Extending zkLedger with private swaps

Alberto Centelles$^{1[0000-0002-6511-0265]}$ and Gabe Dijkstra$^{1[0000-0003-2131-0182]}$

Adjoint Inc.

Abstract. In a distributed ledger with privacy features, it is important to strike a balance between public verifiability and privacy. If transaction data is public, anyone can verify whether the ledger satisfies certain properties. However, public transaction data may leak sensitive trading information or violate data privacy regulations. With encrypted transaction data, public verifiability may be lost, as well as the possibility for third-party auditing, hindering compliance with regulations. By employing homomorphic commitment schemes and non-interactive zero-knowledge proofs, Narula et al. managed to achieve transaction privacy while retaining public verifiability and auditability with zkLedger [12]. This paper extends the zkLedger design to support the asynchronous submission of transfers and thereby avoids the race condition that required parties to recompute and resubmit private transfers. The new design enables complex multiparty workflows to be built on the zkLedger platform. As an example of such a workflow, we present a protocol to atomically and privately swap assets.

Keywords: Swaps · Secure Multiparty Computation · Multiparty Workflows · Pedersen Commitments · Bulletproofs · Arithmetic Circuits.

1 Introduction

Distributed ledger technology (DLT) allows companies to consolidate the infrastructure that reconciles transactions between them or within the companies themselves. When designing such systems, one has to deal with information boundaries. It may be undesirable for company A to see all the transactions happening between company B and C as this may reveal trading strategies or otherwise confidential information. Furthermore, the unrestricted sharing or storage of unencrypted transaction data may violate data privacy laws such as GDPR [8] or violate information boundary policies internal to companies, i.e. the so called “Chinese Walls”.

For the scenarios sketched above, transaction privacy is essential. However, transaction privacy in and of itself may not be enough. One of the core features of a distributed ledger such as Bitcoin [11], is that any participant can evaluate the consistency of the ledger and assess its non-repudiation and no-double-spending properties. Such DLTs are said to enjoy *public verifiability.* zkLedger [12] achieves transaction privacy while retaining public verifiability. The encrypted transactions come with proof objects that any party can verify. These proof objects ascertain that nobody spent more money than they had and that they only spent their own money. The proof objects and their verification do not reveal any other information than just that: they do not reveal the participants in the transfers nor the amounts transferred.

Apart from public verifiability, parties may need to selectively reveal parts of the encrypted data on the ledger to third-party auditors. They may also be required to produce aggregations of the transfers they were involved in and produce statistics such as the average amount transferred and report these to the auditors. On a ledger without transaction privacy, the auditor can simply perform these computations themselves. If the transaction data is encrypted, the auditor will have to defer to the parties having the keys to the relevant data. In this case, it is important that the auditor can still assess the validity of the answers given by the parties. The parties should be able to convince the auditor that they have computed the statistics correctly and have not left out any transfers from the computation. zkLedger achieves this by using verifiable computing techniques over commitments to the transfer amounts.

1.1 Contributions

Asynchronous Transfer Submission and Verification In zkLedger, transfer creation and submission has to happen synchronously with transfer verification. The proofs associated with a transfer have to be based on the ledger history up until the latest transfer. This means that if participant A and B both produce a transfer based on the same ledger state, only one of those transfers will be accepted. If A gets its transfer accepted, B has to create new proofs based on the ledger state that includes A's transfer. In section 4 we provide an addition to zkLedger that allows us to make the submission and verification asynchronous.

Private Swaps In section 5 we describe a protocol that allows for two parties to atomically exchange assets on a private ledger. The protocol does not leak any information about the amounts swapped nor about the participants to any other non-verifying party. We make essential use of asynchronous transfer submission to make the protocol non-blocking for the parties involved.

2 Cryptographic Preliminaries

Our setting relies on some cryptographic primitives and assumptions that are worth reviewing before diving into the content of the paper.

2.1 Assumptions

Let $\mathbb{G}$ be a cyclic group of order q.

Definition 1. *(Discrete Log Problem (DLP)). If g is a generator of the group $\mathbb{G}$, every element h in $\mathbb{G}$ can be written as g^x for some x. Given h and g, the discrete logarithm problem is to find x, i.e. the discrete log to the base g of h in the group $\mathbb{G}$.*

The groups in our setting happen to be elliptic curves over a field $\mathbb{F}_p$, hence the DLP is also known as the Elliptic Curve Discrete Logarithm Problem (ECDLP).

