Privacy-preserving auditable token payments in a permissioned blockchain system

Elli Androulaki[1], Jan Camenisch[2,⋆], Angelo De Caro[1], Maria Dubovitskaya[2,⋆], Kaoutar Elkhiyaoui[1], and Björn Tackmann[2,⋆]

[1] IBM Research - Zurich
{lli,adc,kao}@zurich.ibm.com
[2] DFINITY
{jan,maria,bjoern}@dfinity.org

Abstract. Token management systems were the first application of blockchain technology and are still the most widely used one. Early implementations such as Bitcoin or Ethereum provide virtually no privacy beyond basic pseudonymity: all transactions are written in plain to the blockchain, which makes them perfectly linkable and traceable.
Several more recent blockchain systems, such as Monero or Zerocash, implement improved levels of privacy. Most of these systems target the *permissionless* setting, just like Bitcoin. Many practical scenarios, in contrast, require token systems to be *permissioned*, binding the tokens to user identities instead of pseudonymous addresses, and also requiring auditing functionality in order to satisfy regulation such as AML/KYC.
We present a privacy-preserving token management system that is designed for permissioned blockchain systems and supports fine-grained auditing. The scheme is secure under computational assumptions in bilinear groups, in the random-oracle model.

1 Introduction

1.1 Motivation

Token payment systems were the first application of blockchain technology and are still the most widely used one. Early implementations such as Bitcoin or Ethereum provide virtually no privacy beyond basic pseudonymity: all transactions are written in plain to the blockchain, which makes them linkable and traceable.

Several approaches exist for adding different levels of privacy to blockchain-based transactions. *Tumblers* such as CoinJoin [30] combine several transactions of different users and obscure the relation between payers and payees. In *mix-in*-based systems such as CryptoNote [40], transactions reference multiple superfluous payers that do however not actually participate in the transaction and only serve as a cover-up for the actual payer. Confidential Assets [35] hide the amounts in a payment but leave the payer-payee relation in the open. Finally, advanced systems such as Zerocash [3] both encrypt the amounts and fully hide the payer-payee relation.

While the privacy of transactions is important, it should not void the requirements of transparency and auditability, especially in permissioned networks that come with strong identity management and promise to ensure accountability and non-deniability. This paper introduces a solution dedicated to the permissioned setting to cover this gap: it hides the content of transactions without preventing authorized parties from auditing them.

Another goal of this paper is to move away from complex and non-falsifiable computational assumptions that underpin zkSNARK-based schemes and instead work with more conservative assumptions. Restricting ourselves to the permissioned setting allows us to leverage a combination of signatures and standard ZK-proofs to achieve these goals.

1.2 Related work

Various solutions for improving privacy in blockchain-based token systems exist. We briefly review the most related ones.

⋆ Work done while at IBM Research - Zurich.

Miers et al. [31] introduced Zerocoin, which allows users to anonymize their bitcoins by converting them into *zerocoins* that rely on Pedersen commitments and zero-knowledge proofs. Zerocoins can be changed back to bitcoins without leaking their origin. Zerocoin however does not offer any transacting or auditing capabilites.

Confidential Assets of Poelstra et al. [35] protect privacy (in a limited form) by hiding the types and the values of the traded assets. The idea, similarly to Zerocoin, is to use Pedersen commitments to encode the amount and types of traded assets, and zero-knowledge proofs to show the validity of a transaction. The proposed scheme however does not hide the transaction graph or the public keys of the transactors. While this allows for some form of public auditability, it hinders the privacy of the transacting parties.

Zerocash [3] is the first fully anonymous decentralized payment scheme. It offers unconditional anonymity, to the extent that users can repudiate their participation in a transaction. Thanks to a combination of hash-based commitments and zkSNARKs, Zerocash validates payments and prevents double-spending relatively efficiently. On the downside, Zerocash requires a trusted setup and an expensive transaction generation and its security relies on non-falsifiable assumptions.

Extensions to Zerocash have been proposed [19] to support expressive validity rules to provide accountability: notably, the proposed solution ensures regulatory closure (i.e. allowing exchanges of assets of the same type only) and enforcing spending limits. In terms of accountability, the proposed scheme allows the tracing of certain *tainted* coins, while not really extensively and consistently allowing transactions to be audited. By building on Zerocash, the proposed scheme inherits the same limitations regarding computational assumptions and trusted setup.

QuisQuis [18] and Zether [4] propose solutions that provide partial anonymity. On a high level, instead of sending a transaction that refers only to the accounts of the sender and the recipients of a payment, the sender adds accounts of other users, who act as an anonymity set (similar to CryptoNote [40]). Both schemes couple ElGamal encryption with Schnorr zero-knowledge proofs to ensure that user accounts reflect the correct payment flows. Contrary to Zerocash, QuisQuis and Zether rely on falsifiable assumptions and do not require any trusted setup.

Solidus [15] is a privacy-preserving protocol for asset transfer that is suitable for intermediated bilateral transactions, where banks act as mediators. Solidus conceals the transaction graph and values by using banks as proxies. The authors leverage ORAMs to allow banks to update the accounts of their clients without revealing exactly which accounts are being updated. The novelty of Solidus is PVORM, which is an ORAM that comes with zero-knowledge proofs that show that the ORAM updates are correct with respect to the transaction triggering them. In Solidus there is no dedicated auditing functionality; however banks could open the content of relevant transactions at the behest of authorized auditors.

The zkLedger protocol of Narula, Vasquez, and Virza [33] is a permissioned asset transfer scheme that hides transaction amounts as well as the payer-payee relationship and supports auditing. One main difference with our approach is the end user: zkLedger aims at a setting where the transacting parties are banks, whereas our solution considers the end user to be the client of "a bank." This is why zkLedger enjoys relatively more efficient proofs and could afford a transaction size that grows linearly with the number of total transactors in the platform (i.e. banks), which is inherently small. (In our scheme, transaction sizes do *not* grow with the number of overall parties.) Similarly when it comes to auditability, zkLedger offers richer and more flexible semantics but at the expense of audit granularity. Auditing in zkLedger is limited to banks and does not cover cases where auditors are required to monitor the transaction flow of the clients (of the banks).

1.3 Results

We describe a token management system for permissioned networks that enjoys the following properties:

Privacy: Transactions written on the blockchain conceal both the values that are transferred and the payer-payee relationship. The transaction leaks no information about the tokens spent in this transaction beyond the fact that they are valid and unspent.

Authorization: Users authorize transactions via credentials; i.e., the authorization for spending a token is bound to the user's identity instead of a pseudonym (or address). The authorization makes use of anonymous credentials and is privacy-preserving.

Auditability: Each user has an assigned auditor that is allowed to see the transaction information *related to that particular user*.

Satisfying these three requirements is crucial for implementing a payment system that protects the users' privacy but at the same time complies with regulation.

The system we propose is based on the unspent transaction output (UTXO) model pioneered by Bitcoin [32] and supports multi-input-multi-output transactions. It inherits several ideas from prior work, such as the use of Pedersen commitments from Confidential Assets [35] and the use of serial numbers to prevent double-spending from Zerocash [3]. These are combined with a blind certification mechanism that guarantees the validity of tokens via threshold signatures, and with an auditing mechanism that allows flexible and fine-grained assignment of users to auditors.

We use a selection of cryptographic schemes that are based in the discrete-logarithm or pairing settings and are structure-preserving, such as Dodis-Yampolskiy VRF [16], ElGamal encryption [17], Groth signatures [23], Pedersen commitments [34], and Pointcheval-Sanders signatures [36]. This allows us to use the relatively efficient Groth-Sahai proofs [25] and achieve security under standard assumptions, in the random-oracle model.

Outline The remainder of the paper is structured as follows. In Section 2, we provide further background on several important techniques. Section 3 then shows an overview of our protocol. Section 4 describes the types of cryptographic schemes used in the protocol, before Section 5 specifies the security model. Section 6.5, contains the protocol description and the security statement. In Section 7, we describe the implementation and the performance measurements. Section 8 concludes.

2 Background

2.1 Decentralized token systems

A decentralized token transfer is performed by appending a `transfer` transaction to the blockchain. Such a transaction comprises the transfer details (e.g. sender, receivers, type and value) and a proof that the author of the transaction possesses enough liquidity to perform the transfer. The transaction is then validated against the blockchain state (i.e. the ledger). More precisely, the blockchain checks that the origin of the transaction has the right to transfer the token and that the overall quantity of tokens is preserved during the transfer. Existing decentralized token systems are either *account-based* (e.g. [26]) or *unspent transaction outputs (UTXO)-based* (cf. [32]). A valid transfer in an account-based systems results in updating the accounts of the sender and the receivers. In a UTXO-based token system, a `transfer` transaction includes a set of inputs—tokens to be consumed—and outputs—tokens to be created. A valid transfer in such systems leads to destroying the inputs and adding the outputs to the ledger to be later consumed by subsequent transactions.

2.2 Privacy-preserving token systems

Decentralization of token systems gives rise to serious privacy threats: if transactions contain the transfer information in the clear, then anyone with access to the ledger is able to learn the history of each party's transaction. We call a decentralized token system *privacy-preserving* if it partially or fully hides the transfer details. Examples of decentralized and privacy preserving token systems are Confidential Assets [35], Zether [4], QuisQuis [18] and Zerocash [3], with the last one offering the highest level of privacy protection.

Zerocash. The privacy in Zerocash relies on a combination of commitments, zkSNARKs and Merkle-tree membership proofs. Namely, tokens in Zerocash are computed as a hiding commitment to a value, a type and an owner's pseudonym. After its creation, a token is added to a *public Merkle tree* and during a transfer, the origin of the transaction proves in zero-knowledge that the token is valid (i.e. included in the Merkle tree), that it was not spent before and that she owns it. Thanks to zkSNARKs, transaction validation in Zerocash is quite fast. Yet, this comes at the cost of a *complex trusted setup* and a very expensive proof generation. To obviate these two limitations, we exploit the properties of *permissioned token systems* to replace Merkle trees with signature-based membership proofs, in order to devise a solution that relies only on Groth-Sahai proofs [25].

2.3 Permissioned token systems

In a permissioned token system such as Hyperledger Fabric [1] or Quorum [27], a user is endowed with a long-term credential that reflects her attributes and role. Tokens are introduced by special users, called *issuers*, through

`issue` transactions. These transactions are then validated against predefined policies that reflect existing norms and regulations. For example, issuing policies define which issuers are authorized to create which tokens and under which conditions. Similarly to `issue` transactions, `transfer` can also be validated against policies: the simplest of which is that a transfer can take place only between registered users. A fundamental property of permissioned systems is that transactions are signed using long-term credentials. As a by-product, transactions can be traced back to their origin, enforcing thus the requirements of auditability and accountability.

2.4 Signature-based membership proofs

We use signatures to implement zero-knowledge membership proofs [5]. Roughly speaking, consider a set $\mathbb{S}$ that consists of elements that are signed using a secret key sk associated with $\mathbb{S}$. It follows that proving knowledge of some e in $\mathbb{S}$ in zero-knowledge amounts to **(i)** computing a hiding commitment of e; **(ii)** and then proving knowledge of a signature, computed with sk, on the committed value. In this paper, we use this mechanism for two purposes: **(i)** to prove that a user is in the set of registered users; **(ii)** and to show that a token is in the set of *valid* tokens recorded in the ledger.

2.5 Encryption-based auditability

In an encryption-based auditable token system, transactions carry ciphertexts intended for the authorized auditors. For such a mechanism to be viable, it is important to ensure that **(i)** the ciphertexts encrypt the correct information; **(ii)** and they are computed using the correct keys. This can be achieved through zero-knowledge proofs—computed by the creator of the transaction—that link the ciphertexts to the transfer details and attest that the two requirements listed above are not violated.

3 Overview

3.1 Design Approach

The first component of our solution is *token encoding*. Each token is represented by a hiding commitment (e.g. Pedersen's) that contains the identifier of the token owner, the value of the token and its type. The life-cycle of a token is governed by two transactions: `issue` and `transfer`. An `issue` transaction creates a token of a given type and value and assigns it to the issuer (i.e. author of the transaction). For an `issue` transaction to be valid it should be submitted by the authorized issuer. For ease of exposition, we assume that a token issuer can issue only one type of tokens and we conflate the type of a token with its issuer. Once a token is created it changes ownership through `transfer` transaction. Given that we operate within the UTXO framework, `transfer` transaction consists of a set of input tokens to be consumed and a set of output tokens to be created and it is validated against the following rules:

- The author of the transaction is the rightful owner of the input tokens;
- the owners of the output tokens are registered;
- the type and the value of tokens are preserved;
- the input tokens can be traced back to *valid* transactions in the ledger;
- the input tokens were not consumed before (to prevent double spending).

Our solution moves away from zkSNARK and their trusted setup assumption and relies only on standard NIZK proofs (e.g. Groth-Sahai's [25]). More precisely, it leverages the *permissioned setting* to use ZK signature-based membership proofs to ascertain that a user is registered and that a token belongs to the ledger in a privacy-preserving manner. Namely, we assume that there is a *registration authority* that provides authorized users with long-term credentials (i.e. signatures) with their attributes, and a *certifier* that a user contacts with a *certification* request to vouch for the validity of tokens she owns. A *certification* request contains a token (i.e. commitment) and upon receiving such a request the certifier checks whether the token is included in a valid transaction in the ledger. If so, the certifier *blindly* signs the token and the resulting signature can be used subsequently to prove that the token is legitimate.

To prevent double spending, we leverage serial numbers to identify tokens when they are consumed, as in Zerocash. It is important that these serial numbers satisfy the following security properties: **(i)** collision

resistance: two tokens result in two different serial numbers; **(ii)** determinism: the same token always yields the same serial number; **(iii)** unforgeability: only the owner of the token can produce a valid serial number. We use *verifiable random functions* (e.g. Dodis-Yampolskiy [16]) to generate serial numbers that are a function of the token owner secret key and a randomness that is tied to the token at its creation time.

To enable auditability, we encrypt the information in `transfer` transactions (i.e. sender, receivers, types and values) under the public keys of the sender's and the receivers' auditors. To accommodate real-world use-cases, our solution does not assume a single auditor for all users. This means that the encryption scheme must not only be *semantically-secure* but also *key-private*, such as ElGamal.

3.2 Architectural Model

Participants Our solution involves the following types of users:

Users They own tokens that represent some real-world assets, and wish to exchange their tokens with other users in the network. This is achieved through `transfer` transactions.

Issuers They are users who are authorized to introduce tokens in the system through `issue` transactions. For simplicity purposes, we assume that each issuer is allowed to introduce only one type of token and that the type of token is defined as the identifier of the respective issuer.

Auditors These are entities with the authority and responsibility to inspect transactions of users. We assume that each user in the system is assigned an auditor at registration time and that this assignment is immutable.

Certifier This is a privileged party that provides users with *certificates* that vouch for the validity of their tokens. More specifically, a user who wishes to transfer the ownership of a token contacts the certifier with the token; the certifier in turn inspects whether the token appears in a *valid transaction* in the ledger or not. If so, the certifiers sends a certificate (i.e. signature) to that effect to the user.

