions contain UTXO entries rather than accounts.This is made possible by the use of updatable public keys, a primitive that makes it possible to have a number of different public keys associated with the same private key. All of these keys are derived from the same original public key. In this way, a user can use the same private key to spend all of the UTXOs (denoted by the updated keys) that belong to his or her account. Each public key is used a maximum of twice, once during its generation on the output side and once during its use on the input side of a transaction.

Anonymity is achieved through the combination of anonymity sets with the use of the updatable public key primitive. More specifically, transactions can be thought of as "wealth redistribution" between inputs and outputs Figure 3.4. Input accounts include the senders, the recipients as well as an anonymity set. Output accounts are new, updated but unlikable accounts for the senders, recipients, and decoys.

Confidentiality is achieved through a commitment scheme. Both the balances in the accounts and the transacted amounts are given in a commitment form. A user can change the corresponding commited value using the homomorphic property of the commitment scheme.

Finally, each transaction includes zero-knowledge proofs derived from Σ-protocols in order to ensure its validity.

Updatable Public Keys

The concept of an UPK scheme is that public keys can be updated while remaining indistinguishable from freshly generated keys. A UPK scheme is a tuple of algorithms $(\mathsf{Setup}, \mathsf{KGen}, \mathsf{Update}, \mathsf{VerifyKP}, \mathsf{VerifyUpdate})$.

- Setup generates the public parameters, which are implicitly given as input to all other algorithms, i.e. $\mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$. For instance, pp could be a prime-order group $(\mathbb{G}, g, p)$.

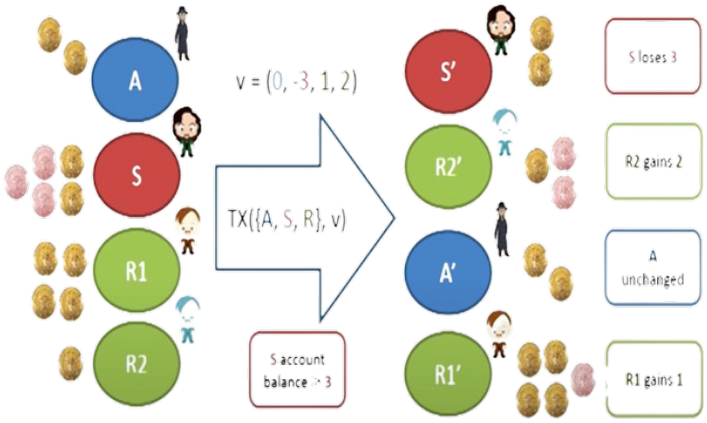


Figure 3.4: Example of redistribution of value in Quisquis

- KGen generates a keypair $(\mathsf{pk}, \mathsf{sk})$. Concretely, it is implemented as: Sample $r, \mathsf{sk} \leftarrow\!\!\$\ \mathbb{F}_p$, calculate $\mathsf{pk} = (g^r, g^{r\cdot\mathsf{sk}})$ and output $(\mathsf{sk}, \mathsf{pk})$.
- Update takes as input a set of public keys $\{\mathsf{pk}\}_{i=1}^n$ and a secret key and generates a new set $\{\mathsf{pk}'\}_{i=1}^n$ where $\mathsf{pk}'_i = \mathsf{pk}_i^r = (g_i^r, g_i^{r\cdot\mathsf{sk}})$ for all i.
- VerifyKP takes as input a keypair $(\mathsf{sk}, \mathsf{pk})$ and checks if it is valid, i.e. if pk corresponds to sk. It is constructed by parsing $\mathsf{pk} = (g', h')$ and outputting the result of the check $(g')^{\mathsf{sk}} \stackrel{?}{=} h'$.
- VerifyUpdate takes as input a pair of public keys and some randomness $(\mathsf{pk}', \mathsf{pk}, r)$ and checks if pk' is a valid update of pk using r. This is done by checking if $\mathsf{Update}(\mathsf{pk}; r) \stackrel{?}{=} \mathsf{pk}'$.

An UPK scheme must satisfy the following properties:

- **Correctness**: All honestly generated keys verify correctly, the update process can be verified and the updated keys also verify successfully.

  Definition 4. *A UPK satisfies perfect correctness if the following three properties hold for all* $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{KGen}]$*:*

  - $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk}) = 1$;
  - $\mathsf{VerifyUpdate}(\mathsf{Update}(\mathsf{pk}; r), \mathsf{pk}, r) = 1\ \forall r \in \mathtt{R}$;
  - $\mathsf{VerifyKP}(\mathsf{pk}', \mathsf{sk}) = 1\ \forall \mathsf{pk}' \in [\mathsf{Update}(\mathsf{pk})]$.

- **Indistinguishability**, meaning that an adversary cannot distinguish between a freshly generated public key and an updated version of public key it already knows.

  Definition 5. *The* advantage *of the adversary in winning the indistinguishability game 3.1 is defined as:* $\mathsf{Adv}^{\mathrm{ind}}_{\mathcal{A}}(\lambda) = \mid \Pr[\mathsf{Exp}^{ind}_{\mathcal{A}}((\lambda)) = 1] - \frac{1}{2} \mid$

  A DPS satisfies indistinguishability *if for every PPT adversary* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{ind}}_{\mathcal{A}}(\lambda)$ *is negligible in* λ.

Game 3.1: Indistinguishability game $\mathsf{Exp}^{ind}_{\mathcal{A}}(\lambda)$

Input : λ
Output: $\{0,1\}$

$b \leftarrow \{0,1\}$
$(\mathsf{pk}^*, \mathsf{sk}^*) \leftarrow \mathsf{KGen}()$
$r \leftarrow\!\!\$\ \mathcal{R}$
$\mathsf{pk}_0 \leftarrow \mathsf{Update}(\mathsf{pk}^*; r)$
$(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathsf{KGen}()$
$b' \leftarrow \mathcal{A}(\mathsf{pk}^*, \mathsf{pk}_b)$
return $(b = b')$

Note that in indistinguishability game 3.1 the challenger can update many times the pk^* before creating pk_0 due to the fact that even with more updates the pk_0 can be described as an update of pk^* with a different randomness.

- **Unforgeability**, meaning that for every honestly generated keypair an adversary cannot learn the secret key of an updated public key without knowing the secret key of the original public key. This is formalized by saying that the adversary cannot generate a public key for which he knows both the secret key and the randomness required to explain that public key as an update of an honestly generated public key.

  Definition 6. *The* advantage *of the adversary in winning the unforgeability game 3.2 is defined as:* $\mathsf{Adv}^{\text{unf}}_{\mathcal{A}}(\lambda) = \mid \Pr[\mathsf{Exp}^{unf}_{\mathcal{A}}((\lambda)) = 1] - \frac{1}{2} \mid$

  A DPS satisfies unforgeability *if for every PPT adversary* $\mathcal{A}$, $\mathsf{Adv}^{\text{unf}}_{\mathcal{A}}(\lambda)$ *is negligible in* λ.

Game 3.2: Unforgeability game $\mathsf{Exp}^{unf}_{\mathcal{A}}(\lambda)$

Input : λ
Output: $\{0,1\}$

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}()$
$(\mathsf{sk}', \mathsf{pk}', r) \leftarrow \mathcal{A}(\mathsf{pk})$
return $\mathsf{VerifyKP}(\mathsf{pk}', \mathsf{sk}') \wedge \mathsf{VerifyUpdate}(\mathsf{pk}', \mathsf{pk}, r)$

If the DDH assumption holds in $(\mathbb{G}, g, p)$ then this construction satisfies correctness, indistinguishability and unforgeability.

Proof. Correctness is straightforward to verify. Indistinguishability derives from the DDH assumption. An adversary $\mathcal{A}$ that can win the indistinguishability game 3.1 can be used to create $\mathcal{B}$ who can distinguish a DDH tuple. That can be proven using the following reduction 3.3.

If *chl* is a DDH tuple then pk' is distributed identically to pk_0. Otherwise pk' is distributed identically to pk_1. Therefore, the reduction has the same non-negligible advantage in the DDH as the $\mathcal{A}$ has in the indistinguishability game 3.1.

Algorithm 3.3: Adversary who wins DDH: $\mathcal{B}(chl)$

Input : $chl = (g, g^x, g^y, g^z)$
Output: $\{0, 1\}$

$r \leftarrow\$ \mathcal{R}$
$\mathsf{pk}^* = (g^r, g^{xr})$
$\mathsf{pk}' = (g^{yr}, g^{zr})$
$b \leftarrow \mathcal{A}(\mathsf{pk}^*, \mathsf{pk}')$
return b

Unforgeability derives from the DL assumption. An adversary $\mathcal{A}$ that can win the unforgeability game 3.2 can be used to create $\mathcal{B}$ who can win the DL game. That can be proven using the following reduction 3.4.

Algorithm 3.4: Adversary who wins DL: $\mathcal{B}(chl)$

Input : $chl = (g, h = g^s)$
Output: s

$t \leftarrow\$ \mathcal{R}$
$(g_0, h_0) = (g^t, h^t)$
$\mathsf{pk} = (g_0, h_0)$
$(\mathsf{sk}', \mathsf{pk}' = (g_1, h_1), r) \leftarrow \mathcal{A}(\mathsf{pk})$
return sk'

The winning condition of the unforgeability definition 6 requires that $(h_1 = g_1^{\mathsf{sk}'})$ and $(g_1, h_1) = (g_0^r, h_0^r) = (g^{rt}, h^{rt})$ thus implying that $g^{\mathsf{sk}'rt} = h^{rt}$ or equivalent that $h = g^{\mathsf{sk}'}$, meaning $\mathsf{sk}' = s$ or that it is a valid solution to the DL oracle.

□

3.2.4 Comparison

Of the three private systems mentioned above, Zerocash offers full anonymity, while the other two offer set anonymity. However, both Zerocash and Monero suffer from a very important limitation. This is the size of the storage cost that each miner must maintain in the system. Since a UTXO input is not identified when spent (to preserve anonymity), these two systems have a growing list of UTXOs. This fact prevents miners from storing a concise version of the blockchain. The primitive updatable public keys used by Quisquis provide a solution to this problem. As a result, Quisquis manages to maintain a constant storage cost with respect to the number of transactions.

Chapter 4

Auditability in Private Payment Systems

Auditability plays a key role in ensuring payment system integrity, transparency and compliance. In financial transactions where trust and security are paramount, auditability provides critical protection against fraud, money laundering, and other illicit activities. These regulatory functions that appear in the literature can be categorized in transaction and user level.
On the transaction level regulatory functions can include data such as: *value limits* (e.g. a threshold in the transfer amount), *tracing tags*, that provide links between transactions, *revealing the transaction value and/or participants*, *tax rate*, deducting a transaction's value portion towards a pre-determined account. On the user level regulatory functions can include: *information of user's sum of values* (e.g. total amount of funds received/spent in a specific period of time), *user revocation*, meaning that specific policies are applied only to users in a "blacklist", *deriving statistical information* (e.g. learning th average transacted value in a time from user's past transactions), *revoking a non compliant user's anonymity.*

There are two approaches in order these regulation to be enforced, *auditability* and *accountability.* On the one hand, auditability refers to a protocol where an external auditor can learn the requested information through the data that are stored on the blockchain. This protocol could be either interactive with the users, meaning their consent is required, or non-interactive. On the other hand, accountability refers to the recurrent execution of policies by system functions when certain predicate is met. In other words, transactions that do not comply with the system's policies are never verified and stored in the blockchain. Therefore, there is no need for active participation of an external auditor, as policies are enforced during the verification phase of the transaction.

Then, an overview of existing distributed payment systems that combine both privacy and auditability is presented. Following the structure of [8], these systems are divided into two categories depending on the power given to auditors from the disclosed information: There are systems that requires a centralized trusted desingated athority to perform the regulation functions and the systems that does not assume any explicit auditor.

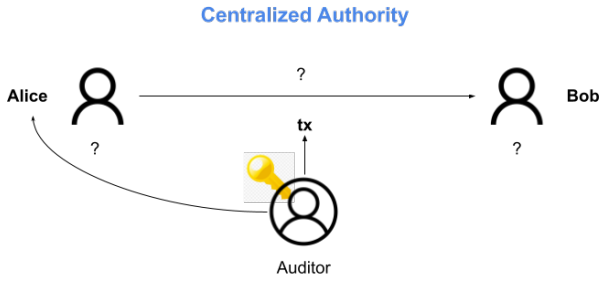


Figure 4.1: Auditability - Centralized Authority

4.1 Centralized Authority

A common way to implement auditability in private systems is to introduce a centralized authority or group of authorities (multi-party computation). Such authority can either be an external designated auditor or can enforce the internal policy rules in each transaction (accountability).

According to this approach, users embed auxiliary information in the transactions, which is encrypted under the public key of a designated trusted auditor (Figure 4.1). Thus, the users' data remains private to the rest of the system's participants, except for the central authority, which can decrypt the auxiliary information at any time without the users' consent.

This method can be a trivial solution for adding auditability to privacy-preserving systems. However, all data is collected by a single centralized authority, which accumulates excessive power. This fact can have a negative impact on user privacy.

An example that implements this approach is presented below:

4.1.1 Zcash extension

[12] is an extension of Zerocash [5] to add privacy-preserving policy enforcement mechanisms that ensure regulatory compliance. It provides a variety of auditability features including regulatory closure, transaction spending limits, and accountable selective user tracing. To achieve this, auxiliary information is added to each Zerocoin (counter, regulatory type) as well as algorithms-policies that are executed each time a coin is spent.

They introduce a new basic building block in Zerocash, *Counters*, which is added to the coin data. Counters can store cumulative information either for a specific physical user or for Zerocash addresses. In the first case, counters should be tied to a specific (unique) identity and issued by a trusted third party. In the second case, a user may have many addresses within the system, and an authority should know the link between the user's real-world identity and the set of internal addresses. To preserve anonymity, these counters must be entered using the same method as coin commitments, which involves proving their inclusion in a Merkle tree.

Counters are important for enforcing spending limits and tax policies. *Spend-*

ing Limit Policy enforces that no transaction with a transfer or counter value over a specified limit is valid unless it is signed by an authority. This provides a form of accountability as it can preemptively review transactions before they are entered into the ledger. On the other hand, *Tax* belongs to the auditability type policy. All outputs sent to other parties are summed up and a percentage of that amount is added to a user's tax counter. After that, the authority is responsible for auditing the users for their tax compliance.

The second piece of information embedded in each coin is the regulatory type. This data is used to achieve regulatory closure, meaning that the system can provide enough information to guarantee the continuation of another regulatory framework that can be enforced with or without zero-knowledge. This policy ensures that all input and output coins in a transaction have the same regulatory type, and thus an adversary cannot cause policies to operate on the wrong data.

Finally, [12] offers the ability of accountable coin tracing. The coins can also contain tracing information encrypted under a unique key held by the tracing authority. The authority could then trace these coins, along with all subsequent coins resulting from transactions involving the original coins, without requiring any interaction with the users. However, there is a mechanism that allows a user to find out whether or not they have been traced. If a user is being traced, the key given by the authority will be a randomized version of their public key, otherwise it will be a randomized version of a null key. Users cannot tell which key they have received at any given time without being aware of the randomness. However, if the authority later reveals the randomness, users can definitively determine whether or not they have been traced.

Although [12] allows the implementation of a wide variety of regulatory policies, it suffers from both efficiency and privacy limitations. Zerocash is already an computationally intensive and storage expensive system since it has a monotonically growing list of UTXO [2]. The addition of auxiliary information and the requirement that transactions be validated by an authority before posting on-chain adds additional communication and computation costs. In addition, the designated authority has too much power with respect to user privacy, since it can learn any transaction information and deanonymize the user. Furthermore, there is no mechanism to prevent censorship of certain transactions by the authority [8].

4.2 General Auditor

To avoid collecting all information at a centralized authority (or group of authorities), a second approach has been proposed. This is an interactive protocol between the user being audited and the auditor. In this case, the auditor, which can be *any* auditing authority, can ask specified questions derived from the system's policies. Users answer these questions with zero-knowledge proofs based on data stored on chain (Figure 4.2).

This protocol implies the consent and cooperation of the audited user. However, this requirement cannot be exploited by non-compliant users, since refusal to cooperate with authorities can be considered equivalent to a failed audit.

Below are some typical examples of auditable private decentralized systems that implement a general auditor:

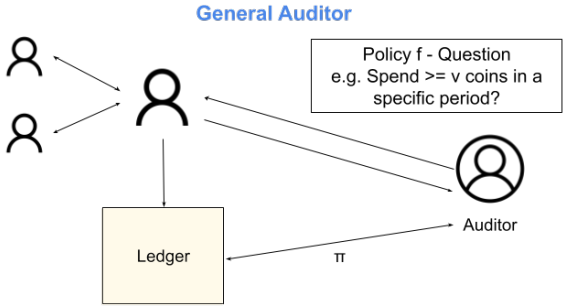


Figure 4.2: Auditability - General Auditor

4.2.1 zkLedger

zkLedger [20] is a permissioned, fully-private payment system that supports auditing by a general auditor. It succeeds to provide strong transaction privacy properties, such as hiding the transfered amounts, the participants as well as the link between the transactions (transaction graph), while allowing an auditor to compute provably correct functions over the on-chain data. Audit procedure takes place as an interactive protocol between the users and an auditor, where the latter queries the former about its contents on the ledger.

Since an auditor cannot distinguish the participants of a transaction, zkLedger should ensure that during auditing a user cannot leave out transactions that participated, in order to be able to receive reliable answer to its queries.

The solution, that zkLedger proposed, to this problem relies on its unique ledger. That is, a table where transactions correspond to rows, and Users (Banks) correspond to columns. Each transaction include information for all other users, even for those who do not participate. To hide the transfered amounts as well as the assets each user holds each entry in a transaction are contains a commitment to a value that is debited or credited to the user. All entries for the non-participants are a commited value to 0 (Figure 4.3). Due to the hiding property of the commitments an adversary cannot distinguish between a zero and a non-zero commited value.

Transactions

The transactions in zkLedger are publicly verifiable and must satisfy the following invariants:

- Transactions conserves assets, meaning a transfer transaction cannot create or destroy assets.
- The spending user gives concent to the transfer and actually own enough assets to execute the transaction.
- After each transactions all users have enough information to open their commitments for the audit functionality.

Therefore, transactions are formed via a combination of commitments and the zero-knowledge proofs and have the following form $\mathsf{tx} = (cm_i, Token_i, \pi^B, \pi^A, \pi^C)$:

ID	Asset	Goldman Sachs	JP Morgan	Barclays
1	€	Depositor, Goldman Sachs, 30M		
2	€	comm(-10M)	comm(10M)	comm(0)
3	€	comm(0)	comm(-1M)	comm(1M)
4	€	comm(0)	comm(-2M)	comm(2M)

Figure 4.3: zkLedger Ledger - naive transaction example

- **Commitment** (cm_i): $(g^{v_i}h^{r_i})$ a Pedersen commitment to the transfered value.
- **Audit token** ($Token_i$): $(pk_i)_i^r$ used in order to enable user to create reliable answers to the audits without knowing the randomness used in the commitment.
- **Proof of balance** (π^B): a zero-knowledge proof asserting the preservation of balance in each transaction $\sum_{k=1}^{n} v_k = 0$.
- **Proof of assets** (π^A): a zero-knowledge proof asserting that the sender has the assets to transfer. In zkLedger a column in the ledger represents all the assets the corresponding user has received or spent. So π^A is proven by creating a commitment to the sum of values for the asset in its column, including the current transaction. If the sum is greater than or equal to 0, then the user has the right to transfer the amount.
- **Proof of consistency** (π^C): a zero-knowledge proof asserting that a malicious user cannot add data to the ledger that would prevent another user to be able to open their commitments.

Finally, whenever a transactions happens, a new row is added to the ledger.

Auditing

The Auditor has access to the ledger and interact with the users to calculate functions on their private data. This functions allow the auditor to issues queries that includes information about ratios of holdings, sums, average, variance, outliers, changes over time. The account holder reveal only the value hidden in commitment necessary for the question, without leaking any other information. However, frequent auditing is possible to reveal more of a transaction contents.

Figure 4.4 illustrate an example. An auditor can ask a user "How many euros do you hold at time t?". The user responds with an answer along with the validity zero-knowledge proof. The auditor can multiply all the commitments on the ledger for the audited user and can verify if the proof and the answer are valid. Since each column of the ledger represents all the assets that the corresponding user has received or spent, the auditor can be sure that user could not hide any of their transaction during the audit.

