Privacy-preserving auditable token payments in a permissioned blockchain system

Elli Androulaki
IBM Research – Zurich
lli@zurich.ibm.com

Jan Camenisch
DFINITY – Switzerland
jan@dfinity.org

Angelo De Caro
IBM Research – Zurich
adc@zurich.ibm.com

Maria Dubovitskaya
DFINITY – Switzerland
maria@dfinity.org

Kaoutar Elkhiyaoui
IBM Research – Zurich
kao@zurich.ibm.com

Björn Tackmann
DFINITY – Switzerland
bjoern@dfinity.org

ABSTRACT

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

Several more recent blockchain systems, such as Monero or Zerocash, implement improved levels of privacy. Most of these systems target the *permissionless* setting, and as such are not suited for *enterprise* networks. These require token systems to be *permissioned* and to bind tokens to user identities instead of pseudonymous addresses. They also require *auditing* functionalities in order to satisfy regulations such as AML/KYC.

We present a privacy-preserving token management system for permissioned blockchains that also supports fine-grained auditing. The scheme is secure under computational assumptions in bilinear groups, in the random-oracle model. We provide performance measurements for our prototype built on top of Hyperledger Fabric.

ACM Reference Format:
Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. 2020. Privacy-preserving auditable token payments in a permissioned blockchain system. In *2nd ACM Conference on Advances in Financial Technologies (AFT '20), October 21–23, 2020, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3419614.3423259

1 INTRODUCTION

1.1 Motivation

As the blockchain technology matures, public and private companies are exploring the technology to streamline their business [31]. An emerging paradigm to accommodate enterprise use-cases is tokenization [32]. Tokens provide a natural abstraction to describe the lifecycle of an asset in terms of simple operations such as issue and transfer. However, moving to blockchain-based token systems raises the challenge of preserving privacy.

Several approaches already exist to add different levels of privacy to blockchain-based token systems. *Tumblers* such as CoinJoin [35] combine several transfer operations from different users and obscure the relation between senders and receivers. In *mix-in*-based systems such as CryptoNote [44], token transactions reference multiple superfluous senders that do not actually participate in the transaction and only serve as a cover-up for the actual sender. Confidential Assets [40] hide the values in a transfer but leave the sender-receiver relation in the open. Finally, advanced systems such as Zerocash [4] both encrypt the values and fully hide the sender-receiver relation.

All of these solutions are however designed for *permissionless* blockchains. Enterprise networks, on the other hand, favor *permissioned* platforms for their higher throughput and built-in governance [30, 31], with the latter being achieved by a combination of identity management, auditability and non-deniability. Although existing solutions can be tailored for permissioned networks, they are not designed with auditability and accountability in mind. This paper bridges that gap: it hides the content of transactions without preventing authorized parties from inspecting them.

Another goal of this paper is to move away from complex computational assumptions that underpin zkSNARK-based schemes and instead work with more conservative assumptions. This is driven by financial use-cases that have a preference for well-established primitives with standard security assumptions. By restricting ourselves to the permissioned setting we were able to leverage a combination of signatures and standard ZK proofs to achieve this goal.

1.2 Results

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

Privacy: Transactions written on the blockchain conceal the sender-receiver relationship[1] and leak no information about the tokens being spent beyond that they are valid and unspent.

Authorization: Only registered users can transfer tokens. This is achieved by binding tokens to the users' long term identities instead of pseudonyms.

Auditability: Each user has an assigned auditor that sees all the transaction information *related to that particular user.*

[1] We do not consider network-level adversaries. To protect against such adversaries, one could use TOR, for example.

AFT '20, October 21–23, 2020, New York, NY, USA

ACM ISBN 978-1-4503-8139-0/20/10…$15.00
https://doi.org/10.1145/3419614.3423259

Satisfying these three requirements is crucial for implementing a token system that ensures governance, protects user privacy and at the same time complies with regulations such as AML/KYC.

The system we propose is based on the unspent transaction output (UTXO) model pioneered by Bitcoin [37] and supports multi-input-multi-output transactions. It inherits several ideas from prior work, such as the use of Pedersen commitments from Confidential Assets [40] and the use of serial numbers to prevent double-spending from Zerocash [4]. 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 [20], ElGamal encryption [21], Groth signatures [26], Pedersen commitments [39], and Pointcheval-Sanders signatures [41]. This allows us to use the relatively efficient Groth-Sahai proofs [27] and achieve security under standard assumptions, in the random-oracle model.

1.3 Related work

Miers et al. [36] introduced Zerocoin, which helps users anonymize their bitcoins by converting them into *zerocoins* using Pedersen commitments and zero-knowledge proofs. Zerocoins can be changed back to bitcoins without leaking their origin. Zerocoin however does not offer any transacting or auditing capabilites.

Confidential Assets of Poelstra et al. [40] protect privacy (in a limited form) by hiding the types and the values of the traded assets. The idea, similarly to Zerocoin, is to use Pedersen commitments to encode the amount and types of traded assets, and zero-knowledge proofs to show the validity of a transaction. The proposed scheme however does not hide the transaction graph or the public keys of the transactors. While this allows for some form of public auditability, it hinders the privacy of the transacting parties. Pretty Good Confidentiality of Chen et al. [19] follows a similar approach but adds explicit auditing functionality.