Registration authority This is a privileged party that generates long-term credentials for all the participants in the system, including users, issuers, auditors and certifiers. Namely, the credentials tie the real-world identity of the requestor to her attributes and her public keys. An example of an attribute in our solution is the role (e.g., "user", "auditor", "certifier") that determines what type of credentials to be generated. A *user credential* is a signature that binds the user public keys to both her identifier and her auditor's identifier; whereas an *auditor credential* is a signature that links the auditor's encryption key to her identifier; finally the certifier's credential is a signature of her public key.

Ledger This is a decentralized data store that keeps records of all `issue` and `transfer` transactions that have been previously submitted. It is accessible to all parties in the system to read from and submit transactions to. The ledger has a *genesis block* that contains **(i)** the system security parameters; **(ii)** the public information of the registration authority and the credentials of the certifier; and **(iii)** the identifiers of the issuers authorized to introduce tokens in the system.

Interactions The interactions between system participants are shown in Figure 1. At first users, issuers, auditors and certifier engage with the registration authority in a *registration* protocol to get long-term credentials for their subsequent interactions.

A genesis block is created that announces the system parameters, the public information of the registration authority, the credentials of the certifiers and an *initial list* of authorized issuers. From now on, the system will be able to accommodate token management requests. More precisely, issuers submit `issue` transactions to the ledger to introduce new tokens, and the ledger ensures that all incoming transactions are correctly stored. Anyone with access to the ledger, in particular the certifier and auditors, can verify whether the transaction is valid or not using the information in the *genesis* block. Subsequently, *token transfer* operations take place between users through `transfer` transactions. The ledger again stores the transaction to make it available to

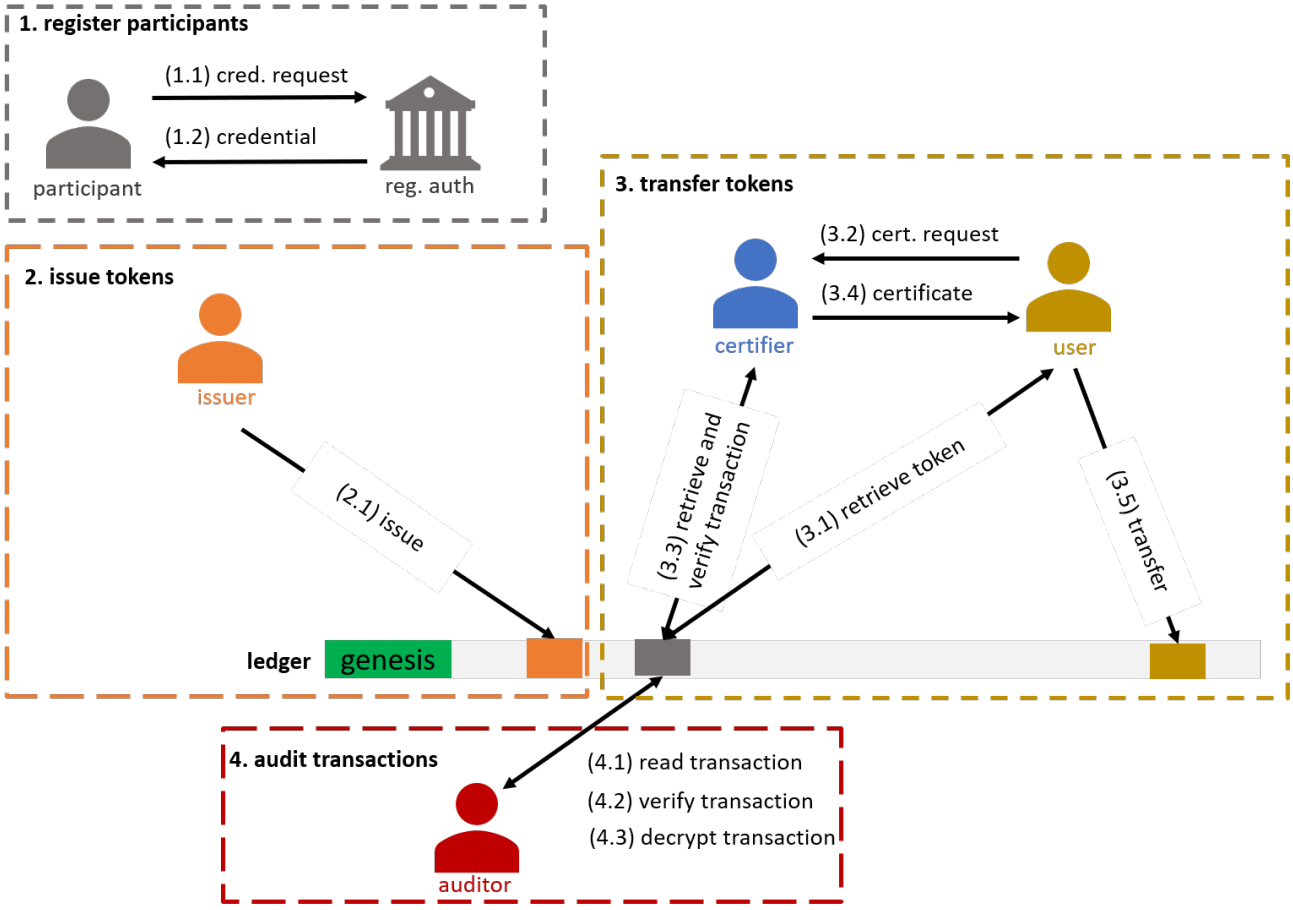

Fig. 1. This figure shows the interactions between the participants in our system. Users, issuers, auditors and the certifier are granted credentials by interacting with the registration authority. Upon an external decision to create new tokens, one or more issuers submit an `issue` transaction to the ledger and the ledger automatically adds the transaction. To transfer a token, a user contacts the certifier with a certification request that references the token to be signed and the transaction that created it. If it is an `issue` transaction, the certifier checks if the author of the transaction is authorized. If it is a `transfer` transaction, the certifier verifies if the ZK proof is valid. Once the owner of a token receives the corresponding certificate, she can transfer it to registered users. Finally, auditors assigned to a user can audit that user's transactions by obtaining access to the ledger.

all participants. For simplicity, we assume that the ledger accepts all transactions without verification. However to transfer a token, a user contacts the certifier that checks if the transaction including the token is valid. Only then the certifier signs the token making it transferable. To inspect a user activity, an auditor reads transactions from the ledger, checks their validity and tries to decrypt them with her secret key only if they are valid.

3.3 Trust Model

Registration authority We assume that the registration authority is trusted to assign *correct credentials* to all parties in the system. A participant presents a set of attributes and her public key to the registration authority and receives in return a credential that binds her attributes to her public key. It is incumbent upon the registration authority to verify the correctness of the attributes of a participant prior to sending the credential. For example, it should verify that the participant knows the secret key underlying the advertised public key; it should also verify in the case of users that the announced auditors are legitimate. Furthermore, the registration authority is trusted to assign *one unique credential* per participant. However, it is not *trusted* regarding the privacy of users.

Notice that the trust assumption in the registration authority can be relaxed via a distributed registration protocol.

Users Users may collude to compromise the security of the system. They may attempt to steal tokens of others, double-spend their own tokens, forge new tokens, transfer tokens to non-registered users, encrypt incorrect information in the auditors' ciphertexts, etc. They may also attempt to undermine the privacy of honest users by de-anonymizing transactors, linking tokens, learning the content of transactions, etc.

Issuers Issuers are users that are trusted to introduce tokens of a certain type. However, issuers may collude to surreptitiously create tokens on behalf of other honest issuers, compromise the privacy of users and obviate auditing among other things.

Certifier To transfer a token, a user contacts the certifier to receive a signature that proves the token validity (i.e. inclusion in a valid transaction in the ledger). Accordingly, the certifier is trusted to generate signatures only for tokens that can be traced back to valid transactions in the ledger. We can relax this trust assumption using a *threshold signature* scheme that distributes the certification process and guarantees its integrity as long as the majority of the signers (i.e. certifiers) is honest.

While certifiers may be able to link `transfer` transactions referencing certified tokens to certification requests, they should not be able to derive any further information about the transactions in the ledger or the tokens they certify.

Auditors Auditors are authorized to only learn the information pertained to their assigned users. That is, colluding users and auditors should not be able to derive any information about the token history of users who are not assigned to the malicious auditors.

Ledger For simplicity purposes, we use the ledger only as a time-stamping service. It does not perform any transaction validation, rather it stores the full transaction including the proofs of correctness. Anyone later can check the transaction, verify the proofs and decide if the transaction is valid or not. We assume however that the ledger is *live* and *immutable*: a transaction submitted to the ledger will eventually be included and cannot be deleted afterwards.

4 Cryptographic schemes

The section presents the cryptographic schemes that will be used to build the protocol. We only present them briefly, and provide more information on concrete instantiations later in the paper. All cryptographic algorithms are parametrized by a so-called *security parameter* $\lambda \in \mathbb{N}$ given (sometimes implicitly) to the algorithms.

4.1 Commitment schemes

A commitment scheme COM consists of three algorithms ccrsgen, commit, and open. The *common reference string (CRS)* generator ccrsgen is probabilistic and, on input the security parameter λ, samples a CRS $crs \leftarrow\!\!\$\ \mathrm{ccrsgen}(\lambda)$. The commitment algorithm is a probabilistic algorithm that, on input of a vector $(m_1, \ldots, m_\ell)$ of messages, outputs a pair $(cm, r_{\mathrm{cm}}) \leftarrow\!\!\$\ \mathrm{commit}(crs, (m_1, \ldots, m_\ell))$ of commitment cm and opening r_{cm}. We sometimes also use the notation $cm \leftarrow \mathrm{commit}(crs, (m_1, \ldots, m_\ell); r_{\mathrm{cm}})$ to emphasize that a specific random string r_{cm} is used. Finally, there is a deterministic opening algorithm $\mathrm{open}(crs, cm, (m_1, \ldots, m_\ell), r_{\mathrm{cm}})$ that outputs either true or false.

Commitments must be *hiding* in the sense that, without knowledge of r_{cm}, they do not reveal information on the committed messages, and they must be *binding* in the sense that it must be infeasible to find a different set of messages $m'_1, \ldots, m'_\ell$ and r'_{cm} that open the same commitment.

4.2 Digital signature schemes

A digital signature scheme SIG consists of three algorithms skeygen, sign, and verify. The key generation algorithm $(sk, pk) \leftarrow\!\!\$\ \mathrm{skeygen}(\lambda)$ takes as input the security parameter λ and outputs a pair of private (or secret) key sk and public key pk. Signing algorithm $s \leftarrow\!\!\$\ \mathrm{sign}(sk, m)$ takes as input private key sk and message m, and produces a signature s. Deterministic verification algorithm $b \leftarrow \mathrm{verify}(pk, m, s)$ takes as input public key pk, message m, and signature s, and outputs a Boolean b that signifies whether s is a valid signature on m relative to public key pk. The standard definition of signature scheme security, existential unforgeability under chosen-message attack, has been introduced by Goldwasser, Micali, and Rivest [22]. It states that the probability for an efficient adversary, given an oracle that generates valid signatures, to output a valid signature on a message that has not been queried to the oracle must be negligible. The security of a signature scheme can also be described by an ideal functionality $\mathcal{F}_{\mathrm{SIG}}$, which can be found in Appendix A.4.

4.3 Threshold signature schemes

A non-interactive threshold signature scheme TSIG consists of four algorithms tkeygen, sign, combine, and verify. Threshold key generation $(sk_1, \ldots, sk_n, pk_1, \ldots, pk_n, pk) \leftarrow\!\!\$\ \mathrm{tkeygen}(\lambda, n, t)$ gets as input security parameter λ, total number of parties n, and threshold t. Each party can sign with their own secret key sk_i as above to generate a partial signature s_i. Any t valid signatures can be combined using combine into a full signature s, which is verified as in the non-threshold case. A signature produced honestly by any t parties verifies correctly, but any signature produced by less than t parties will not verify.

4.4 Public-key encryption.

A public-key encryption scheme PKE consists of three algorithms ekeygen, enc, and dec. Key-generation algorithm $(sk, pk) \leftarrow\!\!\$\ \mathrm{ekeygen}(\lambda)$ takes as input security parameter λ and outputs a pair of private key sk and public key pk. Probabilistic encryption algorithm $c \leftarrow\!\!\$\ \mathrm{enc}(pk, m)$ takes as input message m and public key pk and produces ciphertext c. We also write $c \leftarrow \mathrm{enc}(pk, m; r)$ where we want to emphasize that the encryption uses randomness r. Deterministic decryption $m \leftarrow \mathrm{dec}(sk, c)$ takes as input ciphertext c and private key sk and recovers message m. Correctness requires that $\mathrm{dec}(sk, \mathrm{enc}(pk, m)) = m$ for all (sk, pk) generated by ekeygen. For our work, we require semantic security as first defined by Goldwasser and Micali [21]. The scheme must additionally satisfy key privacy as defined by Bellare, Boldyreva, Desai, and Pointcheval [2], which states that, given a ciphertext c, it must be hard to determine the public key under which the ciphertext is encrypted.

4.5 Verifiable random functions

A verifiable random function VRF consists of three algorithms vkeygen, eval, and check. Key generation $(vsk, vpk) \leftarrow\!\!\$\ \mathrm{vkeygen}(\lambda)$ takes as input the security parameter and outputs a pair of private key vsk and public key vpk. Deterministic evaluation $(y, \pi) \leftarrow \mathrm{eval}(vsk, x)$ takes as input secret key vsk and input value x, and produces as output the value y with proof π. Deterministic verification $b \leftarrow \mathrm{check}(vpk, x, y, \pi)$ takes as input public key vpk, input x, output y, and proof π, and outputs a Boolean that signifies whether the proof should be accepted.

The scheme satisfies *correctness* if honest proofs are always accepted. *Soundness* means that it is infeasible to produce a valid proof for a wrong statement. The scheme must satisfy *pseudo-randomness* which means that, given only vpk, the output y for a fresh input x is indistinguishable from a random output.

4.6 Non-interactive zero-knowledge proofs of knowledge

Let $\mathcal{R}$ be a binary relation. For pairs $(x, w) \in \mathcal{R}$, x is called statement (i.e. public input) whereas w is called witness (i.e. private input). $\mathcal{L} = \{x, \exists\ w \text{ s.t. } (x, w) \in \mathcal{R}\}$ is called language of relation $\mathcal{R}$. A NIZK proof-of-knowledge system NIZK for language $\mathcal{L}$ comprises three algorithms: zkcrsgen, prove and verify. CRS generation $\text{crs} \leftarrow\!\!\$\ \text{zkcrsgen}(\lambda, \mathcal{R})$ takes as input the security parameter λ and a relation $\mathcal{R}$ and outputs a common reference string. On input of $(x, w) \in \mathcal{L}$ and crs, proof generation $\psi \leftarrow\!\!\$\ \text{prove}(x, w, \text{crs})$ returns a proof ψ. Proof ψ is verified by calling algorithm $b \leftarrow \text{verify}(\psi, x, \text{crs})$, which in turn outputs a Boolean that indicates whether the proof is valid or not.

