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 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 approaches exist for adding different levels of privacy to blockchain-based transactions. *Tumblers* combine several transactions of different users and obscure the relation between payers and payees. In *mix-in*-based systems, transactions reference multiple superfluous payers that are however not changed by the transaction and only serve as a cover-up for the actual payer. *Confidential Assets* [31] hide the amounts in a payment but leave the payer-payee relation in the open. Finally, several advanced systems 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 setup assumptions that underpin zkSNARK-based schemes and work with standard assumptions to build our solution. Restricting ourselves to the permissioned setting allows us to leverage a combination of signatures and standard ZK-proofs to achieve our security goals.

1.2 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 outputs spent in this transaction beyond the fact that they are valid and unspent.

⋆ Work done while at IBM Research - Zurich.

Authorization: Users authorize transactions via certificates; i.e., the authorization for spending a token is bound to the user's identity instead of a pseudonym (or address). The authorization 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 [28] and supports multi-input-multi-output transactions. It inherits several ideas from prior work, such as the use of Pedersen commitments from Confidential Assets [31] and the use of serial numbers to prevent double-spending from Zerocash [2]. 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 [15], ElGamal encryption [16], Groth signatures [22], Pedersen commitments [30], and Pointcheval-Sanders signatures [32]. This allows us to use the relatively efficient Groth-Sahai proofs [24] and achieve security under standard assumptions, in the random-oracle model.

1.3 Related work

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

Miers et al. [27] introduced Zerocoin, which allows users to anonymize their bitcoins by converting them into *zerocoins* that rely on Peder 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.

Poelstra et al. [31] protect privacy (in a limited form) by hiding the type 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 [2] is the first fully anonymous decentralized payment scheme. Namely, offers unconditional anonymity, to the extent that users can repudiate their participation in a transaction. Thanks to a combination of hash-based commitments and ZK-SNARKs, 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 [18] 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), enforcing spending limits and tracing tainted coins. 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, i.e. the computational assumptions and the trusted setup.

QuisQuis [17] and Zether [3] 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. 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 [14] 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 [29] 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 transactors (i.e. banks), which is inherently small. 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).

2 Preliminaries

2.1 Notation

We use $\mathsf{sans\text{-}serif}$ fonts to denote special values such as true or false, and $\mathtt{typewriter}$ fonts to denote string constants. All cryptographic algorithms are parametrized by a so-called *security parameter* $\lambda \in \mathbb{N}$ given (sometimes implicitly) to the algorithms. We generally denote the message space of algorithms by $\mathcal{M}$.

To succinctly represent proofs of knowledge, we use the common notation introduced by Camenisch and Stadler [9], namely PK $\{(\textit{witness}) : \textit{statement}\}$ to denote a proof of knowledge of *witness* for *statement*.

2.2 Cryptographic schemes

The section presents cryptographic schemes that will are to build the protocol. We only present them briefly, and provide more information on concrete instantiations later in the paper.

Commitment schemes. A commitment scheme COM consists of three algorithms crsgen, commit, and open. The *common reference string (CRS)* generator crsgen is probabilistic and, on input the security parameter λ, samples a common reference string $crs \leftarrow\$ \mathrm{crsgen}(\lambda)$. 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 an opening r_{cm}. We sometimes also use the notation $cm \leftarrow \mathrm{commit}(crs, (m_1, \ldots, m_\ell); r_{\mathrm{cm}})$ if we want 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 of 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 are valid for the same commitment.

Digital signature schemes. A digital signature scheme SIG consists of three algorithms sigkeygen, sign, and verify. The key generation algorithm $(sk, pk) \leftarrow\$ \mathrm{sigkeygen}(\lambda)$ 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 [21]. It states that the probability for an efficient adversary, given an oracle for producing 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 we included in the appendix.

Threshold signature schemes. A non-interactive threshold signature scheme TSIG consists of four algorithms threshKeygen, sign, combine, and verify. Key generation $(sk_1, \ldots, sk_n, pk_1, \ldots, pk_n, pk) \leftarrow\$ \mathrm{threshKeygen}(\lambda, n, t)$ 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.

Public-key encryption. A public-key encryption scheme PKE consists of three algorithms pkeKeygen, enc, and dec. Key-generation algorithm $(sk, pk) \leftarrow\!\!\$\ \mathrm{pkeKeygen}(\lambda)$ 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 $b \leftarrow \mathrm{dec}(sk, c)$ 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 pkeKeygen. For our work we require semantic security as first defined by Goldwasser and Micali [20]. The scheme must additionally satisfy key privacy as defined by Bellare, Boldyreva, Desai, and Pointcheval [1], which states that, given a ciphertext c, it must be hard to determine the public key under which the ciphertext is encrypted.

Verifiable random functions. A verifiable random function VRF consists of three algorithms vrfKeygen, eval, and check. Key generation $(vsk, vpk) \leftarrow\!\!\$\ \mathrm{vrfKeygen}(\lambda)$ takes as input the security parameter and outputs a pair of private key vsk and public key vpk. Deterministic evaluation $(y, \psi) \leftarrow \mathrm{eval}(vsk, x)$ takes as input secret key vsk and input value x, and produces as output the value y with proof ψ. Deterministic verification $b \leftarrow \mathrm{check}(vpk, x, y, \psi)$ takes as input public key vpk, input x, output y, and proof ψ, 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 proof for a wrong statement. The scheme must satisfy *pseudorandomness* which means that, given only vpk, the output y for a fresh input x is indistinguishable from a random output.

2.3 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 [10, 12, 11].

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. In order to use composition, some additional restrictions on protocols are necessary. In a protocol $\rho^{\phi\to\pi}$, which means that all calls within ρ 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 ρ must be *compliant.* This roughly means that ρ 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 $\rho^{\phi\to\pi}$.

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 $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$, 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 [11]. We say that a protocol π UC-realizes a functionality $\mathcal{F}$ if

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

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 [5] 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.

2.4 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 the appendix.

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 1. In a nutshell, every party can append bit strings to a globally available ledger, and every party can retrieve the current ledger.

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. 1. Ledger functionality.

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.

Non-interactive zero-knowledge. The guarantees provided by non-interactive zero-knowledge proofs of knowledge can be formalized via the functionality $\mathcal{F}_{\text{NIZK}}$ proposed by Groth, Ostrovsky, and Sahai [23]. In a nutshell, the (honest or dishonest) prover inputs statement and witness and obtains a proof, the adversary only learns the statement. The verification operates by inputting statement and proof, and the functionality provides the consistent response.

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 [13], 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.

Registration functionality. 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.

Extended anonymous registration 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 $(\texttt{prove-issue}, crs, cm, r_{\text{cm}}, v, \rho)$ from party P, if $\text{open}(crs, cm, (v, P, \rho), r_{\text{cm}})$ then generate a proof ψ, store the record $(\texttt{issue}, crs, cm, \psi)$ internally and output ψ to P.
- Upon input $(\texttt{prove-transfer}, crs, cm, r_{\text{cm}}, v, \rho, m)$ from party P, if $\text{open}(crs, cm, (v, P, \rho), r_{\text{cm}})$ then generate a proof ψ, store the record $(\texttt{transfer}, crs, cm, \psi, m)$ internally and output ψ to P.
- Upon input $(\texttt{verify-issue}, crs, cm, \tilde{\psi})$ from some P, look up if there is a record $(\texttt{issue}, crs, cm, \tilde{\psi})$ and output success (only) if we did so.
- Upon input $(\texttt{verify-transfer}, crs, cm, \tilde{\psi}, m)$ from some P, look up if there is a record $(\texttt{transfer}, crs, cm, \tilde{\psi}, m)$ and output success (only) if we did so.

Fig. 2. 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 [8]. 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 2.

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 exact reason for this mechanism will become clear in Section 3.1.

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 [6]. We chose this simplified version since this part of the mechanism is not the focus of this work.