Zerocash [4] is the first fully anonymous decentralized payment scheme. It offers unconditional anonymity, to the extent that users can repudiate their participation in a transaction. Thanks to a combination of hash-based commitments and zkSNARKs, Zerocash validates payments and prevents double-spending relatively efficiently. On the downside, Zerocash requires a trusted setup and an expensive transaction generation and its security relies on non-falsifiable assumptions.

Extensions to Zerocash have been introduced in [23] to provide accountability: notably, the proposed solution ensures regulatory closure (i.e. allowing exchanges of assets of the same type only) and enforces spending limits. It also enables the tracing of certain *tainted* tokens, while not really extensively and consistently allowing transactions to be audited. By building on Zerocash, the work in [23] inherits the same limitations regarding computational assumptions and trusted setup.

QuisQuis [22] and Zether [5] propose solutions that provide partial anonymity. On a high level, instead of sending a transaction that refers only to the accounts of the sender and the recipients of a payment, the sender adds accounts of other users, who act as an anonymity set (similar to CryptoNote [44]). 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. Furthermore, both schemes can be extended to provide auditability in a permissioned setting using verifiable public-key encryption. However, their privacy guarantees are weaker than ours: namely, the anonymity set in our solution corresponds to *all system participants*.

Solidus [17] 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 [38] is a permissioned asset transfer scheme that hides transaction amounts as well as the sender-receiver relationship and supports auditing. One main difference with our approach is the end user: zkLedger aims at a setting where the transacting parties are banks, whereas our solution considers the end user to be the client of "a bank." This is why zkLedger enjoys relatively more efficient proofs and could afford a *transaction size that grows linearly with the number of total transactors* in the platform (i.e. banks), which is inherently small. In our scheme, transaction size does *not* grow with the number of overall parties; it only depends on the number of parties involved in the transaction. Concerning auditability, zkLedger offers richer and more flexible semantics but requires the participation of the banks for the audit to take place.

Centralized e-cash. Our solution uses threshold blind signatures to validate tokens and allow them to be transferred. While this shows similarities with centralized e-cash [10, 11, 18, 42], our solution, by leveraging the blockchain, differs in the following fundamental ways: **(i)** double spending detection is performed in real–time by all the nodes in the network; **(ii)** and a receiver of a token is assured of its validity without contacting the issuing bank.

Outline. The remainder of the paper is structured as follows. In Section 2, we provide further background on several important techniques. Section 3 then shows an overview of our protocol. Section 4 describes the types of cryptographic schemes used in the protocol, before Section 5 specifies the security model. Section 6 contains the protocol description and the security statement. In Section 7, we describe the implementation and the performance measurements. Section 8 concludes.

2 BACKGROUND

2.1 Decentralized token systems

A decentralized token transfer is performed by appending a `transfer` transaction to the blockchain. Such a transaction comprises the transfer details (e.g. sender, receivers, type and value) and a proof

that the author of the transaction possesses enough liquidity to perform the transfer. The transaction is then validated against the blockchain state (i.e. the ledger). More precisely, the blockchain checks that the author of the transaction has the right to transfer the token and that the overall quantity of tokens is preserved during the transfer. Existing decentralized token systems can be *account-based* (e.g. [29]) or *unspent transaction outputs (UTXO)-based* (cf. [37]). A valid transfer in an account-based system results in updating the accounts of the sender and the receivers. In a UTXO-based token system, a `transfer` transaction includes a set of inputs (tokens to be consumed) and outputs (tokens to be created). A valid transfer in such systems destroys the inputs and adds the outputs to the ledger to be consumed by subsequent transactions.

2.2 Privacy-preserving token systems

Decentralization of token systems gives rise to serious privacy threats: if transactions contain the transfer information in the clear, then anyone with access to the ledger is able to learn each party's transaction history. We call a decentralized token system *privacy-preserving* if it partially or fully hides the transfer details. Examples of decentralized and privacy-preserving token systems are Confidential Assets [40], Zether [5], QuisQuis [22] and Zerocash [4], with the latter offering the highest level of privacy protection.

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

2.3 Permissioned token systems

In a permissioned token system such as Hyperledger Fabric [1] or Quorum [28], a user is given a long-term credential that reflects her attributes. Tokens are introduced by special users, called *issuers*, through `issue` transactions. These transactions are then validated against predefined policies that reflect existing regulations. For example, issuing policies define which issuers are authorized to create which tokens and under which conditions. Similarly to `issue` transactions, `transfer` are also validated against policies: the simplest of which is that a transfer can take place only between registered users. A fundamental property of permissioned systems is that transactions are signed using long-term credentials. As a by-product, transactions can be traced back to their origin, enforcing thus the requirements of auditability and accountability.

2.4 Signature-based membership proofs

We use signatures to implement zero-knowledge membership proofs [6]. Consider a set $\mathbb{S}$ that consists of elements that are signed using a secret key sk associated with $\mathbb{S}$. Proving knowledge of some element e in $\mathbb{S}$ in zero-knowledge amounts to **(i)** computing a hiding commitment of e; **(ii)** and then proving knowledge of a signature, computed with sk, on e. In this paper, we use this mechanism for two purposes: **(i)** to prove that a user is registered; **(ii)** and to show that a token is one of the *valid* tokens recorded in the ledger.