Correctness for such a proof system means that honestly-generated proofs are always accepted. *Knowledge soundness* implies that a prover that produces a valid proof for some x must know a witness w with $(x, w) \in \mathcal{R}$, in the sense that w can be extracted. Finally, *zero-knowledge* ensures that the verification of correct statements yields nothing beyond the fact that they are correct. We describe the security of NIZK proofs of knowledge more formally using an ideal functionality $\mathcal{F}_{\text{NIZK}}$ deferred to Appendix A.2.

In the remainder of the paper, we succinctly represent zero knowledge proofs of knowledge using the common notation introduced by Camenisch and Stadler [10], namely $\text{PK}\{(x) : w\}$ denotes a proof of knowledge of *witness* w for *statement* x.

5 Security formalization

5.1 Notation

We use sans-serif fonts to denote constants such as true or false, and `typewriter` fonts to denote string constants.

5.2 Universal composition and MUC

In this section, we only recall basic notation and specific parts of the model that we need in this work. Details can be found in [11, 13, 12].

The UC framework follows the simulation paradigm, and the entities taking part in the protocol execution (protocol machines, functionalities, adversary, and environment) are described as *interactive Turing machines* (ITMs). The execution is an interaction of *ITM instances* (ITIs) and is initiated by the environment $\mathcal{Z}$ that provides input to and obtains output from the protocol machines, and also communicates with adversary $\mathcal{A}$ resp. simulator $\mathcal{S}$. The adversary has access to the protocols as well as functionalities used by them. Each ITI has an identity that consists of a party identifier *pid* and a session identifier *sid*. The environment and adversary have specific, constant identifiers, and ideal functionalities have party identifier $\perp$. The understanding here is that all ITIs that share the same code and the same *sid* are considered a *session* of a protocol. It is natural to use the same *pid* for all ITIs that are considered the same party.

ITIs can invoke other ITIs by sending them messages, new instances are created adaptively during the protocol execution when they are first invoked by another ITI. To use composition, some additional restrictions on protocols are necessary. In a protocol $\mu^{\phi \to \pi}$, which means that all calls within μ to protocol ϕ are replaced by calls to protocol π, both protocols ϕ and π must be *subroutine respecting*. This means, in a nutshell, that while those protocols may have further subroutines, all inputs to and outputs from subroutines of ϕ or π must only be given and obtained through ϕ or π, never by directly interacting with their subroutines. (This requirement is natural, since a higher-level protocol should never directly access the internal structure of ϕ or π; this would obviously hurt composition.) Also, protocol μ must be *compliant*. This roughly means that μ should not be invoking instances of π with the same *sid* as instances of ϕ, as otherwise these instances of π would interact with the ones obtained by the operation $\mu^{\phi \to \pi}$.

In summary, a protocol execution involves the following types of ITIs: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, instances of the protocol machines π, and (possibly) further ITIs invoked by $\mathcal{A}$ or any instance of π (or their subroutines). The contents of the environment's output tape after the execution is denoted by the random

variable $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$, where $\lambda \in \mathbb{N}$ is the *security parameter* and $z \in \{0,1\}^*$ is the input to the environment $\mathcal{Z}$. The formal details of the execution are specified in [12]. We say that a protocol π UC-realizes a functionality $\mathcal{F}$ if

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z} : \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}},$$

where "$\approx$" denotes indistinguishability of the respective distribution ensembles, and ϕ is the dummy protocol that simply relays all inputs to and outputs from the functionality $\mathcal{F}$.

Multi-protcocol UC The standard UC framework does not allow to modularly prove protocols in which, e.g., a zero-knowledge proof system is used to prove that a party has performed a certain evaluation of a cryptographic scheme correctly. Camenisch, Drijvers, and Tackmann [6] recently showed how this can be overcome. In a nutshell, they start from the standard $\mathcal{F}_{\text{NIZK}}^{R}$-functionality which is parametrized by a relation R, and show that if R is described in terms of evaluating a protocol, then the protocol can equivalently be evaluated outside of the functionality, and even used to realize another functionality $\mathcal{F}$. This results in a setting where $\mathcal{F}_{\text{NIZK}}$ validates a pair (y, w) of statement y and witness w by "calling out" to the other functionality $\mathcal{F}$. We use this proof technique extensively in this work.

5.3 The privacy-preserving token functionality

The functionality $\mathcal{F}_{\text{TOKEN}}$ realized by our privacy-preserving token system is formalized in Figure 2. To keep the presentation simple, the functionality formalizes the guarantees for the case of a single token issuer I. The functionality initially requires registration authority A, certifier C, and issuer I to initialize. (This corresponds to the fact that all protocol steps depend on those parties' keys.) Likewise, regular parties P have to generate and register their keys before they can perform operations. Each party can then `read` the tokens they own and generate `transfer` transactions that reference those tokens and transfers them to one or more receivers. Issuers can additionally `issue` new tokens. In the inputs and outputs of the functionality, v always represents the value of a token, and *cm* serves as a handle identifying the token (it stems from the commitment that represents the token on the ledger). Finally, the functionality specifies which information is potentially leaked to the adversary, and which operations the adversary can perform in the name of corrupted parties.

5.4 Set-up functionalities

Our protocol requires a number of set-up functionalities to be available. Most of these functionalities are widely used in the literature, which is why we only briefly describe them here and specify them in detail in Appendix A.

Common reference string Functionality $\mathcal{F}_{\text{CRS}}$ provides a string that is sampled at random from a given distribution and accessible to all participants. All parties can simply query $\mathcal{F}_{\text{CRS}}$ for the reference string. The functionality is generally used to generate common public parameters used in a cryptographic scheme.

Transaction ledger We describe a simplified transaction ledger functionality as $\mathcal{F}_{\text{LEDGER}}$ in Figure 3. In a nutshell, every party can append bit strings to a globally available ledger, and every party can retrieve the current ledger.

The functionality intentionally idealizes the guarantees achieved by a real-world ledger; transactions are immediately appended, final, and available to all parties. We also use $\mathcal{F}_{\text{LEDGER}}$ as a local functionality. These simplifications are intended to keep the paper more digestible.

Secure and private message transfer Functionality $\mathcal{F}_{\text{SMT}}$ provides a message transfer mechanism between parties. The functionality builds on the ones described by Canetti and Krawczyk [14], but additionally hides the sender and receiver of a message, if both are honest. This is required since our protocol passes information between transacting parties, and leaking the communication pattern to the adversary would revoke the anonymity otherwise provided by our protocol.

Public-key registration The registration functionality $\mathcal{F}_{\text{REG}}$ models a public-key infrastructure. It allows each party P to input one value $x \in \{0,1\}^*$ and makes the pair (P, x) available to all other parties. This is generally used to publish public keys, binding them to the identity of a party.

Privacy-preserving token functionality $\mathcal{F}_{\text{TOKEN}}$

Functionality $\mathcal{F}_{\text{TOKEN}}$ stores a list of registered users and an initially empty map $\mathsf{Records}$. The session identifier is of the form $sid = (A, C, I, sid')$.

- Upon input $\texttt{init}$ from $P \in \{A, C, I\}$, output to $\mathcal{A}$ $(\texttt{initialized}, P)$. (This must happen for all three before anything else.)
- Upon input $\texttt{register}$ from a party P, if P is unregistered, then mark P as registered and output $(\texttt{registered}, P)$ to $\mathcal{A}$. (Otherwise ignore.)
- Upon input $\texttt{read}$ from a registered party P, issue $(\texttt{read?}, P)$ to $\mathcal{A}$. Upon receiving response $(\texttt{read!}, P)$ from $\mathcal{A}$, return to P a list of all records of the type (cm, v) that belong to P.
- Upon input $(\texttt{issue}, v)$ from I, output $(\texttt{issue}, v)$ to $\mathcal{A}$. Receiving from $\mathcal{A}$ a response $(\texttt{issue}, cm)$, if $\mathsf{Records}[cm] \neq \bot$ then abort, else set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{alive})$. Return $(\texttt{issued}, cm)$ to I.
- Upon receiving an input $(\texttt{issue}, v, cm)$ from $\mathcal{A}$, where I is corrupt, check and record the commitment as in the previous step. Return to $\mathcal{A}$.
- Upon input $\left(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$ from an honest party P, where P and all R_j for $j = 1, \ldots, n$ are registered, proceed as follows.
  1. If, for any $i \in \{1, \ldots, m\}$, $\mathsf{Records}[cm_i] = \bot$ then abort, else set $(v_i^{\text{in}}, P_i', st_i) \leftarrow \mathsf{Records}[cm_i]$.
  2. If, for any $i \in \{1, \ldots, m\}$, $st_i \neq \texttt{alive}$ or $P_i' \neq P$, then abort.
  3. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^n v_j^{\text{out}}$ then abort.
  4. Let L be an empty list. For all $j = 1, \ldots, n$, if R_j is corrupt then append to L the information $(j, P, R_j, v_j^{\text{out}})$. Output $(\texttt{transfer}, m, n, L)$ to $\mathcal{A}$.
  5. Receiving from $\mathcal{A}$ a response $(\texttt{transfer}, (cm_j^{\text{out}})_{j=1}^n)$, if $\mathsf{Records}[cm_j^{\text{out}}] \neq \bot$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\mathsf{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{delayed})$ for all $j \in \{1, \ldots, n\}$ and set $\mathsf{Records}[cm_i] \leftarrow (v_i^{\text{in}}, P', \texttt{consumed})$ for all $i \in \{1, \ldots, m\}$.
  6. Return $(\texttt{transferred}, (cm_j^{\text{out}})_{j=1}^n)$ to P.
- On $(\texttt{transfer}, P, (cm_i^{\text{in}})_{i=1}^m, (R_j, v_j^{\text{out}}, cm_j^{\text{out}})_{j=1}^n)$ where P is corrupt, proceed analogously to above.
- Upon receiving an input $(\texttt{deliver}, cm)$ from $\mathcal{A}$ with $\mathsf{Records}[cm] = (v, P, \texttt{delayed})$ for some v, set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{alive})$. If C is corrupted, then output P to $\mathcal{A}$.

Fig. 2. Privacy-preserving token functionality.

Ledger functionality $\mathcal{F}_{\text{LEDGER}}$

Functionality $\mathcal{F}_{\text{LEDGER}}$ stores an initially empty list L of bit strings.

- Upon input $(\texttt{append}, x)$ from a party P, append x to L. If P is corrupt then send $(\texttt{append}, x, P)$ to $\mathcal{A}$, else return to P.
- Upon input `retrieve` from a party P or $\mathcal{A}$, return L.

Fig. 3. Ledger functionality.

Extended anonymous authentication functionality $\mathcal{F}_{\text{A-AUTH}}$

Functionality $\mathcal{F}_{\text{A-AUTH}}$ is parametrized by a commitment opening algorithm open. It stores an initially empty $\mathcal{U}$ of registered users, and an initially empty list of records.

- Upon input `register` from a party P where $P \notin \mathcal{U}$, set $\mathcal{U} \leftarrow \mathcal{U} \cup \{P\}$ and output $(\texttt{registered}, P)$ to $\mathcal{A}$.
- Upon input $(\texttt{lookup}, P')$ from a party P, return (the result of) $P' \in \mathcal{U}$.
- Upon input $(\texttt{prove}, crs, cm, r_{\text{cm}}, x, y, m)$ from party P, if $\text{open}(crs, cm, (x, P, y), r_{\text{cm}})$ then generate a proof ψ, store the record $(\texttt{transfer}, crs, cm, \psi, m)$ internally and output ψ to P.
- Upon input $(\texttt{verify}, crs, cm, \psi, m)$ from some P, look up if there is a record $(\texttt{transfer}, crs, cm, \psi, m)$ and output success (only) if it exists.

Fig. 4. Extended anonymous registration functionality.

Anonymous authentication As our protocol is in the permissioned setting but supposed to provide privacy, we need anonymous credentials to authorize transactions. Our schemes integrate well with the Identity Mixer family of protocols [9]. Yet, as these topics are not the core interest of this paper, we abstract the necessary mechanisms in the functionality $\mathcal{F}_{\text{A-AUTH}}$ as depicted in Figure 4.

In a nutshell, the functionality allows parties to first register and then "authorize" commitments; the functionality returns "proofs" ψ assuring that the party's identity is contained in a certain position of that commitment. The functionality allows to further bind the proof to a bit string m, which intuitively can be understood as "party P (as referenced in the commitment) signs message m." The exact reason for this mechanism will become clear in the protocol description in Section 6.4.

Our description of $\mathcal{F}_{\text{A-AUTH}}$ is simplistic and tailored to an easy treatment in our proofs. For a complete composable model of anonymous authentication schemes, see e.g. the work of Camenisch, Dubovitskaya, Haralambiev, and Kohlweiss [7].

6 Privacy-preserving auditable UTXO

This section describes the complete protocol. We begin by introducing the core ideas and mechanisms. Section 6.4 then describes multi-input multi-output transactions, followed by Section 6.5 that assembles all pieces and describes the full protocol. Section 6.6 introduces the extension that makes the protocol auditable.

6.1 Core protocol ideas

The protocol represents tokens as commitments $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, P))$ that are stored on $\mathcal{F}_{\text{LEDGER}}$, where v is the value and P is the current owner. Issuers can create new tokens in their own name. Transferring tokens (v, P) to a party R means replacing the commitment to (v, P) with a commitment to (v, R). We now describe the protocol steps in more detail, but still at an informal level.

To issue a token of value v, the issuer generates a new commitment $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, I))$, which means a token with value v is created with owner I. The protocol generates a proof

$$\psi_0 \leftarrow \text{PK}\left\{(r_{\text{cm}}) : \text{open}(crs, cm, r_{\text{cm}}, (v, I)) = \mathsf{true}\right\},$$

which shows that the commitment contains the expected information. Issuer I also creates a signature s on message (v, cm, ψ_0). The information written to $\mathcal{F}_{\text{LEDGER}}$ is $tx = (\texttt{issue}, v, cm, \psi_0, s)$.

A party can transfer a token identified by a commitment cm to a receiver R by generating a new commitment $(cm', r'_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, R))$. She generates a first NIZK ψ_1 showing that cm' contains the correct information and that the receiver is registered, and a second proof ψ_2 of eligibility (i.e. the initiator of the transfer owns cm) using $\mathcal{F}_{\text{A-AUTH}}$. The information written to $\mathcal{F}_{\text{LEDGER}}$ is $tx = (\texttt{transfer}, cm', \psi_1, \psi_2)$. At this point, we cannot yet describe how P proves that (a) cm is a valid commitment on the ledger—we cannot include cm in the transaction as that would hurt privacy—and (b) that P is not double-spending cm. These aspects will be covered in the next steps. Party P also sends the message $(\texttt{token}, cm', r'_{\text{cm}}, v)$ to R *privately*.