3 Warm-up: Simplified protocol

We present in this section a simplified version of the protocol, and add complexity step by step. The goal of this section is to arrive at a still minimal but functional version, which is then extended with further functionality in the subsequent sections.

3.1 First outline

Functionality. The goal of the simplified protocol is the realization of functionality $\mathcal{F}_{\text{S-TOKEN}}$ specified in Figure 3. In a nutshell, the functionality allows for privacy-preserving single-input single-output transactions between users. The functionality has a specific *issuer* I that is allowed to issue new tokens; this can easily be generalized to a set $\mathcal{I}$ of issuers each issuing tokens in their own name. Tokens are always bound to the (identity of the) current owner.

Each party first has to register by sending `register` to $\mathcal{F}_{\text{S-TOKEN}}$. At any later time, a party can send *read* to $\mathcal{F}_{\text{S-TOKEN}}$ to obtain a list of items $(cm, v) \in \{0,1\}^* \times \mathbb{N}$, where cm is an identifier for the token and v is its value. The issuer I can input $(\texttt{issue}, v)$ to mint a new token with value v. The response of the functionality is the identifier cm of the newly minted token. A party that owns a token cm can transfer it to receiver R by inputting $(\texttt{transfer}, cm, R)$ to $\mathcal{F}_{\text{S-TOKEN}}$. The response is the new identifier cm' that can be used by R to address that token.

The adversary learns when new tokens are issued; in particular it learns the value and the issuer. Token transfers between honest users are private: only the fact that a transfer occurs is leaked. If either the sender or the receiver is corrupt, the adversary learns the full details of the transaction.

(Incomplete) protocol. The protocol represents tokens as commitments $(cm, r_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, P))$ that 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

SISO token functionality $\mathcal{F}_{\text{S-TOKEN}}$

Functionality $\mathcal{F}_{\text{S-TOKEN}}$ 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 `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 `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 `read` from a registered party P, issue `read?` to $\mathcal{A}$. Upon receiving response `read!` 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 $(\texttt{transfer}, cm, R)$ from an honest party P, where both P and R are registered, proceed as follows.
  1. If $\mathsf{Records}[cm] = \bot$ then abort, else set $(v, P', st) \leftarrow \mathsf{Records}[cm]$.
  2. If $st \neq \texttt{alive}$ or $P' \neq P$, then abort.
  3. If R and C are honest, output `transfer` to $\mathcal{A}$. If R is honest but C is corrupt, output $(\texttt{transfer}, cm)$ If R is corrupt, output $(\texttt{transfer}, P, R, cm, v)$ to $\mathcal{A}$.
  4. Receiving from $\mathcal{A}$ a response $(\texttt{transfer}, cm')$, if $\mathsf{Records}[cm'] \neq \bot$ then abort, else set $\mathsf{Records}[cm'] \leftarrow (v, R, \texttt{delayed})$ and set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{consumed})$.
  5. Return $(\texttt{transferred}, cm')$ to P.
- Upon receiving an input $(\texttt{transfer}, P_s, R, cm_0, cm_1)$ where P_s is corrupt, proceed analogously to the above. (That is, check whether cm_0 is alive and controlled by P_s, and update the records accordingly.)
- Upon receiving an input $(\texttt{deliver}, cm)$ from $\mathcal{A}$ with $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{delayed})$ for some v, set $\mathsf{Records}[cm] \leftarrow (v, P, \texttt{alive})$.

Fig. 3. SISO token functionality.

name. Transferring tokens (v, P) to a party R means replacing the record with a record (v, R). We now describe all protocol steps in more detail, but still at an informal level.

A party first registers in the system by invoking `register` at its protocol. The protocol then inputs `register` at functionality $\mathcal{F}_{\text{A-AUTH}}$. This step corresponds to registering at an identity provider. The protocol also retrieves the CRS crs used for the commitments from $\mathcal{F}_{\text{CRS}}$.

Issuer I can invoke $(\texttt{issue}, v)$ at its protocol, which leads to the protocol generating 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. It also generates a proof ψ_2 by calling `prove-issue` at $\mathcal{F}_{\text{A-AUTH}}$, which effectively shows that I is authorized to generate this transaction. The information written to $\mathcal{F}_{\text{LEDGER}}$ is $(\texttt{issue}, v, cm, \psi_0, \psi_2)$.

A party can transfer a token identified by a commitment cm to a receiver R by invoking $(\texttt{transfer}, v, R)$. The protocol generates a new commitment $(cm', r'_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, R))$. It generates a NIZK ψ_1 showing that cm' contains the correct information and a proof ψ_2 (via $\mathcal{F}_{\text{A-AUTH}}$) showing that P controls cm. The information written to $\mathcal{F}_{\text{LEDGER}}$ is $(\texttt{transfer}, cm', \psi_1, \psi_2)$. 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 sections. Party P additionally sends the message $(\texttt{token}, cm', r'_{\text{cm}}, v)$ to R via $\mathcal{F}_{\text{SMT}}$.

Furthermore, a party can invoke `read` to obtain the list of tokens a party owns. Each invocation (of `issue`, `transfer`, or `read`) at a party begins with querying $\mathcal{F}_{\text{LEDGER}}$ for new transactions and verifying them, and querying $\mathcal{F}_{\text{SMT}}$ for incoming messages, which are also verified against the information stored on the ledger.

Security. The protocol guarantees privacy since the token data is stored on the ledger in a commitment that is only opened by the sender and the receiver of that transaction, together with zero-knowledge proofs. The proofs obtained via $\mathcal{F}_{\text{A-AUTH}}$ bind the transaction to the owner of a token. The consistency guarantees will only become fully clear in the next subsections, but intuitively it should already be clear that the NIZK ψ_1 in `transfer` will guarantee that the commitments *cm* and *cm'* contain the same token value and issuer.

3.2 Certification via blind signatures

The problem of verification of token validity during transfer will be resolved by *certification.* We consider a specific party, called a *certifier* C, which will vouch for the validity of the token by issuing a signature. We later show in Section 4.2 how the certification task can be distributed, so that no single party has to be trusted for verification.

(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 = (threshKeygen, sign, verify). The functionality keeps an initially empty set $\mathcal{S}$ of signed messages.

- Upon `init` from some C_i, run $(sk_1, \ldots, sk_n, pk_1, \ldots, pk_n, pk) \leftarrow\$ \text{threshKeygen}(\lambda, n, t, \ell)$, 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. 4. Blind signature functionality, threshold version.

Moreover, we use a blind signature protocol that implements the functionality $\mathcal{F}_{\text{BLINDSIG}}$ specified in Figure 4. (We specify the threshold version directly, for the setting described here one uses $\mathcal{F}_{\text{BLINDSIG}}(1)$ with a single certifier C_1. In that case, the threshold signature scheme can be replaced by a simple signature scheme.) The functionality allows all parties to obtain the public key pk of the signature scheme through the `get-pk` call. A party P can subsequently use the call $(\texttt{request}, crs, r_{\text{cm}}, (m_1, \ldots, m_\ell))$ to request a blind signature on the vector $(m_1, \ldots, m_\ell)$ of messages. The functionality provides to the signer C the commitment $cm \leftarrow \text{commit}(crs, (m_1, \ldots, m_\ell); r_{\text{cm}})$. Party C can now check whether *cm* indeed exists in $\mathcal{F}_{\text{LEDGER}}$, and if so, it can input $(\texttt{sign}, cm)$, which in turn leads to a signature s on $(m_1, \ldots, m_\ell)$ being created and output to P. All parties can also use $(\texttt{verify}, pk, s, (m_1, \ldots, m_\ell))$ to verify the signature.

We will provide more information on the protocol that realizes $\mathcal{F}_{\text{BLINDSIG}}$ and the instantiation with a concrete signature scheme later in the paper.