2.5 Encryption-based auditability

In an encryption-based auditable token system, transactions carry ciphertexts intended for the authorized auditors, see e.g. [10, 11, 23]. For such a mechanism to be viable, it is important to ensure that **(i)** the ciphertexts encrypt the correct information; **(ii)** and they are computed using the correct keys. This is achieved through zero-knowledge proofs—computed by the creator of the transaction—that link the ciphertexts to the transfer details and attest that the two requirements listed above are not violated.

3 OVERVIEW

3.1 Design Approach

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

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

Our solution moves away from zkSNARK and their trusted setup assumption and relies only on standard NIZK proofs (e.g. Groth-Sahai's [27]). More precisely, it leverages the *permissioned setting* to use ZK signature-based membership proofs to ascertain that a user is registered and that a token belongs to the ledger in a privacy-preserving manner. We assume that there is a *registration authority* that provides authorized users with long-term credentials, and a *certifier* that a user contacts with a *certification* request to vouch for the validity of her tokens. A *certification* request contains a token, and upon receiving such a request, the certifier checks whether the token is included in a valid transaction in the ledger. If so, the certifier *blindly* signs the token and the resulting signature is used subsequently to prove the legitimacy of the token.

To prevent double spending, we leverage serial numbers to identify tokens when they are consumed as in Zerocash. It is important that these serial numbers satisfy the following security properties: **(i)** collision resistance: two tokens result in two different serial numbers; **(ii)** determinism: the same token always yields the same serial number; **(iii)** unforgeability: only the owner of the token can produce a valid serial number. We use *verifiable random functions* (e.g. Dodis-Yampolskiy [20]) to generate serial numbers that are a function of the token owner's secret key and a randomness that is tied to the token at its creation time.

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

3.2 Architectural Model

3.2.1 Participants. Our solution involves the following parties:

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

Issuers. They are users who are authorized to introduce tokens in the system through `issue` transactions. For simplicity purposes, an issuer is assumed to introduce *one type* of token only and that the type of a token is defined as the identifier of its issuer.

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

Certifier. This is a privileged *potentially-distributed* party that provides users with *certificates* that vouch for the validity of their tokens. More specifically, a user who wishes to transfer a token contacts the certifier with the token; the certifier in turn inspects whether the token appears in a *valid transaction* in the ledger. If so, the certifier sends back a certificate (i.e. signature) to that effect.

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

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

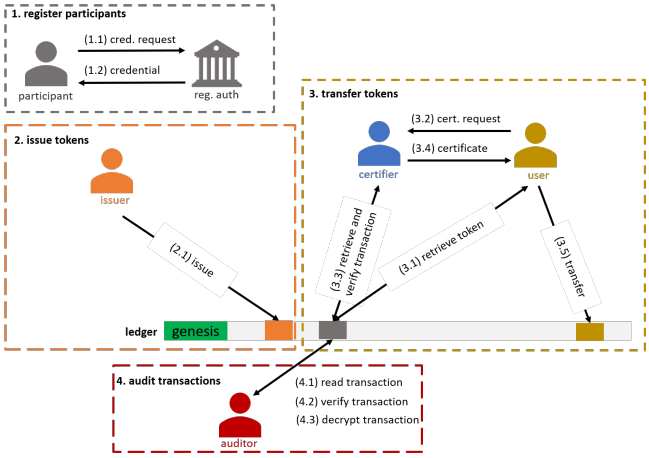

Figure 1: This figure shows the interactions between the participants in our system. Users, issuers, auditors and the certifier are granted credentials by interacting with the registration authority. To create new tokens, an issuer submits an `issue` transaction to the ledger and the ledger automatically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

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

A genesis block is created that announces the system parameters, the public information of the registration authority, the credentials of the certifiers and an *initial list* of authorized issuers. From now on, the system is able to accommodate token management requests. More precisely, issuers submit `issue` transactions to the ledger to introduce new tokens; whereas users spend tokens through `transfer` transactions. For simplicity, we assume that the ledger accepts all incoming transactions without verification. However, anyone with access to the ledger can verify whether a transaction is valid or not using the information in the *genesis* block. To transfer a token, a user contacts the certifier that checks if the transaction including the token is valid. Only then the certifier signs the token making it transferable. To inspect a user's activity, an auditor reads transactions from the ledger, checks their validity and tries to decrypt them with her secret key only if they are valid.

3.3 Trust Model

Registration authority. We assume that the registration authority is trusted to assign *correct credentials* to all parties in the system. A participant presents a set of attributes and her public key to the registration authority and receives in return a credential that binds her attributes to her public key. It is incumbent upon the registration authority to verify the correctness of the attributes of a participant prior to sending the credential. For example, it should verify that the participant knows the secret key underlying the advertised public key. In the case of users, it should also verify that the

announced auditors are legitimate. Furthermore, the registration authority is trusted to assign *one unique credential* per participant. However, it is not *trusted* regarding the privacy of users.