So far, we have shown how to transfer a single token from a party P to a receiver R. Following sections show how to **(i)** make sure that only *valid and unspent tokens* are transferred; and **(ii)** support multi-input multi-output transfers.

6.2 Certification via blind signatures

The problem of verification of token validity during transfer is resolved by *certification*. We consider a specific party, called a *certifier* C, which vouches for the validity of the token (v, P) stored as a commitment cm on $\mathcal{F}_{\text{LEDGER}}$ by issuing a signature s on (v, P). In the proof ψ_1, P refers to signature s instead of commitment cm.

A naive implementation of the above scheme would require party P to reveal the content (v, P) of cm to certifier C, so that the latter issues the corresponding signature s. Namely prior to signing, C checks that cm opens to (v, P) and that cm is stored on $\mathcal{F}_{\text{LEDGER}}$. Disclosing the pair (v, P) to C is both undesired and unnecessary. Instead we rely on a *blind signature* protocol, in which C learns only the commitment cm, but not its contents, and *blindly* signs the contents.

(Threshold) blind signature functionality
$\mathcal{F}_{\text{BLINDSIG}}^{(\text{TSIG},\text{commit})}$

Functionality $\mathcal{F}_{\text{BLINDSIG}}$ requires that $sid = ((C_1, \ldots, C_n), t, \ell, sid')$, where $C_1, \ldots, C_n$ are the party identifiers of the signers. It is parametrized by the (deterministic) commit algorithm of the commitment as well as the a threshold signature scheme TSIG = (tkeygen, sign, verify). The functionality keeps an initially empty set $\mathcal{S}$ of signed messages.

- Upon `init` from some C_i, run $(sk_1, \ldots, sk_n, pk_1, \ldots, pk_n, pk) \leftarrow\!\!\$\ \text{tkeygen}(\lambda, n, t, \ell)$, where λ is obtained from the security parameter tape, and store $(sk_1, \ldots, sk_n, pk)$. Output $(\texttt{init}, C_i)$ to $\mathcal{A}$.
- Upon input `pubkey` from party P, return $(\texttt{pubkey}, pk)$.
- On input $(\texttt{request}, crs, r_{\text{cm}}, (m_1, \ldots, m_\ell))$ from party P:
  1. Compute $cm \leftarrow \text{commit}(crs, (m_1, \ldots, m_\ell); r_{\text{cm}})$ and store it internally along with the messages and randomness.
  2. Send delayed output $(\texttt{request}, P, crs, cm)$ to each C_i, $i \in \{1, \ldots, n\}$.
- Upon input $(\texttt{sign}, cm)$ from C_i:
  1. If no record with commitment cm exists, then abort.
  2. If there is a record $((m_1, \ldots, m_\ell), S) \in \mathcal{S}$ with $S \subseteq \{C_1, \ldots, C_n\}$, update the record with $S \leftarrow S \cup \{C_i\}$. Else set $\mathcal{S} \leftarrow \mathcal{S} \cup \{((m_1, \ldots, m_\ell), \{C_i\})\}$.
  3. If $|S| \geq t$, then compute $s \leftarrow \text{sign}(sk, (m_1, \ldots, m_\ell))$ and output s to requestor P.
- Upon input $(\texttt{verify}, pk', (m_1, \ldots, m_\ell), s)$ from P, compute $b \leftarrow \text{verify}(pk', (m_1, \ldots, m_\ell), s)$. If $pk = pk' \wedge b \wedge \big(((m_1, \ldots, m_\ell), S) \notin \mathcal{S} \vee |S| < t\big)$ then output $(\texttt{result}, \mathsf{false})$ to P. Else output $(\texttt{result}, b)$ to P.
- Upon input $(\texttt{seckey}, i)$ from $\mathcal{A}$, if C_i is corrupted, then return sk_i.

Fig. 5. Blind signature functionality, threshold version.

While C learns cm during the protocol, it will not be able to leverage the data on $\mathcal{F}_{\text{LEDGER}}$ to trace when P makes use of the corresponding signature s. More precisely, within ψ_1, party P only proves knowledge of signature s and does not reveal it.

Note that a malicious certifier can essentially generate tokens by providing its signature without checking for existence of the commitment on $\mathcal{F}_{\text{LEDGER}}$. Therefore, in Appendix D, we describe how the certification task can be distributed, so that no single party has to be trusted for verification. Figure 5 shows a threshold blind signature ideal functionality, which we will use in the description of our solution in Section 6.5.

6.3 Serial numbers prevent double-spending

Double-spending prevention is achieved via a scheme that is inspired by Zerocash [3] in that it uses a VRF to compute serial numbers for tokens when they are spent. The VRF key is here, however, bound to a user identity via a signature from the registration authority. On a very high level, the above protocol is extended as follows.

1. Each user P creates a VRF key pair (vsk, vpk). They obtain a signature s_A from registration authority A that binds vsk to their identity P.
2. Each commitment contains an additional value ρ.
3. During transfer, the value ρ is used to derive the serial number $(sn, \pi) \leftarrow \text{eval}(vsk, \rho)$. The transaction stored in $\mathcal{F}_{\text{LEDGER}}$ also contains sn.
4. We cannot store the VRF proof π on $\mathcal{F}_{\text{LEDGER}}$, as it is bound to vpk and would deanonymize P. Therefore, P proves knowledge of signature s_A, which binds vpk to her identity, and proves that $\text{check}(vpk, \rho, sn, \pi) = \mathsf{true}$ through a NIZK proof.

It is important to note that authority A must be trusted for preventing double-spending, since it could easily register two different VRF keys for the same user. It is therefore recommended to implement A in a distributed fashion.

The proof ψ_1 made by P during a transfer of the token (v, P, ρ^{in}) is then

$$\begin{aligned}\psi_1 \leftarrow \text{PK}\big\{\big(r'_{\text{cm}}, s_A, s_C, R, P, \rho^{\text{in}}, \rho^{\text{out}}, \pi, v\big) :\\ \text{verify}(pk_C, (v, P, \rho^{\text{in}}), s_C) \wedge \text{open}(crs, cm', (v, R, \rho^{\text{out}}), r'_{\text{cm}})\\ \wedge\, \text{verify}(pk_A, (P, vpk), s_A) \wedge \text{check}(vpk, \rho^{\text{in}}, sn, \pi)\big\},\end{aligned}$$

which can be parsed as follows: prior to the transfer, P obtains signature s_C on (v, P, ρ^{in}) under pk_C from C. The first condition in the proof statement checks that P knows signature s_C on the triplet (v, P, ρ^{in}). The second condition checks that the new commitment cm' contains the same value v. These two conditions, together with trust in the correctness of C, ensure that the token corresponding to cm' is properly derived from a token existing on $\mathcal{F}_{\text{LEDGER}}$. The third condition checks that the VRF public key vpk indeed belongs to P, and the fourth condition checks that the computation of the serial number sn is correct. These two conditions, together with trust in the correctness of A, prevent token (v, P, ρ^{in}) from being double-spent.

6.4 Multi-input multi-output transactions

Multi-input multi-output transactions allow a sender to transfer tokens contained in multiple commitments at once, and to split the accumulated value into multiple outputs for potentially different receivers. We therefore modify the transaction format to contain multiple inputs and multiple outputs. We also have to extend the NIZK: besides the fact that we have to prove consistency of multiple inputs and multiple outputs, we now have to show that the *sum* of the inputs equals the *sum* of the outputs.

Due to arithmetics in finite algebraic structures, we also have to prove that no wrap-arounds occur. This is achieved, as in previous work, by the use of range proofs. For a given value $\mathsf{max} \in \{1, \ldots, p\}$, the condition is that $0 \leq v \leq \mathsf{max}$ for any value v that appears in an output commitment.

The proof, in more detail, now becomes

$$\begin{aligned}\psi_1 \leftarrow \mathrm{PK}\big\{\big((s_i, v_i^{\mathrm{in}}, \rho_i^{\mathrm{in}}, \pi_i)_{i=1}^m, P, s_A, (R_j, r_{\mathrm{cm}}^j, v_j^{\mathrm{out}}, \rho_j^{\mathrm{out}}, vpk_j, s_A^j)_{j=1}^n\big) :\\ \forall i \in \{1,\ldots,m\} : \mathrm{verify}(pk_C, (v_i^{\mathrm{in}}, P, \rho_i^{\mathrm{in}}), s_i)\\ \wedge \forall j \in \{1,\ldots,n\} : \mathrm{open}(crs, cm_j, (v_j^{\mathrm{out}}, R_j, \rho_j^{\mathrm{out}}), r_{\mathrm{cm}}^j)\\ \wedge \mathrm{verify}(pk_A, (P, vpk), s_A)\\ \wedge \forall j \in \{1,\ldots,n\} : \mathrm{verify}(pk_A, (R_j, vpk_j), s_A^j)\\ \wedge \forall i \in \{1,\ldots,m\} : \mathrm{check}(vpk, \rho_i^{\mathrm{in}}, sn_i, \pi_i)\\ \wedge \sum_{i=1}^m v_i^{\mathrm{in}} = \sum_{j=1}^m v_j^{\mathrm{out}} \wedge \forall j \in \{1,\ldots,n\} : 0 \le v_j^{\mathrm{out}} \le \mathsf{max}\big\}. \quad (1)\end{aligned}$$

The processing of the transaction is analogously modified to check this more complex NIZK. We now argue that the statement proved in the NIZK indeed guarantees the consistency of the system.

The first sub-statement (together with the honesty of C) guarantees that all commitments used as inputs indeed exist in the ledger, and the fact that the commitment is binding further implies that the values $(v_i^{\mathrm{in}}, P, \rho_i^{\mathrm{in}})$ indeed correspond to the expected state of the system. The next sub-statement shows that the output commitments indeed contain the expected values $(v_j^{\mathrm{out}}, R_j, \rho_j^{\mathrm{out}})$. The subsequent two statements ensure that all parties are properly registered on the system, and the statement $\mathrm{check}(vpk, \rho_i^{\mathrm{in}}, sn_i, \pi_i)$ prevents double-spending by showing that the serial numbers are computed correctly.

The final two equations guarantee the global consistency of the system: the summation equation then shows that no tokens have been created or destroyed in this transaction. Finally, the range proof shows that all outputs contain a value in the valid range, which avoids wrap-arounds.

6.5 The protocol

This section describes the protocol sketched in the above sections more formally. The protocol has a bit *registered* $\leftarrow$ false and keeps an initially empty list of commitments. We begin by describing the protocol for a regular user P of the system.

- Upon input `register`, if *registered* is set, then return. Else, retrieve the public keys of A and C from $\mathcal{F}_{\mathrm{REG}}$. Query *crs* from $\mathcal{F}_{\mathrm{CRS}}$. Generate a VRF key pair (vsk, vpk) and send a message $(\mathtt{register}, vpk)$ to A via $\mathcal{F}_{\mathrm{SMT}}$ to obtain a signature s_A on (P, vpk). If all steps succeeded, then set *registered* $\leftarrow$ true send `register` to $\mathcal{F}_{\text{A-AUTH}}$.
- Process pending messages and retrieve new data from $\mathcal{F}_{\mathrm{LEDGER}}$. This is a subroutine called from functions below.
  - For transactions $tx = (\mathtt{issue}, v, cm, \psi_0, s)$ from $\mathcal{F}_{\mathrm{LEDGER}}$, validate ψ_0 by inputting $(\mathtt{verify}, (crs, cm, v, I), \psi_0)$ to $\mathcal{F}_{\mathrm{NIZK}}$ and verify s via $\mathrm{verify}(pk_I, (v, cm, \psi_0), s)$. If both checks succeed, record cm as a valid commitment.
  - For transactions $tx = (\mathtt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$, check the serial numbers $sn_1, \ldots, sn_m$ for uniqueness, validate ψ_1 via $\mathcal{F}_{\mathrm{NIZK}}$ and verify $\psi_{2,1}, \ldots, \psi_{2,n}$ via $(\mathtt{verify}, crs, cm_i, \psi_{2,i}, \mathbf{m})$ to $\mathcal{F}_{\text{A-AUTH}}$ for $\mathbf{m} = ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$. If all checks succeed, then store $cm_1, \ldots, cm_m$ as valid.
  - For each incoming message $(\mathtt{sent}, S, P, m)$ buffered from $\mathcal{F}_{\mathrm{SMT}}$, parse m as $(\mathtt{token}, cm, r_{\mathrm{cm}}, v, \rho)$ and test whether the commitment is correct, i.e. whether it holds that $\mathrm{open}(crs, cm, r_{\mathrm{cm}}, (v, P, \rho)) = \mathsf{true}$. Check whether there is a `transfer` transaction tx that appears in $\mathcal{F}_{\mathrm{LEDGER}}$ and contains cm. If all checks are successful, then input $(\mathtt{request}, r_{\mathrm{cm}}, (v^{\mathrm{in}}, P, \rho^{\mathrm{in}}))$ to $\mathcal{F}_{\mathrm{BLINDSIG}}$ and wait for a response s_{cm}. Store the complete information in the internal list.
- Upon input `read`, if $\neg$*registered* then abort, else first process pending messages. Then return a list of all unspent assets (cm, v) owned by the party.
- Upon input $(\mathtt{issue}, v)$, assuming that *registered*, process pending messages and proceed as follows.
  1. Choose a uniformly random ρ and create a commitment $(cm, r_{\mathrm{cm}}) \leftarrow\!\!\$\ \mathrm{commit}(crs, (v, I, \rho))$.
  2. Compute a proof
  $$\psi_0 \leftarrow \mathrm{PK}\left\{(r_{\mathrm{cm}}, \rho) : \mathrm{open}(crs, cm, r_{\mathrm{cm}}, (v, I, \rho)) = \mathsf{true}\right\},$$
  where I and v are publicly known; this is achieved by sending $(\mathtt{prove}, x, w)$ to $\mathcal{F}_{\mathrm{NIZK}}$, where the statement is $x = (crs, cm, v, P)$ and the witness is $w = (r_{\mathrm{cm}}, \rho)$. Compute a signature $s \leftarrow\!\!\$\ \mathrm{sign}(sk_I, (v, cm, \psi_0))$.

3. Send to $\mathcal{F}_{\text{LEDGER}}$ the input $(\texttt{append}, (\texttt{issue}, v, cm, \psi_0, s))$.
4. Store tuple $(cm, r_{\text{cm}}, v, \rho)$ internally and return $(\texttt{issued}, cm)$.