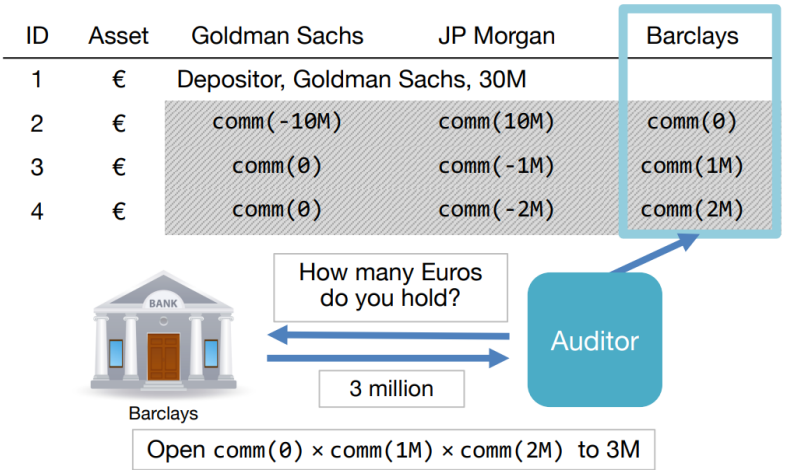


Figure 4.4: zkLedger - audit functionality example

Setbacks

Although zkLedger combines full privacy with auditability and offers a wide variety of audit functionalities, suffers from very limited scalability. There are two significant costs that grow with the number of users. The verification of transactions which increases with the number of the users as well as The sequential steps to create transactions increase linearly.

Considering the verification, zkLedger requires each transaction to include commitments and zero-knowledge proofs for all users. This fact, combined with the ever-increasing ledger that grows with every transaction, results in large computational and storage costs. To overcome this problem an improvement of zkLedger protocol has been proposed called miniLedger [7].

Concerning the second setback, in order a user to be able to produce a new transaction (for exmample transaction n) they must use the state of the ledger right before, meaning the must known the $n-1$ transaction. Therefore, multiple users cannot produce different transactions in parallel, since concurrent transactions always have conflicts.

4.2.2 PGC

PGC [9] is a standalone auditable confidential payment system. In this work they trade anonymity for highly efficient auditing only dependent to the number of past user transactions. Privacy is offered in terms of confidentiality and use pseudonymity as a feature assuming that auditors can link the account addresses with real identities. It proposes two kinds of auditing mechanism: (i) regulation compliance that is achieved through three audit functions, namely transaction limits, tax payment and selective value disclosure. (ii) supervision (tracing functionality) that is achieved through including the necessary auxilary information in the transaction structure. The cryptographic techniques that are used for its implementation is a variant of El Gamal encryption and zk-proofs composed of Σ-protocols.

Entities

More specifically, the system consists of the following entities:

- Users: Individuals that transact with each other and may control several accounts within the system.
- Validators: Validators are responsible for checking the validity of proposed transactions within the system. They ensure that transactions meet the required criteria before being included in the blockchain.
- Regulators: Regulators interact with involved users to verify if a set of transactions complies with system's policies, without holding any secret information.
- Supervisors: Supervisors have access to a global trapdoor, which allows them to monitor and trace transactions without interacting with the involved users.

Accounts

In order to be able to interact with a system a user creates an account. Each account is associated with a secret key sk, a public key pk, which represent the pseudonym of the user within the system, an encoded balance $\tilde{C}$, and an incremental serial number sn used to prevent replay attacks ($\mathsf{acct} = (\mathsf{sk}, \mathsf{pk}, \tilde{C}, sn)$). The balance is encrypted in order privacy to be achieved. Only the owner of sk can decrypt and learn the value but all users should be able to change the encrypted value. PGC implement this through homomorphic encryption.

Transactions

Users can use the accounts to transact within the system. A transaction take place between two participants and need as input the secret key of the sender sk, the transacted value v and the public keys of both sender and receiver $(\mathsf{pk}_s, \mathsf{pk}_r)$. Given the input the algorithm produces (C_s, C_r) that are encoded transferred value v under the public keys of sender and receiver $(\mathsf{pk}_s, \mathsf{pk}_r)$.

The legality of the transaction is proved through the creation of a a zk-proof π_{legal}. π_{legal} consists of the following proofs: (i) π_{equal}: C_s, C_r contains encryption of the same value v (ii) π_{right}: the transfer amount v is within the allowed limits (iii) $\pi_{solvent}$: the sender has enough balance.

Finally, to authenticate that the sender is the owner of the corresponding account the algorithm sign the $(sn, memo = (\mathsf{pk}_s, \mathsf{pk}_r, C_s, C_r), \pi_{legal})$ with the secret key sk producing the signature σ.

The final transaction is $\mathsf{tx} = (sn, memo, \pi_{legal}, \sigma)$.

In order to support supervision under a specified authority with a known public key pk_a the transaction can be extended with an encryption of transacted value C_a under the pk_a. Then the supervisor can inspect any confidential transaction by decrypting C_a using sk_a.

Policies and Audit

In PGC policies are represented as predicates f over a public key pk and related transactions $\{\mathsf{tx}_i\}_{i=1}^{n}$ in which pk participates either as sender or as receiver.

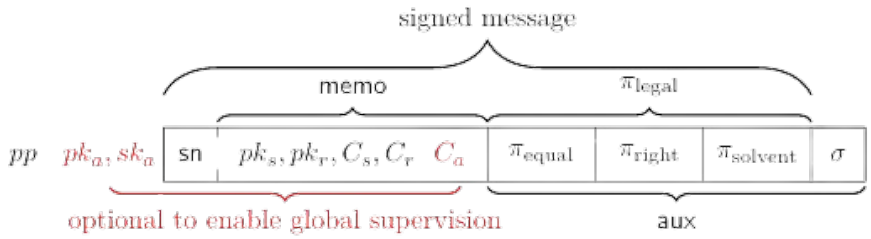


Figure 4.5: Data structure of transaction in PGC

Let v_i the transferred amount in tx_i. They implement the following policies over the values v_i:

- Limit policy $f_{limit}(\mathsf{pk}, \{\mathsf{tx}\}_{i=1}^n)$: Checks that $\sum_i^n v_i \leq a_{max}$, where a_{max} is an upper bound depending on application. The prover can be either the sender or the receiver of tx. This policy is a mechanism used for anti-laundering money.

- Tax policy $f_{tax}(\mathsf{pk}, \mathsf{tx}_1, \mathsf{tx}_2)$: Let pk be recipient in tx_1 and sender in tx_2. Let v_1, v_2 be the transfer amounts in each transaction. The policy checks if $v_1/v_2 = \rho$, where ρ is a rate depending on application. The auditor can use this policy to ensure that user paid appropriate tax.

- Open policy $f_{open}(\mathsf{pk}, \mathsf{tx})$: Checks that the underlying transferred amount v is equal to a v^* ($v = v^*$), where v^* is an application-dependent value. The prover can be either the sender or the receiver of tx. The auditor can use this policy to enforce selective-disclosure.

Figure 4.6: Policies in PGC

In order the auditing to be efficient in PGC it is executed with the aid from the auditee. In particular, it needs the consent of the auditee who creates a zk-proof that proves that they are compliant with the specified policy. The zk-proofs are implemented by using bulletproofs and Σ-protocols.

Setback

Although PGC does not suffer from the scalability issues of zkLedger, it is not fully private. As mentioned before, PGC offers only confidentiality when it comes to privacy and trades anonymity in order to implement efficient auditability.

Chapter 5

AQQUA: Augmenting Quisquis with Auditability

5.1 Overview

Objective

The solutions proposed in the literature that aim to combine privacy and auditability using a general auditor rather than a centralized authority either suffer from limited scalability or do not provide full privacy.

Therefore, we aim to construct an efficient, anonymous, confidential and auditable system that can also support concurrent transactions and maintain a stable state size regardless of the number of users or transaction history. To create such a system, we propose AQQUA, which extends the Quisquis [11] DPS system with a general auditor. Thus, AQQUA combines the anonymity of Quisquis with the policy expressiveness and regulation of PGC [9]. This means that the auditor can perform queries on the upper limit of the amount sent/received by the user in a given period, on the non-participation of a user in a given transaction or period, as well as on the exact value sent/received in a transaction.

We chose to extend Quisquis because, as already mentioned, it is a fully private system, offering both anonymity and confidentiality, while having a constant storage cost with respect to the number of transactions. Moreover, due to the fact that the anonymity set used to hide the participants of a transaction does not include all the users of the system, as in zkLedger, Quisquis can support concurrent transactions.

Challenges

The design of AQQUA had to overcome the following key challenge to enable auditing functionalities while still preserving user privacy. Due to the anonymity property, users can hide their accounts and consequently the amounts necessary for the auditing process. In addition, since Quisquis is permissionless and private, an authority cannot enforce some effective penalties for non-compliant users.

To overcome this challenge, we introduce a registration functionality in Quisquis through a registration authority. This means that users must first register with the systems and provide their real-world credentials. They can then create new, unlinked accounts that are used to transact within the system. The registration functionality provides a way for the system to know which users are using the system and to penalize them in a way that is outside the scope of the system. In addition, AQQUA splits the state into two sets. It keeps the UTXO set used in Quisquis, which contains the user's unspent accounts, but also adds a new set that contains the user's public registration information along with the necessary information to ensure that users cannot hide information during the audit process.

5.2 Preliminaries

5.2.1 Notation

We denote by λ the security parameter. We denote by $\mathcal{M}$ the message space and by $\mathcal{R}$ the randomness space of our cryptographic schemes. $\mathcal{V} = \{0, \ldots, V\}$ is the set that defines the range of valid currency values, where V is an upper bound on the maximum possible number of coins in the system ($|\mathcal{V}| \ll |\mathcal{M}|$). When an element x is sampled uniformly at random from a set $\mathcal{X}$, we write $x \leftarrow\!\!\$\ \mathcal{X}$. Given a tuple $t = (a, b)$ we refer to its parts using the dot notation, i.e. $t.a$ or $t.b$. We denote (a^x, b^x) as $t^x = (a, b)^x$.

5.2.2 Commitments

We use a commitment scheme Commit relative to a public key pk that, given a message $m \in \mathcal{M}$ and randomness $r \in \mathcal{R}$, computes $\boxed{m} \leftarrow \mathsf{Commit}(\mathsf{pk}, m; r)$. Our commitments must satisfy the following properties:

- **Computational hiding**: An adversary has negligible advantage in distinguishing between $\mathsf{Commit}(\mathsf{pk}, m_0; r_0)$ and $\mathsf{Commit}(\mathsf{pk}, m_1; r_1)$, where $r_0, r_1 \leftarrow\!\!\$\ \mathcal{R}$.
- **Unconditional binding**: A commitment cannot be opened to two different messages, even with the knowledge of the secret key sk.
- **Additively homomorphic**: For given operation $\odot$ it holds that $\mathsf{Commit}(\mathsf{pk}, m; r) \odot \mathsf{Commit}(\mathsf{pk}, m'; r') = \mathsf{Commit}(\mathsf{pk}, m + m'; r + r')$.
- **Key-anonymous**: An adversary cannot distinguish between $(m, \mathsf{pk}_0, \mathsf{pk}_1, \mathsf{Commit}(\mathsf{pk}_0, m))$ and $(m, \mathsf{pk}_0, \mathsf{pk}_1, \mathsf{Commit}(\mathsf{pk}_1, m))$ for any honestly generated public keys $\mathsf{pk}_0, \mathsf{pk}_1$ and adversarially chosen message m.

We construct such a scheme using the unconditionally binding commitments of [11]. They are defined in a prime-order p group $(\mathbb{G}, g, p)$ generated by g, where the DDH problem is hard. In essence, ElGamal 'encryption' is used in the exponent where the public keys are of the form $\mathsf{pk} = (g, h) \in \mathbb{G}^2$. Specifically, $\mathsf{Commit}(\mathsf{pk}, m; r)$ yields $\boxed{m} = (c, d)$, where $c = g_i^r$ and $d = g^m h_i^r$.

Using UKPs as the commitment public keys, one can verify and open commitments using the secret key, without needing to know the randomness used.

- $\mathsf{VerifyCom}(\mathsf{sk}, \mathsf{pk}, \mathtt{com}, m)$: Checks if $\mathtt{com} = (c, d)$ is a commitment to m under pk, by checking if $d = g^m c^{\mathsf{sk}}$ holds.

- $\mathsf{OpenCom}(\mathsf{sk}, \boxed{m})$: Given $\boxed{m} = (c, d)$, calculates m by calculating $dc^{-\mathsf{sk}}$ and brute-forcing to obtain m.

5.3 Definition of an Auditable Private Decentralized Payment System

We now describe the components of an auditable and private DPS like AQQUA:

5.3.1 Entities

- *Registration Authority* (`RA`): The role of the `RA` is to enroll new users into the system. Users register by sending their real-world identity information together with an initial public key that they create on their own. The `RA` stores this information off-chain. All the accounts that transact on behalf of this real-world user will originate from this initial public key, through the mechanism of section 3.2.3. The purpose of the registration procedure is essential as it establishes a link between a user's public key and their real identity, to be used for the potential penalization of non-compliant users.

- *Audit Authority* (`AA`): Its role is to initiate the audit procedure in order to verify that users comply with the system's policies. If a user of the system is found to be non-compliant, the `AA` will collaborate with the `RA` to enforce the relative penalties.

- *Users* (`U`): Users of the system that transact with each other.

5.3.2 State

In AQQUA, the state (denoted state) consists of the following two sets:

- $\mathsf{UTXOSet}$: A table containing the 'unspent' accounts, i.e. the accounts that are recorded as outputs of a valid transaction, but have not (yet) been used as inputs.

- $\mathsf{UserSet}$: A table containing one tuple for each physical user, which is composed of the user's initial public key and a commitment to the number of accounts owned by the user.

5.3.3 Accounts

User accounts are of the form $\mathsf{acct} = (\mathsf{pk}, \boxed{\mathtt{bl}}, \boxed{\mathtt{out}}, \boxed{\mathtt{in}})$, where `bl` is the account balance and `out`, `in` is the total amount that the account has sent and received, respectively. Each user may own multiple accounts which are stored in the $\mathsf{UTXOSet}$. The following functionalities create, verify and update accounts.

- $\mathsf{acct} \leftarrow \mathsf{NewAcct}(\mathtt{pk}_0; r_1, r_2, r_3, r_4)$: takes as input a public key $\mathtt{pk}_0$ and outputs a new account of the form $\mathsf{acct} = (\mathsf{pk}, \boxed{\mathtt{bl}}, \boxed{\mathtt{out}}, \boxed{\mathtt{in}})$, where $\mathsf{pk} = \mathsf{Update}(\mathtt{pk}_0; r_1)$, $\boxed{\mathtt{bl}} = \mathsf{Commit}(\mathsf{pk}, 0; r_2)$, $\boxed{\mathtt{out}} = \mathsf{Commit}(\mathsf{pk}, 0; r_3)$ and $\boxed{\mathtt{in}} = \mathsf{Commit}(\mathsf{pk}, 0; r_4)$.
- $0/1 \leftarrow \mathsf{VerifyAcct}(\mathsf{acct}, \mathsf{sk}, \mathtt{bl}, \mathtt{out}, \mathtt{in})$: Parses acct as $(\mathsf{pk}, \mathtt{com}_1, \mathtt{com}_2, \mathtt{com}_3)$ and outputs 1 if

$$\mathsf{VerifyCom}(\mathsf{sk}, \mathsf{pk}, \mathtt{com}_1, \mathtt{bl}) \wedge \mathsf{VerifyCom}(\mathsf{sk}, \mathsf{pk}, \mathtt{com}_2, \mathtt{out}) \wedge$$
$$\mathsf{VerifyCom}(\mathsf{sk}, \mathsf{pk}, \mathtt{com}_3, \mathtt{in}) \wedge (\mathtt{bl}, \mathtt{out}, \mathtt{in} \in \mathcal{V})$$

- $\{\mathsf{acct}'_i\}_{i=1}^n \leftarrow \mathsf{UpdateAcct}(\{\mathsf{acct}_i, \mathtt{v}_{\mathtt{bl}i}, \mathtt{v}_{\mathtt{in}i}, \mathtt{v}_{\mathtt{out}i}\}_{i=1}^n; r_1, r_2, r_3, r_4)$ takes as input a set of accounts $\mathsf{acct}_i = (\mathsf{pk}_i, \mathsf{com}_{\mathtt{bl}i}, \mathsf{com}_{\mathtt{out}i}, \mathsf{com}_{\mathtt{in}i})$ and values $\mathtt{v}_{\mathtt{bl}i}, \mathtt{v}_{\mathtt{out}i}, \mathtt{v}_{\mathtt{in}i} \in \mathcal{V}$ and outputs a new set of accounts $\{\mathsf{acct}'_i\}_{i=1}^n$, where

$$\mathsf{acct}'_i \leftarrow (\mathsf{Update}(\mathsf{pk}; r_1), \mathsf{com}_{\mathtt{bl}i} \odot \mathsf{Commit}(\mathsf{pk}, \mathtt{v}_{\mathtt{bl}i}; r_2),$$
$$\mathsf{com}_{\mathtt{out}i} \odot \mathsf{Commit}(\mathsf{pk}, \mathtt{v}_{\mathtt{out}i}; r_3), \mathsf{com}_{\mathtt{in}i} \odot \mathsf{Commit}(\mathsf{pk}, \mathtt{v}_{\mathtt{in}i}; r_4)).$$

- $0/1 \leftarrow \mathsf{VerifyUpdateAcct}(\{\mathsf{acct}'_i, \mathsf{acct}_i, \mathtt{v}_{\mathtt{bl}i}, \mathtt{v}_{\mathtt{out}i}, \mathtt{v}_{\mathtt{in}i}\}_{i=1}^n; r_1, r_2, r_3, r_4)$: outputs 1 if

$$\{\mathsf{acct}'_i\}_{i=1}^n = \mathsf{UpdateAcct}(\{\mathsf{acct}_i, \mathtt{v}_{\mathtt{bl}i}, \mathtt{v}_{\mathtt{out}i}, \mathtt{v}_{\mathtt{in}i}\}_{i=1}^n; r_1, r_2, r_3, r_4) \wedge (|\mathtt{v}_{\mathtt{bl}}|, \mathtt{v}_{\mathtt{out}}, \mathtt{v}_{\mathtt{in}} \in \mathcal{V}).$$

5.3.4 User information

Each real-world user is associated with a tuple $\mathsf{userInfo} = (\mathtt{pk}_0, \boxed{\mathtt{\#accs}})$, stored in the UserSet. The public key $\mathtt{pk}_0$ is an initial public key provided at the time of registration. The public key of every account owned by the user will share the same secret key with $\mathtt{pk}_0$.

The value #accs is the number of accounts in the UTXOSet that are owned by the user, and is stored as a commitment so that it remains hidden. Keeping track of the number of accounts a user owns is necessary in order to support policies related to value limits, such as the total amount a user has received or sent in a period of time. Otherwise, such policies could be easily bypassed through the creation of sybil identities [8]. The opening of the commitment $\boxed{\mathtt{\#accs}}$ will be revealed only to the AA during the auditing procedure.

The following functions create, verify and update userInfo entries of the UserSet.

- $(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}) \leftarrow \mathsf{GenUser}()$: Picks $r_1, r_2, r_3, r_4, r_5 \leftarrow\$ \mathcal{R}$ and let $\vec{r} = (r_1, r_2, r_3, r_4)$. Then runs $(\mathsf{sk}, \mathtt{pk}_0) \leftarrow \mathsf{KGen}()$, $\mathsf{acct} \leftarrow \mathsf{NewAcct}(\mathtt{pk}_0; \vec{r})$, calculates the tuple $\mathsf{userInfo} = (\mathtt{pk}_0, \mathsf{Commit}(\mathtt{pk}_0, 1; r_5))$ and returns $(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct})$.
- $0/1 \leftarrow \mathsf{VerifyUser}((\mathtt{pk}_0, \mathtt{com}), (\mathsf{sk}, \mathtt{\#accs}))$: outputs 1 if $\mathsf{VerifyCom}(\mathsf{sk}, \mathtt{pk}_0, \mathtt{com}, \mathtt{\#accs}) \wedge (\mathtt{\#accs} \in \mathcal{V})$
- $\{\mathsf{userInfo}'_i\}_{i=1}^n \leftarrow \mathsf{UpdateUser}(\{\mathsf{userInfo}_i, \mathtt{v}_{\#accs_i}\}_{i=1}^n; r)$ takes as input a set of user-value pairs where $\mathsf{userInfo}_i = (\mathsf{pk}_{0_i}, \mathsf{com}_{\mathtt{\#accs}i})$ and $\mathtt{v}_{\#accs_i} \in \mathcal{V}$ and outputs a new set of users $\{\mathsf{userInfo}'_i\}_{i=1}^n = \{(\mathsf{pk}_{0_i}, \mathsf{com}'_{\mathtt{\#accs}i})\}_{i=1}^n$ where

$$\mathsf{com}'_{\mathtt{\#accs}i} = \mathsf{com}_{\mathtt{\#accs}i} \odot \mathsf{Commit}(\mathtt{pk}_0, \mathtt{v}_{\#accs}; r)$$

- $0/1 \leftarrow \mathsf{VerifyUpdateUser}(\{\mathsf{userInfo}'_i, \mathsf{user}_i, \mathtt{v}_{\#accs_i}\}_{i=1}^n; r)$ outputs 1 if

$$\{\mathsf{userInfo}'\}_{i=1}^n = \mathsf{UpdateUser}(\{\mathsf{userInfo}_i, \mathtt{v}_{\#accs_i}\}_{i=1}^n; r) \wedge (\mathtt{v}_{\#accs} \in \mathcal{V})$$

5.3.5 Policies

An auditable DPS should support a rich set of compliance policies. They can be captured as predicates over an initial public key $\mathtt{pk}_0$, a time period represented by a starting state state_1 and an ending state state_2, and auxiliary information $\mathtt{aux}$ which is dependent on an specific compliance goal. In all the policy predicates, we use the notation A_1, A_2 to denote the set of accounts in $\mathsf{state}_1.\mathsf{UTXOSet}, \mathsf{state}_2.\mathsf{UTXOSet}$ that are owned by the owner of $\mathtt{pk}_0$.