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

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

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

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

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

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

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

4 CRYPTOGRAPHIC SCHEMES

This section introduces the cryptographic schemes that we use to build the protocol. We only present them briefly, and provide more information on concrete instantiations in the full version of the paper [2]. All cryptographic algorithms are parametrized by a so-called *security parameter* $\lambda \in \mathbb{N}$ given (sometimes implicitly) to the algorithms.

4.1 Commitment schemes

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

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

4.2 Digital signature schemes

A digital signature scheme SIG consists of algorithms skeygen, sign, and verify. The key generation algorithm $(sk, pk) \leftarrow\!\!\$$ skeygen(λ) takes as input the security parameter λ and outputs a pair of private (or secret) key sk and public key pk. Signing algorithm $s \leftarrow\!\!\$$ sign(sk, m) takes as input private key sk and message m, and produces a signature s. Deterministic verification algorithm $b \leftarrow$ 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 [25]. It states that given an oracle that generates valid signatures, it is infeasible for an efficient adversary to output a valid signature on a message that has not been queried to the oracle.

4.3 Threshold signature schemes

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

4.4 Public-key encryption.

A public-key encryption scheme PKE consists of algorithms ekeygen, enc, and dec. Key-generation algorithm $(sk, pk) \leftarrow\!\!\$$ ekeygen(λ) takes as input security parameter λ and outputs a pair of private key sk and public key pk. Probabilistic encryption algorithm $c \leftarrow\!\!\$$ enc(pk, m) takes as input message m and public key pk and produces ciphertext c. We also write $c \leftarrow$ enc$(pk, m; r)$ to emphasize that the encryption uses randomness r. Deterministic decryption $m \leftarrow$ dec(sk, c) takes as input ciphertext c and private key sk and recovers message m. Correctness requires that dec$(sk,$ enc$(pk, m)) = m$ for all (sk, pk) generated by ekeygen. For our work, we require semantic security as first defined by Goldwasser and Micali [24]. The scheme must additionally satisfy key privacy as defined by Bellare et al. [3], which ensures that, given a ciphertext c, it is hard to determine the public key under which the ciphertext was computed.

4.5 Verifiable random functions

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

The scheme satisfies *correctness* if honest proofs are always accepted. *Soundness* means that it is infeasible to produce a valid proof for an output value that was not correctly generated. The scheme must satisfy *pseudo-randomness* which means that, given only vpk, the output y for a fresh input x is indistinguishable from a random output.

4.6 Non-interactive zero-knowledge proofs of knowledge

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

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

In the remainder of the paper, we succinctly represent zero knowledge proofs of knowledge using the common notation introduced by Camenisch and Stadler [12], namely $\text{PK}\{(w) : x\}$ denotes a proof of knowledge of *witness* w for *statement* x.

5 SECURITY FORMALIZATION

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

5.1 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 [13–15].

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

In summary, a protocol execution involves the following types of ITIs: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, instances of the protocol machines π, and (possibly) further ITIs invoked by $\mathcal{A}$ or any instance of π (or their subroutines). The contents of the environment's output tape after the execution is denoted by the random variable $\text{EXEC}_{\pi, \mathcal{A}, \mathcal{Z}}(\lambda, z)$, where $\lambda \in \mathbb{N}$ is the *security parameter* and $z \in \{0,1\}^*$ is the input to the environment $\mathcal{Z}$. The formal details of the execution are specified in [14]. We say that a protocol π UC-realizes a functionality $\mathcal{F}$ if

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

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

Multi-protocol UC. The standard UC framework does not allow to modularly prove the security of protocols, in which a zero-knowledge proof system is used to show that a party has performed a certain cryptographic operation correctly. Camenisch et al. [7] 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 $\mathcal{F}_{\text{NIZK}}^{R}$ validates a pair (x, w) of statement x and witness w by "calling out" to the other functionality $\mathcal{F}$. We use this proof technique extensively in this work.

5.2 The privacy-preserving token functionality

The functionality $\mathcal{F}_{\text{TOKEN}}$ realized by our privacy-preserving token system is formalized in Figure 2. To keep the presentation simple, the functionality formalizes the guarantees for the case of a single

token issuer I. The functionality initially requires registration authority A, certifier C, and issuer I to initialize. (This corresponds to the fact that all protocol steps depend on those parties' keys.) Likewise, regular parties (i.e. users) have to generate and register their keys before they can perform operations. Each user can then `read` the tokens they own and generate `transfer` transactions that reference those tokens and transfer them to one or more receivers. The issuer can additionally `issue` new tokens. In the inputs and outputs of the functionality, v always represents the value of a token, and cm serves as a handle identifying the token (it stems from the commitment that represents the token on the ledger). Finally, the functionality specifies which information is potentially leaked to the adversary, and which operations the adversary can perform in the name of corrupted parties.

5.3 Set-up functionalities

Our protocol calls for a number of set-up functionalities; most of which are widely used in the literature. This is why we only briefly describe them here and defer the details to [2].

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

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

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

Secure and private message transfer. Functionality $\mathcal{F}_{\text{SMT}}$ provides a message transfer mechanism between parties. The functionality builds on the ones described by Canetti and Krawczyk [16], 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 would revoke the anonymity otherwise provided by our protocol.