– Upon input $\big(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\big)$, assuming that *registered*, query $(\texttt{lookup}, R_j)$ to $\mathcal{F}_{\text{REG}}$ for all $j = 1, \ldots, n$ in order to make sure that R_j is registered. Then process pending messages and proceed as follows.
1. If, for any $i \in \{1, \ldots, m\}$, there is no recorded commitment $(cm_i, r_{\text{cm}}^i, v_i^{\text{in}}, P, \rho_i^{\text{in}})$, then abort.
2. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^m v_j^{\text{out}}$ then abort.
3. Choose uniformly random ρ_j^{out} for $j = 1, \ldots, n$, and create commitments $(cm_j, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (v_j^{\text{out}}, R_j, \rho_j^{\text{out}}))$.
4. Compute the serial numbers as $(sn_i, \pi_i) \leftarrow \text{eval}(vsk, \rho_i^{\text{in}})$, for $i = 1, \ldots, m$.
5. Compute proof ψ_1 as in Equation (1).
6. Set $\mathbf{m} \leftarrow ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$ For each $i = 1, \ldots, m$, send $(\texttt{prove}, cm_i^{\text{in}}, r_i, v_i^{\text{in}}, \rho_i^{\text{in}}, \mathbf{m})$ to obtain $\psi_{2,i}$.
7. Send $(\texttt{token}, cm_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ to R_j via $\mathcal{F}_{\text{SMT}}$, for each $1 \leq j \leq m$, and send to $\mathcal{F}_{\text{LEDGER}}$ the input

$$(\texttt{append}, (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)).$$

8. Delete cm_i^{in} from the internal state and return $(\texttt{transferred}, (cm_j)_{j=1}^n)$.

– Upon receiving $(\texttt{sent}, S, P, m)$ from $\mathcal{F}_{\text{SMT}}$, buffer it for later processing. Respond $\texttt{ok}$ to sender S.

The protocol machines for parties C and A are easier to describe. Certifier C checks the validity of a commitment and signs if it finds the commitment in the ledger. In more detail:

1. Upon input $\texttt{init}$ obtain crs from $\mathcal{F}_{\text{CRS}}$ and input $\texttt{init}$ to $\mathcal{F}_{\text{BLINDSIG}}$.
2. Upon receiving $(\texttt{request}, P, crs', cm)$ from $\mathcal{F}_{\text{BLINDSIG}}$, check that $crs = crs'$. Query $\mathcal{F}_{\text{LEDGER}}$ for the entire ledger. For each yet unprocessed transaction tx on $\mathcal{F}_{\text{LEDGER}}$, validate the proofs as described in the party protocol. Check whether cm is marked as a valid commitment.
3. If the above check is successful, send $(\texttt{sign}, cm)$ to $\mathcal{F}_{\text{BLINDSIG}}$.

Certification authority A signs VRF public keys of parties.

1. Upon $\texttt{init}$, generate a key pair $(sk_A, pk_A) \leftarrow\!\!\$\ \text{skeygen}(\lambda)$ for the signature scheme and input $(\texttt{register}, pk_A)$ to $\mathcal{F}_{\text{REG}}$.
2. When activated, input $\texttt{retrieve}$ to $\mathcal{F}_{\text{SMT}}$ to obtain the next message. Let it be m from P. If no message has been signed for P yet, then sign $s_A \leftarrow \text{sign}(sk_A, (P, m))$ and send s_A via $\mathcal{F}_{\text{SMT}}$ back to P.

6.6 Auditing

The auditing capability we implement associates to each user U an auditor AU. Auditor AU has the capabilities to decrypt all transaction information associated to U, such as the transaction outputs that are associated with U, as well as the full transactions issued by U. The set of auditors is denoted by $\mathbf{AU}$.

We formalize the guarantees in a functionality $\mathcal{F}_{\text{ATOKEN}}$ described in the following. Functionality $\mathcal{F}_{\text{ATOKEN}}$ stores a list of registered users and an initially empty map $\mathsf{Records}$. The session identifier is of the form $sid = (A, C, I, \mathbf{AU}, sid')$.

– Upon input $\texttt{init}$ from $P \in \{A, C\} \cup \mathbf{AU}$, output to $\mathcal{A}$ $(\texttt{initialized}, P)$. (This must happen for all before anything else.)
– Inputs $\texttt{register}$, $\texttt{read}$, and $\texttt{issue}$ are treated as in $\mathcal{F}_{\text{TOKEN}}$.
– Upon input $(\texttt{bind}, U, AU)$ from A, where U is a registered user and $AU \in \mathbf{AU}$ is an initialized auditor, and there is not yet a pair (U, AU') with $AU \neq AU' \in \mathbf{AU}$, record (U, AU) and output $(\texttt{bound}, U, AU)$ to $\mathcal{A}$.
– Upon input $\big(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\big)$ from an honest party P, where P and all R_j for $j = 1, \ldots, n$ are registered, proceed as follows.
1. If, for any $i \in \{1, \ldots, m\}$, $\mathsf{Records}[cm_i] = \bot$ then abort, else set $(v_i^{\text{in}}, P_i', st_i) \leftarrow \mathsf{Records}[cm_i]$.
2. If, for any $i \in \{1, \ldots, m\}$, $st_i \neq \texttt{alive}$ or $P_i' \neq P$, then abort.
3. If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^n v_j^{\text{out}}$ then abort.

4. Let L be an empty list. For all $j = 1, \ldots, n$, if R_j or its auditor AU_j are corrupt, then append to L the information $(P, R_j, v_j^{\text{out}})$. If the auditor AU of P is corrupt, include the information for all inputs and all outputs. Output $(\texttt{transfer}, L)$ to $\mathcal{A}$.
5. Receiving from $\mathcal{A}$ a response $(\texttt{transfer}, (cm_j^{\text{out}})_{j=1}^n)$, if $\mathsf{Records}[cm_j^{\text{out}}] \neq \perp$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\mathsf{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{alive})$ for all $j \in \{1, \ldots, n\}$ and set $\mathsf{Records}[cm_i] \leftarrow (v_i^{\text{in}}, P, \texttt{consumed})$ for all $i \in \{1, \ldots, m\}$.
6. Return $(\texttt{transferred}, (cm_j^{\text{out}})_{j=1}^n)$ to P.

– Upon input $(\texttt{audit}, cm)$ from auditor AU, if $\mathsf{Records}[cm] = \perp$ then return $\perp$. Otherwise, set $(v^{\text{in}}, P, st) \leftarrow \mathsf{Records}[cm]$. If P is not audited by AU, then return $\perp$, else return (v, P).

The protocol is adapted as follows. First, each commitment also contains the identity of the previous owner. This is not technically necessary but allows to prove that the auditable information is correct while keeping the description here compact. The binding between the auditor and the user is achieved through a (structure-preserving) signature from A. The auditing functionality is implemented as follows: A party P that executes a transfer encrypts the following information:

– To its own auditor, for each input the value v^{in} and current owner P. For each output the value v_j^{out}, sender P, and receiver R_j.
– For each output to R_j, to the auditor of R_j the value v_j^{out}, sender P, and receiver R_j.

This is achieved by encrypting the information, including the resulting ciphertexts in the transfer, and proving that the encryption is consistent with the information in the commitment.

For concreteness, consider an input described by commitment $cm = \text{commit}(crs, (v^{\text{in}}, P, P', \rho^{\text{in}}); r_{\text{cm}})$. We encrypt current owner $c_1 = \text{enc}(pk_{AU}, P; r_1)$ and value $c_2 = \text{enc}(pk_{AU}, v, r_2)$. Then we generate a NIZK proof:

$$\begin{aligned}\text{PK}\big\{\big(v^{\text{in}}, P, P', \rho^{\text{in}}, s, pk_{AU}, r_1, r_2\big) :\ & \\ \text{verify}(pk_C, (v^{\text{in}}, P, P', \rho^{\text{in}}), s) \wedge \text{verify}(pk_A, (P, pk_{AU}), s_A) & \\ \wedge\, c_1 = \text{enc}(pk_{AU}, P; r_1) \wedge c_2 = \text{enc}(pk_{AU}, v^{\text{in}}, r_2)\big\} &\end{aligned}$$

where pk_C and pk_A are public, and c_1 and c_2 are part of the transaction.

Similarly, for a transfer from P to R and an output commitment $cm = \text{commit}(crs, (v^{\text{out}}, R, P, \rho^{\text{out}}); r_{\text{cm}})$, we encrypt to the auditor (here we use the one of P) the sender $c_1 = \text{enc}(pk_{AU}, P; r_1)$, the receiver $c_2 = \text{enc}(pk_{AU}, R; r_2)$, and the value $c_3 = \text{enc}(pk_{AU}, v^{\text{out}}; r_3)$. We then generate a NIZK proof:

$$\begin{aligned}\text{PK}\big\{\big(v^{\text{out}}, R, P, \rho^{\text{out}}, r_{\text{cm}}, pk_{AU}, s_A\big) :\ & \\ \text{open}(crs, cm, (v^{\text{out}}, R, P, \rho^{\text{out}}), r_{\text{cm}}) \wedge \text{verify}(pk_A, (P, pk_{AU}), s_A) & \\ \wedge\, c_1 = \text{enc}(pk_{AU}, P; r_1) \wedge c_2 = \text{enc}(pk_{AU}, R, r_2) \wedge c_3 = \text{enc}(pk_{AU}, v^{\text{out}}; r_3)\big\} &\end{aligned}$$

with public parameters crs and pk_A, as well as cm, c_1, c_2, and c_3 taken from the transaction.

6.7 Security analysis

This section contains the main result of the paper, namely that the protocol in Section 6.5 instantiates functionality $\mathcal{F}_{\text{TOKEN}}$.

Theorem 1. *Assume that* $\text{COM} = (\text{ccrsgen}, \text{commit}, \text{open})$ *is a commitment scheme that is perfectly hiding and computationally binding. Assume that* $\text{VRF} = (\text{vkeygen}, \text{eval}, \text{check})$ *is a verifiable random function. Then* π_{TOKEN} *realizes* $\mathcal{F}_{\text{TOKEN}}$ *with static corruption. Corruption is malicious for* I *and users, and honest-but-curious for* C*. A is required to be honest, but is inactive during the main protocol phase.*

The restriction that C can only be corrupted in an honest-but-curious model is necessary: Otherwise C can issue signatures on arbitrary commitments, even ones that are not stored in $\mathcal{F}_{\text{LEDGER}}$. The proof is deferred to Appendix B.

Appendix C provides details on how to instantiate the above protocol using well-established primitives that do not require any complex setup assumptions.

7 Implementation and performance

To evaluate the feasibility of our protocol, we implemented a prototype using the primitives described in Appendix C and measured its performance. By design, our prototype is compatible with Hyperledger Fabric and requires minimal changes to be integrated. This section elaborates on the integration effort and measures the overhead incurred by our scheme.

7.1 Hyperledger Fabric

Hyperledger Fabric is a permissioned blockchain system. Hyperledger Fabric entities exchange messages, called *transactions*, over the Hyperledger Fabric network. A transaction is used to introduce either a new smart contract (*chaincode* in Hyperledger Fabric terms) into the system or changes to the state of an already existing chaincode (i.e. *execute*). The first process is called *chaincode instantiation*, whereas the latter process is referred to as *chaincode invocation*. A special type of transactions, *reconfiguration transactions*, is used to introduce changes to the system configuration.

In a Hyperledger Fabric network, we identify three types of participants: **(i)** *clients* who submit transactions to the network in order to instantiate or invoke chaincodes, or to reconfigure the system; **(ii)** *peers* which execute chaincodes, validate transactions and maintain a (consistent) copy of the ledger; and **(iii)** *orderers* which jointly decide the order in which transactions would appear in the ledger.

For the proper operation of the system, each instance of Hyperledger Fabric considers one or more membership service providers (in short, MSPs) that issue long-term identities to parties falling under their authority. These identities allow system entities to securely interact with each other; essentially, MSPs provide the required abstractions to validate identities; namely, to compute and verify signatures. The configuration of valid MSPs is included in the genesis block of each Hyperledger Fabric instance and can be updated via reconfiguration transactions.

Hyperledger Fabric follows an *execute-order-validate* model. Here, chaincodes are speculatively *executed* on one or more peers upon a client request—called *chaincode proposal*—prior to submitting the resulting transaction for ordering. Execution results are signed by the peers that generated them in *chaincode endorsements* and are returned to the client who requested them. Endorsements (i.e. peer signatures) are included in the transaction that the client constructs and sends to the ordering service. The latter *orders* the transactions it receives and outputs a first version of the ledger called *raw ledger*. Raw ledger is provided to the peers of the network upon demand. Upon receiving the raw ledger, peers *validate* the ordered transactions against the *endorsement policy* of their origin chaincodes. An endorsement policy specifies the endorsements that a transaction should carry to be deemed valid. If validation completes successfully, then the transaction is committed to the ledger.

Notice that although there is a separation in Hyperledger Fabric between clients and peers, there still is a communication channel between the two, leveraged by clients to acquire endorsements on the chaincodes they wish to invoke and perform queries on the ledger state. In the following section, we show how to make use of this channel to extend Hyperledger Fabric with our protocol.

7.2 Integration architecture

We first require that each issuer, user and auditor operates a Hyperledger Fabric client. These clients are used to generate an `issue` or `transfer` transactions, submit token certification requests and read from the ledger. Along these lines, we outsource the cryptographic operations required to generate token transactions to a *prover chaincode* in the aim of alleviating the load at the client. This setting assumes that each client possesses a peer that she trusts with the computation of the zero-knowledge proofs and serial numbers. We contend that this is a reasonable assumption especially for Hyperledger Fabric that focuses on enterprise applications.

We also make use of the already-existing communication protocol between the clients and the peers to implement what we call, for convenience, *certifier chaincode*. This is a chaincode that runs only on a selective set of peers chosen at setup time and trusted to jointly certify valid tokens, following the protocol in Appendix D. Each such a peer is endowed with a share of the certification signing key, and whenever invoked, provides its share to the certifier chaincode.

Finally, we leverage the membership service infrastructure of Hyperledger Fabric to grant long-term identities to issuers and users. In particular, we integrate the identity mixer MSP of Hyperledger Fabric with our solution to allow privacy preserving user authentication. When it comes to assigning auditors to users, we use an off-band

channel to bind identity mixer user identities with auditor encryption public keys. In a real implementation, this could be accommodated by an external identity management service, preferably distributed[3].

Notice that our protocol uses the ledger only as a time-stamping service, without any validation functionalities; those are offloaded indirectly to certifiers and auditors. This could be supported in Hyperledger Fabric directly by setting the endorsement policy of the prover chaincode to `any`. We note that we plan to extend our prototype to allow the ledger to also validate token transactions. More concretely, we intend to exploit the fact that Hyperledger Fabric supports pluggable transaction validation [28] that allows chaincodes to specify their own custom validation rules, in our case, the custom validation would consist of executing the verification of the ZK proofs.

7.3 Performance numbers

We installed Hyperledger Fabric client and peer infrastructure with our custom validation process on a MacBook Pro (15-inch, 2016), with 2.7 GHz Intel Core i7, and 16 GB of RAM. We implemented our prototype in golang, as this is the core language of Hyperledger Fabric, and used EC groups in BN256 curves. We instantiated both prover and certifier chaincodes on all the peers in the network, while disseminating the secret shares needed for token certification only to the peers reserved for that purpose. For efficiency reasons, we used Schnorr [38] proofs to implement some of the zero-knowledge proofs; this however comes at the cost of formally losing composable security.