- Sending limit policy f_{slimit}: The total amount a real-world user can send within a specific period. It can be determined by the $\mathtt{AA}$ off-chain and announced to the user for a specific period, depending on the application. The $\mathsf{state}_1, \mathsf{state}_2$ are the states of the blockchain at the beginning and end of the period, respectively.

$$f_{\mathsf{slimit}}(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), \mathtt{a}_{max}) = 1 \iff \left\{ \left(\sum_{\mathsf{acct} \in A_2} \mathtt{out} - \sum_{\mathsf{acct} \in A_1} \mathtt{out} \right) \leq \mathtt{a}_{max} \right\}$$

  where $\mathtt{out}$ is the opening of $\boxed{\mathtt{out}}$ of an account acct, using the account's secret key sk.

- Receiving Limit policy f_{rlimit}: Similarly, the total amount a 'physical' user can receive from other accounts.

$$f_{\mathsf{rlimit}}(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), \mathtt{a}_{max}) = 1 \iff \left\{ \left(\sum_{\mathsf{acct} \in A_2} \mathtt{in} - \sum_{\mathsf{acct} \in A_1} \mathtt{in} \right) \leq \mathtt{a}_{max} \right\}$$

  where $\mathtt{in}$ is the opening of $\boxed{\mathtt{in}}$ for account acct, calculated using the account's secret key sk.

- Open policy f_{open}: The value of the amount sent or received by a user in a transaction.

$$f_{\mathsf{open}}(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), v_{\mathsf{open}}) = 1 \iff \left\{ \begin{array}{l} (v = \left(\sum_{\mathsf{acct} \in A_2} \mathtt{bl} - \sum_{\mathsf{acct} \in A_1} \mathtt{bl} \right) \in \mathcal{V} \\ \vee\, v = \left(\sum_{\mathsf{acct} \in A_1} \mathtt{bl} - \sum_{\mathsf{acct} \in A_2} \mathtt{bl} \right) \in \mathcal{V}) \\ \wedge\, v = v_{\mathsf{open}} \end{array} \right\}$$

  where $\mathtt{bl}$ is the opening of $\boxed{\mathtt{bl}}$ of an acct.

- Transaction Value Limit f_{txlimit}: Upper bound to the total transferred amount that can be sent in a transaction.

$$f_{\mathsf{txlimit}}(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), v_{\max}) = 1 \iff \left\{ v = \left(\sum_{\mathsf{acct} \in A_1} \mathtt{bl} - \sum_{\mathsf{acct} \in A_2} \mathtt{bl} \right) \leq v_{\max} \right\}$$

- Non-participation f_{np}: Non-participation in a specific transaction tx or inactivity of the user for a time period. The states $\mathsf{state}_1, \mathsf{state}_2$ are the

states before and after a transaction is applied or at the beginning and end of the period.

$$f_{\mathsf{np}}(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2)) = 1 \iff \left\{ \begin{array}{l} \wedge \left(\sum_{\mathsf{acct} \in A_1} \mathtt{out} - \sum_{\mathsf{acct} \in A_2} \mathtt{out} \right) = 0 \\ \wedge \left(\sum_{\mathsf{acct} \in A_1} \mathtt{in} - \sum_{\mathsf{acct} \in A_2} \mathtt{in} \right) = 0 \end{array} \right\}$$

5.3.6 Functionalities

An auditable private decentralized payment system is a tuple of polynomial-time algorithms defined as below:

- $(\mathsf{state}_0, \mathsf{pp}) \leftarrow \mathsf{Setup}(\lambda)$: Generates the initial state of the system state_0 and the public parameters $\mathtt{pp}$, which are implicitly given as input to all other algorithms.
- $(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}, \pi) \leftarrow \mathsf{Register}()$: Used by a user to create the registration information $\mathsf{userInfo}$ and their first account acct.
- $0/1 \leftarrow \mathsf{VerifyRegister}(\mathsf{userInfo}, \mathsf{acct}, \pi, \mathsf{state})$: Used by the Registration Authority to verify the registration information and the account of a user.
- $\mathsf{state}' \leftarrow \mathsf{ApplyRegister}(\mathsf{userInfo}, \mathsf{acct}, \mathsf{state})$: Used by the Registration Authority to add a user to the system after their successful registration.
- $\mathsf{tx} = (\{\mathsf{acct}\}_{i=1}^n, \{\mathsf{acct}'\}_{i=1}^n, \pi) \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v}_\mathtt{S}}, \overrightarrow{\mathtt{v}_\mathtt{R}}, \mathtt{A})$: Used by the sender with secret key sk to create a transaction that redistributes their coins from their accounts in $\mathtt{S}$ among the recipients accounts in $\mathtt{R}$. The vectors $\overrightarrow{\mathtt{v}_\mathtt{S}}, \overrightarrow{\mathtt{v}_\mathtt{R}}$ describe the changes in the values in $\mathtt{S}, \mathtt{R}$ respectively. To hide the participating accounts, an anonymity set $\mathtt{A}$ is passed as input.
- $\mathsf{tx}_{\mathsf{CA}} = (\mathsf{acct}, \{\mathsf{userInfo}_i\}_{i=1}^n, \{\mathsf{userInfo}'_i\}_{i=1}^n, \pi) \leftarrow \mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A})$: Creates a transaction to create a new account for the owner of $\mathsf{userInfo}.\mathtt{pk}_0$ and appropriately updates the value of the commitment to the number of accounts they own, $\mathsf{userInfo}.\mathtt{com}_{\#\mathtt{accs}}$. To hide the link between the newly created account acct and the corresponding $\mathtt{pk}_0$, an anonymity set $\mathtt{A}$ is given.
- $\mathsf{tx}_{\mathsf{DA}} = (\{\mathsf{acct}\}_{i=1}^n, \{\mathsf{acct}'\}_{i=1}^n, \{\mathsf{userInfo}\}_{i=1}^n, \{\mathsf{userInfo}'\}_{i=1}^n \pi) \leftarrow \mathsf{DeleteAcct}(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}_\mathsf{D}, \mathsf{acct}_\mathsf{C}$ Delete a zero-balance account acct_D from the UTXO set from owner of sk, and adding its auditing info $(\mathtt{out}, \mathtt{in})$ to another account acct_C that shares the same sk. Anonymity sets $\mathtt{A}_1, \mathtt{A}_2$ are included to hide acct_C and $\mathsf{userInfo}$, respectively.
- $0/1 \leftarrow \mathsf{VerifyTrans}(\mathsf{tx}, \mathsf{state})$: It is a public verification algorithm that checks the validity of a transaction tx given the current state and outputs 1 if and only if it is valid.
- $\mathsf{state}' \leftarrow \mathsf{ApplyTrans}(\mathsf{tx}, \mathsf{state})$: Used to apply to the current state a transaction tx, after its verification.

- $\mathsf{auditInfo} = (\pi, \mathtt{\#accs}_1, \{\mathsf{acct}_{1i}\}_{i=1}^{\mathtt{\#accs}_1}, \mathtt{\#accs}_2, \{\mathsf{acct}_{2i}\}_{i=1}^{\mathtt{\#accs}_2}) \leftarrow \mathsf{Audit}(\mathsf{sk}, \mathtt{pk}_0\mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}))$: Used by a user with secret key sk and initial public key $\mathtt{pk}_0$ to generate a proof π for being compliant with policy f, concerning a specific period of time defined by two blockchain snapshots $\mathsf{state}_1, \mathsf{state}_2$. The $\mathtt{aux}$ variable contains the auxiliary information needed for the policy.

- $0/1 \leftarrow \mathsf{VerifyAudit}(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}), \mathsf{auditInfo})$. Used by the Audit Authority to check if the user with initial public key $\mathtt{pk}_0$ is compliant with policy f.

5.4 Security Model

An anonymous payment system should provide *anonymity* and *theft prevention*. Anonymity requires that an observer of the system cannot find the identities of senders and the receivers of a transaction if they don't own the sender's private key, and that even the recipient of a transaction cannot know the sender. Theft prevention means that users can only move funds from accounts they own. For the definitions of the anonymity and theft prevention properties, we adapt the definitions of Quisquis for the corresponding properties to AQQUA. Additionally, an auditable payment system requires the security property of *audit soundness*, which means that there cannot be a successfully verified audit generated by a user who is non-compliant.

We formally define these properties, using security games where the adversary has access to the following oracles.

- $\mathsf{sk} \leftarrow \mathsf{OCorrupt}(\mathsf{pk}, \mathsf{state})$: Returns the secret key that corresponds to a public key. The public key should belong either in an account or a user information entry of the state.

- $\mathsf{state} \leftarrow \mathsf{ORegister}()$: Creates a keypair and registers the public key. Returns the new state.

- $(\mathsf{tx}_{\mathsf{CA}}, \mathsf{state}) \leftarrow \mathsf{OCreateAcct}(\mathsf{userInfo}, \mathtt{A})$: Creates a new account for a $\mathsf{userInfo}$ entry using the anonymity set $\mathtt{A}$. Returns the corresponding transaction and resulting state after the transaction application.

- $(\mathsf{tx}_{\mathsf{DA}}, \mathsf{state}) \leftarrow \mathsf{ODeleteAcct}(\mathsf{userInfo}, \mathsf{acct}_{\mathsf{C}}, \mathsf{acct}_{\mathsf{D}}, \mathtt{A}_1, \mathtt{A}_2)$: Creates and applies a transaction to delete an account by calling $\mathsf{DeleteAcct}$. Returns the transaction and the resulting state after the transaction application.

- $(\mathsf{tx}, \mathsf{state}) \leftarrow \mathsf{OTrans}(\mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v}_\mathtt{S}}, \overrightarrow{\mathtt{v}_\mathtt{R}}, \mathtt{A})$: Creates and applies a transaction, returns the transaction and the new state.

- $\mathsf{state} \leftarrow \mathsf{OApplyTrans}(\mathsf{tx})$: Checks if a transaction is valid and if so, applies it. Returns the resulting state.

- $\mathsf{auditInfo} \leftarrow \mathsf{OAudit}(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}))$: Creates and returns an audit proof.

Our games make use of bookkeeping functionalities that can be called by the challenger and the available oracles. The bookkeeping keeps a list states of consecutive states created through oracle queries, a set $\mathsf{entries}$ containing all the

secret keys that control the accounts appearing in these states, and a partition of the keys set into honest and corrupt (controlled by the adversary) keys, honest and $\mathsf{corrupt}$, respectively. The bookkeeping functionalities are:

- $\mathsf{sk} \leftarrow \mathsf{findSecretKey}(\mathsf{pk}, \mathsf{state})$: Finds the secret key corresponding to a public key present in a state.
- $s \leftarrow \mathsf{totalWealth}(\mathsf{set}, \mathsf{state})$: Counts and returns the total amount of funds of the accounts of state that are owned by a set of secret keys ($\mathsf{set} = \mathsf{honest}$ or $\mathsf{set} = \mathsf{corrupt}$).
- $0/1 \leftarrow \mathsf{verifyPolicy}(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}))$: Checks whether $\mathtt{pk}_0$ is compliant with policy f for the time period represented by $\mathsf{state}_1, \mathsf{state}_2$.

The bookkeeping functionalities are presented in algorithm 5.1, and the oracles the adversary has access to are presented in algorithm 5.2.

5.4.1 Anonymity

In the anonymity game, the challenger first picks a bit $b \leftarrow\!\!\$\ \{0,1\}$. The adversary, after interacting with the oracles, has to output two sender accounts $\mathsf{acct}_0, \mathsf{acct}_1$, two receiver accounts $\mathsf{acct}'_0, \mathsf{acct}'_1$, two amounts v_0, v_1 and an anonymity set $\mathtt{A}$. Then, the challenger creates a transaction in which acct_b sends amount v_b to acct'_b using $\mathtt{A} \cup \{\mathsf{acct}_{1-b}\}$ as the anonymity set. Finally, the adversary has to guess b, and if they guess correctly, they win the game.

In the anonymity game, the following rules must be enforced or else the adversary could trivially guess b.

- Both senders must be honest. If one of the senders were corrupted, the adversary would be able to see whose account's balance decreases.
- Both receivers are honest. If both were corrupted then $\mathsf{acct}'_0 = \mathsf{acct}'_1$ and $v_0 = v_1$. If one is corrupted, the adversary would be able to see which account's balance increased or the amount by which it increased.

The anonymity game is presented in Game 5.3.

Definition 7. *The* advantage *of the adversary in winning the anonymity game is defined as:* $\mathsf{Adv}^{\mathrm{anon}}_{\mathcal{A}}(\lambda) = \mid \Pr[\mathsf{Exp}^{\mathrm{anon}}_{\mathcal{A}}(\lambda) = 1] - \frac{1}{2} \mid$

A DPS satisfies anonymity *if for every PPT adversary* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{anon}}_{\mathcal{A}}(\lambda)$ *is negligible in* λ.

5.4.2 Theft Prevention

In order for the adversary to win the theft prevention game, they have to output a valid transaction that, when applied, either increases the wealth of the users they control, decreases the wealth of the honest parties, or alters the total wealth of all the users (i.e. the adversary's transaction either created or destroyed wealth). The theft prevention game is presented in Game 5.4.

Definition 8. *The* advantage *of the adversary in winning the theft prevention game is defined as* $\mathsf{Adv}^{\mathrm{theft}}_{\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\mathrm{theft}}_{\mathcal{A}}(\lambda) = 1]$ *A DPS satisfies* theft prevention *if for every PPT adversary* $\mathcal{A}$, $\mathsf{Adv}^{\mathrm{theft}}_{\mathcal{A}}(\lambda)$ *is negligible in* λ.

Algorithm 5.1: bookkeeping functionalities

```
entries ← ∅                                   // set of all secret keys
corrupt ← ∅                                   // set of corrupt secret keys
honest ← ∅                                    // set of honest secret keys
states ← []          // list of states, updated through oracles
Function findSecretKey(pk, state)
    if state ∉ states then
        return ⊥
    for sk ∈ entries do
        for acct ∈ state.UTXOSet do
            if acct.pk = pk ∧ VerifyKP(sk, acct.pk) = 1 then
                return sk
        for userInfo ∈ state.UserSet do
            if userInfo.pk_0 = pk ∧ VerifyKP(sk, userInfo.pk) = 1 then
                return sk
    return ⊥
Function totalWealth(set, state)
    s ← 0
    for sk ∈ set do
        for acct ∈ state.UTXOSet do
            if VerifyKP(sk, acct.pk) then
                s ← s + OpenCom(sk, acct.com_bl)
    return s
Function verifyPolicy(pk_0, state_1, state_2, f, aux)
    if state_1, state_2 ∉ states ∨ state_1 is not older than state_2 then
        return ⊥
    A_1, A_2 ← ∅, ∅
    sk ← findSecretKey(pk_0, state_1)
            // Find accounts owned by sk in state_1.UTXOSet and
     state_2.UTXOSet resp.
    for acct ∈ state_1.UTXOSet do
        if VerifyKP(sk, acct.pk) then
            A_1 ← A_1 ∪ {acct}
    for acct ∈ state_2.UTXOSet do
        if VerifyKP(sk, acct.pk) then
            A_2 ← A_2 ∪ {acct}
    if f(pk_0, (state_1, state_2), aux) = 1 then
                                  // Check if f holds using A_1, A_2, sk
        return 1
    return 0
```

Algorithm 5.2: Oracles for security definitions

Oracle OCorrupt(pk, state)
  // pk should be a key of an account or user information in state, aborts otherwise
  sk ← findSecretKey(pk, state)
  honest ← honest \ {sk}
  corrupt ← corrupt ∪ {sk}
  return sk
Oracle ORegister()
  state ← bookkeeping.states[−1] // most recent state of bookkeeping
  (sk, userInfo, acct, π) ← Register()
  if VerifyRegister(userInfo, acct, π, state) = 0 **then**
    return ⊥ // cannot be registered given current state
  entries ← entries ∪ {sk}
  honest ← honest ∪ {sk}
  state′ ← ApplyRegister(userInfo, acct, state); states ← states ∪ [state′]
  return state′
Oracle OCreateAcct(userInfo, A)
  state ← bookkeeping.states[−1] // most recent state of bookkeeping
  tx_{CA} ← CreateAcct(userInfo, A)
  if VerifyTrans(tx_{CA}, state) = 0 **then**
    return ⊥ // transaction cannot be applied to state
  state′ ← ApplyTrans($\mathsf{tx}_{\mathsf{CA}}$, state); states ← states ∪ [state′]
  return tx_{CA}, state′
Oracle ODeleteAcct(userInfo, acct_C, acct_D, $\mathtt{A}_1$, $\mathtt{A}_2$)
  state ← bookkeeping.states[−1]
  sk ← findSecretKey(acct_C)
  $\mathsf{tx}_{\mathsf{DA}}$ ← DeleteAcct(sk, userInfo, $\mathsf{acct}_{\mathsf{C}}$, $\mathsf{acct}_{\mathsf{D}}$, $\mathtt{A}_1$, $\mathtt{A}_2$)
  if VerifyTrans(tx_{DA}, state) = 0 **then**
    return ⊥ // transaction cannot be applied to state
  state′ ← ApplyTrans($\mathsf{tx}_{\mathsf{DA}}$, state); states ← states ∪ [state′]
  return tx_{DA}, state′
Oracle OTrans(S, R, $\vec{v_{\mathtt{S}}}$, $\vec{v_{\mathtt{R}}}$, A)
  state ← bookkeeping.states[−1] // most recent state of bookkeeping
  for sk ∈ entries **do**
    Take an arbitrary acct ∈ S
    if VerifyKP(sk, acct.pk) = 1 **then**
      tx ← Trans(S, R, $\vec{v_{\mathtt{S}}}$, $\vec{v_{\mathtt{R}}}$A) // If sk is not the owner of all accounts in S, the transaction will not be created.
      if VerifyTrans(tx, state) = 0 **then**
        return ⊥ // transaction cannot be applied to state
      state′ ← ApplyTrans(tx, state); states ← states ∪ [state′]
      return tx, state′
  return ⊥
Oracle OApplyTrans(tx)
  if VerifyTrans(tx, state) = 0 **then**
    return ⊥
  state′ ← ApplyTrans(tx, state)
  states ← states ∪ [state′]; **return** state′
Oracle OAudit(pk$_0$, state$_1$, state$_2$, f, aux)
  sk ← findSecretKey(pk$_0$, state$_1$)
  if state$_1$, state$_2$ ∈ states ∧ state$_1$ **is older than** state$_2$ **then**
    auditInfo ← Audit(sk, pk$_0$, state$_1$, state$_2$, f, aux)
    if VerifyAudit(pk$_0$, state$_1$, state$_2$, (f, aux), auditInfo) **then**
      return auditInfo
  return ⊥ // pk was invalid for the snapshots

Game 5.3: Anonymity game $\mathsf{Exp}^{\mathrm{anon}}_{\mathcal{A}}(\lambda)$

Input : λ
Output: $\{0,1\}$

$b \leftarrow \{0,1\}$
$(\mathsf{state}_0, \mathsf{pp}) \leftarrow \mathsf{Setup}(\lambda)$
$(\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1, \mathtt{A}, v_0, v_1) \leftarrow$
$\mathcal{A}^{\mathsf{OCorrupt},\mathsf{ORegister},\mathsf{OCreateAcct},\mathsf{ODeleteAcct},\mathsf{OTrans},\mathsf{OApplyTrans}}(\mathsf{state}_0)$
$\mathsf{state} \leftarrow \mathsf{states}[-1]$ `// most recent state of bookkeeping`
$\mathsf{sk}_0 \leftarrow \mathsf{findSecretKey}(\mathsf{acct}_0.\mathsf{pk}, \mathsf{state})$; $\mathsf{sk}_1 \leftarrow \mathsf{findSecretKey}(\mathsf{acct}_1.\mathsf{pk}, \mathsf{state})$

$\mathsf{sk}'_0 \leftarrow \mathsf{findSecretKey}(\mathsf{acct}'_0.\mathsf{pk}, \mathsf{state})$; $\mathsf{sk}'_1 \leftarrow \mathsf{findSecretKey}(\mathsf{acct}'_1.\mathsf{pk}, \mathsf{state})$

if
$(\mathsf{sk}_0 \in \mathsf{corrupt} \vee \mathsf{sk}_1 \in \mathsf{corrupt}) \vee ((\mathsf{sk}'_0 \in \mathsf{corrupt} \vee \mathsf{sk}'_1 \in \mathsf{corrupt}) \wedge ((\mathsf{acct}'_0 \neq \mathsf{acct}'_1) \vee (\mathsf{acct}'_0 = \mathsf{acct}'_1 \wedge v_0 \neq v_1))) \vee (\mathsf{acct}_0.\mathtt{bl} < v_0 \vee \mathsf{acct}_1.\mathtt{bl} < v_1)$
then
  return $\perp$
for $y \in \{0,1\}$ **do**
  $\mathtt{A}_y \leftarrow \mathtt{A}$
  if $\mathsf{sk}_0 \neq \mathsf{sk}_1$ **then**
    $\mathtt{A}_y \leftarrow \mathtt{A} \cup \{\mathsf{acct}_{1-y}\}$
  if $\mathsf{sk}'_0 \neq \mathsf{sk}'_1$ **then**
    $\mathtt{A}_y \leftarrow \mathtt{A} \cup \{\mathsf{acct}'_{1-y}\}$
  $\mathsf{tx}_y \leftarrow \mathsf{Trans}(\mathsf{sk}_y, \{\mathsf{acct}_y\}, \{\mathsf{acct}'_y\}, (-v_y), (v_y), \mathtt{A}_y)$
  if $\mathsf{VerifyTrans}(\mathsf{tx}_y, \mathsf{state}) = 0$ **then**
    return $\perp$
$\mathsf{state}' \leftarrow \mathsf{ApplyTrans}(\mathsf{tx}_b, \mathsf{state})$
$b' \leftarrow \mathcal{A}(\mathsf{state}')$
return $(b = b')$

Game 5.4: Theft prevention game $\mathsf{Exp}^{\text{theft}}_{\mathcal{A}}(\lambda)$

Input : λ
Output: $\{0,1\}$

$(\mathsf{state}_0, \mathsf{pp}) \leftarrow \mathsf{Setup}(\lambda)$
$\mathsf{tx} \leftarrow \mathcal{A}^{\mathsf{OCorrupt},\mathsf{ORegister},\mathsf{OCreateAcct},\mathsf{ODeleteAcct},\mathsf{OTrans},\mathsf{OApplyTrans}}(\mathsf{state}_0)$
$\mathsf{state} \leftarrow \mathsf{states}[-1]$ `// most recent state of bookkeeping`
$s_h \leftarrow \mathsf{totalWealth}(.\mathsf{honest}, \mathsf{state})$
$s_c \leftarrow \mathsf{totalWealth}(\mathsf{corrupt}, \mathsf{state})$
if $\mathsf{VerifyTrans}(\mathsf{tx}, \mathsf{state}) = 0$ **then**
  return $\perp$
$\mathsf{state}' \leftarrow \mathsf{ApplyTrans}(\mathsf{tx}, \mathsf{state})$
$s'_h \leftarrow \mathsf{totalWealth}(\mathsf{honest}, \mathsf{state}')$
$s'_c \leftarrow \mathsf{totalWealth}(\mathsf{corrupt}, \mathsf{state}')$
return $(s'_h < s_h) \vee (s'_c > s_c) \vee (s'_c + s'_h \neq s_c + s_h)$

5.4.3 Audit soundness

In order for the adversary to win the audit soundness game for a policy f, they have to output a valid audit proof for a user that is non-compliant regarding the particular policy. The audit soundness game is presented in Game 5.5.