Functionality $\mathcal{F}_{\text{BLINDSIG}}$ is used by a party P during transfer: the user provides as input the contents (v, P) of the commitment *cm* they want to spend, along with the opening r_{cm}, which leads to C learning *cm*. Certifier C then checks whether *cm* is valid (i.e. appears on $\mathcal{F}_{\text{LEDGER}}$), and if so C signs. Thereby, P learns a signature of C on (v, P) and can prove knowledge of this signature to other parties.

3.3 Serial numbers prevent double-spending

Double-spending prevention is achieved via a scheme that is inspired by Zerocash [2] 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 a certification authority. On a very high level, the protocol described in Section 3.2 is extended as follows.

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

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

We first describe the complete protocol $\pi_{\text{S-TOKEN}}$ in the following. We then go on to prove that it realizes $\mathcal{F}_{\text{S-TOKEN}}$ if the commitment is perfectly hiding and computationally binding, and the VRF is secure. The protocol uses functionalities $\mathcal{F}_{\text{BLINDSIG}}$, $\mathcal{F}_{\text{SIG}}$, and $\mathcal{F}_{\text{NIZK}}$.

Complete protocol $\pi_{\text{S-TOKEN}}$. 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 $\mathcal{F}_{\text{REG}}$. Query crs from $\mathcal{F}_{\text{CRS}}$. Generate a VRF key pair (vsk, vpk) and send a message $(\texttt{register}, vpk)$ to A via $\mathcal{F}_{\text{SMT}}$ 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. (This is a subroutine called from functions below.) Retrieve new data from $\mathcal{F}_{\text{LEDGER}}$.
  - For transaction $tx = (\texttt{issue}, v, cm, \psi_0, \psi_2)$ from $\mathcal{F}_{\text{LEDGER}}$, validate ψ_0 by inputting $(\texttt{verify}, (crs, cm, v, I), \psi_0)$ to $\mathcal{F}_{\text{NIZK}}$ and verify ψ_2 by giving $(\texttt{verify-issue}, cm, \psi_2)$ to $\mathcal{F}_{\text{A-AUTH}}$. If both checks succeed, record cm as a valid commitment.
  - For transactions $tx = (\texttt{transfer}, sn, cm, \psi_1, \psi_2)$, check the serial number sn for uniqueness, validate ψ_1 via $\mathcal{F}_{\text{NIZK}}$ and verify ψ_2 via $(\texttt{verify-transfer}, cm, \psi_2, (sn, cm, \psi_1))$ to $\mathcal{F}_{\text{A-AUTH}}$. If all checks succeed, then store cm as valid.
  - For each incoming message $(\texttt{token}, cm, r_{\text{cm}}, v, \rho)$ buffered from $\mathcal{F}_{\text{SMT}}$, test whether the commitment is correct, i.e. whether it holds that $\text{open}(crs, cm, r_{\text{cm}}, (v, P, \rho)) = \mathsf{true}$. Check whether there is a transaction tx that appears in $\mathcal{F}_{\text{LEDGER}}$ with $tx = (\texttt{transfer}, sn, cm, \psi_1, \psi_2)$. If all checks are successful, store the 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 $(\texttt{issue}, v)$, assuming that *registered*, process pending messages and proceed as follows.
  1. Choose uniformly random $\rho \leftarrow\!\!\$\, \mathcal{M}$. Create a commitment $(cm, r_{\text{cm}}) \leftarrow\!\!\$\, \text{commit}(crs, (v, P, \rho))$.
  2. Compute a proof
  $$\psi_0 \leftarrow \text{PK}\left\{(r_{\text{cm}}, \rho) : \text{open}(crs, cm, r_{\text{cm}}, (v, P, \rho)) = \mathsf{true}\right\},$$
  where P and v are publicly known. (This is achieved by sending $(\texttt{prove}, y, w)$ to $\mathcal{F}_{\text{NIZK}}$, where the statement is $y = (crs, cm, v, P)$ and the witness is $w = (r_{\text{cm}}, \rho)$.)
  3. Send $(\texttt{prove-issue}, cm, r_{\text{cm}}, v, \rho)$ to obtain ψ_2.
  4. Send to $\mathcal{F}_{\text{LEDGER}}$ the input $(\texttt{append}, (\texttt{issue}, v, cm, \psi_0, \psi_2))$.
  5. Store tuple $(cm, r_{\text{cm}}, v, \rho)$ internally and return $(\texttt{issued}, cm)$.
– Upon input $(\texttt{transfer}, cm_0, R)$, assuming that *registered*, query $(\texttt{lookup}, R)$ to $\mathcal{F}_{\text{REG}}$ in order to make sure that R is registered. Then process pending messages and proceed as follows.
  1. If there is no record $(cm_0, r^0_{\text{cm}}, v, \rho^{\text{in}})$, then abort.
  2. Choose uniformly random $\rho^{\text{out}} \leftarrow\!\!\$\, \mathcal{M}$. Create a commitment $(cm_1, r^1_{\text{cm}}) \leftarrow\!\!\$\, \text{commit}(crs, (v, R, \rho^{\text{out}}))$.
  3. Compute the serial number as $(sn_0, \pi_0) \leftarrow \text{eval}(vsk, \rho^{\text{in}})$.
  4. Input $(\texttt{request}, r^0_{\text{cm}}, (v, P, \rho^{\text{in}}))$ to $\mathcal{F}_{\text{BLINDSIG}}$, and wait for a response s_C.

5. Compute a proof

$$\psi_1 \leftarrow \mathrm{PK}\big\{\big(r_{\mathrm{cm}}, s_A, s_C, R, P, \rho^{\mathrm{in}}, \rho^{\mathrm{out}}, \pi_0, v\big) : \\ \mathrm{verify}(pk_C, (v, P, \rho^{\mathrm{in}}), s_C) \wedge \mathrm{open}(crs, cm, (v, R, \rho^{\mathrm{out}}), r_{\mathrm{cm}}) \\ \wedge\, \mathrm{verify}(pk_A, (P, vpk), s_A) \wedge \mathrm{check}(vpk, \rho^{\mathrm{in}}, sn_0, \pi_0)\big\}$$

by inputting statement $y = (pk_C, crs, cm, pk_A, sn_0)$ and witness $w = (r_{\mathrm{cm}}, s_A, s_C, R, P, \rho^{\mathrm{in}}, \rho^{\mathrm{out}}, \pi_0, v)$ to $\mathcal{F}_{\text{NIZK}}$.
6. Send $(\texttt{prove-transfer}, cm_0, r^0_{\mathrm{cm}}, v, \rho^{\mathrm{in}}, (sn_0, cm_1, \psi_1))$ to obtain ψ_2.
7. Send $(\texttt{token}, cm_1, r^1_{\mathrm{cm}}, v, \rho^{\mathrm{out}})$ to R via $\mathcal{F}_{\text{SMT}}$ and send input $(\texttt{append}, (\texttt{transfer}, sn_0, cm_1, \psi_1, \psi_2))$ to $\mathcal{F}_{\text{LEDGER}}$.
8. Delete cm_0 from the internal state of the protocol and return output $(\texttt{transferred}, cm_1)$.

– Upon receiving $(\texttt{sent}, S, P, m)$ from $\mathcal{F}_{\text{SMT}}$, buffer it for later processing. Respond 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 init obtain crs from $\mathcal{F}_{\text{CRS}}$ and input 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 init, generate a key pair $(sk_A, pk_A) \leftarrow\$\, \mathrm{sigkeygen}(\lambda)$ for the signature scheme and input $(\texttt{register}, pk_A)$ to $\mathcal{F}_{\text{REG}}$.
2. When activated, input 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 $s_A \leftarrow \mathrm{sign}(sk_A, (P, m))$ and send s_A via $\mathcal{F}_{\text{SMT}}$ back to P.

We use the composition result of [5] to prove this, since we want to prove correctness of the evaluation of the verification algorithm.

Theorem 1. *Assume that* COM = (crsgen, commit, open) *is a commitment scheme that is perfectly hiding and computationally binding. Assume that* VRF = (vrfKeygen, eval, check) *is a verifiable random function. Then* $\pi_{\text{S-TOKEN}}$ *realizes* $\mathcal{F}_{\text{S-TOKEN}}$ *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 $\mathcal{F}_{\text{LEDGER}}$.