Definition 2. *(Decision Diffie-Hellman assumption (DDH))* [4]. *A probabilistic polynomial-time adversary has a negligible probability of distinguishing:*

- (g^a, g^b, g^{ab}) *for random* $a, b \in \mathbb{Z}_q^*$
- (g^a, g^b, g^c) *for random* $a, b, c \in \mathbb{Z}_q^*$

with generator $g \in \mathbb{G}$.

Definition 3. *(Random Oracle Model (ROM))* [3]. *Given a security parameter* k*, a* random oracle R *is a map from* $\{0,1\}^*$ *to* $\{0,1\}^{l(k)}$ *chosen by selecting each bit of* $R(x)$ *independently and uniformly, for every query* x. $l(\cdot)$ *is a length function that determines the set of possible functions.*

Fiat and Shamir [9] use the ROM model to convert three-message identification schemes to an efficient non-interactive signature scheme, while preserving the security of the original scheme.

2.2 Cryptographic primitives

Definition 4. *(Pedersen commitment). Let* $g, h \in \mathbb{G}$. *Given a message* $v \in \mathbb{Z}_q$ *and a value* $r \in \mathbb{Z}_q$ *drawn uniformly at random, a* Pedersen commitment *[13] is a non-interactive, perfectly hiding, homomorphic commitment defined as:*

$$\mathsf{Com} : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{G}$$

$$\mathsf{Com}(v, r) = g^v h^r$$

From now on, we will use commitment to refer to Pedersen commitment.

Definition 5. *(Non-interactive zero-knowledge proof (*NIZK*)). A* non-interactive zero-knowledge proof *is a protocol in which a prover convinces a verifier that some statement holds without revealing anything about that statement apart from that it holds.*

These proofs are non-interactive in that any participant can create or validate proofs without interacting with the other party in the protocol. Our NIZKproofs are constructed by applying the Fiat-Shamir transform [9] on the Generalized Schnorr Protocol [14,10], Bulletproofs and the OR-construction form [6]. Since we are using the Fiat-Shamir transform, we rely on a random oracle to achieve security and full zero-knowledge [3].

Definition 6. *(Range Proof). A* range proof *with respect to a commitment scheme* C *is a proof of set membership in which the set* ϕ *is a continuous sequence of integers* $\phi = [a, b]; a, b \in \mathbb{N}$.

Given that we are working with Pedersen commitments in a group $\mathbb{G}$ of order q, we are essentially working with integers modulo q. We still need to be able to talk about positive and negative integers. Given $n \in \mathbb{N}$ such that $2^{n-1} < q < 2^n$, we designate values $x \in \mathbb{Z}_q$ as positive if $0 < x < 2^{n-1}$ and negative otherwise.

3 Private Ledger

The aim of the private ledger is to maintain the privacy of its participants while making it possible for a third party to get reliable answers. Both participants' privacy and auditing capabilities are guaranteed.

As in the original zkLedger paper [12], the amount transferred and the participants involved are obfuscated and the transaction graph is hidden. Only the time of the transfer and the type of asset transferred are public.

3.1 Setting

While the zkLedger paper leaves it open how to exactly maintain the ledger, this paper is more specific to a certain setting. In this setting we have three types of roles interacting with the system:

- *depositors*: parties allowed to submit public transfers,
- *account holders*: parties allowed to submit private transfers,
- *verifier*: the party who accepts/rejects transfers issued to the system

We will have a central (but possibly distributed) store for the private ledger and associated data. Account holders submit their private transfers to a queue on this store. Analogously, depositors send their public transfers to a similar queue on the same store. The verifier regularly polls the queue and verifies the private and public transfers against the current ledger state. It then decides to accept or reject the transfers, depending on whether the proofs can be verified successfully or not.

Adding the notion of a *verifier* is key to being able to process transfers more asynchronously, as we will see in section 4. The verifier will receive a bit more information than the other parties involved in the system to make its decisions. However, the verifier will also provide proofs that it has processed this extra information correctly. These proofs are public to any party interacting with the system.