Game 5.5: Audit soundness game $\mathsf{Exp}^{\text{ausound}}_{\mathcal{A},f}(\lambda)$

Input : λ
Output: $\{0,1\}$

$b \leftarrow \{0,1\}$
$\mathsf{state}_0, \mathsf{pp} \leftarrow \mathsf{Setup}(\lambda)$
$(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, f, \mathtt{aux}, \mathsf{auditInfo}) \leftarrow$
$\mathcal{A}^{\mathsf{OCorrupt},\mathsf{ORegister},\mathsf{OCreateAcct},\mathsf{ODeleteAcct},\mathsf{OTrans},\mathsf{OApplyTrans},\mathsf{OAudit}}(\mathsf{state}_0)$
if $\mathsf{VerifyAudit}(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}), \mathsf{auditInfo}) = 1$ **then**
  `// run bookkeeping and check if` f `is satisfied and that` $\mathsf{state}_1, \mathsf{state}_2$ `are valid`
  if $\mathsf{verifyPolicy}(\mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux})) = 1$ **then**
    return 0
  else
    return 1
else
  return $\perp$

Definition 9. *The* advantage *of the adversary in winning the audit soundness game for policy f is defined as:* $\mathsf{Adv}^{\text{ausound}}_{\mathcal{A},f}(\lambda) = \Pr[\mathsf{Exp}^{\text{ausound}}_{\mathcal{A},f}(\lambda) = 1]$ *A DPS satisfies* audit soundness *for a policy f if for every PPT adversary $\mathcal{A}$, $\mathsf{Adv}^{\text{ausound}}_{\mathcal{A},f}(\lambda)$ is negligible in λ.*

5.5 Our construction

We now define AQQUA by realising the functionalities of an auditable payment scheme.

The Register algorithm performs the following steps:

1. Run $(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}) \leftarrow \mathsf{GenUser}()$.
2. Create a zero-knowledge proof π of the relation $R(x, w)$, where $x = (\mathsf{acct}, \mathsf{userInfo}), w = (\mathsf{sk})$ and $R(x, w) = 1$ if:

$$\begin{aligned}
&\mathsf{VerifyCom}(\mathsf{userInfo.pk}_0, \mathsf{userInfo.com}_{\#\mathtt{accs}}, (\mathsf{sk}, 1)) = 1 \\
&\wedge \mathsf{VerifyKP}(\mathsf{userInfo.pk}_0, \mathsf{sk}) = 1 \\
&\wedge \mathsf{VerifyKP}(\mathsf{acct.pk}, \mathsf{sk}) = 1 \\
&\wedge \mathsf{VerifyCom}(\mathsf{acct.pk}, \mathsf{acct.com}_{\mathtt{bl}}, (\mathsf{sk}, 0)) = 1 \\
&\wedge \mathsf{VerifyCom}(\mathsf{acct.pk}, \mathsf{acct.com}_{\mathtt{out}}, (\mathsf{sk}, 0)) = 1 \\
&\wedge \mathsf{VerifyCom}(\mathsf{acct.pk}, \mathsf{acct.com}_{\mathtt{in}}, (\mathsf{sk}, 0)) = 1
\end{aligned}$$

3. Return $(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}, \pi)$.

Figure 5.1: The Register algorithm.

5.5.1 Setup

The Setup algorithm takes as input the security parameter λ and returns the output of UPK.Setup and the initial state which contains an empty UserSet and UTXOSet.

5.5.2 Registration

In order for users to register in the system, they first use the Register algorithm to create a secret key, a userInfo entry and a first empty account acct. The Register algorithm also provides proofs that userInfo, acct have been properly created. Then, the user sends userInfo, acct and the proofs to the RA and the RA verifies the proofs using the VerifyRegister algorithm. If the proofs verify, the RA adds userInfo to the UserSet and acct to the UTXOSet using the ApplyRegister algorithm.

Register

The Register algorithm creates a secret key sk, the entry $\mathsf{userInfo} = (\mathsf{pk}_0, \boxed{1})$ that will be later stored in the UserSet, the user's first account $\mathsf{acct} = (\mathsf{pk}, \boxed{0}, \boxed{0}, \boxed{0})$ and a zero-knowledge proof π for the fact that the commitments $\boxed{1}$ of userInfo and $\boxed{0}, \boxed{0}, \boxed{0}$ of acct are indeed commitments to the correct values. The proof π can be posted on-chain for public verification.

The user must keep sk secret, and sends through a secure channel $\mathsf{userInfo}, \mathsf{acct}, \pi$ to the RA, together with their real-world identity information. The detailed description of the Register algorithm is depicted in Figure 5.1.

Verify Register

The $\mathsf{VerifyRegister}(\mathsf{userInfo}, \mathsf{acct}, \pi, \mathsf{state})$ algorithm guarantees the validity of the registration information. It first checks that the $\mathsf{userInfo.pk}_0$ does not already exist in a userInfo entry of UserSet. Afterwards, it executes the verification algorithm for the NIZK argument π and returns its result.

Apply Register

The ApplyRegister(userInfo, acct, state) algorithm runs after the registration verification and adds a new record to the UserSet containing the userInfo as well as a new record to the UTXOSet containing the newly created account acct.

5.5.3 Transactions

Trans Algorithm

Transactions enable a sender to redistribute their wealth to one or more recipients. Similarly to Quisquis [11], transactions are composed of input and output sets, which both include the sender and the intended recipients, and a NIZK proof that the output list has been computed according to the protocol specification. We assume that the size of each of the inputs and outputs sets is a predetermined number $\mathtt{n}$.

The Trans algorithm is used to create a transaction that redistributes a number of coins from a set of sender accounts, which are owned by the same secret key, to a set of receiver accounts. In order to substract an amount from a sender account or add an amount to a receiver account, the homomorphic property of the commitment scheme is used. Furthermore, in the algorithm the total amount sent and total amount received of the sender and receiver accounts is also updated appropriately. Finally, the account public keys are re-randomized in order to hide the connection between the input and output accounts.

In order to hide the participating accounts, an anonymity set is included. The balances of the accounts belonging to the anonymity set do not change, however the commitments and the public keys are re-randomized in order to be indistinguishable from the actual participating accounts. The account updates happen though the invocation of the UpdateAcct algorithm, and the outputs set is composed of these updated accounts.

The ordering of the accounts in the input and output sets should not remain the same, since this trivially reveals the link between every account and its update. Therefore, the input and output lists are always ordered in some canonical order. This can be thought of as applying a random permutation to shuffle the updated accounts.

The detailed description of the Trans algorithm is depicted in Figure 5.2. It takes as input the sender's secret key sk, the set of sender accounts $\mathtt{S}$, the set of receiver accounts $\mathtt{R}$, two vectors $\overrightarrow{\mathtt{v_S}}, \overrightarrow{\mathtt{v_R}}$ containing the desired changes to the balances of the sender and receiver accounts respectively, and an anonymity set $\mathtt{A}$. It returns a transaction $\mathsf{tx} = (\mathtt{inputs}, \mathtt{outputs}, \pi)$, where π is a zero-knowledge proof that $\mathtt{outputs}$ is created correctly.

Due to the way transactions are generated, every address appears at most twice: once when it is created in the output of some transaction, and once when included in the inputs of another transaction (regardless of whether it serves as the actual sender or is only included for anonymity).

Our transaction construction is similar to the one of Quisquis [11], with the difference that we introduce the vectors $\overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}$ to perform the updates to the associated total amount sent and total amount received of the accounts.

The algorithm $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \vec{\mathtt{v}_\mathtt{S}}, \vec{\mathtt{v}_\mathtt{R}}, \mathtt{A})$ performs the following steps:

1. Ensure that for each $\mathsf{acct} \in \mathtt{S}$, $\mathsf{VerifyKP}(\mathsf{sk}, \mathsf{acct.pk}) = 1$, and that $|\mathtt{S}| = |\vec{\mathtt{v}_\mathtt{S}}|, |\mathtt{R}| = |\vec{\mathtt{v}_\mathtt{R}}|$.
2. Let $\mathtt{I_S} = \{1, \ldots, |\mathtt{S}|\}$. For all $i \in \mathtt{I_S}$, calculate the opening of the committed balance $\boxed{\mathtt{bl}_i}$ of $\mathsf{acct}_i \in \mathtt{S}$, denoted $\mathtt{bl}_i$.
3. Let $\vec{\mathtt{v}_\mathtt{bl}} = \vec{\mathtt{v}_\mathtt{S}} || \vec{\mathtt{v}_\mathtt{R}}$, where $||$ denotes vector concatenation. Let also $\mathtt{I_R} = \{|\mathtt{S}| + 1, \ldots, |\mathtt{S}| + |\mathtt{R}|\}$. Ensure that:
   (a) $\sum_{i \in \mathtt{I_S} \cup \mathtt{I_R}} \mathtt{v}_{\mathtt{bl}i} = 0$
   (b) $\forall i \in \mathtt{I_R} : \mathtt{v}_{\mathtt{bl}i} \in \mathcal{V}$
   (c) $\forall i \in \mathtt{I_S} : -\mathtt{v}_{\mathtt{bl}i} \in \mathcal{V} \wedge \mathtt{bl}_i + \mathtt{v}_{\mathtt{bl}i} \in \mathcal{V}$
4. Construct $\vec{\mathtt{v}_\mathtt{out}}, \vec{\mathtt{v}_\mathtt{in}}$ as follows:
   (a) $\vec{\mathtt{v}_\mathtt{out}} = \vec{\mathtt{v}_\mathtt{S}} || \underbrace{(0, \ldots, 0)}_{\text{length } |\mathtt{R}|} || \underbrace{(0, \ldots, 0)}_{\text{length } |\mathtt{A}|}$
   (b) $\vec{\mathtt{v}_\mathtt{in}} = \underbrace{(0, \ldots, 0)}_{\text{length } |\mathtt{S}|} || \vec{\mathtt{v}_\mathtt{R}} || \underbrace{(0, \ldots, 0)}_{\text{length } |\mathtt{A}|}$.

   Furthermore, expand $\vec{\mathtt{v}_\mathtt{bl}}$ too with zero values for each $\mathsf{acct} \in \mathtt{A}$.
5. Order $\mathtt{P} \cup \mathtt{A}$ in some canonical order and let $\mathtt{inputs}$ be the result. Let also $\vec{\mathtt{v}_\mathtt{bl}}', \vec{\mathtt{v}_\mathtt{out}}', \vec{\mathtt{v}_\mathtt{in}}'$ be the permutation of $\vec{\mathtt{v}_\mathtt{bl}}, \vec{\mathtt{v}_\mathtt{out}}, \vec{\mathtt{v}_\mathtt{in}}$ in the same order. Let $\mathtt{I}^*_\mathtt{S}, \mathtt{I}^*_\mathtt{R}, \mathtt{I}^*_\mathtt{A}$ denote the indices of the respective accounts of the sender, the recipients and the anonymity set in this list.
6. Pick $r_1, r_2, r_3, r_4 \leftarrow\!\!\$\ \mathcal{R}$ and let $\vec{r} = (r_1, r_2, r_3, r_4)$.
   Perform $\mathsf{UpdateAcct}(\mathtt{inputs}, \vec{\mathtt{v}_\mathtt{bl}}', \vec{\mathtt{v}_\mathtt{out}}', \vec{\mathtt{v}_\mathtt{in}}'; \vec{r})$, order the result in some canonical order, and denote by $\mathtt{outputs}$ the final result.
7. Let $\psi : [\mathtt{n}] \to [\mathtt{n}]$ be the implicit permutation mapping $\mathtt{inputs}$ into $\mathtt{outputs}$; such that accounts $\mathsf{acct}_i \in \mathtt{inputs}$ and $\mathsf{acct}'_{\psi(i)} \in \mathtt{outputs}$ share the same secret key.
8. Form a zero-knowledge proof π of the relation $R(x, w)$, where $x = (\mathtt{inputs}, \mathtt{outputs}), w = (\mathsf{sk}, \{\mathtt{bl}_i, \mathtt{out}_i, \mathtt{in}_i\}_{i \in \mathtt{I}^*_\mathtt{S}}, \vec{\mathtt{v}_\mathtt{bl}}', \vec{\mathtt{v}_\mathtt{out}}', \vec{\mathtt{v}_\mathtt{in}}', \vec{r}, \psi, \mathtt{I}^*_\mathtt{S}, \mathtt{I}^*_\mathtt{R}, \mathtt{I}^*_\mathtt{A})$, and $R(x, w) = 1$ if

$$
\begin{aligned}
&\mathsf{VerifyUpdateAcct}(\mathsf{acct}'_{\psi(i)}, \mathsf{acct}_i, 0, 0, 0; \vec{r}) = 1 \ \forall i \in \mathtt{I}^*_\mathtt{A} \\
&\wedge (\mathsf{VerifyUpdateAcct}(\mathsf{acct}'_{\psi(i)}, \mathsf{acct}_i, \mathtt{v}_{\mathtt{bl}i}', \mathtt{v}_{\mathtt{out}i}', \mathtt{v}_{\mathtt{in}i}'; \vec{r}) = 1 \wedge \mathtt{v}_{\mathtt{bl}i}', \mathtt{v}_{\mathtt{out}i}', \mathtt{v}_{\mathtt{in}i}' \in \mathcal{V}) \ \forall i \in \mathtt{I}^*_\mathtt{R} \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathsf{acct}'_{\psi(i)}, \mathsf{acct}_i, \mathtt{v}_{\mathtt{bl}i}', \mathtt{v}_{\mathtt{out}i}', \mathtt{v}_{\mathtt{in}i}'; \vec{r}) = 1 \ \forall i \in \mathtt{I}^*_\mathtt{S} \\
&\wedge \mathsf{VerifyAcct}(\mathsf{acct}'_{\psi(i)}, \mathsf{sk}, \mathtt{bl}_i + \mathtt{v}_{\mathtt{bl}i}', \mathtt{out}_i + \mathtt{v}_{\mathtt{out}i}', \mathtt{in}_i + \mathtt{v}_{\mathtt{in}i}') = 1 \ \forall i \in \mathtt{I}^*_\mathtt{S} \\
&\wedge \sum_{i \in \mathtt{I}^*_\mathtt{S} \cup \mathtt{I}^*_\mathtt{R} \cup \mathtt{I}^*_\mathtt{A}} \mathtt{v}_{\mathtt{bl}i}' = 0 \\
&\wedge \mathtt{v}_{\mathtt{bl}i}' = \mathtt{v}_{\mathtt{out}i}' \ \forall i \in \mathtt{I}^*_\mathtt{S} \\
&\wedge \mathtt{v}_{\mathtt{bl}i}' = \mathtt{v}_{\mathtt{in}i}' \ \forall i \in \mathtt{I}^*_\mathtt{R} \\
&\wedge \mathtt{v}_{\mathtt{out}i}' = \mathtt{v}'_{\mathtt{in}i} = 0 \ \forall i \in \mathtt{I}^*_\mathtt{A}
\end{aligned}
$$

The transaction created is $\mathsf{tx} = (\mathtt{inputs}, \mathtt{outputs}, \pi)$.

Figure 5.2: The Trans algorithm.

The algorithm $\mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A})$ performs the following steps:

1. Pick $r_1, r_2, r_3, r_4 \leftarrow\!\!\$\ \mathcal{R}$ and let $\vec{r} = (r_1, r_2, r_3, r_4)$. Let $\mathsf{acct} = (\mathsf{pk}, \boxed{0}, \boxed{0}, \boxed{0})$ be the output of $\mathsf{NewAcct}(\mathsf{userInfo}.\mathsf{pk}_0; \vec{r})$.
2. Let $\mathtt{inputs} = \{\mathsf{userInfo}\} \cup \mathtt{A}$ in some canonical order. Let $\mathtt{c}, \mathtt{I_A}$ be the indices of the chosen initial public key for which we wish to construct the new account, and the anonymity set in this list.
3. Construct $\vec{\mathtt{v}}$ as follows: $\mathtt{v}_i = 0\ \forall i \in \mathtt{I_A}$ and $\mathtt{v_c} = 1$.
4. Pick $r_5 \leftarrow\!\!\$\ \mathcal{R}$ and let $\mathtt{outputs}$ be the output of $\mathsf{UpdateUser}(\mathtt{inputs}, \vec{\mathtt{v}}; r_5)$.
5. Form a zero-knowledge proof π of the relation $R(x, w)$, where $x = (\mathsf{acct}, \mathtt{inputs}, \mathtt{outputs}), w = (c, \vec{\mathtt{v}}, \vec{r}, r_5)$ and $R(x, w) = 1$ if $\forall i \in \{\mathtt{c}\} \cup \mathtt{I_A},\ \mathsf{userInfo}_i \in \mathtt{inputs}, \mathsf{userInfo}'_i \in \mathtt{outputs}$ we have that:

$$\begin{aligned}
&\mathsf{VerifyUpdateUser}(\mathsf{userInfo}'_i, \mathsf{userInfo}_i, 0; r_5) = 1\ \forall i \in \mathtt{I_A} \\
&\wedge\ \mathsf{VerifyUpdateUser}(\mathsf{userInfo}'_\mathtt{c}, \mathsf{userInfo}_\mathtt{c}, 1; r_5) = 1 \\
&\wedge\ \mathsf{VerifyUpdate}(\mathsf{acct.pk}, \mathsf{userInfo}_c.\mathsf{pk}_0, r_1) = 1 \\
&\wedge\ \mathsf{Commit}(\mathsf{acct.pk}, 0; r_2) = \mathsf{acct.com}_{\mathtt{bl}} \\
&\wedge\ \mathsf{Commit}(\mathsf{acct.pk}, 0; r_3) = \mathsf{acct.com}_{\mathtt{out}} \wedge \mathsf{Commit}(\mathsf{acct.pk}, 0; r_4) = \mathsf{acct.com}_{\mathtt{in}}
\end{aligned}$$

The final transaction returned by the algorithm is $\mathsf{tx_{CA}} = (\mathsf{acct}, \mathtt{inputs}, \mathtt{outputs}, \pi)$.

Figure 5.3: The CreateAcct algorithm.

Create Account Algorithm

Within the system every user can create a new account for any other registered user, which improves the efficiency of the system [11]. Since each account can appear only once as input in a transaction, if two concurrent transactions include the same account in their input set, one of them should be rejected. As the number of accounts within the system increases, the probability of a non-empty intersection between two transaction input sets decreases. In addition, creating new accounts allows users to own a fixed key that can be used to receive funds, instead of the key constantly changing. Therefore, it improves the overall communication overhead.