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

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

$\mathcal{F}_{\text{A-AUTH}}$ 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. $\mathcal{F}_{\text{A-AUTH}}$ also allows to bind the proof to a bit string m, which intuitively can be understood as "party P (as referenced in the commitment) signs message m." The exact reason for this mechanism will become clear in the protocol description in Section 6.5.

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 for e.g. the work of Camenisch, Dubovitskaya, Haralambiev, and Kohlweiss [8].

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

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

Figure 2: Privacy-preserving token functionality

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

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

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

Figure 3: Ledger functionality

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

Functionality $\mathcal{F}_{\text{A-AUTH}}$ is parametrized by a commitment opening algorithm open. It stores an initially empty set $\mathcal{U}$ 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 (registered, P) to $\mathcal{A}$.
- Upon input (lookup, P') from a party P, return (the result of) $P' \in \mathcal{U}$.
- Upon input (prove, crs, cm, r_{cm}, x, y, m) from a registered party P, if open(crs, cm, (x, P, y), r_{cm}) then generate a proof ψ, store the record (transfer, crs, cm, ψ, m) internally and output ψ to P.
- Upon input (verify, crs, ψ, m) from some P, look up if there is a record (transfer, crs, cm, ψ, m) and output success (only) if it exists.

Figure 4: Extended anonymous registration functionality

6 PRIVACY-PRESERVING AUDITABLE UTXO

This section describes the complete protocol. We begin by introducing the core ideas and mechanisms in Sections 6.1 – 6.3. Section 6.4 introduces multi-input multi-output transactions, followed by Section 6.5 that assembles all pieces and describes the full protocol. Section 6.6 presents the extension that makes the protocol auditable and Section 6.7 states the main security result of the paper.

6.1 Core protocol ideas

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

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

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

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

A party P transfers a token (i.e. a commitment cm) to a receiver R by computing a new commitment $(cm', r'_{\text{cm}}) \leftarrow\!\!\$\ \text{commit}(crs, (v, R))$. She generates a first NIZK ψ_1 showing that cm' contains the correct information and that the receiver is registered, and a second proof ψ_2 of eligibility (i.e. the initiator of the transfer owns cm) using $\mathcal{F}_{\text{A-AUTH}}$. The information written to $\mathcal{F}_{\text{LEDGER}}$ is $tx = (\text{transfer}, cm', \psi_1, \psi_2)$. 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 subsequently. Party P also sends the message (token, cm', r'_{cm}, v) to R *privately*.

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

6.2 Certification via blind signatures

(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 and ℓ is the size of the vectors of messages to be signed. It is parametrized by the commit algorithm of the commitment as well as the a threshold signature scheme TSIG = (tkeygen, sign, verify). The functionality keeps an initially empty set Σ of signed messages.

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

Figure 5: Blind signature functionality, threshold version.

The problem of verification of token validity during transfer is resolved by *certification*. We consider a specific party, called a *certifier* C, which vouches for the validity of a token (v, P) stored

as a commitment cm on $\mathcal{F}_{\text{LEDGER}}$ by issuing a signature s on (v, P). In the proof ψ_1, P refers to signature s instead of commitment cm.

A naive implementation of the above scheme would require party P to reveal the content (v, P) of cm to certifier C, so that the latter issues the corresponding signature s: prior to signing, C checks that cm opens to (v, P) and that cm is stored on $\mathcal{F}_{\text{LEDGER}}$. Disclosing the pair (v, P) to C is both undesired and unnecessary. Instead we rely on a *blind signature* protocol, in which C learns only the value of commitment cm and *blindly* signs its content.

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

Note that a malicious certifier can essentially generate tokens by providing its signature without checking for existence of the commitment on $\mathcal{F}_{\text{LEDGER}}$. This mandates that the certification task be distributed so that no single party is trusted for verification. Figure 5 shows a threshold blind signature ideal functionality $\mathcal{F}_{\text{BLINDSIG}}$, which we will use in the description of our solution in Section 6.5. In [2], we describe how to realize $\mathcal{F}_{\text{BLINDSIG}}$ using Pointcheval-Sanders signatures [41].

6.3 Serial numbers prevent double-spending

Double-spending prevention is achieved via a scheme that is inspired by Zerocash [4] 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 via a signature from the registration authority. On a very high level, the above protocol is extended as follows.

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

We stress that authority A must be trusted to prevent double-spending, otherwise it could register multiple VRF keys for the same user. It is thus recommended to implement A in a distributed fashion.

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

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

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

6.4 Multi-input multi-output transactions

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

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

The proof, in more detail, now becomes

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

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

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

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

6.5 The protocol π_{TOKEN}

This section describes the protocol sketched previously more formally. The protocol has a bit *registered* ← false and keeps an initially empty list of commitments. We begin by describing the protocol for a user P of the system.