There are several efficiency downsides in a private ledger compared to a public one. Firstly, proofs involved in the transaction creation and validation are computationally expensive, especially range proofs. Despite recent improvements in the time and size of the range proofs made by Bulletproofs [5], they are still the slowest proofs to create. Secondly, private transactions are sized order the number of participants, i.e. the verification of transactions increases quadratically with the number of accounts, making the private ledger more suitable for a small number of them.

Our setting includes optimizations to address some efficiency challenges. As we will see later in detail, the proof of assets guarantees that an account has enough of an asset to transfer. The verifier proves the spender account liquidity by calculating the product of commitments of all the transactions in its account. This computation can become too slow as the number of transactions grow. A commitment cache is constantly kept with the rolling product of the commitments to avoid having to compute this calculation for all previous transactions.

Furthermore, participants can construct proofs of the transfer non-interactively, which reduces communication costs.

3.2 Private Ledger

Definition 7. *A* private ledger *containing m accounts consists of:*

- *a set of public keys $\{\mathsf{pk}_i\}_{i \in accts}$ for every account,*
- *for every asset α, a list of private transfers and public transfers $(t_0, \ldots, t_n)$*

A ledger can be visualized as a table of ordered transactions and accounts. Transactions correspond to rows and accounts correspond to columns. An account in a private ledger can only issue private transfers, except for the depositor's account. A depositor in the ledger is a special type of account that is capable of issuing and withdrawing assets publicly. On the other hand, private transfers can be issued by any other account and are publicly verifiable, i.e. any participant can validate a private transaction in real time.

ID	**Time**	**Asset**	**Account** A pk_A	**Account** B pk_B	...	**Account** M pk_M
0	-	α_0	t_0			
1	-	α_1	t_1			
...			...			
n	-	α_n	t_n			

As the private ledger only stores encrypted information, all accounts must keep their own data separately in order to be able to answer queries. Our setting guarantees that an auditor will have reliable answers from queries to any account based on its encrypted data.

3.3 Private transfers

A private transaction pt consists of a set of commitments, one for each account, in which an empty entry is indistinguishable from an entry of an account involved in the transfer. It also contains a set of proofs that allow a public verifier check that the transaction conserves assets and the spending account has enough balance of an asset to transfer. With both commitments and proofs, the integrity and privacy of the ledger is guaranteed and the transaction graph remains hidden. With audit tokens, any participant can provide provably correct answers to queries. More formally:

Definition 8. *A* private transfer *is a tuple $(\mathsf{cm}_o, \ldots, \mathsf{cm}_n)$ of commitments to integer values $v_0, \ldots, v_n$ along with for every element of the tuple:*

- *a* proof of assets π_i^A
- *a* proof of balance π_i^B

ID	**Time**	**Asset**	**Account** A	**Account** B	$\dots$	**Account** M
			pk_A	pk_B		pk_M
-	-	α	$\mathsf{Com}(X)$ token_A $\pi_A^A\ \pi_A^C$	$\mathsf{Com}(Y)$ token_B $\pi_B^A\ \pi_B^C$	$\dots$	$\mathsf{Com}(Z)$ token_C $\pi_C^A\ \pi_C^C$
			π^B			

- *a* token token_i
- *a* proof of token consistency π_i^C

When account holders transfer assets, we make sure that such transfers never create or destroy assets. In other words, given a transfer $(v_0, \dots, v_n)$, we want $\sum_{i=0}^n v_i = 0$. To ascertain that this is the case, the account holder creating the private transfer can pick the blinders r_i such that $\sum_{i=0}^n r_i = 0$, then $\prod_{i=0}^n \mathsf{cm}_i = \prod_{i=0}^n g^{v_i} h^{r_i} = g^{\sum_{i=0}^n v_i} h^{\sum_{i=0}^n r_i} = g^{\sum_{i=0}^n v_i}$, which will be 1 if and only if $\sum_{i=0}^n v_i = 0$.