New accounts are composed of updates of the initial public key stored in the user's userInfo and commitments to zero values for the other attributes related to bl, out, in. Moreover, userInfo is updated, by increasing the committed value for the number of accounts the user owns. This is achieved by using the homomorphic property of the commitment scheme.

In order to hide the userInfo that corresponds to the user, an anonymity set A is used. The values of the commitments of the userInfo that belong to the anonymity set are re-randomized without changing their committed values. That is, transactions that create new accounts are composed of input and output sets, which both include the intended user's userInfo, and also the newly created account. The userInfo updates happen through the invocation of the UpdateUser algorithm, and the outputs set is composed of these updated userInfo.

The detailed description of the CreateAcct algorithm is depicted in Figure 5.3. It takes as input the userInfo of the intended user and an anonymity set A. It returns a transaction $\mathsf{tx_{CA}} = (\mathsf{acct}, \mathtt{inputs}, \mathtt{outputs}, \pi)$.

Delete Account Algorithm

Allowing users to delete zero-balance accounts reduces the storage overhead of AQQUA, since accounts that have no balance left to spend might be abandoned and thus not needed to be stored in the UTXOSet. Furthermore, due to the fact that senders usually create new accounts for their intended recipients, the number of accounts in the UTXOSet increases if the option to remove zero-balance accounts is not given. We note that users should be incentivized to delete the zero-balance accounts they own and don't need to keep. The mechanism to do so is left for future work.

In order to delete an account, the information containing the total amount $\texttt{out}, \texttt{in}$ sent and received by the account must be transferred to another account acct_C of the corresponding owner. In order to hide acct_C an anonymity set is included.

The detailed description of the DeleteAcct algorithm is depicted in Figure 5.4. The algorithm takes as input the secret key sk, the account to be deleted acct_D, the account acct_C to which $\texttt{out}, \texttt{in}$ of acct_D will be transferred, and anonymity sets $\mathtt{A}_1$ for the UTXOSet and $\mathtt{A}_2$ for the UserSet respectively. It returns a transaction $\mathsf{tx}_\mathsf{DA} = (\texttt{inputs}, \texttt{outputs}, \pi)$.

Trans Verification

The $\mathsf{VerifyTrans}(\mathsf{tx}, \mathsf{state})$ algorithm guarantees the validity of transaction tx. Depending on they transaction type $(\mathsf{tx}, \mathsf{tx}_\mathsf{CA}, \mathsf{tx}_\mathsf{DA})$ performs the following steps:

- if tx is an output of the Trans algorithm, then it first checks that all the accounts listed in $\mathsf{tx}.\texttt{inputs}$ are deemed unspent in the current state, meaning for each $\mathsf{acct} \in \mathsf{tx}.\texttt{inputs}, \mathsf{acct} \in \mathsf{state}.\mathsf{UTXOSet}$. Afterwards, it executes the verification algorithm for the NIZK argument π and returns its result.

- if tx_CA is an output of the CreateAcct algorithm, then it first checks that all the userInfo listed in $\mathsf{tx}_\mathsf{CA}.\texttt{inputs}$ are registered, meaning, for each $\mathsf{userInfo} \in \mathsf{tx}_\mathsf{CA}.\texttt{inputs}$ we have that $\mathsf{userInfo} \in \mathsf{state}.\mathsf{UserSet}$. It also ensures that $\mathsf{tx}_\mathsf{CA}.\mathsf{acct} \notin \mathsf{state}.\mathsf{UTXOSet}$. Afterwards, it executes the verification algorithm for the NIZK argument π and returns its result.

- if tx_DA is an output of the DeleteAcct algorithm, then it first checks that all the accounts listed in $\mathsf{tx}.\texttt{inputs}_{\mathsf{UTXOSet}}$ belong to state.UTXOSet and similarly for $\texttt{inputs}_{\mathsf{userInfo}}$. Afterwards, it executes the verification algorithm for the NIZK argument π and returns its result.

Apply Transaction

The $\mathsf{ApplyTrans}(\mathsf{tx}, \mathsf{state})$ algorithm is executed after the verification of the transaction. It applies the transaction tx by updating the current state, adding $\mathsf{tx}.\texttt{outputs}$ and removing $\mathsf{tx}.\texttt{inputs}$.

- If tx is the result of the Trans algorithm, it updates only the state.UTXOSet with the new accounts.

- If tx is the result of the CreateAcct algorithm, it updates the state.UserSet and adds the newly created account in the state.UTXOSet.

The algorithm $\mathsf{DeleteAcct}(\mathsf{sk}, \mathsf{userInfo}, \mathsf{acct}_\mathsf{D}, \mathsf{acct}_\mathsf{C}, \mathtt{A}_1, \mathtt{A}_2)$ performs the following steps:

1. For the account acct_D, calculate the opening of the commitments $\mathsf{acct}_\mathsf{D}.\mathtt{com}_{\mathtt{out}}, \mathsf{acct}_\mathsf{D}.\mathtt{com}_{\mathtt{in}}$, denoted $\mathtt{out}_\mathsf{D}, \mathtt{in}_\mathsf{D}$, using the secret key sk.
2. Let $\mathtt{inputs}_{\mathsf{UTXOSet}} = \{\mathsf{acct}_\mathsf{C}\} \cup \mathtt{A}_1$ in some canonical order. Let $c^*, \mathtt{I}_{\mathtt{A}1}$ denote the indices of the account to be added the information and the accounts of the anonymity set in this list.
3. Construct $\overrightarrow{\mathtt{v}_{\mathtt{bl}}}, \overrightarrow{\mathtt{v}_{\mathtt{out}}}, \overrightarrow{\mathtt{v}_{\mathtt{in}}}$ as follows:
   - $\overrightarrow{\mathtt{v}_{\mathtt{bl}}} = 0 \ \forall i \in \{c^*\} \cup \mathtt{I}_{\mathtt{A}1}$
   - $\overrightarrow{\mathtt{v}_{\mathtt{out}}} = 0 \ \forall i \in \mathtt{I}_{\mathtt{A}1}$ and $\mathtt{v}_{\mathtt{out}\,c^*} = \mathtt{out}_\mathsf{D}$
   - $\overrightarrow{\mathtt{v}_{\mathtt{in}}} = 0 \ \forall i \in \mathtt{I}_{\mathtt{A}1}$ and $\mathtt{v}_{\mathtt{in}\,c^*} = \mathtt{in}_\mathsf{D}$
4. Pick $r_1, r_2, r_3, r_4 \leftarrow\!\!\$\ \mathcal{R}$. and let $\vec{r} = (r_1, r_2, r_3, r_4)$. Let $\mathtt{outputs}_{\mathsf{UTXOSet}}$ be the output of $\mathsf{UpdateAcct}(\mathtt{inputs}_{\mathsf{UTXOSet}}, \overrightarrow{\mathtt{v}_{\mathtt{bl}}}, \overrightarrow{\mathtt{v}_{\mathtt{out}}}, \overrightarrow{\mathtt{v}_{\mathtt{in}}}; \vec{r})$ in some canonical order.
5. Let $\psi : [\mathtt{n}] \to [\mathtt{n}]$ be the implicit permutation mapping $\mathtt{inputs}_{\mathsf{UTXOSet}}$ into $\mathtt{outputs}_{\mathsf{UTXOSet}}$; such that accounts $\mathsf{acct}_i \in \mathtt{inputs}_{\mathsf{UTXOSet}}$ and $\mathsf{acct}'_{\psi(i)} \in \mathtt{outputs}_{\mathsf{UTXOSet}}$ share the same secret key.
6. Form a zero-knowledge proof π_1 of the relation $R(x, w)$, where $x = (\mathsf{acct}_\mathsf{D}, \mathtt{inputs}_{\mathsf{UTXOSet}}, \mathtt{outputs}_{\mathsf{UTXOSet}}), w = (\mathsf{sk}, \mathtt{out}_\mathsf{D}, \mathtt{in}_\mathsf{D}, \vec{r}, \psi, c^*, \mathtt{I}_{\mathtt{A}1})$, and $R(x, w) = 1$ if

$$\begin{aligned}
&\mathsf{VerifyKP}(\mathsf{sk}, \mathsf{acct}_\mathsf{D}.\mathsf{pk}) = 1 \wedge \mathsf{VerifyKP}(\mathsf{sk}, \mathsf{acct}_{c^*}.\mathsf{pk}) = 1 \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathsf{acct}'_{\psi(i)}, \mathsf{acct}_i, 0, 0, 0; \vec{r}) = 1 \ \forall i \in \mathtt{I}_{\mathtt{A}1} \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathsf{acct}'_{\psi(c^*)}, \mathsf{acct}_{c^*}, 0, \mathtt{out}_\mathsf{D}, \mathtt{in}_\mathsf{D}; \vec{r}) = 1 \\
&\wedge \mathsf{VerifyCom}(\mathsf{acct}_\mathsf{D}.\mathsf{pk}, \mathsf{acct}_\mathsf{D}.\mathtt{com}_{\mathtt{bl}}, (\mathsf{sk}, 0)) = 1
\end{aligned}$$

7. Let $\mathtt{inputs}_{\mathsf{UserSet}} = \{\mathsf{userInfo}\} \cup \mathtt{A}_2$ in some canonical order. Let $s^*, \mathtt{I}_{\mathtt{A}2}$ denote the indices of the chosen initial public key for which we wish to construct the new account, and the anonymity set in this list.
8. Construct $\vec{\mathtt{v}}$ as follows: $\mathtt{v}_i = 0 \ \forall i \in \mathtt{I}_{\mathtt{A}2}$ and $\mathtt{v}_{s^*} = -1$.
9. Pick $r \leftarrow\!\!\$\ \mathcal{R}$ and let $\mathtt{outputs}_{\mathsf{UserSet}}$ be the output of $\mathsf{UpdateUser}(\mathtt{inputs}_{\mathsf{UserSet}}, \vec{\mathtt{v}}; r)$.
10. Form a zero-knowledge proof π_2 of the relation $R(x, w)$, where $x = (\mathtt{inputs}_{\mathsf{UserSet}}, \mathtt{outputs}_{\mathsf{UserSet}}), w = (\mathsf{sk}, r, s^*, \mathtt{I}_{\mathtt{A}2})$ and $R(x, w) = 1$ if $\forall i \in \{s^*\} \cup \mathtt{I}_{\mathtt{A}2}$ $\mathsf{userInfo}_i \in \mathtt{inputs}_{\mathsf{UserSet}}, \mathsf{userInfo}'_i \in \mathtt{outputs}_{\mathsf{UserSet}}$ we have that:

$$\begin{aligned}
&\mathsf{VerifyKP}(\mathsf{sk}, \mathsf{userInfo}_{s^*}.\mathsf{pk}_0) = 1 \\
&\wedge \mathsf{VerifyUpdateUser}(\mathsf{userInfo}'_i, \mathsf{userInfo}_i, 0; r) = 1 \ \forall i \in \mathtt{I}_{\mathtt{A}2} \\
&\wedge \mathsf{VerifyUpdateUser}(\mathsf{userInfo}'_{s^*}, \mathsf{userInfo}_{s^*}, -1; r) = 1
\end{aligned}$$

The final transaction returned by the algorithm is
$\mathsf{tx}_{\mathsf{DA}} = (\mathtt{inputs}_{\mathsf{UTXOSet}}, \mathtt{outputs}_{\mathsf{UTXOSet}}, \mathtt{inputs}_{\mathsf{UserSet}}, \mathtt{outputs}_{\mathsf{UserSet}}, \pi = (\pi_1, \pi_2))$.

Figure 5.4: The $\mathsf{DeleteAcct}$ algorithm.

- If tx is the result of the DeleteAcct algorithm, it updates both state.UserSet and state.UTXOSet.

Similarly to [11], upon receiving a new state, users whose accounts are included in a tranction's `inputs` should identify their updated accounts in `outputs`. This can be accomplished by iterating through every $\mathsf{acct} \in \texttt{outputs}$ and using $\mathsf{VerifyKP}(\mathsf{sk}, \mathsf{acct.pk})$. Once the user identifies an updated account, they can check whether their account was used as part of the anonymity set or as a recipient, by running $\mathsf{VerifyCom}(\mathsf{sk}, \mathsf{acct.pk}, \mathsf{acct.com}_{\texttt{bl}}, \texttt{bl})$, passing as input the account's previous balance `bl`. If the result is 1, then the account was used as part of the anonymity set. Otherwise, the user must find out the new value for the balance. The value is small enough so that the computation of its discrete logarithm takes place in a reasonable time.

5.5.4 Audit

Audit Algorithm

In the audit procedure, the `AA` selects a user by their initial public key $\texttt{pk}_0$ and a time period which is represented by two snapshots of the blockchain $(\mathsf{state}_1, \mathsf{state}_2)$. For the policies that are applied to transactions (namely $f_{\mathsf{txlimit}}, f_{\mathsf{open}}$), the snapshot state_2 should be the state that results from applying the transaction to state_1. In the case where the policy is applied to a specified period (for example in $f_{\mathsf{slimit}}, f_{\mathsf{rlimit}}, f_{\mathsf{np}}$), the snapshots $\mathsf{state}_1, \mathsf{state}_2$ should be the states right before the beginning and after the end of the period, respectively.

The user which participates in the auditing should open for each of the two snapshots the committed value of the number of accounts they own ($\boxed{\texttt{\#accs}}$ field of userInfo). Then, they should reveal their accounts in each of the two snapshots' UTXOSet. The number of accounts they reveal in each snapshot should be equal to the opening of the corresponding commitment. Revealing the accounts does not hurt the anonymity of the user, since from the indistinguishability property of the UPK scheme and the hiding property of the commitment scheme, the `AA` cannot link the accounts that will be revealed with updated versions of them that will appear as a result of the user participating in any new transaction.

After opening the commitment and revealing the account, the user creates a zero-knowledge proof that the sets of accounts satisfy the required policy predicate, as defined in subsection 5.3.5.

The detailed description of the Audit algorithm is depicted in Figure 5.5. It takes as input the user's secret key sk, the two blockchain snapshots $(\mathsf{state}_1, \mathsf{state}_2)$, and the policy f along with the necessary information `aux`.

Audit Verification

The VerifyAudit algorithm is executed by the `AA` to check the compliance of the user with a specific policy. Initially, the algorithm checks that the user has revealed `#accs` accounts that belongs to each selected snapshot, calculate the necessary values (i.e. multiplication of committed amounts), and then runs the verification algorithm for the NIZK argument π and returns its result.

The algorithm $\mathsf{auditInfo} \leftarrow \mathsf{Audit}(\mathsf{sk}, \mathtt{pk}_0, \mathsf{state}_1, \mathsf{state}_2, (f, \mathtt{aux}))$ performs the following steps:

1. Ensure that $\mathsf{VerifyKP}(\mathsf{sk}, \mathtt{pk}_0)$. For each snapshot $\mathsf{state}_j, j = 1, 2$ find the $\mathsf{userInfo}_j$ that contains $\mathtt{pk}_0$, and calculate $\mathtt{\#accs}_j = \mathsf{OpenCom}(\mathsf{sk}, \mathsf{userInfo}.\boxed{\mathtt{\#accs}_j})$
2. For each snapshot find the set of accounts $A_j = \{\mathsf{acct}_i\}_{i=1}^{\mathtt{\#accs}_j}$ that belong to the user. That is, $\forall \mathsf{acct} \in \mathsf{state}_j.\mathsf{UTXOSet}$, if $\mathsf{VerifyKP}(\mathsf{acct.pk}, \mathsf{sk}) = 1$, then add acct to A_j.
3. Form a zero-knowledge proof π_1 of the relation $R(x, w)$, where $x = (\mathtt{pk}_0, \{\mathtt{\#accs}_j, \boxed{\mathtt{\#accs}_j}, \{\mathsf{acct}_{ji}\}_{i=1}^{\mathtt{\#accs}_j}\}_{j=1}^{2}), w = (\mathsf{sk})$ and $R(x, w) = 1$ if:

$$\begin{aligned}&\mathsf{VerifyCom}(\mathtt{pk}_0, \boxed{\mathtt{\#accs}_j}, (\mathsf{sk}, \mathtt{\#accs}_j)) = 1 \ \forall j \in \{1, 2\}\\ &\wedge \mathsf{VerifyKP}(\mathtt{pk}_0, \mathsf{sk}) = 1\\ &\wedge \mathsf{VerifyKP}(\mathsf{acct}_{ji}.\mathsf{pk}, \mathsf{sk}) = 1 \ \forall i \in \{1, \ldots, \mathtt{\#accs}_j\}, \ \forall j \in \{1, 2\}\end{aligned}$$

If $f \in \{f_{\mathsf{slimit}}, f_{\mathsf{rlimit}}, f_{\mathsf{np}}\}$ then:

4. For each snapshot calculate $\boxed{\mathtt{out}_j^*} = \prod_{i=1}^{\mathtt{\#accs}_j} \mathsf{acct}_{ji}.\boxed{\mathtt{out}}, \boxed{\mathtt{in}_j^*} = \prod_{i=1}^{\mathtt{\#accs}_j} \mathsf{acct}_{ji}.\boxed{\mathtt{in}}$.
   Then calculate $\boxed{\mathtt{out}^*} = \boxed{\mathtt{out}_2^*} \cdot \left(\boxed{\mathtt{out}_1^*}\right)^{-1}, \boxed{\mathtt{in}^*} = \boxed{\mathtt{in}_2^*} \cdot \left(\boxed{\mathtt{in}_1^*}\right)^{-1}$.
   Finally, calculate $\mathtt{out}^* = \mathsf{OpenCom}(\mathsf{sk}, \boxed{\mathtt{out}^*}), \mathtt{in}^* = \mathsf{OpenCom}(\mathsf{sk}, \boxed{\mathtt{in}^*})$. These values represent the total amount of coins that the user spent/received in the selected period of time.
5. Form a zero-knowledge proof π_2 of the relation $R(x, w)$ where $x = (\{\mathsf{acct}_{1i}\}_{i=1}^{\mathtt{\#accs}_j}, \{\mathsf{acct}_{2i}\}_{i=1}^{\mathtt{\#accs}_j}, \boxed{\mathtt{out}^*}, \boxed{\mathtt{in}^*}, \mathtt{aux}), w = (\mathtt{out}^*, \mathtt{in}^*)$ and $R(x, w) = 1$ if:

$$f(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), \mathtt{aux}) = 1$$

If $f \in \{f_{\mathsf{txlimit}}, f_{\mathsf{open}}\}$ then:

4. For each snapshot calculate $\boxed{\mathtt{bl}_j^*} = \prod_{i=1}^{\mathtt{\#accs}_j} \mathsf{acct}_{ji}.\boxed{\mathtt{bl}}$. Then calculate $\boxed{\mathtt{bl}^*} = \boxed{\mathtt{bl}_2^*} \cdot \left(\boxed{\mathtt{bl}_1^*}\right)^{-1}$ and $\mathtt{bl}^* = \mathsf{OpenCom}(\mathsf{sk}, \boxed{\mathtt{bl}^*})$.
5. Form a zero-knowledge proof π_2 of the relation $R(x, w)$ where $x = (\{\mathsf{acct}_{1i}\}_{i=1}^{\mathtt{\#accs}_j}, \{\mathsf{acct}_{2i}\}_{i=1}^{\mathtt{\#accs}_j}, \boxed{\mathtt{bl}^*}, \mathtt{aux}), w = (\mathtt{bl}^*)$ and $R(x, w) = 1$ if:

$$f(\mathtt{pk}_0, (\mathsf{state}_1, \mathsf{state}_2), \mathtt{aux}) = 1$$

The final output is $\mathsf{auditInfo} = (\pi = (\pi_1, \pi_2), \mathtt{\#accs}_1, \{\mathsf{acct}_{1i}\}_{i=1}^{\mathtt{\#accs}}, \mathtt{\#accs}_2, \{\mathsf{acct}_{2i}\}_{i=1}^{\mathtt{\#accs}})$.

Figure 5.5: The Audit algorithm.

5.6 Instantiating the Zero-knowledge Proof

In order to implement the proof system of zero-knowledge arguments we use the following Σ-protocols defined in [11] as well as Σ-protocols for discrete logarithm equality due to Chaum and Pedersen. For completeness we describe it below.

- Σ_{vu}: proof a valid update. Prover shows knowledge of w such that $\mathsf{pk}' = pk^w$.

Prover$(\mathsf{pk}, \mathsf{pk}', w)$		**Verifier**$(\mathsf{pk}, \mathsf{pk}')$
$s \leftarrow\!\!\$\ \mathbb{F}_p$		
$\alpha \leftarrow \mathsf{pk}^s = (g^s, h^s)$	$\xrightarrow{\alpha}$	
	$\xleftarrow{c}$	$c \leftarrow\!\!\$\ \{0,1\}^\kappa$
$z \leftarrow cw + s$	$\xrightarrow{z}$	
		Check $\mathsf{pk}^z = (\mathsf{pk}')^c \cdot \alpha$

- Σ_{com} : proof of knowledge of two commitments of the same value v under different public keys. Prover shows knowledge of $w = (v, r_1, r_2)$ such that $\mathtt{com}_1 = \mathsf{Commit}(\mathsf{pk}_1, v; r_1), \mathtt{com}_2 = \mathsf{Commit}(\mathsf{pk}_2, v; r_2)$.