Proof. We use the proof technology of Camenisch et al. [5] 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 $\mathcal{F}_{\text{S-TOKEN}}$, 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 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 $\mathcal{F}_{\text{S-TOKEN}}$.
- If $tx = (\texttt{transfer}, sn, cm_1, \psi_1, \psi_2)$, then issue $(\texttt{verify}, y, \psi_1)$ to $\mathcal{A}$ as an output of $\mathcal{F}_{\text{NIZK}}$, with $y = (pk_C, crs, cm_1, pk_A, sn)$. Expect as response from adversary $\mathcal{A}$ a witness $w = (r^1_{\text{cm}}, s_A, s_C, R, P, \rho^{\text{in}}, \rho^{\text{out}}, \pi_0, v)$. If w is valid, and (ψ_2 is valid according to the simulated instance of $\mathcal{F}_{\text{A-AUTH}}$), and a corresponding message $(\texttt{token}, cm_1, r^1_{\text{cm}}, v, \rho^{\text{out}})$ has been sent to R, then provide a request $(\texttt{transfer}, P, R, cm_0, cm_1)$ to $\mathcal{F}_{\text{S-TOKEN}}$.

– Upon receiving `read?` from $\mathcal{F}_{\text{S-TOKEN}}$, process pending messages and return `read!` to $\mathcal{F}_{\text{S-TOKEN}}$.
– Upon receiving $(\texttt{issue}, v)$ from $\mathcal{F}_{\text{S-TOKEN}}$, first process pending messages. Then, generate a new all-zero commitment $(cm^*, r^*_{\text{cm}}) \leftarrow\$ \operatorname{commit}(crs, (0,0,0))$. 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 $\mathcal{F}_{\text{S-TOKEN}}$.
– Upon receiving `transfer` from $\mathcal{F}_{\text{S-TOKEN}}$, first process pending messages. Then generate a random serial number sn^* and a commitment $(cm^*, r^*_{\text{cm}}) \leftarrow\$ \operatorname{commit}(crs, (0,0,0))$. Next, emulate the output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ for instance $y = (pk_C, crs, cm^*, pk_A, sn^*)$ and record the proof ψ^*_1 returned by $\mathcal{A}$. Emulate the proof ψ^*_2 as in $\mathcal{F}_{\text{A-AUTH}}$. Append transaction $(\texttt{append}, sn^*, cm^*, \psi^*_1, \psi^*_2)$ to the internal ledger and emulate transmission of a message of the same length as $(\texttt{token}, cm^*, r^1_{\text{cm}}, v, \rho)$ on $\mathcal{F}_{\text{SMT}}$ (i.e., append the length to the internal queue). Respond with $(\texttt{transfer}, cm^*)$ to $\mathcal{F}_{\text{S-TOKEN}}$. Emulate a private delayed message on $\mathcal{F}_{\text{SMT}}$; when $\mathcal{A}$ delivers this message, input $(\texttt{deliver}, cm^*)$ to $\mathcal{F}_{\text{S-TOKEN}}$.
– 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 receiving $(\texttt{transfer}, P, R, v, cm)$ from $\mathcal{F}_{\text{S-TOKEN}}$, first process all pending messages. Generate at random a value $\rho^{\text{out}} \leftarrow\$ \mathcal{M}$ and a random serial number sn^* and compute $(cm^*, r^*_{\text{cm}}) \leftarrow\$ \operatorname{commit}(crs, (v, R, \rho^{\text{out}}))$. Emulate output $(\texttt{prove}, y)$ from $\mathcal{F}_{\text{NIZK}}$ with $y = (pk_C, crs, cm^*, pk_A, sn^*)$ as above and wait for $\mathcal{A}$ to supply the proof ψ^*_1. Emulate the proof ψ^*_2 as above. Append $(\texttt{append}, sn^*, cm^*, \psi^*_1, \psi^*_2)$ to the internal ledger. If serial number sn^* does not appear on the ledger, then respond with $(\texttt{transfer}, cm^*)$ to $\mathcal{F}_{\text{S-TOKEN}}$. Emulate a public delayed message $(P, R, (\texttt{token}, cm^*, r^*_{\text{cm}}, v, \rho^{\text{out}}))$ on $\mathcal{F}_{\text{SMT}}$; once this message is delivered by $\mathcal{A}$, provide input $(\texttt{deliver}, cm^*)$ to $\mathcal{F}_{\text{S-TOKEN}}$.
– 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-issue`, `prove-transfer`, `verify-issue`, as well as `verify-transfer` 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 provide input $(\texttt{transfer}, P, R, cm_0, cm_1)$ with the respective values from tx to $\mathcal{F}_{\text{S-TOKEN}}$. 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 $(\texttt{transfer}, cm, R)$ at an honest party P, where R is also honest, are replaced by commitments generated via $(cm, r_{\text{cm}}) \leftarrow\$ \operatorname{commit}(crs, (0,0,0))$. 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 $(\texttt{transfer}, cm, R)$ from an honest user P, if not both of P and R are registered, then the request is ignored in both cases. Also if no transferable commitment cm exists and is associated to user P, both invocations abort. The protocol $\pi_{\text{S-TOKEN}}$ then generates a new commitment cm' and sends it for the proof to $\mathcal{F}_{\text{NIZK}}$, which requests a proof ψ_1 from $\mathcal{A}$. Upon return, $\pi_{\text{S-TOKEN}}$ generates an additional proof ψ_2 via $\mathcal{F}_{\text{A-AUTH}}$, sends the transaction $(\texttt{transfer}, sn, cm', \psi_1, \psi_2)$ to $\mathcal{F}_{\text{LEDGER}}$ and the `token` message to $\mathcal{F}_{\text{SMT}}$. If R is corrupt, this latter invocation means that $\mathcal{A}$ learns $(r^1_{\text{cm}}, v, \rho^{\text{out}})$ as well as the sender P via $\mathcal{F}_{\text{SMT}}$ in addition.
The functionality $\mathcal{F}_{\text{S-TOKEN}}$ provides either just `transfer`—if R is honest—or $(\texttt{transfer}, P, R, v)$—if R is corrupt. 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 $\mathcal{F}_{\text{S-TOKEN}}$. (Note that this processing in $\mathcal{F}_{\text{S-TOKEN}}$ 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 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 $\pi_{\text{S-TOKEN}}$ 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 $\mathcal{F}_{\text{S-TOKEN}}$. 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 $tx = (\texttt{transfer}, sn^*, cm^*, \psi^*_1, \psi^*_2)$ input by the adversary. When this is verified by the honest party, then $\mathcal{A}$ is given the statement $y = (pk_C, crs, cm^*, pk_A, sn^*)$ and provides a witness $w = (r_{\text{cm}}, s_A, s_C, R, P, \rho^{\text{in}}, \rho^{\text{out}}, \pi_0, v)$, which satisfies the PK-statement

$$\begin{aligned}\psi_1 \leftarrow \text{PK}\big\{\big(r_{\text{cm}}, s_A, s_C, R, P, \rho^{\text{in}}, \rho^{\text{out}}, \pi_0, v\big) :\\ \text{verify}(pk_C, (v, P, \rho^{\text{in}}), s_C) \wedge \text{open}(crs, cm^*, (v, R, \rho^{\text{out}}), r_{\text{cm}})\\ \wedge \text{verify}(pk_A, (P, vpk), s_A) \wedge \text{check}(vpk, \rho^{\text{in}}, sn^*, \pi_0)\big\}.\end{aligned}$$

As $\text{verify}(pk_C, (v, P, \rho^{\text{in}}), s_C)$ is evaluated via $\mathcal{F}_{\text{BLINDSIG}}$, and C checks the correctness of crs, we also know that s_C was generated for an input $(\texttt{request}, crs, r^*_{\text{cm}}, (v, P, \rho^{\text{in}}))$, and commitment $cm^* = \text{commit}(crs, (v, P, \rho^{\text{in}}); r^*_{\text{cm}})$ indeed exists on the ledger. Then either cm^* was created during a previous transaction with the same input (v, P, ρ^{in}) 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} are correct. As $\text{check}(vpk, \rho^{\text{in}}, sn^*, \pi_0) = \mathsf{true}$, either $(sn^*, \_) \leftarrow \text{eval}(vsk, \rho^{\text{in}})$ or we can turn $\mathcal{Z}$ into an adversary against the soundness of VRF. This means that sn^* is also correct, no double-spending occurred, and we know an opening to the new commitment cm^*, namely $\text{open}(crs, cm^*, (v, R, \rho^{\text{out}}), r_{\text{cm}})$, which means that the state transition is consistent.