- Upon input `register`, if *registered* is set, then return. Else, retrieve the public keys of A, C and I 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* ← true send `register` to $\mathcal{F}_{\text{A-AUTH}}$.
- Process pending messages and retrieve new data from $\mathcal{F}_{\text{LEDGER}}$. This is a subroutine called from the functions below.
  - For transactions $tx = (\texttt{issue}, v, cm, \psi_0, s)$ 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 s via $\text{verify}(pk_I, (v, cm, \psi_0), s)$. If both checks succeed, record cm as a valid commitment.
  - For $tx = (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$, check the serial numbers $sn_1, \ldots, sn_m$ for uniqueness, validate ψ_1 via $\mathcal{F}_{\text{NIZK}}$ and verify $\psi_{2,1}, \ldots, \psi_{2,n}$ by calling $\mathcal{F}_{\text{A-AUTH}}$ with input $(\texttt{verify}, crs, \psi_{2,i}, \mathbf{m})$ for $\mathbf{m} = ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$. If all checks succeed, then store $cm_1, \ldots, cm_n$ as valid.
  - For each incoming message $(\texttt{sent}, S, P, m)$ buffered from $\mathcal{F}_{\text{SMT}}$, parse m as $(\texttt{token}, cm, r_{\text{cm}}, v, \rho)$ and test whether $\text{open}(crs, cm, r_{\text{cm}}, (v, P, \rho)) = \text{true}$ holds. Check whether there is a valid `transfer` transaction tx that appears in $\mathcal{F}_{\text{LEDGER}}$ and contains cm. If all checks are successful, input $(\texttt{request}, r_{\text{cm}}, (v, P, \rho))$ to $\mathcal{F}_{\text{BLINDSIG}}$ and wait for a response s_C. Store the complete information in the internal list.
- Upon input `read`, if ¬*registered* then abort, else first process pending messages. Then return a list of all of P's unspent tokens (cm, v).
- Upon input $\left(\texttt{transfer}, (cm_i^{\text{in}})_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$, assuming that *registered*, query $(\texttt{lookup}, R_j)$ to $\mathcal{F}_{\text{REG}}$ for all $j = 1, \ldots, n$ in order to make sure that R_j is registered. Then process pending messages and proceed as follows.
  (1) If, for any $i \in \{1, \ldots, m\}$, there is no internal record of commitment $(cm_i^{\text{in}}, r_{\text{cm}}^i, v_i^{\text{in}}, P, \rho_i^{\text{in}})$, then abort.
  (2) If $\sum_{i=1}^m v_i^{\text{in}} \neq \sum_{j=1}^m v_j^{\text{out}}$ then abort.
  (3) Choose uniformly a random ρ_j^{out} for $j = 1, \ldots, n$ and compute $(cm_j, r_{\text{cm}}^j) \leftarrow\!\$\ \text{commit}(crs, (v_j^{\text{out}}, R_j, \rho_j^{\text{out}}))$.
  (4) Compute the serial numbers $(sn_i, \pi_i) \leftarrow \text{eval}(vsk, \rho_i^{\text{in}})$, for $i = 1, \ldots, m$.
  (5) Call $\mathcal{F}_{\text{NIZK}}$ to generate proof ψ_1 as in Equation (1).
  (6) Set $\mathbf{m} \leftarrow ((sn_i)_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$. For $i = 1, \ldots, m$ send $(\texttt{prove}, cm_i^{\text{in}}, r_{\text{cm}}^i, v_i^{\text{in}}, \rho_i^{\text{in}}, \mathbf{m})$ to $\mathcal{F}_{\text{A-AUTH}}$ to obtain $\psi_{2,i}$.
  (7) Send $(\texttt{token}, cm_j, r_{\text{cm}}^j, v_j^{\text{out}}, \rho_j^{\text{out}})$ to R_j via $\mathcal{F}_{\text{SMT}}$ for $j = 1, \ldots, n$ and call $\mathcal{F}_{\text{LEDGER}}$ with input $(\texttt{append}, tx)$ for $tx = (\texttt{transfer}, (sn_i, \psi_{2,i})_{i=1}^m, (cm_j)_{j=1}^n, \psi_1)$.
  (8) Delete the internal records of cm_i^{in} for $i = 1, \ldots, m$ and return $(\texttt{transferred}, (cm_j)_{j=1}^n)$.
- Upon receiving $(\texttt{sent}, S, P, m)$ from $\mathcal{F}_{\text{SMT}}$, buffer it for later processing. Respond `ok` to sender S.

The protocol machines for parties I, C and A are easier to describe. Issuer I behaves the same as a user P except when introducing new tokens. More specifically:

(1) Upon `init`, generate a key pair $(sk_I, pk_I) \leftarrow\!\$\ \text{skeygen}(\lambda)$ for the signature scheme and input $(\texttt{register}, pk_I)$ to $\mathcal{F}_{\text{REG}}$.
(2) Upon $(\texttt{issue}, v)$, choose randomly ρ and compute commitment $(cm, r_{\text{cm}}) \leftarrow\!\$\ \text{commit}(crs, (v, I, \rho))$ and proof $\psi_0 \leftarrow \text{PK}\{(r_{\text{cm}}, \rho) : \text{open}(crs, cm, r_{\text{cm}}, (v, I, \rho)) = \text{true}\}$, where I and v are publicly known. This is achieved by sending $(\texttt{prove}, x, w)$ to $\mathcal{F}_{\text{NIZK}}$, where $x = (crs, cm, v, I)$ and $w = (r_{\text{cm}}, \rho)$. Next compute $s \leftarrow\!\$\ \text{sign}(sk_I, (v, cm, \psi_0))$ and send $(\texttt{append}, tx)$, with $tx = (\texttt{issue}, v, cm, \psi_0, s)$, to $\mathcal{F}_{\text{LEDGER}}$. Finally, internally store $(cm, r_{\text{cm}}, v, \rho)$ and return $(\texttt{issued}, cm)$.