	$\mathsf{pk}_1 = (g_1, h_1), \mathtt{com}_1 = (c_1, d_1)$ $\mathsf{pk}_2 = (g_2, h_2), \mathtt{com}_2 = (c_2, d_2)$	
Prover(v, r_1, r_2)		**Verifier**
$v', r_1', r_2' \leftarrow\!\!\$\ \mathbb{F}_p$		
$(e_1, f_1) \leftarrow (g_1^{r_1'}, g^{v'} h_1^{r_1'})$		
$(e_2, f_2) \leftarrow (g_2^{r_2'}, g^{v'} h_2^{r_2'})$	$\xrightarrow{\alpha}$	
	$\xleftarrow{x}$	$x \leftarrow\!\!\$\ \{0,1\}^\kappa$
$(z_v, z_{r1}, z_{r_2}) \leftarrow x(v, r_1, r_2) + (v', r_1', r_2')$	$\xrightarrow{(z_v, z_{r1}, z_{r_2})}$	
		Check for $i = 1, 2$:
		$g_i^{z_r i} = c_i^x \cdot e_1$
		$g^{z_v} h_i^{z_r i} = d_i^x \cdot f_i$

5.6.1 zk-proof of transactions

In each transaction created from Trans algorithm a prover essentially has to prove that:

1. accounts in `outputs` are proper updates of `inputs`
2. the updates of balances satisfy preservation of value
3. balances in accounts of recipients and anonymity set do not decrease
4. the sender account in `outputs` contain a balance in $\mathcal{V}$
5. the vectors $\overrightarrow{\mathtt{v_{bl}}'}, \overrightarrow{\mathtt{v_{out}}'}$ have the same values for the sender accounts and $\overrightarrow{\mathtt{v_{bl}}'}, \overrightarrow{\mathtt{v_{in}}'}$ for the receivers accounts and $(\overrightarrow{\mathtt{v_{out}}'}, \overrightarrow{\mathtt{v_{in}}'})$ have zero value for the rest.

Properties 3,4 can be proved by range proofs and we implement them with Bulletproofs [6]. For the properties 1,2,5 we are doing the following analysis similar to Quisquis[11].

Let the sender's accounts be $\mathtt{inputs}_1, \ldots, \mathtt{inputs}_s$ and the receivers' accounts be $\mathtt{inputs}_{s+1}, \ldots, \mathtt{inputs}_t$.

In order to easily verify the validity of the updates, the prover creates accounts $\vec{\mathsf{acct}}_\delta$, where $\mathsf{acct}_{\delta,i} = (\mathsf{pk}_i, \boxed{\mathtt{v}_{\mathtt{bl}i}}, \boxed{\mathtt{v}_{\mathtt{out}i}}, \boxed{\mathtt{v}_{\mathtt{in}i}})$. Now in order to prove property 5, the prover shows that for $\mathsf{acct}_{\delta,1}, \ldots, \mathsf{acct}_{\delta,s}$ the values under the $\boxed{\mathtt{v}_{\mathtt{bl}i}}$ and $\boxed{\mathtt{v}_{\mathtt{out}i}}$ are the same. Respectively for the recipients, for $\mathsf{acct}_{\delta,s+1}, \ldots, \mathsf{acct}_{\delta,t}$ the values under the $\boxed{\mathtt{v}_{\mathtt{bl}i}}$ and $\boxed{\mathtt{v}_{\mathtt{in}i}}$ are the same.

Since the sender-prover knows all the values of the acct_δ, they can create commitments for the same values under a different public key $\mathsf{pk}_\epsilon = (g, h)$. So the prover creates $\vec{\mathsf{acct}}_\epsilon$ where $\mathsf{acct}_{\epsilon i} = ((g,h), \boxed{\mathtt{v}_{\mathtt{bl}i}}_\epsilon, \boxed{\mathtt{v}_{\mathtt{out}i}}_\epsilon, \boxed{\mathtt{v}_{\mathtt{in}i}}_\epsilon)$. Then they use the homomorphic property of the commitment in order to prove the preservation of value, since $\sum_i \mathtt{v}_{\mathtt{bl}i} = 0 \iff \prod_i \boxed{\mathtt{v}_{\mathtt{bl}i}}_\epsilon$ is a commitment of 0 under $\mathsf{pk}_\epsilon = (g, h)$. The values in $\mathsf{acct}_{\epsilon,s+1}, \ldots, \mathsf{acct}_{\epsilon,t}$ will be used to prove that balances of recipients set and anonymity set is not decreased, meaning $\mathtt{v}_{\mathtt{bl}\epsilon,s+1}, \ldots, \mathtt{v}_{\mathtt{bl}\epsilon,n} \in \mathcal{V}$.

Now in order to hide the sender's and the receiver's position in `inputs` and `outputs` we first shuffle `inputs` list to $\mathtt{inputs}'$ before the updates, then we execute the updates to produce $\mathtt{outputs}'$, and finally we shuffle again after the updates to get `outputs` in arbitrary order. The first shuffle uses the aforementioned permutation where senders' accounts are first,followed by recipients' accounts and then the anonymity set. The second shuffle uses a permutation in order to order the `outputs` lexicographically.

Therefore, we need some auxiliary functions for the proof that are defined as following:

- $\mathsf{CreateDelta}(\{\mathsf{acct}_i\}_{i=1}^n, \{\mathtt{v}_{\mathtt{bl}i}\}_{i=1}^n, \{\mathtt{v}_{\mathtt{out}i}\}_{i=1}^n, \{\mathtt{v}_{\mathtt{in}i}\}_{i=1}^n)$: Creates a set of accounts that contains the differences between accounts' variables $\mathtt{bl}, \mathtt{out}, \mathtt{in}$ in the input and output accounts, and another set of accounts that also contains these differences but all with the global public key (g, h):

  1. Parse $\mathsf{acct}_i = (\mathsf{pk}_i, \mathsf{com}_{\mathtt{bl},i}, \mathsf{com}_{\mathtt{out},i}, \mathsf{com}_{\mathtt{in},i})$. Sample $r_{(\mathtt{bl}|\mathtt{out}|\mathtt{in}),1}, \ldots, r_{(\mathtt{bl}|\mathtt{out}|\mathtt{in}),n-1} \leftarrow\$ \mathbb{F}_p$ and set $r_{(\mathtt{bl}|\mathtt{out}|\mathtt{in}),n} = -\sum_{i=1}^{n-1} r_{(\mathtt{bl}|\mathtt{out}|\mathtt{in}),i}$.
  2. Set $\mathsf{acct}_{\delta,i} = (\mathsf{pk}_i, \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{bl}i}; r_{\mathtt{bl},i}), \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{out}i}; r_{\mathtt{out},i}), \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{in}i}; r_{\mathtt{in},i}))$
  3. Set $\mathsf{acct}_{\epsilon,i} = ((g,h), \mathsf{Commit}((g,h), \mathtt{v}_{\mathtt{bl}i}; r_{\mathtt{bl},i}), \mathsf{Commit}((g,h), \mathtt{v}_{\mathtt{out}i}; r_{\mathtt{out},i}), \mathsf{Commit}((g,h), \mathtt{v}_{\mathtt{in}i}$
  4. Output $(\{\mathsf{acct}_{\delta,i}\}_{i=1}^n, \{\mathsf{acct}_{\epsilon,i}\}_{i=1}^n, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}})$

- $\mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}_{i=1}^n, \{\mathsf{acct}_{\epsilon,i}\}_{i=1}^n, \vec{\mathtt{v}_{\mathtt{bl}}}, \vec{\mathtt{v}_{\mathtt{out}}}, \vec{\mathtt{v}_{\mathtt{in}}}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}})$: Verifies that accounts created using $\mathsf{CreateDelta}$ are consistent:

  1. Parse $\mathsf{acct}_{\delta,i} = (\mathsf{pk}_i, \boxed{\mathtt{v}_{\mathtt{bl}i}}, \boxed{\mathtt{v}_{\mathtt{out}i}}, \boxed{\mathtt{v}_{\mathtt{in}i}})$ and $\mathsf{acct}_{\epsilon,i} = (\mathsf{pk}_{\epsilon,i}, \mathsf{com}_{\epsilon,i})$
  2. If $\prod_{i=1}^n \mathsf{com}_{\epsilon,i} = (1,1)$ and $\forall i$ $\boxed{\mathtt{v}_{\mathtt{bl}i}} = \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{bl}i}; r_{\mathtt{bl},i}) \wedge \boxed{\mathtt{v}_{\mathtt{out}i}} = \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{out}i}; r_{\mathtt{out},i}) \wedge \boxed{\mathtt{v}_{\mathtt{in}i}} = \mathsf{Commit}(\mathsf{pk}_i, \mathtt{v}_{\mathtt{in}i}; r_{\mathtt{in},i}) \wedge \mathsf{acct}_{\epsilon,i} = ((g,h), \mathsf{Commit}((g,h), (\mathtt{v}_{\mathtt{bl}i}; r_{\mathtt{bl},i}))$ output 1. Else output 0.

- $\mathsf{VerifyNonNegative}(\mathsf{acct}_\epsilon, \mathtt{v}_{\mathtt{bl}}, r_{\mathtt{bl}})$: Verifies that an account contains a balances in $\mathcal{V}$:
  1. If $\mathsf{acct}_\epsilon = ((g,h),(g^r, g^v h^r)) \wedge v \in \mathcal{V}$ outputs 1. Else output 0.

- $\mathsf{UpdateDelta}(\{\mathsf{acct}_i\}_{i=1}^n, \{\mathsf{acct}_{\delta,i}\}_{i=1}^n)$: Updates the input accounts by $\mathtt{v}_{\mathtt{bl}i}, \mathtt{v}_{\mathtt{out}i}, \mathtt{v}_{\mathtt{in}i}$ but with the public key unchanged:
  1. Parse $\mathsf{acct}_i = (\mathsf{pk}_i, \mathsf{com}_{\mathtt{bl},i}, \mathsf{com}_{\mathtt{out},i}, \mathsf{com}_{\mathtt{in},i})$ and $\mathsf{acct}_{\delta,i} = (\mathsf{pk}'_i, \boxed{\mathtt{v}_{\mathtt{bl}i}}, \boxed{\mathtt{v}_{\mathtt{out}i}}, \boxed{\mathtt{v}_{\mathtt{in}i}})$.
  2. If $\mathsf{pk}_i = \mathsf{pk}'_i \ \forall i$ output $\{(\mathsf{pk}_i, \mathsf{com}_{\mathtt{bl},i} \cdot \boxed{\mathtt{v}_{\mathtt{bl}i}}, \mathsf{com}_{\mathtt{out},i} \cdot \boxed{\mathtt{v}_{\mathtt{out}i}}, \mathsf{com}_{\mathtt{in},i} \cdot \boxed{\mathtt{v}_{\mathtt{in}i}})\}$, else output $\perp$.

- $\mathsf{VerifyUD}(\mathsf{acct}, \mathsf{acct}', \mathsf{acct}_\delta)$: Verifies that $\mathsf{UpdateDelta}$ was performed correctly:
  1. Parse $\mathsf{acct} = (\mathsf{pk}, \mathsf{com}_{\mathtt{bl}}, \mathsf{com}_{\mathtt{out}}, \mathsf{com}_{\mathtt{in}})$, $\mathsf{acct}' = (\mathsf{pk}, \mathsf{com}'_{\mathtt{bl}}, \mathsf{com}'_{\mathtt{out}}, \mathsf{com}'_{\mathtt{in}})$ and $\mathsf{acct}_\delta = (\mathsf{pk}_\delta, \boxed{\mathtt{v}_{\mathtt{bl}}}, \boxed{\mathtt{v}_{\mathtt{out}}}, \boxed{\mathtt{v}_{\mathtt{in}}})$.
  2. Check that $\mathsf{pk} = \mathsf{pk}' = \mathsf{pk}_\delta \ \wedge \ \mathsf{com}'_{\mathtt{bl}} = \mathsf{com}_{\mathtt{bl}} \cdot \boxed{\mathtt{v}_{\mathtt{bl}}} \ \wedge \ \mathsf{com}'_{\mathtt{out}} = \mathsf{com}_{\mathtt{out}} \cdot \boxed{\mathtt{v}_{\mathtt{out}}} \ \wedge \ \mathsf{com}'_{\mathtt{in}} = \mathsf{com}_{\mathtt{in}} \cdot \boxed{\mathtt{v}_{\mathtt{in}}}$.

- $\mathsf{VerifyDeltaSender}(\mathsf{acct}_\delta, v, r_{\mathtt{bl}}, r_{\mathtt{out}})$: Verifies that sender's value $\mathtt{out}$ is correct.
  1. Parse $\mathsf{acct}_\delta = (\mathsf{pk}_\delta, \boxed{\mathtt{v}_{\mathtt{bl}}}, \boxed{\mathtt{v}_{\mathtt{out}}}, \boxed{\mathtt{v}_{\mathtt{in}}})$.
  2. If $\boxed{\mathtt{v}_{\mathtt{bl}}} = \mathsf{Commit}(\mathsf{pk}_\delta, v; r_{\mathtt{bl}}) \wedge \boxed{\mathtt{v}_{\mathtt{out}}} = \mathsf{Commit}(\mathsf{pk}_\delta, v; r_{\mathtt{out}})$ then return 1. Else return 0.

- $\mathsf{VerifyDeltaReceiver}(\mathsf{acct}_\delta, v, r_{\mathtt{bl}}, r_{\mathtt{in}})$: Verifies that receiver's value $\mathtt{in}$ is correct.
  1. Parse $\mathsf{acct}_\delta = (\mathsf{pk}_\delta, \boxed{\mathtt{v}_{\mathtt{bl}}}, \boxed{\mathtt{v}_{\mathtt{out}}}, \boxed{\mathtt{v}_{\mathtt{in}}})$.
  2. If $\boxed{\mathtt{v}_{\mathtt{bl}}} = \mathsf{Commit}(\mathsf{pk}_\delta, v; r_{\mathtt{bl}}) \wedge \boxed{\mathtt{v}_{\mathtt{in}}} = \mathsf{Commit}(\mathsf{pk}_\delta, v; r_{\mathtt{in}})$ then return 1. Else return 0.

Then the $\mathsf{NIZK.Prove}_{\mathsf{Trans}}(x, w)$ performs the following steps:

1. Parse $x = (\mathtt{inputs}, \mathtt{outputs}), w = (\mathsf{sk}, \{\mathtt{bl}_i, \mathtt{out}_i, \mathtt{in}_i\}_{i \in \mathtt{I}^*_{\mathtt{S}}}, \overrightarrow{\mathtt{v}_{\mathtt{bl}}}', \overrightarrow{\mathtt{v}_{\mathtt{out}}}', \overrightarrow{\mathtt{v}_{\mathtt{in}}}', \vec{r}, \psi, \mathtt{I}^*_{\mathtt{S}}, \mathtt{I}^*_{\mathtt{R}}, \mathtt{I}^*_{\mathtt{A}})$. If $R(x,w) = 0$ abort;
2. Let ψ_1 be a permutation such that $\psi_1(\mathtt{I}^*_{\mathtt{S}}) = [1,s], \psi_1(\mathtt{I}^*_{\mathtt{R}}) = [s+1,t]$ and $\psi_1(\mathtt{I}^*_{\mathtt{A}}) = [t+1,n]$;
3. Sample $\rho_1, \rho_2, \rho_3, \rho_4 \leftarrow\$ \mathbb{F}_p$ and let $\vec{\rho} = (\rho_1, \rho_2, \rho_3, \rho_4)$;
4. Set $\mathtt{inputs}' = \mathsf{UpdateAcct}(\{\mathtt{inputs}_{\psi_1(i)}, 0, 0, 0\}_i; \vec{\rho})$;

5. Set vectors $\overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}$ such that $\mathtt{v_{bl}}_i = \mathtt{v_{bl}}'_{\psi(i)}$, $\mathtt{v_{out}}_i = \mathtt{v_{out}}'_{\psi(i)}$,$\mathtt{v_{in}}_i = \mathtt{v_{in}}'_{\psi(i)}$;

6. Set $(\{\mathsf{acct}_{\delta,i}\}, \{\mathsf{acct}_{\epsilon,i}\}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}}) \leftarrow\$ \mathsf{CreateDelta}(\mathtt{inputs}', \overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}})$;

7. Update $\mathtt{outputs}' \leftarrow \mathsf{UpdateDelta}(\mathtt{inputs}', \{\mathsf{acct}_{\delta,i}\})$;

8. Let $\psi_2 = \psi_1^{-1} \circ \psi, \rho'_1 = \frac{r_1}{\rho_1}, \vec{\rho'_2} = \frac{r_2-\rho_2}{\rho_1} - r_{\mathtt{bl}\,i}, \vec{\rho'_3} = \frac{r_3-\rho_3}{\rho_1} - r_{\mathtt{out}\,i}, \vec{\rho'_4} = \frac{r_4-\rho_4}{\rho_1} - r_{\mathtt{in}\,i}$ and let $\vec{\rho'} = (\rho'_1, \vec{\rho'_2}, \vec{\rho'_3}, \vec{\rho'_4})$.

9. Update $\mathtt{outputs} = \mathsf{UpdateAcct}(\{\mathtt{outputs}'_{\psi_2(i)}, 0, 0, 0\}_i; \vec{\rho'})$

10. Generate a ZK proof $\pi = (\mathtt{inputs}', \mathtt{outputs}', \mathsf{acct}_\delta, \mathsf{acct}_\epsilon, \pi_1, \pi_2, \pi_3)$ for the relation $R_1 \wedge R_2 \wedge R_3$ where:

$$
\begin{aligned}
R_1 = \{&(\mathtt{inputs}, \mathtt{inputs}', (\psi_1, \vec{\rho}))| \\
&\mathsf{VerifyUpdateAcct}(\{\mathtt{inputs}'_i, \mathtt{inputs}_{\psi_1(i)}, 0, 0, 0\}_i; \vec{\rho}) = 1\}, \\
R_2 = \{&((\mathtt{inputs}', \mathtt{outputs}', \mathsf{acct}_\delta, \mathsf{acct}_\epsilon), (\mathsf{sk}, \{\mathtt{bl}, \mathtt{out}, \mathtt{in}\}_{i=0}^{s}, \overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}}))| \\
&\mathsf{VerifyUD}(\mathtt{inputs}'_i, \mathtt{outputs}'_i, \mathsf{acct}_{\delta,i}) = 1 \ \forall i \\
&\wedge \mathsf{VerifyUpdateAcct}(\mathtt{inputs}'_i, \mathtt{outputs}'_i, 0, 0, 0; 1, r_{\mathtt{bl},i}, r_{\mathtt{out},i}, r_{\mathtt{in},i}) = 1 \ \forall i \in [t+1, n] \\
&\wedge \mathsf{VerifyNonNegative}(\mathsf{acct}_{\epsilon,i}, \mathtt{v_{bl}}_i, r_{\mathtt{bl},i}) = 1 \ \forall i \in [s+1, t] \\
&\wedge \mathsf{VerifyAcct}(\mathtt{outputs}'_i, (\mathsf{sk}, \mathtt{bl}_i + \mathtt{v_{bl}}_i)) = 1 \ \forall i \in [1, s] \\
&\wedge \mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}, \{\mathsf{acct}_{\epsilon,i}\}, \overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}}) = 1 \\
&\wedge \mathsf{VerifyDeltaSender}(\mathsf{acct}_{\delta,i}, \mathtt{v_{bl}}_i, r_{\mathtt{bl},i}, r_{\mathtt{out},i}) = 1 \ \forall i \in [1, s] \\
&\wedge \mathsf{VerifyDeltaReceiver}(\mathsf{acct}_{\delta,i}, \mathtt{v_{bl}}_i, r_{\mathtt{bl},i}, r_{\mathtt{in},i}) = 1 \ \forall i \in [s+1, t]\}, \\
R_3 = \{&(\mathtt{outputs}', \mathtt{outputs}, (\psi_2, \vec{\rho'}))| \\
&\mathsf{VerifyUpdateAcct}(\{\mathtt{outputs}_i, \mathtt{outputs}'_{\psi_1(2)}, 0, 0, 0\}_i; \vec{\rho'}) = 1\}
\end{aligned}
$$

Now R_1, R_3 can be proven using a slight modification of the Bayer-Groth shuffle argument [4]. The Σ_2 protocol that proves R_2 consists of the following sub-protocols:

1. Σ_{vu}: trivial check of $\mathsf{VerifyUD}$.