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

3.4 Instantiation

We describe briefly the concrete schemes that are used to instantiate the mechanisms described in the previous subsections efficiently.

Pedersen commitments. The commitment scheme is instantiated with Pedersen commitments [30] on multiple values. Consider a group $\mathcal{G}$ and generators $g_0, g_1, \ldots, g_\ell \in \mathcal{G}$ such that the relative discrete logarithms between the g_i are not known. A commitment to a vector $(x_1, \ldots, x_\ell \in \{1, \ldots, |\mathcal{G}|\})$ of inputs is computed by choosing a uniformly random $r \in \{1, \ldots, |\mathcal{G}|\}$ and computing $(cm, r_{\mathrm{cm}}) \leftarrow (g_0^r g_1^{x_1} \cdots g_\ell^{x_\ell}, r)$. Pedersen commitments are perfectly hiding and computationally binding under the discrete-logarithm assumption in group $\mathcal{G}$.

Pointcheval-Sanders (PS) signatures. We use the signature scheme of Pointcheval and Sanders [32] to implement the blind signatures. We modify the signature generation slightly to make it deterministic; security still holds in the random-oracle model. We parametrize the signature algorithm by a collision-resistant function $f : \mathbb{Z}_p^\ell \times \{0,1\}^* \to \{0,1\}^k$ and allow it to take as input an auxiliary string aux. The scheme operates in an asymmetric pairing setting with groups $\mathcal{G}_1$ and $\mathcal{G}_2$ of size p, and target group $\mathcal{G}_\mathrm{T}$ and a bilinear map $e : \mathcal{G}_1 \times \mathcal{G}_2 \to \mathcal{G}_\mathrm{T}$. Key generation sigkeygen chooses $\tilde{g} \in \mathcal{G}_2$ and $(x, y_1, \ldots, y_\ell) \in \mathbb{Z}_p^{\ell+2}$ and sets $sk \leftarrow x, y_1, \ldots, y_\ell)$ and $pk \leftarrow (\tilde{g}, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_\ell}) = (\tilde{g}, X, Y_1, \ldots, Y_\ell)$. A signature $s = \mathrm{sign}(sk, (m_1, \ldots, m_\ell), aux)$ on message vector $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$ is computed as $a \leftarrow f(m_1, \ldots, m_\ell, aux)$ and sign $\leftarrow (h, h^{x+\sum_j y_j m_j})$ with $h \leftarrow H_{\mathcal{G}_1}(a)$. Verification of signature $s = (s_1, s_2)$ is performed by checking $s \neq 1_{\mathcal{G}_1}$ and $e(s_1, X \prod_j Y_l^{m_j}) = e(s_2, \tilde{g})$.

PS signatures are CMA under an interactive computational assumption. In follow-up work, Pointcheval and Sanders [33] showed that the scheme can be modified to be secure under a non-interactive assumption, by adding another random element, which can be instantiated with $m_0 \leftarrow H_{\mathbb{Z}_p}(cm)$ in our case to keep the scheme deterministic. For simplicity, we keep the simpler version in this description.

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.

- Upon input `init`, certifier C generates a key pair (sk, pk) with $sk = (y, y_1, \ldots, y_\ell)$ and $pk = (\tilde{g}, X, Y_1, \ldots, Y_\ell)$, 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 $(\texttt{request}, crs, r_{\mathrm{cm}}, (m_1, \ldots, m_\ell))$, proceed as follows.
  1. Pick $r \leftarrow\!\!\$\, \mathbb{Z}_p$ and compute $V \leftarrow g^r$. Compute $cm \leftarrow g_0^r \prod_{i=1}^{\ell} g_i^{m_i}$ and $Q \leftarrow H_{\mathcal{G}}(cm)$.
  2. For each $i = 1, \ldots, \ell$, choose $r_i \leftarrow\!\!\$\, \mathbb{Z}_p$, $A_i \leftarrow V^{r_i}$, and $B_i \leftarrow Q^{m_i} g^{r_i}$.
  3. Obtain the proof
  $$\zeta \leftarrow \mathrm{PK}\big\{\big((m_i, r_i)_{i=1}^{\ell}, r_{\mathrm{cm}}\big) : \bigwedge_{i=1}^{\ell} (A_i = V^{r_i} \wedge B_i = Q^{m_i} g^{r_i}) \wedge cm = g_0^{r_{\mathrm{cm}}} \prod_{i=1}^{\ell} g_i^{m_i}\}$$
  on input $(\texttt{prove}, y, w)$ at $\mathcal{F}_{\text{NIZK}}$ with $y = (cm, Q, V, A_1, \ldots, A_\ell, B_1, \ldots, B_\ell)$ and $w = ((m_i, r_i)_{i=1}^{\ell}, r_{\mathrm{cm}})$.
  4. Call $\mathcal{F}_{\text{SMT}}$ with $(\texttt{send}, C, (\zeta, crs, cm, V, (A_i, B_i)_{i=1}^{\ell}))$.
- Upon receiving $(\texttt{sent}, P, (\zeta, crs, cm, V, (A_i, B_i)_{i=1}^{\ell}))$ from $\mathcal{F}_{\text{SMT}}$, certifier C proceeds as follows:
  1. Verify ζ via $\mathcal{F}_{\text{NIZK}}$ and compute $Q \leftarrow H_{\mathcal{G}}(cm)$. If verification fails, input $(\texttt{send}, P, \perp)$ to $\mathcal{F}_{\text{SMT}}$ and stop.
  2. Store $(\zeta, crs, cm, V, (A_i, B_i)_{i=1}^{\ell}, Q)$ 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 $m_0 \leftarrow H_{\mathbb{Z}_p}(cm)$, pick $\bar{r} \leftarrow\!\!\$\, \mathbb{Z}_p$.
  3. Compute $\bar{B} \leftarrow g^{\bar{r}} Q^x \prod_{i=1}^{\ell} A_i^{y_i}$ and $\bar{A} \leftarrow V^{\bar{r}} \prod_{i=1}^{\ell} A_i^{y_i}$.
  4. 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 $(\bar{A}, \bar{B}) \in \mathcal{G}^2$ output $(\texttt{result}, \perp)$ and stop.
  2. Compute $Q' \leftarrow \bar{B}\bar{A}^{1/r}$ and check $e(Q, \tilde{X} \prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}) \stackrel{?}{=} e(Q', g)$. If the check fails, output $(\texttt{result}, \perp)$ and stop. Else output $(\texttt{result}, (Q, Q'))$.

Lemma 1. *Protocol π_{BLINDSIG} realizes $\mathcal{F}_{\text{BLINDSIG}}$ under Assumption 2 of Pointcheval and Sanders [32], 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 [34], but the protocol there is slightly less efficient.