We measured the time required to produce and validate a `transfer` transaction as these operations are the most computationally-heavy. We produced our results using the measurements of 100 runs of each operation.

Our results are shown in Table 1 for transfers with two inputs and two outputs. Although our scheme supports an arbitrary number of inputs and outputs, we opt for this combination as it is the common configuration in existing schemes. We assume that there is one certifier and that the maximum liquidity that can be issued or transferred at anytime is capped at 2^{16}.

In the performance evaluation of transaction generation and validation, we present separately the overhead resulting from (**i)** checking that the input and outputs preserve value and type; (**ii)** hiding the transaction graph, cf. entries token validity and serial numbers; (**iii)** and auditability. Our measurements show that the overall transaction construction time is little less than 2s, whereas transaction validation takes a little less than 3s. Auditability is the most expensive operation as it requires the generation and the verification of multiple proofs of correct encryption under obfuscated public keys; more than 2/3 of the overall computation time. Second comes the operations that hide the transaction graph with proof generation time of almost 0.5s and verification time of roughly 0.7s. This shows that in applications where auditability and full privacy are not a priority, our solution performs relatively-well, less than 158ms for transaction generation and 287ms for its verification. Our performance figures exclude proofs of ownership as the performance of those is outweighed by the Identity Mixer overhead.

While these numbers are not yet favorable to a wide adoption, we would like to stress that the AMCL library underlying our implementation is not optimized. An optimization in the crypto libraries is expected to bring in a speedup of at least one order of magnitude [29]. We also note that the current implementation did not investigate possibilities of parallelization.

We also measured the time it takes to get a token certificate. Table 1 shows that the computation at the user takes around 199ms, whereas the overhead at the certifier is 123ms.

8 Conclusion

We described a privacy-preserving and auditable token management scheme for permissioned blockchains, which is instantiated without complex setup and relies only on falsifiable assumptions. Through the use of structure-preserving primitives, we achieve practical transaction sizes and near-practical computation times that are expected to become practical once an optimized implementation of the underlying schemes is available.

[3] The auditor assignment requires structure preserving signatures, which as of now lack single-round distributed instantiations.

	token certification	
user	$198,90$	
certifier	$123,36$	
	proof generation	**proof validation**
overall computation	$1992,844$	$2885,134$
in-out consistency	$157,43$	$287,21$
token validity	$263,02$	$322,34$
serial number	$208,24$	$452,55$
auditability	$1361,99$	$1823,01$

Table 1. This table shows the performance numbers of token certification and transfer in milliseconds (ms). We note that the transaction size is a little over 63KB; this figure however can be further optimized.

Acknowledgement

This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE.

References

1. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.
2. Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT*, volume 2248 of *LNCS*, pages 566–582. Springer, 2001.
3. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.
4. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptology ePrint Archive*, 2019.
5. Jan Camenisch, Rafik Chaabouni, and abhi shelat. Efficient protocols for set membership and range proofs. In Josef Pieprzyk, editor, *Advances in Cryptology — ASIACRYPT*, volume 5350 of *LNCS*, pages 234–252. Springer, 2008.
6. Jan Camenisch, Manu Drijvers, and Björn Tackmann. Multi-protocol UC and its use for building modular and efficient protocols. Cryptology eprint archive, report 2019/065, January 2019.
7. Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT*, volume 9453 of *LNCS*, pages 262–288. Springer, 2015.
8. Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT*, volume 10032 of *LNCS*, pages 807–840. Springer, 2016.
9. Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.
10. Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burton S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO*, volume 1294 of *LNCS*, pages 410–424. Springer, 1997.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science*. IEEE, 2001.
12. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology eprint archive, report 2000/067, December 2018.
13. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil Vadhan, editor, *Theory of Cryptography*, volume 4392 of *LNCS*, pages 61–85. Springer, 2007.
14. Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT*, volume 2332 of *LNCS*, pages 337–351. Springer, 2002.
15. Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via PVORM. In *ACM CCS*, pages 701–717. ACM, 2017.
16. Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In Serge Vaudenay, editor, *Public Key Cryptography — PKC*, volume 3386 of *LNCS*, pages 416–431. Springer, 2005.
17. Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
18. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018.
19. Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, volume 9603 of *LNCS*, pages 81–98. Springer, 2016.
20. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.
21. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.
22. Shafi Goldwasser, Silvio Micali, and Ron Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.
23. Jens Groth. Efficient fully structure-preserving signatures for large messages. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology — ASIACRYPT*, volume 9452 of *LNCS*, pages 239–259. Springer, 2015.
24. Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM*, 59(3), June 2012.
25. Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel Smart, editor, *Advances in Cryptology — EUROCRYPT*, volume 4965 of *LNCS*, pages 415–432. Springer, 2008.

26. https://github.com/ethereum/wiki/wiki/White Paper.
27. https://www.goquorum.com/.
28. Hyperledger Fabric Maintainers. Hyperledger Fabric pluggable endorsement and validation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html.
29. Vlad Krasnov. Go crypto: bridging the performance gap. https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/, May 2015.
30. Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. `bitcointalk.org`, August 2013.
31. Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 397–411. IEEE, 2013.
32. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.
33. Neha Narula, Willy Vasquez, and Madars Virza. zkledger: Privacy-preserving auditing for distributed ledgers. In *Symposium on Networked Systems Design and Implementation*, pages 65–80. USENIX, 2018.
34. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO*, volume 576 of *LNCS*, pages 129–140. Springer, 1991.
35. Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. Confidential assets. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sata, editors, *Financial Cryptography and Data Security*, volume 10958 of *LNCS*, pages 43–63. Springer, 2018.
36. David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *Proceedings of the Cryptographers Track at the RSA Conference*, volume 9610 of *LNCS*, pages 111–126. Springer, 2016.
37. David Pointcheval and Olivier Sanders. Reassessing security of randomizable signatures. In Nigel Smart, editor, *Topics in Cryptology — CT-RSA*, volume 10808 of *LNCS*, pages 319–338. Springer, 2018.
38. C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan 1991.
39. Alberto Sonnino, Mustafa Al-Bassam, Sherar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. arXiv:1802.07344, August 2018.
40. Nicolas van Saberhagen. CryptoNote v 2.0. `https://cryptonote.org/whitepaper.pdf`, October 2013.

A Functionalities

We describe here in more detail some ideal functionalities that are commonly used and that we therefore omitted from the preliminaries of the paper.

Functionality $\mathcal{F}_{\text{CRS}}^{\text{crsgen}}$

$\mathcal{F}_{\text{CRS}}$ is parametrized by a probabilistic algorithm crsgen. Initially, it sets $crs \leftarrow\!\!\$\ \text{crsgen}(\lambda)$.

1. On input `read` from a party P, return crs to P.

Fig. 6. Common reference string.

A.1 Common reference string

Functionality $\mathcal{F}_{\text{CRS}}$ is parametrized by a CRS generator crsgen, which on input security parameter λ samples a fresh string $crs \leftarrow\!\!\$\ \text{crsgen}(\lambda)$.

Functionality $\mathcal{F}_{\text{NIZK}}^{R}$

$\mathcal{F}_{\text{NIZK}}$ is parametrized by a relation R for which we can efficiently check membership. It keeps an initially empty list L of proven statements and a list L_0 of proofs that do not verify.

1. On input $(\texttt{prove}, y, w)$ from a party P, such that $(y, w) \in R$,[a] send $(\texttt{prove}, y)$ to $\mathcal{A}$.
2. Upon receiving a message $(\texttt{done}, \psi)$ from $\mathcal{A}$, with $\psi \in \{0,1\}^*$, record (y, ψ) in L and send $(\texttt{done}, \psi)$ to P.
3. Upon receiving $(\texttt{verify}, y, \psi)$ from some party P, check whether $(y, \psi) \in L$, then return 1 to P, or whether $(y, \psi) \in L_0$, then return 0 to P. If neither, then output $(\texttt{verify}, y, \psi)$ to $\mathcal{A}$ and wait for receiving answer $(\texttt{witness}, w)$. Check $(y, w) \in R$ and if so, store (y, ψ) in L, else store it in L_0. If (y, ψ) is valid, then output 1 to P, else output 0.

[a] Inputs that do not satisfy the respective relation are ignored.

Fig. 7. Non-interactive zero-knowledge functionality based on the one described by Groth et al. [24].

A.2 Non-interactive zero-knowledge

Our functionality $\mathcal{F}_{\text{NIZK}}$ is adapted from the work of Groth et al. [24], with a few modifications of which most are mainly stylistic. The most relevant difference is that we store a set L_0 of false statements that have been verified; we need this to ensure that a statement that was evaluated as false by one honest party will also be evaluated as false by all other honest parties. Otherwise $\mathcal{F}_{\text{NIZK}}$ has the two expected types of inputs `prove` and `verify`, and the adversary is allowed to delay proof generation unless $\mathcal{F}_{\text{NIZK}}$ is used in the context of responsive environments [8].

Secure message transmission functionality $\mathcal{F}_{\text{SMT}}$

Functionality $\mathcal{F}_{\text{SMT}}$ is for transmitting messages in a secure and *private* manner.

- Upon input $(\texttt{send}, R, m)$ from a party S:
  - If both S and R are honest, provide a private delayed output $(\texttt{sent}, S, R, m)$ to R.
  - If at least one of S and R is corrupt, provide a public delayed output $(\texttt{sent}, S, R, m)$ to $\mathcal{A}$'s queue.

Fig. 8. Secure message transmission functionality.

A.3 Secure message transmission

Functionality $\mathcal{F}_{\text{SMT}}$ models a secure channel between a sender S and a receiver R. In comparison to the functionality introduced by Canetti and Krawczyk [14], however, our functionality additionally provides privacy and hides the parties that are involved in the transmission.

Functionality $\mathcal{F}_{\text{SIG}}^{(\text{skeygen},\text{sign},\text{verify})}$

Functionality $\mathcal{F}_{\text{SIG}}$ requires that $sid = (S, sid')$, where S is the party identifier of the sender. Set $\mathcal{C}$, initially empty, specifies the set of currently corrupted parties. The functionality keeps a set $\mathcal{S}$ of properly signed messages.

0. Upon the first activation from S, run $(sk, pk) \leftarrow\!\!\$\ \text{skeygen}(\lambda)$, where λ is obtained from the security parameter tape, and store (sk, pk).
1. Upon input $\texttt{pubkey}$ from party S, output $(\texttt{pubkey}, pk)$ to S.
2. Upon input $(\texttt{sign}, m)$ from party S with $m \in \{0,1\}^*$, compute $s \leftarrow\!\!\$\ \text{sign}(sk, m)$. Set $\mathcal{S} \leftarrow \mathcal{S} \cup \{m\}$ and output s to S.
3. Upon input $(\texttt{verify}, pk', m', s')$ from party P, compute $b \leftarrow \text{verify}(pk', m', s')$. If $S \notin \mathcal{C} \wedge pk = pk' \wedge b \wedge m' \notin \mathcal{S}$ then output $(\texttt{result}, 0)$ to P. Else output $(\texttt{result}, b)$ to P.
4. Upon input $(\texttt{corrupt}, P)$ from the adversary, set $\mathcal{C} \leftarrow \mathcal{C} \cup \{P\}$. If $P = S$, then additionally output sk to $\mathcal{A}$.

Fig. 9. Signature functionality

A.4 Digital signatures

We use the variant of the signature functionality $\mathcal{F}_{\text{SIG}}$ that was introduced by Camenisch et al. [6]. This version of the functionality is compatible with the modular NIZK proof technique introduced in the same paper.

Functionality $\mathcal{F}_{\text{DKG}}$

$\mathcal{F}_{\text{DKG}}$ is parameterized by a PPT algorithm tkeygen. The session identifier *sid* specifies the total number of certifiers n and the threshold bound t.

- Upon input `init` from a party C_i:
  - If no keys $(sk_1, \ldots, sk_n, pk)$ are stored yet, generate $(sk_1, \ldots, sk_n, pk) \leftarrow\!\!\$\ \text{tkeygen}(\lambda, n, t)$.
  - Return (sk_i, pk) to C_i.

Fig. 10. Distributed key generation functionality

A.5 Distributed key generation

Functionality $\mathcal{F}_{\text{DKG}}$ idealizes a distributed key-generation protocol such as, for discrete-log based schemes, the one of Gennaro et al. [20]. The simplified functionality given in Figure 10 is not directly realizable since it does not model that, e.g., the communication may be delayed or prevented by the adversary. We decided to still use this version to simplify the overall treatment.

B Security proof

This section proves Theorem 1. We use the composition result of [6] to prove this, since we want to prove correctness of the evaluation of the verification algorithm.

Proof. We use the proof technique of Camenisch et al. [6] in instantiating the functionalities $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{SIG}}$, and $\mathcal{F}_{\text{BLINDSIG}}$ in a way that $\mathcal{F}_{\text{NIZK}}$ can call out to $\mathcal{F}_{\text{SIG}}$ and $\mathcal{F}_{\text{BLINDSIG}}$ for the verification of signatures. This has the advantage that the respective clauses in the statement are ideally verified.

We then need to describe a simulator. Simulator $\mathcal{S}$ emulates functionalities $\mathcal{F}_{\text{LEDGER}}$, $\mathcal{F}_{\text{REG}}$, $\mathcal{F}_{\text{A-AUTH}}$, $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{SIG}}$, and $\mathcal{F}_{\text{SMT}}$. To emulate $\mathcal{F}_{\text{LEDGER}}$, $\mathcal{S}$ manages an initially empty internal ledger and allows $\mathcal{A}$ to read it via `retrieve` or append messages as described below. $\mathcal{S}$ initially sets *initialized* $\leftarrow$ false. We start by describing the behavior of $\mathcal{S}$ upon outputs provided by $\mathcal{F}_{\text{LEDGER}}$.