2. Σ_δ: prover shows knowledge of $\overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}}$ such that $\mathsf{VerifyDelta}(\{\mathsf{acct}_{\delta,i}\}_{i=1}^n, \{\mathsf{acct}_{\epsilon,i}\}_{i=1}^n, \overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}, \vec{r_{\mathtt{bl}}}, \vec{r_{\mathtt{out}}}, \vec{r_{\mathtt{in}}}) = 1$.
Σ_δ can be implemented by using Σ_{com}:
$\Sigma_\delta = \wedge_{i=1}^n \Sigma_{com}((\mathsf{pk}_{\delta,i}, \mathsf{com}_{\delta,i}), (\mathsf{pk}_{\epsilon,i}, \mathsf{com}_{\epsilon,i}); (\mathtt{v_{bl}}, r_{\mathtt{bl},i}, r_{\mathtt{bl},i}))$
$\wedge_{i=1}^n \Sigma_{com}((\mathsf{pk}_{\delta,i}, \mathsf{com}_{\delta,i}), (\mathsf{pk}_{\epsilon,i}, \mathsf{com}_{\epsilon,i}); (\mathtt{v_{out}}, r_{\mathtt{out},i}, r_{\mathtt{out},i}))$
$\wedge_{i=1}^n \Sigma_{com}((\mathsf{pk}_{\delta,i}, \mathsf{com}_{\delta,i}), (\mathsf{pk}_{\epsilon,i}, \mathsf{com}_{\epsilon,i}); (\mathtt{v_{in}}, r_{\mathtt{in},i}, r_{\mathtt{in},i}))$, but the verifier additionally checks that $\mathsf{pk}_{\epsilon,i} = (g, h)\ \forall i$ and that $\prod_{i=1}^n \boxed{\mathtt{v_{bl}}i}_\epsilon = (1, 1)$.

3. Σ_{zero}^i: prover shows knowledge of $r_{\mathtt{bl},i}, r_{\mathtt{out},i}, r_{\mathtt{in},i}$ such that $\mathsf{VerifyUpdateAcct}(\mathtt{inputs}'_i, \mathtt{outputs}'_i, 0, 0, 0; (1, r_{\mathtt{bl},i}, r_{\mathtt{out},i}, r_{\mathtt{in},i})) = 1$.
The sub-argument can be written as follows:
given $\mathsf{acct}_1 = (\mathsf{pk}, \boxed{\mathtt{v_{bl}}}_1, \boxed{\mathtt{v_{out}}}_1, \boxed{\mathtt{v_{in}}}_1), \mathsf{acct}_2 = (\mathsf{pk}, \boxed{\mathtt{v_{bl}}}_2, \boxed{\mathtt{v_{out}}}_2, \boxed{\mathtt{v_{in}}}_2)$,

the prover knows $r_{\mathtt{bl}}, r_{\mathtt{out}}, r_{\mathtt{in}}$ such that $\boxed{\mathtt{v_{bl}}}_1 = \boxed{\mathtt{v_{bl}}}_2 \cdot \mathsf{pk}^{r_{\mathtt{bl}}}, \boxed{\mathtt{v_{out}}}_1 = \boxed{\mathtt{v_{out}}}_2 \cdot \mathsf{pk}^{r_{\mathtt{out}}}, \boxed{\mathtt{v_{in}}}_1 = \boxed{\mathtt{v_{in}}}_2 \cdot \mathsf{pk}^{r_{\mathtt{in}}}$. The equation is equivelant to $\wedge_{i=\{\mathtt{bl},\mathtt{out},\mathtt{in}\}} \mathsf{VerifyUpdate}(\mathsf{pk}, \frac{com_{2,i}}{com_{1,i}}, r_i) = 1$, hence can be done using AND-proofs of Σ_{vu}.

4. Σ^i_{vds}: prover shows knowledge of $v, r_{\mathtt{bl},i}, r_{\mathtt{out},i}$ such that $\mathsf{acct}_{\delta,i}$ has the same value under commitments $\boxed{\mathtt{v_{bl}}}, \boxed{\mathtt{v_{out}}}$. Σ^i_{vds} can be implemented by using $\Sigma_{com}((\mathsf{pk}_{\delta,i}, \boxed{\mathtt{v_{bl}}}_i), (\mathsf{pk}_{\delta,i}, \boxed{\mathtt{v_{out}}}_i); (\mathtt{v_{bl}}_i, r_{\mathtt{bl},i}, r_{\mathtt{out},i}))$.

5. Σ^i_{vdr}: prover shows knowledge of $v, r_{\mathtt{bl},i}, r_{\mathtt{in},i}$ such that $\mathsf{acct}_{\delta,i}$ has the same value under commitments $\boxed{\mathtt{v_{bl}}}, \boxed{\mathtt{v_{in}}}$. Σ^i_{vds} can be implemented by using $\Sigma_{com}((\mathsf{pk}_{\delta,i}, \boxed{\mathtt{v_{bl}}}_i), (\mathsf{pk}_{\delta,i}, \boxed{\mathtt{v_{in}}}_i); (\mathtt{v_{bl}}_i, r_{\mathtt{bl},i}, r_{\mathtt{in},i}))$.

6. Σ_{range}: prover shows knowledge of $\mathsf{acct}_\epsilon, v, r$ such that $\mathsf{VerifyNonNegative}(\mathsf{acct}_\epsilon, v, r) = 1$. In order to implement this we use Bulletproofs [6].

7. Finally in order to prove $\mathsf{VerifyAcct}(\mathsf{acct}, \mathsf{sk}, \mathtt{bl})$:

   (a) the prover shows knowledge of sk using Σ_{dlog}.

   (b) Since sender may not know the randomness used to open his commitment, the prover opens the commitment with the sk and finds the value $\mathtt{bl}$.

   (c) Chooses a new randomness $r \leftarrow\!\!\$\ \mathbb{F}_p$ and constructs $\mathsf{acct}_\epsilon = ((g,h), \mathsf{Commit}((g,h), \mathtt{bl}; r))$.

   (d) Proves using Σ_{com} that these two accounts has the same $\mathtt{bl}$.

   (e) Proves using $\Sigma_{range}(\mathsf{acct}_\epsilon, \mathtt{bl}, r)$ that $\mathtt{bl} \in \mathcal{V}$.

   So $\Sigma_{range,\mathsf{sk}} = \Sigma_{dlog} \ \wedge \ \Sigma_{com} \ \wedge \ \Sigma_{range}$.

Hence $\Sigma_2 = \Sigma_{vud} \wedge \Sigma_\delta \wedge \left(\wedge^t_{i=s+1} \Sigma_{range}(\mathsf{acct}_{\delta,i}, \mathtt{v}'_{\mathtt{bl}i}, r_{\mathtt{bl},i})\right) \wedge \left(\wedge^n_{i=t+1} \Sigma^i_{zero}\right) \wedge (\wedge^s_{i=1} \Sigma_{range,\mathsf{sk}}(\mathtt{outputs}'_i, \mathtt{bl}_i + \mathtt{v}_{\mathtt{bl}i}, \mathsf{sk})) \ \wedge \ \left(\wedge^s_{i=1} \Sigma^i_{vds}\right) \wedge \left(\wedge^t_{i=s+1} \Sigma^i_{vdr}\right)$. Σ_2 is a public-coin SHVZK argument of knowledge of the relation R_2 as follows from the properties of AND-proofs.

The full SHVZK argument knowledge of Trans is then $\Sigma := \Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3$.

5.6.2 zk-proof of Audit and Register

Both the $\mathsf{Register}$ and Audit functionalities need a zero-knowledge proof for the statements:

- $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk})$: prover shows knowledge of a valid $(\mathsf{pk}, \mathsf{sk})$ key-pair. The corresponding language can be written as:

$$L_{vu} := \{pk = (X = g^r, Y = g^{r \cdot sk}) \,\big|\, \exists \mathsf{sk} \text{ s.t. } \ Y = X^{\mathsf{sk}}\}$$

  That can be proven through Σ_{dlog} with arguments (X, Y, sk).

- $\mathsf{VerifyCom}(\mathsf{pk}, \mathtt{com}, \mathsf{sk}, v)$: prover shows knowledge of secret key sk that opens the commitment $\mathtt{com}$ to value v. The corresponding language can be written as:

$$L_{open(sk)} := \{(\mathtt{com} = (X = h^r, Y = g^v h^{sk \cdot r}), v) \,\big|\, \exists \mathsf{sk} \text{ s.t. } \ Y/g^v = X^{\mathsf{sk}}\}$$

  That can be proven through Σ_{dlog} with arguments $(X, Y/g^v, \mathsf{sk})$.

The proof needed for Register results from the composition of these Σ-protocols and a range proof for showing that $bl \in \mathcal{V}$.

The Audit proof uses the same combination of these Σ-protocols and appropriate range proofs for each policy $f_{\mathsf{slimit}}, f_{\mathsf{rlimit}}, f_{\mathsf{open}}, f_{\mathsf{txlimit}}, f_{\mathsf{np}}$.

5.7 Analysis

Proof of anonymity

Intuitively, we argue that any PPT adversary $\mathcal{A}$ capable of distinguishing between $\mathsf{tx}_0, \mathsf{tx}_1$ in the anonymity game (find if $b' = b$) can be used to break either the indistinguishability of UPK scheme, the hiding property of commitment scheme, or the zero-knowledge property of the NIZK proofs.

Transactions consist of $\mathtt{inputs}, \mathtt{outputs}$, and a zk-proof π (and if it is CreateAcct or DeleteAcct a newly created account acct). One way $\mathcal{A}$ could determine b is based on π, but that violates the zero-knowledge property of the NIZK proofs. Another way that $\mathcal{A}$ could determine b is to distinguish between $\mathsf{tx}_0, \mathsf{tx}_1$ through the $\mathtt{outputs}$ sets of each tx. The only differences in the two $\mathtt{outputs}$ sets $\mathsf{tx}_0.\mathtt{outputs}, \mathsf{tx}_1.\mathtt{outputs}$ are the accounts which are used in $\mathtt{P}$ and in $\mathtt{A}$ as well as the amount v used to increase/decrease the variables in the accounts of $\mathtt{P}$. However, since both the accounts' amounts and transferred value v are presented in a committed form, if $\mathcal{A}$ can determine b based on the different values v_0, v_1 then the hiding property of the commitment scheme is violated. In addition, since all the accounts participating in the transaction are updated and randomly permuted, $\mathcal{A}$ cannot use $\mathtt{P}_0, \mathtt{A}_0, \mathtt{P}_1, \mathtt{A}_1$ to distinguish the two transactions without violating the indistinguishability property of UPK scheme.

Theorem 1. *AQQUA satisfies anonymity, as defined in 7*

Proof. We prove the theorem using a sequence of 14 hybrid games, as follows. Hybrid 0 and Hybrid 7 are the anonymity game for $b = 0, b = 1$ respectively. Each of the rest hybrids differs in oracles' functionalities in a way that the successive hybrids are indistinguishable from the view of the adversary. We use these hybrids to prove that the adversary cannot distinguish anonymity game for $b = 0$ and anonymity game with $b = 1$.

Hybrid 0. The anonymity game for $b = 0$.

Hybrid 1. Same as Hybrid 0, but here we run the NIZK extractor on each transaction generated by the adversary. That means, when $\mathcal{A}$ runs the $\mathsf{OApplyTrans}(\mathsf{tx})$ Oracle, the Oracle verifies tx by running $\mathsf{VerifyTrans}(\mathsf{tx}, \mathsf{state})$ depending on the transaction tx and if it is successful the oracle runs $\mathsf{state}' \leftarrow \mathsf{ApplyTrans}(\mathsf{state}, \mathsf{tx})$, as well as uses the NIZK extractor to extract the witness used to generate tx, including sk.

Hybrid 2. Same as Hybrid 1, but here the zero-knowledge arguments of the each transaction is replaced with the output of the corresponding simulator of the zero-knowledge property of NIZK. In order to achieve this we change the following oracles' functionality:

- when $\mathcal{A}$ or the challenger creates tx through the $\mathsf{OTrans}(\mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v}_\mathtt{S}}, \overrightarrow{\mathtt{v}_\mathtt{R}}, \mathtt{A})$ Oracle, the Oracle runs $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v}_\mathtt{S}}, \overrightarrow{\mathtt{v}_\mathtt{R}}, \mathtt{A})$, but replaces the zero-

knowledge arguments in tx with a simulated argument.

- when $\mathcal{A}$ or the challenger creates tx through the $\mathsf{OCreateAcct}(\mathsf{userInfo}, \mathtt{A})$ Oracle, the Oracle runs $\mathsf{tx} \leftarrow \mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A})$, but replaces the zero-knowledge arguments in tx with a simulated argument.

Hybrid 3. Same as Hybrid 2, but now the challenger replaces the potential senders' and receivers' accounts of the challenge transaction tx_0 $(\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1)$, with new accounts that have a freshly created key pair $(\mathsf{sk}, \mathsf{pk})$ derived from the output of the $\mathsf{KGen}()$. In order to achieve this we change the following oracles' functionality:

- when $\mathcal{A}$ creates one of these accounts acct_i through the OTrans Oracle (these accounts are presented in $\mathsf{tx}.\mathtt{outputs}$), the Oracle runs $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v_S}}, \overrightarrow{\mathtt{v_R}}, \mathtt{A}), (\mathsf{pk}'_i, \mathsf{sk}'_i) \leftarrow \mathsf{KGen}$ and then return tx', where $\mathsf{tx}' = \mathsf{tx}$ except that each $\mathsf{acct}_i \in \{\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1\}$ is replaced with $\mathsf{acct}'_i = (\mathsf{pk}'_i, \mathtt{com}_{\mathtt{bl}i}, \mathtt{com}_{\mathtt{out}i}, \mathtt{com}_{\mathtt{in}i})$.

- when $\mathcal{A}$ creates one of these accounts acct_i through the $\mathsf{OCreateAcct}$ Oracle, the Oracle runs $\mathsf{tx} \leftarrow \mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A}), (\mathsf{pk}'_i, \mathsf{sk}'_i) \leftarrow \mathsf{KGen}$ and then return tx', where $\mathsf{tx}' = \mathsf{tx}$ except that each $\mathsf{acct}_i \in \{\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1\}$ is replaced with $\mathsf{acct}'_i = (\mathsf{pk}'_i, \boxed{0}, \boxed{0}, \boxed{0})$.

Hybrid 4. Same as Hybrid 3, but here the challenger replaces also the commitments of the accounts $(\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1)$ with newly created commitments to the same values with different randomness. In order to achieve this we change the following oracles' functionality:

- when $\mathcal{A}$ creates one of these accounts acct_i through the OTrans Oracle (these accounts are presented in $\mathsf{tx}.\mathtt{outputs}$), the Oracle runs $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v_S}}, \overrightarrow{\mathtt{v_R}}, \mathtt{A}), (r_1, r_2, r_3) \leftarrow\!\!\$\ \mathcal{R}, \mathtt{bl}_i \leftarrow \mathsf{OpenCom}(\mathsf{sk}, \mathsf{acct}_i.\mathtt{com}_{\mathtt{bl}}), \mathtt{out}_i \leftarrow \mathsf{OpenCom}(\mathsf{sk}, \mathsf{acct}_i.\mathtt{com}_{\mathtt{out}}), \mathtt{in}_i \leftarrow \mathsf{OpenCom}(\mathsf{sk}, \mathsf{acct}_i.\mathtt{com}_{\mathtt{in}}), \mathtt{com}'_{\mathtt{bl}} \leftarrow \mathsf{Commit}(\mathsf{pk}', \mathtt{bl}_i; r_1), \mathtt{com}'_{\mathtt{out}} \leftarrow \mathsf{Commit}(\mathsf{pk}', \mathtt{out}_i; r_2), \mathtt{com}'_{\mathtt{in}} \leftarrow \mathsf{Commit}(\mathsf{pk}', \mathtt{in}_i; r_3)$ and then return tx', where $\mathsf{tx}' = \mathsf{tx}$ except that each $\mathsf{acct}_i \in \{\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1\}$ is replaced with $\mathsf{acct}' = (\mathsf{pk}, \mathtt{com}'_{\mathtt{bl}}, \mathtt{com}'_{\mathtt{out}}, \mathtt{com}'_{\mathtt{in}})$. ($\mathsf{pk} = \mathsf{pk}'$ as in the Hybrid 3).

- when $\mathcal{A}$ creates one of these accounts acct_i through the $\mathsf{OCreateAcct}$ Oracle, the Oracle runs $\mathsf{tx} \leftarrow \mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A}), (r_1, r_2, r_3) \leftarrow\!\!\$\ \mathcal{R}, \mathtt{com}'_{\mathtt{bl}} \leftarrow \mathsf{Commit}(\mathsf{pk}'0; r_1), \mathtt{com}'_{\mathtt{out}} \leftarrow \mathsf{Commit}(\mathsf{pk}', 0; r_2), \mathtt{com}'_{\mathtt{in}} \leftarrow \mathsf{Commit}(\mathsf{pk}', 0; r_3)$ and then return tx', where $\mathsf{tx}' = \mathsf{tx}$ except that each $\mathsf{acct}_i \in \{\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1\}$ is replaced with $\mathsf{acct}' = (\mathsf{pk}, \mathtt{com}'_{\mathtt{bl}}, \mathtt{com}'_{\mathtt{out}}, \mathtt{com}'_{\mathtt{in}})$. ($\mathsf{pk} = \mathsf{pk}'$ as in the Hybrid 3).

Hybrid 5. Same as Hybrid 4, but here also the updated accounts of $(\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1)$ in the challenge $\mathsf{tx}.\mathtt{outputs}$ are replaced by accounts with freshly created public key pk'.

Hybrid 6. Same as Hybrid 5, but here also the updated accounts of $(\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1)$ in the challenge $\mathsf{tx}.\mathtt{outputs}$ are replaced by accounts with freshly created commitments to the same value.

Afterwards, we create Hybrids 7-13 that are the same with Hybrids 0-6 with the difference that are made for the anonymity game with $b = 1$.

Note that in Hybrid 6 and in Hybrid 13 all accounts of the potential senders' and receivers' accounts of the challenge transaction tx_b (both in $\mathtt{inputs}$ and $\mathtt{outputs}$) are fresh accounts, where in $\mathtt{outputs}$ have been generated with values corresponding to the case $b = 0$ — $b = 1$.

Now we will prove that $\mathcal{A}$ has negligible advantage of distinguish Hybrid 0 and Hybrid 7.

Lemma 2. *Hybrid 0 and Hybrid 1 are indistinguishable.*

Corollary 1. *Hybrid 7 and Hybrid 8 are indistinguishable.*

Proof. The adversary's view in the two hybrids' game are identical. □

Lemma 3. *Hybrid 1 and Hybrid 2 are indistinguishable.*

Corollary 2. *Hybrid 8 and Hybrid 9 are indistinguishable.*

Proof. Let $\mathcal{A}$ be an adversary that can distinguish Hybrid 1 and Hybrid 2 with advantage ϵ. We construct an adversary $\mathcal{B}$ that breaks the zero-knowledge property of the NIZK proof π of transaction tx with probability ϵ.

Let $\mathsf{O}_{\mathsf{zk}}(\cdot)$ be an oracle that on input $(\mathsf{tx}.\mathtt{inputs}, \mathsf{tx}.\mathtt{outputs})$ creates a valid zero-knowledge proof for the transaction. Then $\mathcal{B}$ wins if they can decide wether $\mathsf{O}_{\mathsf{zk}}(\cdot)$ is a prover or simulator oracle.

$\mathcal{B}$ takes as input the $\mathsf{O}_{\mathsf{zk}}(\cdot)$ and runs as follows:

1. $\mathcal{B}$ generates $\mathsf{state} \leftarrow \mathsf{Setup}(\lambda)$;
2. When $\mathcal{A}$ queries the $\mathsf{OTrans}(\mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v_S}}, \overrightarrow{\mathtt{v_R}}, \mathtt{A})$ oracle then $\mathcal{B}$ runs $\mathsf{tx} \leftarrow \mathsf{Trans}(\mathsf{sk}, \mathtt{S}, \mathtt{R}, \overrightarrow{\mathtt{v_S}}, \overrightarrow{\mathtt{v_R}}, \mathtt{A})$ with the difference that $\mathcal{B}$ replace the proof with the output of $\mathsf{O}_{\mathsf{zk}}(\mathsf{tx}[\mathtt{inputs}], \mathsf{tx}[\mathtt{outputs}])$
3. When $\mathcal{A}$ queries the $\mathsf{OCreateAcct}(\mathsf{userInfo}, \mathtt{A})$ oracle then $\mathcal{B}$ runs $\mathsf{tx} \leftarrow \mathsf{CreateAcct}(\mathsf{userInfo}, \mathtt{A})$ with the difference that $\mathcal{B}$ replace the proof with the output of $\mathsf{O}_{\mathsf{zk}}(\mathsf{tx}[\mathtt{inputs}], \mathsf{tx}[\mathtt{outputs}])$
4. $\mathcal{B}$ runs $b \leftarrow \mathcal{A}(state)$;

If $\mathcal{A}$ answers Hybrid 0 then $\mathsf{O}_{\mathsf{zk}}(\cdot)$ is a prover oracle. If $\mathcal{A}$ answers Hybrid 1 then $\mathsf{O}_{\mathsf{zk}}(\cdot)$ is a simulator oracle. So $\mathcal{B}$ wins with probability ϵ. □

Lemma 4. *Hybrid 2 and Hybrid 3 are indistinguishable.*

Corollary 3. *Hybrid 9 and Hybrid 10 are indistinguishable.*

Proof. Note that $\mathcal{A}$ cannot distinguish Hybrid 2 and Hybrid 3 from the fact that commitments are under different public key on the grounds that this breaks the key-anonymous property of the commitment scheme. Let $\mathcal{A}$ be an adversary that can distinguish Hybrid 2 and Hybrid 3 with advantage ϵ. We construct an

adversary $\mathcal{B}$ that breaks the indistinguishability property of the UPK scheme with probability ϵ.