Groth signatures. We use Groth's structure preserving signatures [22]. The signature scheme operates in a pairing setting with groups $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_\mathrm{T}$ and g_2 is a generator of $\mathcal{G}_2$. Key generation sigkeygen(λ, ℓ) selects a vector $sk = (x, y_1, \ldots, y_{\ell-1}) \leftarrow\!\!\$\, \mathbb{Z}_p^\ell$ and a random $\tilde{g} \leftarrow\!\!\$\, \mathcal{G}_1$, and computes $pk \leftarrow (\tilde{g}, X, Y_1, \ldots, Y_\ell) = (\tilde{g}, g_2^x, g_2^{y_1}, \ldots, g_2^{y_\ell})$. Signature sign$(sk, (m_1, \ldots, m_\ell))$ selects uniformly at random $r \leftarrow\!\!\$\, \mathbb{Z}_p$, computes $R \leftarrow {g_2^{1/r}}$, $S \leftarrow (\tilde{g} g_1^x)^r$, and $T \leftarrow (\tilde{g}^x m_n \prod_{i=1}^{\ell} m_i^{y_i})^r$, and sets $s \leftarrow (R, S, T)$. Verification of signature $s = (R, S, T)$ for messages $(m_1, \ldots, m_\ell)$ operates by verifying two pairing equations $e(S, R) = e(\tilde{g}, g_2) e(g, X)$ as well as

$$e(T, R) = e(\tilde{g}, X) e(m_n, g_2) \prod_{i=1}^{\ell} e(m_i, Y_i).$$

Dodis-Yampolskiy VRF We use the VRF of Dodis and Yampolskiy [15] that operates in the pairing setting. Key generation vrfKeygen(λ) chooses a random $sk \leftarrow\!\!\$\, \mathbb{Z}_p$ and sets $pk \leftarrow g_1^{sk}$. Evalutation eval(sk, x) aborts if $sk + x \notin \mathbb{Z}_p^\times$. It computes output $y \leftarrow e(g_1, g_2)^{1/(sk+x)} \in \mathcal{G}_\mathrm{T}$ and proof $\pi \leftarrow {g_2^{1/(sk+x)}} \in \mathcal{G}_2$. Verification check$(pk, x, y, \pi)$ checks whether $e(g_1, \pi) = y$ and $e(pk \cdot g_1^x, \pi) = 1$.

Groth-Sahai NIZK We use Groth-Sahai proofs [24] to instantiate $\mathcal{F}_{\text{NIZK}}$ for relations in bilinear groups. Since all equations we have to verify—for the Pointcheval-Sanders signatures, the Pedersen commitment, the Groth signatures, and the Dodis Yampolskiy VRF—are defined in terms of bilinear groups,

4 Privacy-preserving auditable UTXO

In this section we describe three mechanisms in the protocol that are crucial for its use in practice, and that extend the simpler protocol presented in Section 3. We start in Section 4.1 with multi-input multi-output transactions. Section 4.2 shows how the certification mechanism described in Section 3.2 can be thresholdized to protect against misbehaving certifiers. Section 4.3 finally introduces the extension that makes the protocol auditable.

4.1 Multi-input multi-output transactions

Multi-input multi-output transaction 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 arithmetic 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 functionality changes as described in Figure 5. The interface that a user employs to transfer tokens has become more complex: they can specify multiple commitments as input and multiple value/receiver pairs for the outputs.

MIMO protocol. The protocol π_{TOKEN} that realizes $\mathcal{F}_{\text{TOKEN}}$ is based on protocol $\pi_{\text{S-TOKEN}}$. Most parts of the protocol can remain unchanged, only those that deal with generating or processing `transfer` transaction must be adapted.

Upon input $\big(\mathtt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\big)$, assuming that *registered*, query $(\mathtt{lookup}, R_j)$ 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 $(cm_i, r_{\text{cm}}^i, v_i^{\text{in}}, \rho_i^{\text{in}})$, 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 $\rho_j^{\text{out}} \leftarrow\!\!\$\, \mathcal{M}$ 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. Input $(\mathtt{request}, r_{\text{cm}}^i, (v_i^{\text{in}}, P, \rho_i^{\text{in}}))$ to $\mathcal{F}_{\text{BLINDSIG}}$, and wait for a response s_i, for each $i = 1, \ldots, m$.

MIMO 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 `init` from $P \in \{A, C\}$, output to $\mathcal{A}$ ($\texttt{initialized}, P$). (This must happen for both before anything else.)
- Upon input `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 `read` from a registered party P, issue `read?` to $\mathcal{A}$. Upon receiving response `read!` 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 $\big(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\big)$ 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 is corrupt then append to L the information $(P, R_j, v_j^{\text{out}})$. Output $(\texttt{transfer}, L)$ to $\mathcal{A}$.
  5. Receiving from $\mathcal{A}$ a response $(\texttt{transfer}, (cm_j^{\text{out}})_{j=1}^n)$, if $\mathsf{Records}[cm_j^{\text{out}}] \neq \bot$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\mathsf{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{alive})$ 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 receiving an input $(\texttt{transfer}, P_s, (cm_i^{\text{in}})_{i=1}^m, (R_j, v_j^{\text{out}}, cm_j^{\text{out}})_{j=1}^n)$ where P_s is corrupt, proceed analogously to the above. (That is, check whether all cm_i^{in} are alive and controlled by P_s, and whether the sums add up, and update the records accordingly.)

Fig. 5. MIMO token functionality.

6. Compute a proof

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

7. Set $tx^* \leftarrow ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$ For each $i = 1, \ldots, m$, send $(\texttt{prove-transfer}, cm_i^{\text{in}}, r_i, v_i^{\text{in}}, \rho_i^{\text{in}})$ to obtain $\psi_{2,i}$.
8. 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 $j \in \{1, \ldots, n\}$ and send to $\mathcal{F}_{\text{LEDGER}}$ the input
$$(\texttt{append}, (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)).$$
9. Delete cm_i^{in} from the internal state and return $(\texttt{transferred}, (cm_j)_{j=1}^n)$.

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

$(v_i^{\text{in}}, P, \rho_i^{\text{in}})$ indeed correspond to the expected state of the system. The second sub-statement shows that the output commitments indeed contain the expected values $(v_j^{\text{out}}, R_j, \rho_j^{\text{out}})$. The subsequent two statements prevent double-spending by showing that *vpk* is the user's VRF public key, and the serial numbers are computed correctly. This is all analogous to the SISO case.

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

4.2 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. [19]). We describe the key generation algorithm threshKeygen 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 $(m_1, \ldots, m_n)$, signer C_i calls algorithm $(h, h') \leftarrow\!\!\$\, \mathrm{sign}(sk_i, (m_1, \ldots, m_n)))$ with $sk_j = (x_j, y_{0j}, \ldots, y_{\ell j})$. The resulting signature share is a valid Pointcheval-Sanders signature for public key $pk_j = (\tilde{X}_j, \tilde{Y}_{0j}, \ldots, \tilde{Y}_{\ell j})$.

Algorithm threshKeygen(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 $k-1$ with coefficients from $\mathbb{Z}_p$.
- Compute $\tilde{X} \leftarrow g^{p_x(0)}, \tilde{Y}_0 \leftarrow g^{p_{y_0}(0)}, \ldots, \tilde{Y}_\ell \leftarrow g^{p_{y_\ell}(0)}$.
- Compute all $\tilde{X}_j = g^{x_j}$ and $\tilde{Y}_{ij} \leftarrow g^{y_{ji}}$.
- Set $pk = (\tilde{X}, \tilde{Y}_0 = g^{p_{y_0}(0)}, \ldots, \tilde{Y}_\ell = g^{p_{y_\ell}(0)})$, 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 $\mathrm{verify}((\tilde{X}_i, \tilde{Y}_{1i}, \ldots, \tilde{Y}_{\ell i}), (m_1, \ldots, m_\ell), (h, h_i'))$ 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 $(m_0, h, h' = \prod_{j\in S} {h'_j}^{\lambda_j})$.

Protocol π_{BLINDSIG} from Section 3.4 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.

4.3 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 $\mathcal{AU}$. Furthermore, there is a *binder* B whose role is to set up the connection between auditors and binders.