Certifier C signs a commitment if it finds it in a valid transaction in the ledger. In more detail:

(1) Upon `init` get 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 previously. Check whether cm is marked as a valid commitment.
(3) If the above check is successful, send $(\texttt{sign}, cm)$ to $\mathcal{F}_{\text{BLINDSIG}}$.

Registration authority A signs VRF public keys of the parties.

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

6.6 Auditing

Each user in the system is assigned an auditor, which can decrypt all transaction information related to the user. Notably, the transaction outputs owned by the user and the full content of her transactions. We denote the set of auditors by **AUD**.

We formalize next our security guarantees using functionality $\mathcal{F}_{\text{ATOKEN}}$. $\mathcal{F}_{\text{ATOKEN}}$ stores a list of registered users and an initially empty map Records and has as a session identifier $sid = (A, C, I, \textbf{AUD}, sid')$.

- Upon input `init` from $P \in \{A, C, I\} \cup \textbf{AUD}$, output to $\mathcal{A}$ $(\texttt{initialized}, P)$. (This must happen for all before anything else.)
- For inputs `register`, `read`, `issue` and `deliver` see $\mathcal{F}_{\text{TOKEN}}$.
- Upon input $(\texttt{bind}, P, Aud)$ from A, where P is a registered user and $Aud \in \textbf{AUD}$ is an initialized auditor, and there is not yet a pair (P, Aud') with $Aud \neq Aud' \in \textbf{AUD}$, record (P, Aud) and output $(\texttt{bound}, P, Aud)$ to $\mathcal{A}$.
- Upon input $\left(\texttt{transfer}, (cm_i)_{i=1}^m, (v_j^{\text{out}}, R_j)_{j=1}^n\right)$ from an honest party P, where P and all R_j for $j = 1, \ldots, n$ are registered, proceed as follows.
  (1) If, for any $i \in \{1, \ldots, m\}$, $\text{Records}[cm_i] = \perp$ then abort, else set $(v_i^{\text{in}}, P_i', st_i) \leftarrow \text{Records}[cm_i]$.
  (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 Aud_j are corrupt, then append to L the information $(P, R_j, v_j^{\text{out}})$. If the auditor Aud 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 $\text{Records}[cm_j^{\text{out}}] \neq \perp$ for any $j \in \{1, \ldots, n\}$ then abort, else set $\text{Records}[cm_j^{\text{out}}] \leftarrow (v_j^{\text{out}}, R_j, \texttt{delayed})$ for all $j \in \{1, \ldots, n\}$ and for all $i \in \{1, \ldots, m\}$ set $\text{Records}[cm_i] \leftarrow (v_i^{\text{in}}, P, \texttt{consumed})$.
(6) Return $(\texttt{transferred}, (cm_j^{\text{out}})_{j=1}^n)$ to P.

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

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

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

Then P includes the resulting ciphertexts in the transfer transaction, and proves that the encryption is consistent with the information in the commitments.

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 $c_1 = \text{enc}(pk_{Aud}, P; r_1)$ and value $c_2 = \text{enc}(pk_{Aud}, v^{\text{in}}; r_2)$. Then we generate a NIZK proof:

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

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

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

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

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

6.7 Security analysis

This section shows that the protocol in Section 6.5 instantiates functionality $\mathcal{F}_{\text{TOKEN}}$.

Theorem 6.1. *Assume that* COM = (ccrsgen, commit, open) *is a perfectly hiding and computationally binding commitment scheme. Assume that* VRF = (vkeygen, eval, check) *is a verifiable random function. Then π_{TOKEN} realizes $\mathcal{F}_{\text{TOKEN}}$ with static corruption. Corruption is malicious for I and users, and honest-but-curious for C. A is required to be honest, but is inactive during the main protocol phase.*

The restriction that C is only honest-but-curious is necessary; otherwise C can sign arbitrary commitments, even ones that are not stored in $\mathcal{F}_{\text{LEDGER}}$.

Due to space limitations, the proof of Theorem 6.1 is deferred to the full version of the paper [2].

7 IMPLEMENTATION AND PERFORMANCE

To evaluate our protocol, we implemented a prototype on top of Hyperledger Fabric that requires minimal changes to be integrated. The details of the primitives underlying our implementation can be found in [2]. This section elaborates on the integration effort and measures the overhead incurred by our scheme.

7.1 Hyperledger Fabric

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

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

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

Hyperledger Fabric follows an *execute-order-validate* model. Here, chaincodes are speculatively *executed* on one or more peers upon a client request, called *chaincode proposal*, prior to submitting the resulting transaction for ordering. Execution results are signed by the peers that generated them in *chaincode endorsements* and are returned to the client who requested them. Endorsements (i.e. peer signatures) are included in the transaction that the client constructs and sends to the ordering service. The latter *orders* the transactions it receives and outputs a first version of the ledger called *raw ledger*. Raw ledger is provided to the peers of the network upon

demand. Upon receiving the raw ledger, peers *validate* the ordered transactions against the *endorsement policy* of their origin chaincodes. An endorsement policy specifies the endorsements that a transaction should carry to be deemed valid. If validation completes successfully, then the transaction is committed to the ledger.