Definition 9. *Given a tuple of commitments* $(\mathsf{cm}_0, \dots, \mathsf{cm}_n)$ *with* $\mathsf{cm}_i = g^{v_i} h^{r_i}$*, a* proof of balance *consists of the prover choosing blinders* r_i *such that* $\sum_{i=0}^n r_i = 0$*. The verifier checks whether* $\prod_{i=0}^n \mathsf{cm}_i = 1$*, which implies that* $\sum_{i=0}^n v_i = 0$*.*

In order for account holders to make statements about their holdings, they need some form of access to the blinders used in the relevant parts of previous private transfers. We cannot simply make the blinders public to everybody, as that would allow anyone to check who is involved in a private transfer. Instead, the issuer of a private transfer provides a token:

Definition 10. *Given a commitment* $\mathsf{cm}_i = g^{v_i} h^{r_i}$*, the* token *is* $\mathsf{pk}_i^{r_i}$*, denoted* token_i*.*

Given a commitment cm_i and a token token_i, the verifier needs to be able to check whether the token indeed represents the blinder r_i. For this, we need a *proof of consistency*:

Definition 11. *A* proof of token consistency *for a commitment* cm_i *and* token_i *is given by the* Σ_ϕ *protocol with* $\phi_i : \mathbb{Z}^2 \to \mathbb{G}^2$*,* $\phi_i(v, r) = (g^v h^r, \mathsf{pk}_i^r)$*.*

Most importantly, we want to be able to verify that a private transfer only spends money that is currently held by the spender, i.e. no account holder has a negative balance.

Definition 12. *A* proof of assets *for a tuple of commitments* $(\mathsf{cm}_{k0}, \dots, \mathsf{cm}_{kn})$*, with* $\mathsf{cm}_{ki} = g^{v_{ki}} h^{r_{ki}}$*, consists of recommitments* $(\mathsf{rcm}_{k0}, \dots, \mathsf{rcm}_{kn})$ *with* $\mathsf{rcm}_{ki} = g^{v'_{ki}} h^{r'_{ki}}$ *where* $v'_{ki} = v_{ki}$ *if* $v_{ki} \geq 0$ *(i receives money) and* $v'_{ki} = \sum_{x=0}^k v_{xi}$ *(i spends money and* v'_{ki} *is i's balance after the current transfer).*

The recommitments are accompanied with a range proof proving that $v'_i \in [0, 2^k)$ *and a consistency proof which consists of the OR-construction applied to two Generalized Schnorr Protocols. The homomorphisms used are* $\phi_1, \phi_2 : \mathbb{Z}^3 \to \mathbb{G}^2$ *with:*

- *(i is not a spender)* $\phi_1(v, r, r') = (g^v h^r, g^v h^{r'})$*, where the issuer shows it has a preimage of* $(\mathsf{cm}_{ki}, \mathsf{rcm}_{ki})$*, namely* $(v_{ki}, r_{ki}, r'_{ki})$*,*
- *(i is a spender)* $\phi_2(v, r, r') = (g^v \left(\prod_{x=0}^{k} \mathsf{token}_{xi}\right)^r, g^v h^{r'})$*, where the issuer shows that it has a preimage of* $(\prod_{x=0}^{k} \mathsf{cm}_{xi}, \mathsf{rcm}_{ki})$*, namely* $(v'_{ki}, sk_i^{-1}, r'_{ki})$*.*

The proof of assets also doubles as a proof of spender: only holders to the secret key of account i can issue a private transfer that spends holdings of account i.

3.4 Operations

As described above, anybody can verify the correctness of the proofs provided in the private ledger. Furthermore, any account holder can verify whether their balance agrees with the current ledger state. Let b be the account holder, k its expected balance, and suppose the ledger contains n transfers. We want to check whether $\prod_{i=0}^{n} \mathsf{cm}_{ki}$ opens up to the value b. k can compute g^b and has access to the secret key sk_k such that $h^{\mathsf{sk}_k} = \mathsf{pk}_k$. k can therefore also compute $(\prod_{i=0}^{n} \mathsf{cm}_{ki} g^{-b})^{\mathsf{sk}_k}$ and $\prod_{i=0}^{n} \mathsf{token}_{ki}$. If b matches the balance in the ledger $\sum_{i=0}^{n} v_{ki}$, then we have the following:

$$\begin{aligned}
\left(g^{-b} \prod_{i=0}^{n} \mathsf{cm}_{ki}\right)^{\mathsf{sk}_k} &= \left(g^{-b} g^{\sum_{i=0}^{n} v_{ki}} h^{\sum_{i=0}^{n} r_{ki}}\right)^{\mathsf{sk}_k} \\
&= (h^{\sum_{i=0}^{n} r_{ki}})^{\mathsf{sk}_k} \\
&= \mathsf{pk}_k^{(\sum_{i=0}^{n} r_{ki})} \\
&= \prod_{i=0}^{n} \mathsf{token}_{ki}
\end{aligned}$$