- Upon receiving $(\texttt{initialized}, P)$ for $P \in \{A, C\}$, generate a signature key pair for the respective party and simulate the public key of the respective party being registered at $\mathcal{F}_{\text{REG}}$. After receiving this for both A and C, set *initialized* $\leftarrow$ true.
- Upon receiving $(\texttt{registered}, P)$ from $\mathcal{F}_{\text{TOKEN}}$, mark P as registered and generate output $(\texttt{registered}, P)$ as a message from $\mathcal{F}_{\text{A-AUTH}}$ to $\mathcal{A}$.
- Processing of pending messages (several occasions, see below) for party P: For every record tx marked for delayed processing, proceed as follows.
  - If I is corrupt and $tx = (\texttt{issue}, P', v, cm_0, \psi_0, \psi_2)$, then issue $(\texttt{verify}, y, \psi_0)$ to $\mathcal{A}$ as an output of $\mathcal{F}_{\text{NIZK}}$, with $y = (crs, cm_0, v, P')$, and expect as response a witness w. If $w = (r_{\text{cm}}, \rho^{\text{in}})$ is valid for cm_0, and proof ψ_2 is valid according to the simulated instance of $\mathcal{F}_{\text{A-AUTH}}$, then provide the input $(\texttt{issue}, v, cm_0)$ to $\mathcal{F}_{\text{TOKEN}}$.
  - If $tx = (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$, then issue $(\texttt{verify}, y, \psi_1)$ to $\mathcal{A}$ as an output of $\mathcal{F}_{\text{NIZK}}$, with $y = (pk_C, crs, (cm_j)_{j=1}^n, pk_A, (sn_i)_{i=1}^m)$. Expect as response from adversary $\mathcal{A}$ a witness $w = ((s_i, v_i^{\text{in}}, \rho_i^{\text{in}}, \pi_i)_{i=1}^m, P, s_A, (R_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})_{j=1}^n)$. If w is valid, and $\psi_{2,1}, \ldots, \psi_{2,n}$ are valid according to the simulated instance of $\mathcal{F}_{\text{A-AUTH}}$, and corresponding messages have been sent, then mark tx as valid. Provide a request $(\texttt{transfer}, P, (cm_i)_{i=1}^m, (R_j, v_j^{\text{out}}, cm[out]_j)_{j=1}^n)$ to $\mathcal{F}_{\text{TOKEN}}$.
  - For all valid transactions tx where in the meantime the corresponding message $(\texttt{coin}, cm_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ is sent, input $(\texttt{deliver}, cm_j)$ to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{read?}, P)$ from $\mathcal{F}_{\text{TOKEN}}$, process pending messages and return $(\texttt{read!}, P)$ to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{issue}, v)$ from $\mathcal{F}_{\text{TOKEN}}$, first process pending messages. Then, generate a new all-zero commitment $(cm^*, r_{\text{cm}}^*) \leftarrow\!\!\$\ \text{commit}(crs, (0, 0, 0))$. Next, emulate an output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ for the statement $y = (crs, cm^*, v, P)$ and proceed upon an input $(\texttt{done}, \psi_0^*)$ for the same instance of $\mathcal{F}_{\text{NIZK}}$. Emulate

the proof ψ_2^* as in $\mathcal{F}_{\text{A-AUTH}}$, storing the respective instance as a record. Append $(\texttt{issue}, v, cm^*, \psi_0^*, \psi_2^*)$ to the internal ledger. Input $(\texttt{issue}, cm^*)$ to $\mathcal{F}_{\text{TOKEN}}$.

- Upon receiving $(\texttt{transfer}, m, n, L)$ from $\mathcal{F}_{\text{TOKEN}}$, first process pending messages. For each $i = 1, \ldots, m$, generate a random seruak number sn_i^*. Then proceed as follows for $j = 1, \ldots, n$. If there is no entry for j in L, then generate a commitment $(cm_j^*, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (0, 0, 0))$. If there is an entry $(j, P, R_j, v_j^{\text{out}})$, then generate a random value $\rho_j^{\text{out}} \leftarrow\!\!\$\ \mathcal{M}$ and compute commitment $(cm_j^*, r_{\text{cm}}^j) \leftarrow\!\!\$\ \text{commit}(crs, (v_j^{\text{out}}, R_j, \rho_j^{\text{out}}))$. Next, emulate the output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ for instance $y = (pk_C, crs, (cm_j^*)_{j=1}^n, pk_A, (sn_i^*)_{i=1}^m)$ and record the proof ψ_1^* returned by $\mathcal{A}$. Emulate the proofs $\psi_{2,1}^*, \ldots, \psi_{2,i}^*$ as in $\mathcal{F}_{\text{A-AUTH}}$. Append transaction $(\texttt{transfer}, (sn_i^*, \psi_{2,i})_{i=1}^m, (cm_j^*)_{j=1}^n, \psi_1^*)$ to the internal ledger and emulate transmission of n messages of the same length as $(\texttt{token}, cm_j^*, r_{\text{cm}}^j, v_j, \rho_j^{\text{out}})$ on $\mathcal{F}_{\text{SMT}}$ (i.e., append the length to the internal queue). Respond with $(\texttt{transfer}, (cm_j^*)_{j=1}^n)$ to $\mathcal{F}_{\text{TOKEN}}$. When $\mathcal{A}$ delivers a message on $\mathcal{F}_{\text{SMT}}$, input $(\texttt{deliver}, cm_j^*)$ for the corresponding j to $\mathcal{F}_{\text{TOKEN}}$.
- Upon receiving $(\texttt{transfer}, cm)$, first process pending messages. Record cm in the state of the simulated party C, and proceed as in the above case.
- Upon input $(\texttt{append}, s, P')$ from $\mathcal{A}$ for a corrupt P', append s to the ledger and mark for delayed processing. Return to $\mathcal{A}$.

If $\mathcal{S}$ obtains from $\mathcal{A}$ a query to $\mathcal{F}_{\text{A-AUTH}}$ in the name of a corrupt party P that is marked as registered, then $\mathcal{S}$ internally handles the inputs `prove` and `verify` just like $\mathcal{F}_{\text{A-AUTH}}$. If $\mathcal{A}$ provides an input message x to $\mathcal{F}_{\text{SMT}}$ on behalf of a corrupted party P, then the message is ignored unless it is of the format $x = (\texttt{token}, cm, r_{\text{cm}}, v, \rho)$. If it has the right format, then $\mathcal{S}$ checks whether the corresponding transaction tx exists on $\mathcal{F}_{\text{LEDGER}}$; if it does, then input the respective `deliver` message to $\mathcal{F}_{\text{TOKEN}}$. If such a transaction does not exist, then store the message x for later.

Our goal is now to prove that if the commitment and the VRF are secure, then the ideal and real experiments are indistinguishable. We prove this by describing a sequence of experiments, where EXEC_0 is the real experiments and we transform it step-by-step into the ideal experiment, showing for each adjacent pair of steps that they are indistinguishable. The overall statement then follows via the triangle inequality.

Experiment EXEC_1 is almost the same as EXEC_0 but commitments generated during $(\texttt{issue}, v)$ at an honest party P as well as commitments generated during $(\texttt{transfer}, \ldots)$ at an honest party P, where R is also honest, are replaced by commitments generated via $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (0, 0, 0))$. Functionality $\mathcal{F}_{\text{NIZK}}$ is changed so that it does not actually check the input of honest parties.

Experiments EXEC_0 and EXEC_1 are equivalent since the commitment is perfectly hiding and therefore the distribution of the output to the adversary is unchanged. As all inputs of the honest parties' protocols to $\mathcal{F}_{\text{NIZK}}$ are correct, omitting the checks has no effect.

Experiment EXEC_2 is almost the same as EXEC_1 but serial numbers output by honest parties are replaced by uniformly random values from the same set. Experiments EXEC_2 and EXEC_1 are indistinguishable because of the pseudorandomness of VRF, which is easily proved by reduction. Note that the environment never sees honestly generated proofs.

In the following, we describe the response to environment queries in both EXEC_2 and the ideal experiment and point out the differences. We assume that the state in terms of valid commitments is the same prior to the input, and show when the output to the query is the same and when the state in terms of valid commitments remains consistent. The consistency of the input-output behavior is relatively straightforward to check for most inputs. We focus here on the ones used in `transfer`.

- On input $(\texttt{transfer}, \ldots)$ from an honest user P, if not both of P and all $R_1, \ldots, R_j$ are registered, then the request is ignored in both cases. Also if not all transferable commitments $cm_1, \ldots, cm_i$ exist and are associated to user P, both invocations abort. The protocol π_{TOKEN} then generates new commitments $cm_1', \ldots, cm_j'$ and send them for the proof to $\mathcal{F}_{\text{NIZK}}$, which requests a proof ψ_1 from $\mathcal{A}$. Upon return, π_{TOKEN} generates an additional proof ψ_2 via $\mathcal{F}_{\text{A-AUTH}}$, sends the transaction $(\texttt{transfer}, \ldots)$ to $\mathcal{F}_{\text{LEDGER}}$ and the `token` messages to $\mathcal{F}_{\text{SMT}}$. If any R_j is corrupt, this latter invocation means that $\mathcal{A}$ learns $(r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ as well as the sender P via $\mathcal{F}_{\text{SMT}}$ in addition.
  The functionality $\mathcal{F}_{\text{TOKEN}}$ provides either just m, n—if all $R_1, \ldots, R_j$ are honest—or the values $(j, P, R_j, v_j^{\text{out}})$ for each corrupt R_j. In the first case, $\mathcal{S}$ generates a commitment to all-zero messages and requests the proof ψ_1 from $\mathcal{A}$ via $\mathcal{F}_{\text{NIZK}}$-interaction, in the second case $\mathcal{S}$ has all the data available to perform the same computations as the protocol.

The output distribution is the same since in both cases the commitment is an all-zero commitment and the serial number is uniformly random.

– Processing of pending transactions. For all new (possibly adversarial) transactions on $\mathcal{F}_{\text{LEDGER}}$, the honest parties first attempt to verify the proofs via $\mathcal{F}_{\text{NIZK}}$. For adversarially-generated proofs, the first attempt for each such proof may lead to a message from $\mathcal{F}_{\text{NIZK}}$ requesting the witness from $\mathcal{A}$. The same messages are generated by $\mathcal{S}$, which then records the messages and issues the proper requests to $\mathcal{F}_{\text{TOKEN}}$. (Note that this processing in $\mathcal{F}_{\text{TOKEN}}$ takes place at this point in time, but the timing is indistinguishable from that in the protocol as each honest user input leads to that user processing the pending transactions in the protocol.)
For (passively) corrupt C, if a commitment cm is delivered to the receiver, the simulator learns the receiver's identity and emulates the behavior in the real protocol where C also learns both the commitment and the identity of the receiver.
For adversarial transfers sent to the party via $\mathcal{F}_{\text{SMT}}$, it may mean that the message sent on $\mathcal{F}_{\text{SMT}}$ is not proper (so it is ignored by both π_{TOKEN} and $\mathcal{S}$), or that it parses correctly does not have a corresponding transaction in $\mathcal{F}_{\text{LEDGER}}$ (in the sense that the commitment cm^* in the message does not exist there—then it is also ignored), or that both message and transaction can be found, in which case the view of the party changes when the tokens are found.

The only difference between the two above executions is when the adversary fabricates a transaction in the name of a corrupt party that makes a state transition that is different from the one that is done in $\mathcal{F}_{\text{TOKEN}}$. Let us first consider `issue` transactions, where the statement is $y = (crs, cm^*, v, P')$. When an honest party verifies the proof with $\mathcal{F}_{\text{NIZK}}$, then $\mathcal{A}$ has to provide a proper witness (r^*_{cm}, ρ) such that the commitment opens to $\text{open}(crs, cm^*, (v, P, \rho^*), r^*_{\text{cm}}) = \mathsf{true}$.

Consider a transaction $tx = (\texttt{transfer}, (sn^*_i, \psi_{2,i})^m_{i=1}, (cm^*_j)^m_{j=1}, \psi^*_1)$ input by the adversary. When ψ^*_1 is verified by the honest party, then $\mathcal{A}$ is given the statement $y = (pk_C, crs, (cm^*_j)^n_{j=1}, pk_A, (sn^*_i)^m_{i=1})$ and provides a witness $w = ((s_i, v^{\text{in}}_i, \rho^{\text{in}}_i, \pi_i)^m_{i=1}, P, s_A, (R_j, r^j_{\text{cm}}, v^{\text{out}}_j, \rho^{\text{out}}_j)^n_{j=1})$, which satisfies the PK-statement

$$
\begin{aligned}
\text{PK}\big\{\big((s_i, v^{\text{in}}_i, \rho^{\text{in}}_i, \pi_i)^m_{i=1}, P, s_A, (R_j, r^j_{\text{cm}}, v^{\text{out}}_j, \rho^{\text{out}}_j)^n_{j=1}\big) :\\
\forall i \in \{1,\dots,m\} : \text{verify}(pk_C, (v^{\text{in}}_i, P, \rho^{\text{in}}_i), s_i)\\
\wedge\, \forall j \in \{1,\dots,n\} : \text{open}(crs, cm_j, (v^{\text{out}}_j, R_j, \rho^{\text{out}}_j), r^j_{\text{cm}})\\
\wedge\, \text{verify}(pk_A, (P, vpk), s_A)\\
\wedge\, \forall i \in \{1,\dots,m\} : \text{check}(vpk, \rho^{\text{in}}_i, sn_i, \pi_i)\\
\wedge \sum_{i=1}^{m} v^{\text{in}}_i = \sum_{j=1}^{m} v^{\text{out}}_j \wedge \forall j \in \{1,\dots,n\} : 0 \le v^{\text{out}}_j \le \mathsf{max}\big\}.
\end{aligned}
$$

As $\text{verify}(pk_C, (v^{\text{in}}_j, P, \rho^{\text{in}}_i), s_i)$ are evaluated via $\mathcal{F}_{\text{BLINDSIG}}$, and C checks the correctness of crs, we also know that $s_1, \dots, s_m$ were generated for inputs $(\texttt{request}, crs, r^i_{\text{cm}}, (v^{\text{in}}_i, P, \rho^{\text{in}}_i))$, and the commitment $cm^*_i = \text{commit}(crs, (v^{\text{in}}_i, P, \rho^{\text{in}}_i); r^i_{\text{cm}})$ indeed exists on the ledger. Then either cm^i was created during a previous transaction with the same input $(v^{\text{in}}_i, P, \rho^{\text{in}}_i)$ or we can turn $\mathcal{Z}$ into an adversary that breaks the binding property of COM.

As $\text{verify}(pk_A, (P, vpk), s_A)$ is evaluated via a call to $\mathcal{F}_{\text{SIG}}$, and the correctness of both $\mathcal{F}_{\text{SIG}}$ and the honesty of A implies that vpk is the *unique* VRF public key associated to P. So at this point we know that vpk and ρ^{in}_i are correct. As $\text{check}(vpk, \rho^{\text{in}}_i, sn^*_i, \pi_i) = \mathsf{true}$, either $(sn^*_i, \_) \leftarrow \text{eval}(vsk, \rho^{\text{in}}_i)$ or we can turn $\mathcal{Z}$ into an adversary against the soundness of VRF. This means that sn^*_i is also correct, no double-spending occurred. The last two lines mean that the sum of all output values and the sum of all input values are the same, so the overall value is preserved (and the input provided by the simulator is accepted by $\mathcal{F}_{\text{TOKEN}}$).

The construction has negligible correctness error due to collision of sequence numbers.

C Instantiation

C.1 Pedersen commitments

The commitment scheme is instantiated with Pedersen commitments [34] on multiple values. Consider a group $\mathcal{G}$ and generators $g_0, g_1, \dots, g_\ell \in \mathcal{G}$ such that the relative discrete logarithms between the g_i are not known. A commitment to a vector $(x_1, \dots, x_\ell) \in \{1, \dots, |\mathcal{G}|\}^\ell$ of inputs is computed by choosing a uniformly random

$r \in \{1, \ldots, |\mathcal{G}|\}$ and computing $(cm, r_{\text{cm}}) \leftarrow (g_0^r g_1^{x_1} \cdots g_\ell^{x_\ell}, r)$. Pedersen commitments are perfectly hiding and computationally binding under the discrete-logarithm assumption in group $\mathcal{G}$.