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

7.2 Integration architecture

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

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

Finally, we leverage the membership service infrastructure of Hyperledger Fabric to grant long-term identities to issuers and users. In particular, we integrate the identity mixer MSP of Hyperledger Fabric with our solution to allow privacy preserving user authentication. To assign auditors to users, we use an off-band channel to bind identity mixer user identities with auditor encryption public keys. In a real implementation, this could be accommodated by an external identity management service, preferably distributed[2].

Notice that our protocol uses the ledger only as a time-stamping service, without any validation functionalities; those are offloaded indirectly to certifiers and auditors. This could be supported in Hyperledger Fabric directly by setting the endorsement policy of the prover chaincode to any. We note that we plan to extend our prototype to allow the ledger to also validate token transactions. More concretely, we intend to exploit the fact that Hyperledger Fabric supports pluggable transaction validation [33] that allows chaincodes to specify their own custom validation rules, in our case, the custom validation would consist of verifying the ZK proofs.

7.3 Performance numbers

We installed Hyperledger Fabric client and peer infrastructure on a MacBook Pro (15-inch, 2016), with 2.7 GHz Intel Core i7, and 16 GB of RAM. We implemented our prototype in golang, as this is the core language of Hyperledger Fabric, and used EC groups in BN256 curves. We instantiated both prover and certifier chaincodes on all the peers in the network, while disseminating the secret shares needed for token certification only to the peers reserved for that purpose. For efficiency reasons, we used Schnorr proofs [43] to implement some of the zero-knowledge proofs[3].

	token certification	
user	198.90	
certifier	123.36	
	proof generation	**proof validation**
overall computation	1992.844	2885.134
in-out consistency	157.43	287.21
token validity	263.02	322.34
serial number	208.24	452.55
auditability	1361.99	1823.01

Table 1: Performance numbers of token certification and transfer in milliseconds (ms). Transaction size is a little over 63KB; this figure however can be further optimized.

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

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

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

While these numbers are not yet favorable to a wide adoption, we would like to stress that the AMCL library underlying our implementation is not optimized. An optimization in the crypto libraries is expected to bring in a speedup of at least one order of magnitude [34]. We also note that the current implementation did not investigate possibilities of parallelization.

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

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

[3]To preserve universal composability one would need to use an online extractable variant of Schnorr proofs.

8 CONCLUSION

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

REFERENCES

[1] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM.

[2] Elli Androulaki, Jan Camenisch, Angelo De Caro, Maria Dubovitskaya, Kaoutar Elkhiyaoui, and Björn Tackmann. Privacy-preserving auditable token payments in a permissioned blockchain system. Cryptology ePrint Archive, Report 2019/1058, 2019. https://eprint.iacr.org/2019/1058.

[3] 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.

[4] 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.

[5] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. *IACR Cryptology ePrint Archive*, 2019.

[6] 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.

[7] 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.

[8] 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.

[9] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM CCS*, pages 21–30. ACM, 2002.

[10] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Compact e-cash. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, pages 302–321, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[11] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. Balancing accountability and privacy using e-cash (extended abstract). In Roberto De Prisco and Moti Yung, editors, *Security and Cryptography for Networks*, pages 141–155, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[12] 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.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science*. IEEE, 2001.

[14] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology eprint archive, report 2000/067, December 2018.

[15] 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.

[16] 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.

[17] 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.

[18] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.

[19] Yu Chen, Xuecheng Ma, Cong Tang, and Man Ho Au. PGC: decentralized confidential payment system with auditability. Cryptology eprint archive, May 2019.

[20] 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.

[21] Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[22] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. Quisquis: A new design for anonymous cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018.

[23] 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.

[24] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[25] 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.

[26] 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.

[27] 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.

[28] https://consensys.net/quorum/.

[29] https://ethereum.org/en/whitepaper/.

[30] https://www2.deloitte.com/ch/en/pages/risk/articles/security-controls-for-blockchain-applications.html. https://www2.deloitte.com/ch/en/pages/risk/articles/security-controls-for-blockchain-applications.html.

[31] https://www2.deloitte.com/content/dam/Deloitte/cz/Documents/financial-services/cz-2018-deloitte-global-blockchain-survey.pdf, 2018.

[32] https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/financial-services/lu-tokenization-of-assets-disrupting-financial-industry.pdf.

[33] Hyperledger Fabric Maintainers. Hyperledger Fabric pluggable endorsement and validation. https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html.

[34] Vlad Krasnov. Go crypto: bridging the performance gap. https://blog.cloudflare.com/go-crypto-bridging-the-performance-gap/, May 2015.

[35] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. bitcointalk.org, August 2013.

[36] 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.

[37] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Tomas Sander and Amnon Ta-Shma. Auditable, anonymous electronic cash. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 555–572, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[43] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, Jan 1991.

[44] Nicolas van Saberhagen. CryptoNote v 2.0. https://cryptonote.org/whitepaper.pdf, October 2013.