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 Records. The session identifier is of the form $sid = (A, C, I, B, \mathcal{AU}, sid')$.

– Upon input `init` from $P \in \{A, B, C\} \cup \mathcal{AU}$, output to $\mathcal{A}$ $(\texttt{initialized}, P)$. (This must happen for both before anything else.)
– Inputs `register`, `read`, and `issue` are treated as in $\mathcal{F}_{\text{TOKEN}}$.
– Upon input $(\texttt{bind}, U, AU)$ where U is a registered user and $AU \in \mathcal{AU}$ is an initialized auditor, and there is not yet a pair (U, AU') with $AU \neq AU' \in \mathcal{AU}$, record the pair (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 $i \in \{1, \ldots, m\}$, $\mathsf{Records}[cm_i] = \perp$ 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 helpful for proving that the auditable information is correct. The binding between the auditor and the user is achieved through a Groth 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 ciphertext 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 $\wedge c_2 = \text{enc}(pk_{AU}, v, r_2)$. 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_B 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.

4.4 Instantiation

Most of the schemes have already been introduced in Section 3.4. We provide here descriptions of the schemes for range proofs and encryption.

Range proof. The MIMO version of the protocol requires a range proof to ensure that no field wrap-arounds are used and that the number of existing coins is not changed in a transfer. The range proof we use is based on the work of Camenisch, Chaabouni, and shelat [4], instantiated with Pointcheval-Sanders signatures.

ElGamal public-key encryption We use ElGamal encryption [16] encryption. Key generation pkeKeygen(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 under the Decisional Diffie-Hellman assumption.

5 Implementation and performance

We invented a slightly modified variant of the scheme where some instances of the GS protocol were replaced by instances of Schnorr's protocol, for efficiency purposes. The transaction sizes for that protocol variant are described in Table 1. Further details can be found in Appendix B.

Setup			
Parameter generation	$Me_1 + 7e_2 + t_c e_2$		
Audit init	$7e_1 + 1e_2$		
Issue	*Transaction generation*	*Transaction validation*	
		Ledger	*Certification*
Output consistency	$(9e_1 + 2p)n_o$	$3e_2 n_o$	(same as below)
Transfer	*Transaction generation*	*Transaction validation*	
		Ledger	*Certification*
Input validity	$18e_2 n_i$	$(6e_1 + 19e_2 + 2p)n_i$	(per output)
			C:$(48 + t_c + N_c)e_1 + 6t_c e_2 + 2t_c p$
In-out consistency	$(4n_i + 3n_o + 5)e_1$	$(5n_i + 4n_o + 4)e_1$	
Input ownership	$(20e_1 + 10e_2 + 2p)n_i$	$(28e_1 + 10e_2 + 18p)n_i$	
			S: $44e_1$
Auditability	$(136e_1 + 8e_2)n_o$	$(64e_1 + 64p)n_o$	

Table 1. This table shows the computation overhead of each phase in our system, setup, asset issue and asset transfer in terms of number of exponentiations, in $\mathcal{G}_1$, $\mathcal{G}_2$ denoted by e_1, e_2, respectively and pairings in the two groups by p. Our results consider the number of request's inputs n_i, and outputs n_o, the overall N_c and threshold t_c of certifiers, the base used for range proofs M. Finally in certification phase "C" and "S" represent the client and server computation respectively.

6 Conclusion

We described a privacy-preserving and auditable token-management scheme for permissioned blockchains, which can be instantiated without knowledge assumptions. Through the use of structure-preserving primitives, we achieve practical transaction sizes and near-practical computation times that can, however, expected to become practical through the use of optimized implementations for the underlying schemes.

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. 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.
2. 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.
3. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptology ePrint Archive*, 2019.
4. 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.
5. 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.
6. 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.
7. 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.
8. Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.
9. 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.
10. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science*. IEEE, 2001.
11. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology eprint archive, report 2000/067, December 2018.
12. 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.
13. 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.
14. 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.
15. 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.
16. Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
17. Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018.
18. 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.
19. 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.
20. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.
21. 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.
22. 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.
23. Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *Journal of the ACM*, 59(3), June 2012.
24. 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.
25. Hyperledger Fabric Maintainers. Hyperledger Fabric pluggable endorsement and validation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html.
26. Vlad Krasnov. Go crypto: bridging the performance gap. https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/, May 2015.
27. 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.

28. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.
29. 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.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.

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

$\mathcal{F}_{\text{NIZK}}$ 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.

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. [23].

Non-interactive zero-knowledge. Our functionality $\mathcal{F}_{\text{NIZK}}$ is adapted from the work of Groth et al. [23], 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 [7].

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.

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 [13], however, our functionality additionally provides privacy and hides the parties that are involved in the transmission.

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

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 $(sk, pk) \leftarrow\!\$\ \text{sigkeygen}(\lambda)$, where λ is obtained from the security parameter tape, and store (sk, pk).
1. Upon input `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

Digital signatures. We use the variant of the signature functionality $\mathcal{F}_{\text{SIG}}$ that was introduced by Camenisch et al. [5]. 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 threshKeygen. 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 $(sk_1, \ldots, sk_n, pk) \leftarrow\!\$\ \text{threshKeygen}(\lambda, n, t)$.
  - Return (sk_i, pk) to C_i.

Fig. 10. Distributed key generation functionality

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. [19]. 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 Implementation and measurements

Our token management system is currently in the process of being implemented on Hyperledger Fabric as a way of enabling token management on this platform. In this section we elaborate on the integration of the token management system in Hyperledger Fabric. We first start with a short overview of how Hyperledger Fabric operates.

Background. Hyperledger Fabric is a permissioned blockchain system. Hyperledger Fabric entities exchange messages, called *transactions*, over the Hyperledger Fabric network. A transaction can be used to introduce a new smart contract (*chaincode* in Hyperledger Fabric terms) into the system—*chaincode instantiation*—or to introduce changes to the state (i.e., *execute*) of an already instantiated chaincode. The latter process is referred to as *chaincode invocation*. Special types of transactions, *reconfiguration transactions* are used to introduce changes to the system's configuration.

In a Hyperledger Fabric network, we identify three types of participating entities. There are clients that submit transactions to the network in order to instantiate, invoke chaincodes, or to reconfingure the system, peers that actively participate in the chaincode execution process, and maintain a (consistent) copy of the ledger, and a set of orderers constituting the *ordering service* of the system, that jointly decide the order in which transactions appear in the ledger of the system. For the proper operation of the system, Hyperledger Fabric installations consider one or more membership service providers (in short, MSPs) that issue long-term identities to the system entities that fall under their authority. These identities would allow system entities to securely interact with each other. Essentially, MSPs provide the required abstractions to validate identities, compute and verify signatures. The configuration of each MSP considered by a Hyperledger Fabric system is included in the genesis block of its instance and is reconfigurable via reconfiguration transactions.

Hyperledger Fabric follows an *execute-order-validate* model. Here, chaincodes are speculatively *executed* on one or more peers upon a client's request prior to adding the corresponding transaction to the ledger (*ordering*). Client requests for this purpose are called *chaincode proposals*. Execution results are signed by the peers that generated them in *chaincode endorsements* and are returned to the client who requested them. Endorsements are included in the transaction that the client constructs and is sent to the ordering service. The latter *order* the transaction among the rest of the advertised transactions to the network outputting 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, each peer *validates* each transaction that appears in it to ensure that the chaincode execution results included in the transaction have been generated correctly. After validation completes successfully, the transaction is committed, and its speculative execution results are integrated into the ledger's state.

It is easy to see that although there is a separation in Hyperledger Fabric between clients and peers, there is a clear communication channel between the two, such that clients can acquire endorsements on the chaincodes they wish to invoke, or perform queries on the ledger state. In these requests, known as *proposals*, clients provide the chaincode execution arguments, including the name of chaincode that is to be executed. It is also important to note that Hyperledger Fabric supports pluggable transaction validation [25], such that in principle each chaincode can come with its own rules on what constitutes valid transactions that trigger its execution.