In order to create $\mathcal{B}$, we define five sub-hybrids. Let h_0 be Hybrid 2 and for each $i \in \{1, 2, 3, 4\}$ h_i would be a sub-hybrid where we replace the account $\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1$ respectively. In hybrid h_4 all of the accounts will be changed, therefore h_4 is Hybrid 3. Lets $\mathcal{A}$ be an adversary that can distinguish h_i from h_{i+1}. Let acct_c be the account that we are replacing in this hybrid. Then:
$\mathcal{B}$ gets as input the tuple $(\mathsf{acct}^*, \mathsf{acct}_b)$ from the indistinguishability game and runs as follows:

1. $\mathcal{B}$ generates $\mathsf{state} \leftarrow \mathsf{Setup}(\lambda)$.
2. when $\mathcal{A}$ uses the $\mathsf{ORegister}$ Oracle to create the initial account that share the same secret key with acct_c, $\mathcal{B}$ replaces this account with acct^*.
3. when $\mathcal{A}$ uses OTrans or $\mathsf{OCreateAcct}$ Oracle to create the account acct_c, $\mathcal{B}$ replaces acct_c with acct_b.
4. $\mathcal{B}$ reply to all other queries in the oracles as in the Hybrid h_0.
5. $\mathcal{B}$ outputs $b' \leftarrow \mathcal{A}(\mathsf{state})$.

We know that $\mathcal{A}$ did not query the corrupt oracle on acct_c or on any other account that shares the same secret key with acct_c cause it would have immediately lost the anonymity game. Note that if $b = 0$ then the distribution of the game is the same as hybrid h_i and if $b = 1$ then the game has the same distribution as hybrid h_{i+1}. Hence $\mathcal{B}$ answer b' and solves the indistinguishability game with probability ϵ. □

Lemma 5. *Hybrid 3 and Hybrid 4 are indistinguishable.*

Corollary 4. *Hybrid 10 and Hybrid 11 are indistinguishable.*

Proof. The only difference from this two Hybrids are the randomness to the commitments of the real participants accounts. Therefore, they produce a computationally indistinguishable distribution, due to the hiding property if the used commitment scheme. □

Corollary 5. *Hybrid 4 and Hybrid 5 are indistinguishable.*
Hybrid 11 and Hybrid 12 are indistinguishable.
It can be proven the same way as Hybrid 2 and Hybrid 3 are indistinguishable.

Corollary 6. *Hybrid 5 and Hybrid 6 are indistinguishable.*
Hybrid 12 and Hybrid 13 are indistinguishable.
It can be proven the same way as Hybrid 3 and Hybrid 4 are indistinguishable.

Lemma 6. *Hybrid 6 and Hybrid 13 are indistinguishable.*

Proof. Hybrid 6 and Hybrid 13 differ to (1) the accounts that are included in `P` and in `A` as well as to (2) the balances that are stored in the real participants' accounts in the challenge query ($\mathsf{acct}_i =\in \{\mathsf{acct}_0, \mathsf{acct}_1, \mathsf{acct}'_0, \mathsf{acct}'_1\}$). Concerning the former (1), in both Hybrids the `inputs` that $\mathcal{A}$ sees is obtained by permuting $(\mathtt{P}_x \cup \mathtt{A}_x)$ with a random permutation ψ. But the union of these set in both cases ($x = \{0, 1\}$) produces identical distributions. As a result $\mathcal{A}$

cannot distinguish the two Hybrids from (1). The second change (2) produces a computationally indistinguishable distribution, due to the hiding property of the commitment scheme. Therefore, if $\mathcal{A}$ could distinguish these Hybrids based on (2) then $\mathcal{A}$ could break the hiding property of Commit. □

Using the above lemmas and the triangle inequality, we prove that there is not a PPT adversary $\mathcal{A}$ that can distinguish Hybrid 0 and Hybrid 7 with more than negligible advantage.

□

Proof of theft prevention

Intuitively, we argue that any PPT adversary $\mathcal{A}$ capable of winning the theft-prevention game can be used to break either the unforgeability property of UPK scheme, the binding property of commitment scheme, or the soundness property of the NIZK proofs.

In order to win the theft-prevention game, $\mathcal{A}$ should submit a transaction tx that either increases the total balance of the corrupted users, decreases the balance of honest users, or does not maintain preservation of value. This can happen in the following ways: The first way is if the adversary is able to transfer some amount from a honest user's account. However, this means that $\mathcal{A}$ can compute the sk of the honest account, thus the unforgeability property of the UPK scheme is violated. Secondly, if $\mathcal{A}$ manages to transfer more coins than the corrupted account holds. But in order for such a transaction to be valid, the adversary should either be able to make a zk-proof that violates the soundness property, or to compute an opening to a commitment with balance different from the real one, hence breaking the binding property of the commitment scheme. The third way is by creating a transaction that breaks preservation of value, but in order for such a transaction to be valid, $\mathcal{A}$ should again be able to construct an unsound zk-proof or break the binding property of the commitment scheme.

Theorem 7. *AQQUA satisfies theft prevention, as defined in 8.*

Proof. Assume that there exists a PPT $\mathcal{A}$ that wins the theft prevention game of Game 5.4 with non-negligible probability. Using the notation of the game, we have that $\mathcal{A}$ outputted a valid transaction tx that verifies and that results in one of the three winning conditions of the game being satisfied.

We have that $\mathsf{tx} = (\mathtt{inputs}, \mathtt{outputs}, \pi)$, where π is a ZK-proof for the relation $R(x, w)$ as defined in Figure 5.2, with $x = (\mathtt{inputs}, \mathtt{outputs})$ and $w = (\mathsf{sk}, \mathtt{bl}, \mathtt{out}, \mathtt{in}, \overrightarrow{\mathtt{v_{bl}}}, \overrightarrow{\mathtt{v_{out}}}, \overrightarrow{\mathtt{v_{in}}}, \vec{r}, \psi, \mathtt{I}^*_{\mathtt{S}}, \mathtt{I}^*_{\mathtt{R}}, \mathtt{I}^*_{\mathtt{A}})$.

From the soundness property of the NIZK argument of the Trans algorithm, we can extract a witness $w^* = (\mathsf{sk}^*, \mathtt{bl}^*, \cdots, \overrightarrow{\mathtt{v}^{*\prime}_{\mathtt{bl}}}, \cdots, \vec{r^*}, \cdots)$ such that $R(x, w^*) = 1$.

Let $\mathsf{acct} \in \mathtt{inputs}$ be the account such that $\mathsf{VerifyKP}(\mathsf{sk}^*, \mathsf{acct.pk}) = 1$. We divide into two cases.

1. It holds that $\mathsf{sk}^* \in \mathsf{honest}$. In this case, we construct an adversary $\mathcal{B}$ that breaks the unforgeability property of the UPK scheme with non-negligible probability.

   The reduction works as follows. The adversary $\mathcal{B}$ takes as input a public key pk^*. It also keeps a directed tree with root $(\mathsf{pk}^*, 1)$ and whose nodes

will be tuples of the form (pk, r). The tree will be updated so that for every edge of the form $((\mathsf{pk}_1, \cdot), (\mathsf{pk}_2, r_2))$ it will hold that $\mathsf{VerifyUpdate}(\mathsf{pk}_2, \mathsf{pk}_1, r_2) = 1$.

$\mathcal{B}$ answers to $\mathcal{A}$'s oracle queries as follows.

- When $\mathcal{A}$ queries the ORegister oracle and this query results in the Register algorithm to generate sk^*, $\mathcal{B}$ replaces $\mathsf{userInfo}.\mathtt{pk}_0$ with pk^*, and when NewAcct is called in the procedure, $\mathcal{B}$ gives as input pk^*. The adversary $\mathcal{B}$ stores the public key of the newly created account and the randomness used as a child of $(\mathsf{pk}^*, 1)$ in the tree. For the rest of the ORegister queries, $\mathcal{B}$ answers honestly.
- When $\mathcal{A}$ queries the OCreateAcct oracle for an account whose public key pk is contained in a leaf of the tree, $\mathcal{B}$ answers honestly and adds a child to the leaf, composed of the updated public key of the updated account and the randomness used.
- When $\mathcal{A}$ queries the OTrans oracle, the adversary $\mathcal{B}$ acts as follows.
  - If the public keys of the accounts in S are contained in leaves of the tree, $\mathcal{B}$ creates an outputs set and creates a simulated proof for the transaction. $\mathcal{B}$ also updates the tree by creating new children containing the updates of the public keys and the randomness.
  - If there exist public keys of accounts in the anonymity set that are contained in leaves of the tree, $\mathcal{B}$ creates new children containing the updates of the public keys and the randomness.
- When $\mathcal{A}$ queries the OApplyTrans with a transaction whose inputs contain a leaf of the tree, $\mathcal{B}$ uses the proof contained in the transaction to extract the witness. Then, $\mathcal{B}$ creates new children for the updates of the public keys, storing also the randomness of the witness.
- For the rest of the oracle queries, $\mathcal{B}$ answers honestly.

Finally, when $\mathcal{A}$ outputs the transaction tx of the theft prevention game, $\mathcal{B}$ finds the $\mathsf{acct} \in \mathtt{inputs}$ for which $\mathsf{VerifyKP}(\mathsf{sk}^*, \mathsf{acct.pk}) = 1$, and finds the leaf (pk, r) of the tree for which $\mathsf{acct.pk} = \mathsf{pk}$. Let r' be the multiplication of all randomnesses stored in the path from that leaf to the root. $\mathcal{B}$ returns (pk, r').

If $\mathcal{A}$ wins the theft prevention game, we have that $\mathsf{VerifyKP}(\mathsf{pk}, \mathsf{sk}^*) = 1$ and $\mathsf{VerifyUpdate}(\mathsf{pk}, \mathsf{pk}^*, r') = 1$. Since $\mathcal{A}$ can win with non-negligible probability, $\mathcal{B}$ breaks unforgeability with non-negligible probability.

2. It holds that $\mathsf{sk}^* \in \mathsf{corrupt}$.

   Assume w.l.o.g. that the transaction tx that $\mathcal{A}$ outputs is the first transaction that results in winning the game (that is, there is no transaction submitted to OApplyTrans oracle prior to this point that would result in $\mathcal{A}$ winning).

   Since $\mathcal{A}$ wins the game, we have that the sum of the openings of the committed balances of all the accounts (stored in the bookkeeping) of inputs is different from those of outputs.

From the soundness property of the NIZK argument of the Trans algorithm, we have that for every sender account acct' of $\mathtt{outputs}$, $\mathsf{VerifyAcct}(\mathsf{acct}', \mathsf{sk}^*, \mathtt{bl}^* + \mathtt{v}'^*_{\mathtt{bl}}, \cdot, \cdot) = 1$.

Since $\mathsf{VerifyAcct}$ returns 1, and also $\sum_{\mathtt{v}'^*_{\mathtt{bl}} \in \overrightarrow{\mathtt{v}'^*_{\mathtt{bl}}}} \mathtt{v}'^*_{\mathtt{bl}} = 0$, and since $\mathcal{A}$ wins the game, there exists an account $\mathsf{acct} \in \mathtt{outputs}$ for which $\mathtt{acct.com}_{\mathtt{bl}}$ has two different openings: one resulting from the bookkeeping, and one derived from the extracted witness (one of the values of the form $\mathtt{bl}^* + \mathtt{v}'^*_{\mathtt{bl}}$ for some sender account). This trivially breaks the binding property of the commitment scheme.

□

Proof of audit soundness

Intuitively, we argue that any PPT adversary $\mathcal{A}$ capable of winning the audit soundness game can be used to break either the binding property of commitment scheme or the soundness property of the NIZK proofs.

In order to win the the audit soundness game, $\mathcal{A}$ should either create a valid zero-knowledge proof without knowing the corresponding witness, or hide some of their accounts from the $\mathtt{AA}$. However, the former attack violates the soundness property of the zero-knowledge proof. The latter requires the $\mathcal{A}$ to be able to open their commitment $\boxed{\#\mathtt{accs}}$ to a different value, but this breaks again the binding property of the commitment scheme.

Theorem 8. *AQQUA satisfies audit soundness, as of 9*

Proof. Assume that there exist a PPT $\mathcal{A}$ that wins the audit soundness game of Game 5.5 with non-negligible probability. Using the notation of the game, we have that $\mathcal{A}$ outputted a proof $\pi = (\pi_1, \pi_2)$ that verifies but $\mathcal{A}$ is not compliant with the specified policy.

$\mathcal{A}$ choose a policy f with its auxiliary parameters $\mathtt{aux}$, an initial public key $\mathtt{pk}_0$ and two snapshots from the blockchain $\mathsf{state}_1, \mathsf{state}_2$. Then $\mathcal{A}$ constructs $\pi = (\pi_1, \pi_2)$ which as defined in Figure 5.5 is a ZK-proof for the relations $R_1(x, w)$, with $x = (\mathtt{pk}_0, \{\#\mathtt{accs}_j, \boxed{\#\mathtt{accs}_j}, \{\mathsf{acct}_{ji}\}_{i=1}^{\#\mathtt{accs}_j}\}_{j=1}^{2})$ and $w = (\mathsf{sk})$ and $R_2(x, w)$, with $x = (\{\mathsf{acct}_{1i}\}_{i=1}^{\#\mathtt{accs}_j}, \{\mathsf{acct}_{2i}\}_{i=1}^{\#\mathtt{accs}_j}, \boxed{\mathtt{v}}, \mathtt{aux})$ and $w = (\mathsf{sk}, \mathtt{v})$, where $\mathtt{v}, \mathtt{aux}$ are values that depend on the policy.

From the soundness property of the NIZK argument of the π_1, we can extract a witness $w^* = \mathsf{sk}^*$ such that $R_1(x, w^*) = 1$. We have that every $\mathsf{pk} \in \{\mathtt{pk}_0\} \cup \{\mathsf{acct}_{ji}.\mathsf{pk}\}_{i=1}^{\#\mathtt{accs}_j}$, $\mathsf{VerifyKP}(\mathsf{sk}^*, \mathsf{pk})$. Therefore similarly to theft-prevention proof we can prove that if $\mathsf{sk}^* \in \mathsf{honest}$ then $\mathcal{A}$ can be used to break the unforgeability property of UPK scheme. Else if $\mathsf{sk}^* \in \mathsf{corrupt}$ then since $\mathcal{A}$ wins the game, we have that the opening to the commitment of $\boxed{\#\mathtt{accs}}$ is different from the one that resulting from bookkeeping. This trivially breaks the binding property of the commitment scheme.

From the soundness property of the NIZK argument of the π_2, we can extract a witness $w^* = \mathtt{v}^*$ such that $R_1(x, w^*) = 1$. Again since $\mathcal{A}$ wince the game the sum of the openings of the commited value of all the accounts that belongs to $\mathcal{A}$ is different from the one that resulting from bookkeeping, so this breaks the binding property of the commitment scheme.

□

Chapter 6

Conclusion

We presented AQQUA, a decentralized private and auditable payment system. Our accounts extend Quisquis accounts in order to record (in hidden form) the total influx and outflux. While we introduce two authorities in AQQUA, it remains decentralized since the `RA` and `AA` do not intervene in the normal flow of transactions. A major direction for possible future research involves strengthening the privacy provided by AQQUA even more. Firstly, the fact that the audit proofs leak account information between the audit states could be addressed. Secondly, another direction could be to convert audit proofs to be designated-verifier [13]. As a result, the `AA` will be able to simulate them, and thus it will be the only entity convinced about the audit results. This may increase the privacy of the participants, but it will interfere with the trust dynamics of the system. As a result, further research is needed for this integration.

Bibliography

[1] https://dx.doi.org/10.57713/kallipos-492.

[2] Ghada Almashaqbeh and Ravital Solomon. *SoK: Privacy-Preserving Computing in the Blockchain Era*. Cryptology ePrint Archive, Paper 2021/727. https://eprint.iacr.org/2021/727. 2021. URL: https://eprint.iacr.org/2021/727.

[3] Nasser Alsalami and Bingsheng Zhang. "SoK: A Systematic Study of Anonymity in Cryptocurrencies". In: *2019 IEEE Conference on Dependable and Secure Computing (DSC)*. 2019, pp. 1–9. DOI: 10.1109/DSC47296.2019.8937681.

[4] Stephanie Bayer and Jens Groth. "Efficient Zero-Knowledge Argument for Correctness of a Shuffle". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 263–280. ISBN: 978-3-642-29011-4.

[5] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 459–474. DOI: 10.1109/SP.2014.36.

[6] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020. URL: https://doi.org/10.1109/SP.2018.00020.

[7] Panagiotis Chatzigiannis and Foteini Baldimtsi. "MiniLedger: Compact-Sized Anonymous and Auditable Distributed Payments". In: *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part I*. Ed. by Elisa Bertino, Haya Schulmann, and Michael Waidner. Vol. 12972. Lecture Notes in Computer Science. Springer, 2021, pp. 407–429. DOI: 10.1007/978-3-030-88418-5_20. URL: https://doi.org/10.1007/978-3-030-88418-5_20.

[8] Panagiotis Chatzigiannis, Foteini Baldimtsi, and Konstantinos Chalkias. "SoK: Auditability and Accountability in Distributed Payment Systems". In: *Applied Cryptography and Network Security: 19th International Conference, ACNS 2021, Kamakura, Japan, June 21–24, 2021, Proceedings,*

Part II. Kamakura, Japan: Springer-Verlag, 2021, 311–337. ISBN: 978-3-030-78374-7. DOI: 10.1007/978-3-030-78375-4_13. URL: https://doi.org/10.1007/978-3-030-78375-4_13.

[9] Yu Chen, Xuecheng Ma, Cong Tang, and Man Ho Au. "PGC: Decentralized Confidential Payment System with Auditability". In: *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part I*. Ed. by Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider. Vol. 12308. Lecture Notes in Computer Science. Springer, 2020, pp. 591–610. DOI: 10.1007/978-3-030-58951-6_29. URL: https://doi.org/10.1007/978-3-030-58951-6_29.

[10] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "Sok: Transparent dishonesty: front-running attacks on blockchain". In: *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*. Springer. 2020, pp. 170–189.

[11] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. "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*. Springer. 2019, pp. 649–678.

[12] Christina Garman, Matthew Green, and Ian Miers. "Accountable Privacy for Decentralized Anonymous Payments". In: *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*. Ed. by Jens Grossklags and Bart Preneel. Vol. 9603. Lecture Notes in Computer Science. Springer, 2016, pp. 81–98. DOI: 10.1007/978-3-662-54970-4_5. URL: https://doi.org/10.1007/978-3-662-54970-4_5.

[13] Markus Jakobsson, Kazue Sako, and Russell Impagliazzo. "Designated Verifier Proofs and Their Applications". In: *Advances in Cryptology — EUROCRYPT '96*. Ed. by Ueli Maurer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 143–154. ISBN: 978-3-540-68339-1.

[14] Aggelos Kiayias, Markulf Kohlweiss, and Amirreza Sarencheh. "PEReDi: Privacy-Enhanced, Regulated and Distributed Central Bank Digital Currencies". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. ACM, 2022, pp. 1739–1752. DOI: 10.1145/3548606.3560707. URL: https://doi.org/10.1145/3548606.3560707.

[15] Ya-Nan Li, Tian Qiu, and Qiang Tang. "Pisces: Private and Compliable Cryptocurrency Exchange". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1317. URL: https://eprint.iacr.org/2023/1317.

[16] *Making sense of bitcoin, cryptocurrency and blockchain*. https://www.pwc.com/us/en/industries/financial-services/fintech/bitcoin-blockchain-cryptocurrency.html.

[17] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. "A fistful of bitcoins: characterizing payments among men with no names". In: *Proceedings of the 2013 conference on Internet measurement conference*. 2013, pp. 127–140.

[18] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. "A fistful of Bitcoins: characterizing payments among men with no names". In: *Commun. ACM* 59.4 (2016), pp. 86–93. DOI: 10.1145/2896384. URL: https://doi.org/10.1145/2896384.

[19] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf.

[20] Neha Narula, Willy Vasquez, and Madars Virza. "zkLedger: Privacy-Preserving Auditing for Distributed Ledgers". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 65–80. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/narula.

[21] Shen Noether. *Ring Signature Confidential Transactions for Monero*. Cryptology ePrint Archive, Paper 2015/1098. https://eprint.iacr.org/2015/1098. 2015. URL: https://eprint.iacr.org/2015/1098.

[22] Fergal Reid and Martin Harrigan. *An analysis of anonymity in the bitcoin system*. Springer, 2013.

[23] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret: Theory and Applications of Ring Signatures". In: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 164–186. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_7. URL: https://doi.org/10.1007/11685654_7.

[24] Dorit Ron and Adi Shamir. "Quantitative analysis of the full bitcoin transaction graph". In: *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*. Springer. 2013, pp. 6–24.

[25] Florian Tschorsch and Björn Scheuermann. "Bitcoin and beyond: A technical survey on decentralized digital currencies". In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 2084–2123.