C.2 Pointcheval-Sanders (PS) signatures

We use the signature scheme of Pointcheval and Sanders [36] to implement the blind signature used for token certification. The scheme operates in an asymmetric pairing setting with groups $\mathcal{G}_1$ and $\mathcal{G}_2$ of size p, with target group $\mathcal{G}_{\text{T}}$ and bilinear map $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_{\text{T}}$. Key generation skeygen chooses $\tilde{g} \in \mathcal{G}_2$ and $(x, y_1, \ldots, y_\ell) \in \mathbb{Z}_p^{\ell+1}$ and sets $sk \leftarrow (x, y_1, \ldots, y_\ell)$ and $pk \leftarrow (\tilde{g}, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_\ell}) = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$. A signature $s = \text{sign}(sk, (m_1, \ldots, m_\ell))$ on message vector $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$ is computed as $\text{sign} \leftarrow (h, h^{x + \sum_j y_j m_j})$ with h is randomly-chosen in $\mathcal{G}_1$. Verification of signature $s = (s_1, s_2)$ is performed by checking $s_1 \neq 1_{\mathcal{G}_1}$ and $e(s_1, \tilde{X} \prod_j \tilde{Y}_j^{m_j}) = e(s_2, \tilde{g})$.

PS signatures are CMA under an interactive computational assumption. In follow-up work, Pointcheval and Sanders [37] showed that the scheme can be modified to be secure under a non-interactive assumption, by adding and signing another random element m_0. For simplicity, we use the original version in this instantiation.

C.3 Certification through blind signatures

The functionality $\mathcal{F}_{\text{BLINDSIG}}$ is instantiated by the following protocol π_{BLINDSIG}, which operates in the $\{\mathcal{F}_{\text{NIZK}}, \mathcal{F}_{\text{REG}}, \mathcal{F}_{\text{SMT}}\}$-hybrid model. Let $H_{\mathcal{G}} : \mathcal{G}_1 \to \mathcal{G}_1$ denote a cryptographic hash function modeled as a random oracle.

- Upon input `init`, certifier C generates a new key pair (sk, pk) with $sk = (x, y_1, \ldots, y_\ell)$ and $pk = (\tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$, and sends $(\texttt{register}, pk)$ to $\mathcal{F}_{\text{REG}}$.
- Upon input `pubkey`, P sends $(\texttt{query}, C)$ to $\mathcal{F}_{\text{REG}}$ and outputs the result.
- Upon input $(\texttt{request}, crs, r_{\text{cm}}, (m_1, \ldots, m_\ell))$, proceed as follows.
  1. Pick $z \leftarrow\!\!\$\, \mathbb{Z}_p$ and compute $u \leftarrow g^z$. Compute $cm \leftarrow g_0^{r_{\text{cm}}} \prod_{i=1}^\ell g_i^{m_i}$ and $h \leftarrow H_{\mathcal{G}}(cm)$.
  2. For each $i = 1, \ldots, \ell$, choose $r_i \leftarrow\!\!\$\, \mathbb{Z}_p$, $a_i \leftarrow u^{r_i}$, and $b_i \leftarrow h^{m_i} g^{r_i}$.
  3. Obtain the proof

$$\zeta \leftarrow \text{PK}\Big\{\big((m_i, r_i)_{i=1}^\ell, r_{\text{cm}}\big) : \bigwedge_{i=1}^{\ell} (a_i = u^{r_i} \wedge b_i = h^{m_i} g^{r_i}) \wedge cm = g_0^{r_{\text{cm}}} \prod_{i=1}^{\ell} g_i^{m_i}\Big\}$$

     on input $(\texttt{prove}, y, w)$ at $\mathcal{F}_{\text{NIZK}}$ with $y = (cm, h, u, a_1, \ldots, a_\ell, b_1, \ldots, b_\ell)$ and $w = ((m_i, r_i)_{i=1}^\ell, r_{\text{cm}})$.
  4. Call $\mathcal{F}_{\text{SMT}}$ with $(\texttt{send}, C, (\zeta, crs, cm, u, (a_i, b_i)_{i=1}^\ell))$.
- Upon receiving $(\texttt{sent}, P, (\zeta, crs, cm, u, (a_i, b_i)_{i=1}^\ell))$ from $\mathcal{F}_{\text{SMT}}$, certifier C proceeds as follows:
  1. Verify ζ via $\mathcal{F}_{\text{NIZK}}$ and compute $h \leftarrow H_{\mathcal{G}}(cm)$. If verification fails, input $(\texttt{send}, P, \perp)$ to $\mathcal{F}_{\text{SMT}}$ and stop.
  2. Store $(\zeta, crs, cm, u, (a_i, b_i)_{i=1}^\ell, h)$ internally and output to signer C message $(\texttt{request}, P, crs, cm)$.
- Upon input $(\texttt{sign}, cm)$, signer C proceeds as follows:
  1. If no record for commitment cm is stored, stop.
  2. Compute $\bar{b} \leftarrow h^x \prod_{i=1}^\ell b_i^{y_i} = g^{\bar{r}} h^x \prod_{i=1}^\ell h^{m_i y_i}$ and $\bar{a} \leftarrow \prod_{i=1}^\ell a_i^{y_i} = u^{\bar{r}}$.
  3. Call $\mathcal{F}_{\text{SMT}}$ with $(\texttt{send}, P, (\bar{a}, \bar{b}))$.
- Upon receiving $(\texttt{sent}, C, \bar{m})$ from $\mathcal{F}_{\text{SMT}}$, receiver P proceeds as follows:
  1. If $\bar{m}$ cannot be parsed as $(\bar{a}, \bar{b}) \in \mathcal{G}_1^2$ output $(\texttt{result}, \perp)$ and stop.
  2. Compute $h' \leftarrow \bar{b}\bar{a}^{-1/z}$ and check $e(h, \tilde{X} \prod_{i=1}^\ell \tilde{Y}_i^{m_i}) \stackrel{?}{=} e(h', \tilde{g})$. If the check fails, output $(\texttt{result}, \perp)$ and stop. Else output $(\texttt{result}, (h, h'))$.

Note that in the above description we use a deterministic variant of Pointcheval-Sanders signatures. Namely, generator h is not selected randomly in $\mathcal{G}_1$, rather it is computed as the hash of commitment cm. The reason behind this slight modification is to enable a non-interactive distributed signature (i.e. signers do not need to interact), see Appendix D for further details. It is easy to show that the security of this variant holds in the random-oracle model.

Lemma 1. *Protocol π_{BLINDSIG} realizes $\mathcal{F}_{\text{BLINDSIG}}$ under Assumption 2 of Pointcheval and Sanders [36], given that C is honest-but-curious and $\mathcal{A}$ does not have access to the secret key of C.*

A similar protocol has been provided as part of the Coconut systems by Sonnino, Al-Bassam, Bano, Meiklejohn, and Danezis [39], but the protocol there is slightly less efficient. Furthermore, Appendix D shows how the above token certification can be distributed.

C.4 Groth signatures

We use Groth's structure preserving signatures [23] to bind a user public key to an auditor public key. The signature scheme operates in a pairing setting with groups $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_\mathrm{T}$, and on messages in $\mathcal{G}_1$. Let g and $\tilde{g}$ be random generators of $\mathcal{G}_1$ and $\mathcal{G}_2$ respectively. Key generation skeygen(λ, ℓ) selects a vector $sk = (x, y_1, \ldots, y_{\ell-1}) \leftarrow\!\!\$\ \mathbb{Z}_p^\ell$ and a random generator $h \leftarrow\!\!\$\ \mathcal{G}_1$, and computes $pk \leftarrow (h, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_{\ell-1}) = (h, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_{\ell-1}})$. Signature sign$(sk, (m_1, \ldots, m_\ell))$ selects uniformly at random $r \leftarrow\!\!\$\ \mathbb{Z}_p$, computes $\tilde{a} \leftarrow \tilde{g}^{1/r}$, $b \leftarrow (hg^x)^r$, and $c \leftarrow (h^x m_\ell \prod_{i=1}^{\ell-1} m_i^{y_i})^r$, and sets $s \leftarrow (\tilde{a}, b, c)$. Verification of signature $s = (\tilde{a}, b, c)$ for messages $(m_1, \ldots, m_\ell)$ proceeds by verifying two pairing equations $e(b, \tilde{a}) = e(h, \tilde{g})e(g, \tilde{X})$ as well as

$$e(c, \tilde{a}) = e(h, \tilde{X})e(m_\ell, \tilde{g}) \prod_{i=1}^{\ell-1} e(m_i, \tilde{Y}_i).$$

Dodis-Yampolskiy VRF We use the VRF of Dodis and Yampolskiy [16] that operates in the pairing setting. Key generation vkeygen(λ) chooses a random $sk \leftarrow\!\!\$\ \mathbb{Z}_p$ and sets $pk \leftarrow g^{sk}$. Evaluation eval(sk, x) aborts if $sk + x \notin \mathbb{Z}_p^\times$. It computes output $y \leftarrow e(g, \tilde{g})^{1/(sk+x)} \in \mathcal{G}_\mathrm{T}$ and proof $\pi \leftarrow \tilde{g}^{1/(sk+x)} \in \mathcal{G}_2$. Verification check$(pk, x, y, \pi)$ checks whether $e(g, \pi) = y$ and $e(pk \cdot g^x, \pi) = e(g, \tilde{g})$; if so it outputs $b = 1$.

C.5 Groth-Sahai NIZK

Since all equations we have to verify — for the Pointcheval-Sanders signatures, the Pedersen commitments, the Groth signatures, and the Dodis Yampolskiy VRF — are defined in terms of bilinear groups, we propose to use Groth-Sahai proofs [25] to instantiate $\mathcal{F}_{\text{NIZK}}$ in our solution.

C.6 ElGamal public-key encryption

We use ElGamal encryption [17]. Key generation ekeygen(λ) chooses a uniformly random exponent $sk \leftarrow\!\!\$\ \{1, \ldots, |\mathcal{G}|\}$ and computes $pk \leftarrow g^{sk}$. Encryption enc(pk, m) chooses a uniformly random $r \leftarrow\!\!\$\ \{1, \ldots, |\mathcal{G}|\}$ and computes $c \leftarrow (g^r, pk^r m)$. Decryption dec$(sk, c)$ with $c = (c_1, c_2)$ computes $m \leftarrow c_2 c_1^{-sk}$. The encryption scheme is semantically secure and key private under the Decisional Diffie-Hellman assumption.

C.7 Range proofs

Our protocol requires range proofs to ensure that no field wrap-arounds are exploited to increase the quantity of tokens in a transfer. The range proof we use is based on the work of Camenisch, Chaabouni, and shelat [5], instantiated with Pointcheval-Sanders signatures.

D Distributing certification

The Pointcheval-Sanders signature scheme can be extended into a non-interactive t-out-of-n threshold signature scheme. Consider n signers $C_1, \ldots, C_n$ from which a recipient P collects at least t signature shares that can be combined into a complete signature. We describe the process with a trusted key generation, however, notices that it is straightforward to convert the key generation mechanism into a multiparty computation between the signers (see e.g. [20]). We describe the key generation algorithm tkeygen and the reconstruction algorithm combine. The algorithm to produce a signature share is identical to original signing algorithm (taking secret key share as input instead of the overall secret key). That is, to sign a message $(m_1, \ldots, m_\ell)$, signer C_i calls algorithm $(h, h') \leftarrow\!\!\$\ \text{sign}(sk_i, (m_1, \ldots, m_\ell)))$ with $sk_j = (x_j, y_{1j}, \ldots, y_{\ell j})$. The resulting signature share is a valid Pointcheval-Sanders signature for public key $pk_j = (\tilde{X}_j, \tilde{Y}_{1j}, \ldots, \tilde{Y}_{\ell j})$.

Algorithm tkeygen(λ, n, t, ℓ) computes $(sk_j, pk_j)_{j=1}^n, pk$ as follows:

- Pick $\ell + 1$ random polynomials p_x, $p_{y_1}, \ldots p_{y_\ell}$ of degree $t - 1$ with coefficients from $\mathbb{Z}_p$.
- Compute $\tilde{X} \leftarrow g^{p_x(0)}, \tilde{Y}_1 \leftarrow g^{p_{y_1}(0)}, \ldots, \tilde{Y}_\ell \leftarrow g^{p_{y_\ell}(0)}$.
- Compute all $\tilde{X}_j = g^{x_j}$ and $\tilde{Y}_{ij} \leftarrow g^{y_{ji}}$.

– Set $pk = (\tilde{X}, \tilde{Y}_1 = g^{p_{y_1}(0)}, \ldots, \tilde{Y}_\ell = g^{p_{y_\ell}(0)})$, and $pk_j = (\tilde{X}_1, \tilde{Y}_{01}, \ldots, \tilde{Y}_{\ell 1})$. Set $sk_j = (p_x(j), p_{y_0}(j), \ldots, p_{y_\ell}(j))$ and output $(sk_1, \ldots, sk_n, pk_1, \ldots, pk_n, pk)$.

Algorithm combine, on input $\{(s_i, pk_i)\}_{i \in S}, (m_1, \ldots, m_\ell)$, for a set $S \subseteq \{1, \ldots, n\}$ with $|S| = t$, proceeds as follows.

– Output $\perp$ if not all $\{(s_i, pk_i)\}_{i \in S}$ with $s_i = (h_i, h'_i)$ have the same h and if verify$((\tilde{X}_i, \tilde{Y}_{1i}, \ldots, \tilde{Y}_{\ell i}), (m_1, \ldots, m_\ell), (h, h'_i))$ does not hold for all $i \in S$.
– Compute Lagrange coefficients $\lambda_j = \prod_{i \in S \setminus j} \frac{i}{i-j}$ for all $j \in S$.
– Compute and output $(h, h' = \prod_{j \in S} {h'_j}^{\lambda_j})$.

Protocol π_{BLINDSIG} from Appendix C.3 has to be modified as follows:

– Instead of generating a key locally at C, all signers $C_1, \ldots, C_n$ together use $\mathcal{F}_{\text{DKG}}$ to generate the set of keys. Signer C_1 registers the public key pk at $\mathcal{F}_{\text{REG}}$.
– Requestor P sends the request message to parties $C_1, \ldots, C_n$ until it has collected t signatures that verify. It then uses combine to combine that into a single signature that verifies relatively to pk.

Theorem 2. *Let $n \in \mathbb{N}$ and $t < n$. The above-described variant of the protocol realizes the threshold variant of $\mathcal{F}_{\text{BLINDSIG}}$.*

No further adaptations to the users' protocol beyond the use of the threshold functionality are necessary, as the verification equation for the signatures remains the same.