Integration architecture. For our prototype implementation we used the architecture depicted in Figure 11.

We first require that each token user, issuer and auditor operates a Hyperledger Fabric client. It is easy to separate the operations of the protocol to the ones that are used by the client side to generate the cryptographic material for requesting a token's issue or transfer, and the ones used by a verifier to perform the cryptographic verification of the requests. In our prototype implementation on Hyperledger Fabric, we integrate all the client operations (i.e., proof generation) into a *prover chaincode*, and we require that each client is in possession of a peer that it trusts and which will run the prover chaincode. We claim that this is a reasonable requirement given that it fits the setup of multiple enterprise level clients of Hyperledger Fabric.

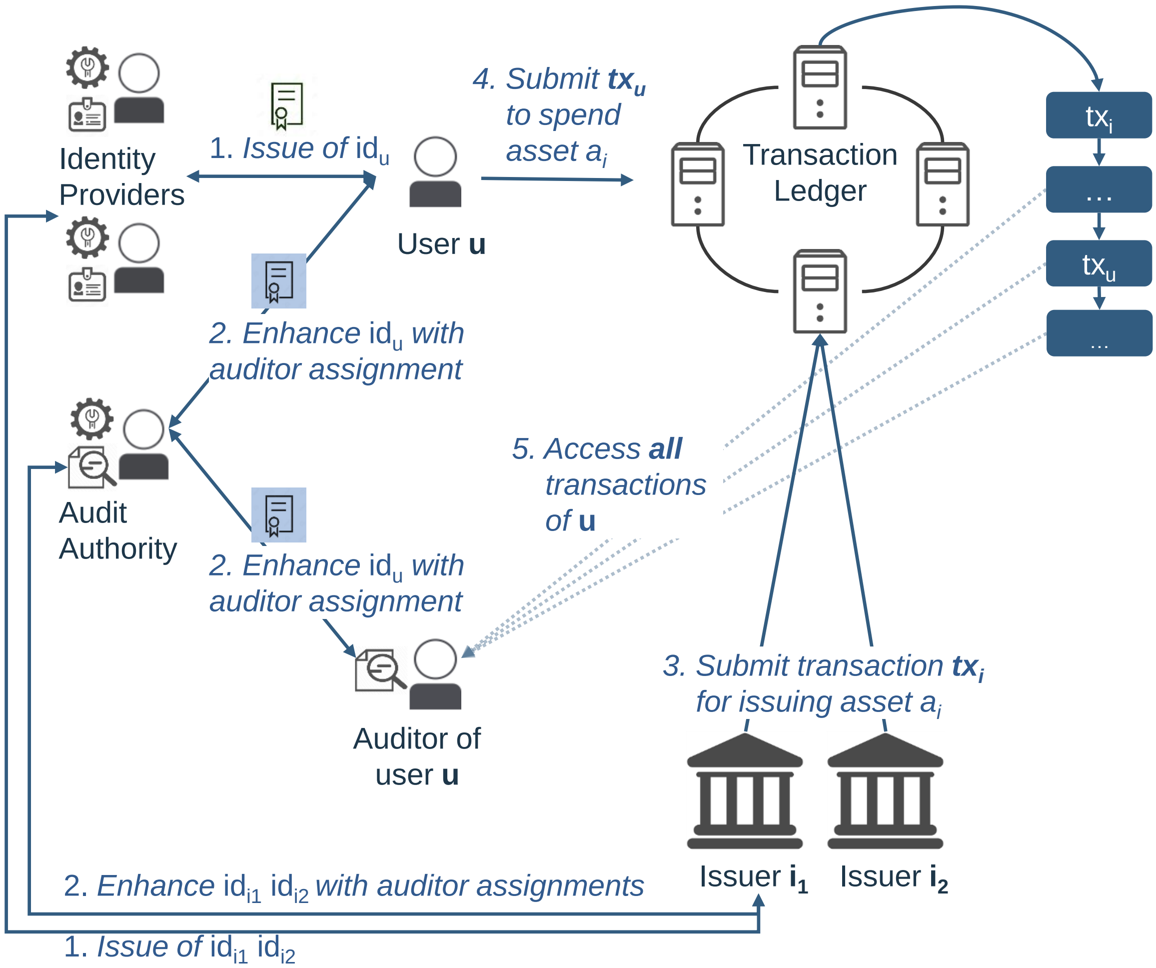

Fig. 11. This figure shows the interactions between the entities in our system. Users and issuers are granted identities and auditor credentials via interactions with the MSPs. Upon a decision to issue a token, one or more token issuers are requested to submit an "issue" request to the transaction ledger of the system. The latter would process the request as long as it authenticates the transaction to have originated from a valid issuer. Users use their credentials to construct transactions to transfer tokens they own to other users, and validation of these transactions take place by the transaction ledger that adds it to the system's immutable ledger. Finally, auditors assigned to a user audit that user's transactions by reading from the ledger.

In our prototype we integrate the verification components into a new validation module that we plug into Hyperledger Fabric. In this way, transactions that refer to the prover chaincode go through our custom validation component that performs changes to the ledger directly. This is an example of how Hyperledger Fabric architecture can be extended to support post-ordering execution based applications.

To accommodate output certification, we leverage the communication protocol between the clients and the peers and an extension to the prover chaincode—we call it for convenience *certifier chaincode*—that is only functional on a selective set of peers. These peers are chosen by the system at setup time as trusted to jointly certify valid outputs that appear in the ledger. At setup, each such peer acquires a share of the output certification signing key; this share is passed to the certifier chaincode, whenever it processes a client request for output certification.

At the same time, we leverage the membership service infrastructure of Hyperledger Fabric to grant identities to the users, since they are also Hyperledger Fabric clients. In particular we leverage the identity mixer MSP feature of Hyperledger Fabric that allows privacy preserving user authentication, using constructions that are compatible with the ones of the token system. For our prototype implementation, we issue audit credentials and assign them to users off-band, assuming that this be accommodated by an offline service of the identity management infrastructure of the system. Audit credentials bind user-identities generated using the Hyperledger Fabric identity mixer MSP to auditors.

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. All our chaincodes and client implementation are in golang as this is the core language used in Hyperledger Fabric, while for a token system instantiation we used EC groups on BN256 curves. We instantiated both prover and certifier chaincodes on all the peers in the network, while we disseminated shares of the output cetification key only to the peers that perform output certification. We measured the time required to produce a token transfer request, and to validate it. Our measurements focused on setup and token transfer as these are the more costly operations in traditional token management systems. We produced our results based the measurements of forty repetitions of each operation.

Our results are shown in Table 2 for a transfer with two inputs and two outputs. Although our schemes support any number of inputs, and outputs in transactions, we chose this combination as it is a common configuration of SoA schemes. In the performance evaluation of transaction generation and validation we present separately the part of theirs that refers to the auditing needs. Our results show an overall transaction construction time of a little more than 1s, whereas transaction validation takes a little less than 2s. While this can be considered poor performance for some use-cases we need to emphasise that the go libraries we use for EC operations are not optimized. An optimisation in the crypto libraries is expected to bring in a speedup of at least one order of magnitude [26]. Notice that auditability operations constitute approximately one third of this time. Output certification consumes 109.07ms on the client side and 116.65ms on the certifier side for each certifier.

	Setup	
Param gen	6.74	
Auditor credential	183.88	
	Transaction generation	*Transaction validation*
Transfer data	631.72	1329.86
Transfer auditability	467.77	400.88
Output certification	$[109.07 \leftrightarrow 116.65]_{t_c}$	117.14

Table 2. This table shows the actual performance of our prototype implementation on Hyperledger Fabric, for token issue and transfer in miliseconds (ms). The $\leftrightarrow$ denotes an interactive process between the client (on the left) and the certifier (on the right), while $[\ldots]_x$ denote a repetition of the interactive round x times. t_c denotes the threshold number of certifiers.