Therefore the account holder k can check whether its own balance matches with the private ledger state by checking:

$$\left(g^{-b} \prod_{i=0}^{n} \mathsf{cm}_{ki}\right)^{\mathsf{sk}_k} \stackrel{?}{=} \prod_{i=0}^{n} \mathsf{token}_{ki}$$

4 Accepting transfers asynchronously

The zkLedger approach requires every issuer of a private transfer to have seen every previous transfer (or a summary of that) in order to produce a valid proof of assets. If we have a system in which various issuers may submit private transfers concurrently, certain submissions can be dropped because it is based on a previous ledger state. For example, issuers A and B poll the ledger state at the same time and build their private transfer pt_A and pt_B with its proof objects based on that state. If A first submits pt_A, then pt_B will be rejected by the system, as its proofs are based on a ledger state that did not include pt_A yet. Once B learns that pt_B has been rejected, it could poll to receive the new ledger state and generate new proofs. However, B can end up in the situation that right before every submission, A submits a new private transfer which then gets accepted. This means that B will never have its transfers accepted by the ledger.

In this section we will see how we can provide the verifier with just enough information (but not more) to be able to assess whether we can accept transfers such as pt_B or not, without introducing possible double spending.

The idea is that along with the private transfer, the issuer also sends a "proof of spenders": the index of the latest transfer the issuer has seen accepted to the ledger and for every participant a commitment indicating whether they are a spending party or not. The verifier then has to asses whether the spenders of the submitted transfer are distinct from the spenders of the intermediate transfers, i.e. all the transfers accepted to the ledger after the latest transfer seen by the issuer.

The design goals for this extension are that non-verifier participants should not learn anything new from the proof of spenders, the verifier can non-interactively assess the validity of the private transfer, and public verifiability should be retained.

4.1 Proof of spending

Given a commitment $\mathsf{cm} = g^v h^r$, the issuer needs to produce a "spending bit commitment" $\mathsf{scm} = g^b h^{r'}$ with $b = \textbf{if } v < 0 \textbf{ then } 1 \textbf{ else } 0$. Following [15], we can encode this computation as an arithmetic circuit. Using Bulletproofs we can therefore produce a proof that the committed value of scm has been computed correctly from the committed value of cm, which can be verified non-interactively without having to open either commitments.

Definition 13. *A* proof of spenders *for a private transfer* pt *with commitments* $(\mathsf{cm}_0, \ldots, \mathsf{cm}_n)$ *consists of:*

- *an index of the previous transfer* pt *its proof of assets is based on,*
- *spending bit commitments* $(\mathsf{scm}_0, \ldots, \mathsf{scm}_n)$*,*
- *proofs asserting that if* cm_i *opens up to* v*, then* scm_i *opens up to* b *such that* $b = \textbf{if } v < 0 \textbf{ then } 1 \textbf{ else } 0$*.*

When a verifier receives a private transfer pt_m with proof of spenders, it has to check the proof of assets up until the index n given in the proof of spenders. It then has to check that since that transfer, the spenders of the later transfers are disjoint from the spenders of pt_m. This amounts to checking for every participant k, that $\prod_{i=n+1}^{n+k} \mathsf{scm}_{ik}$ opens up to 0 if scm_{mk} opens up to 1. If scm_{mk} opens up to 0, we do not care what values the other commitments contain. In other words, if scm_{im} opens up to b and $\prod_{x=n+1}^{n+k} \mathsf{scm}_{ix}$ opens up to b', then $bb' = 0$.

The verifier cannot make the above assertions without knowing the openings to all spending bit commitments. Therefore every issuer sends the openings to the verifier (and no-one else) when submitting a private transfer. Given these openings, the verifier can both perform the check, as well as a proof that the property holds using Bulletproofs. While the non-verifier participants do not learn from this proof object who the spenders of the private transfers are, they can use it to check whether the verifier has been honest or not when accepting the transfer, hence we retain public verifiability.

5 Private swaps

Apart from simply transfer a certain amount of assets between parties, we might want to perform more complicated transfers. One example of this is a *swap*: a party wants to transfer an amount X of asset α to the counterparty and get an amount Y of asset β in return from the counterparty. In this section we will introduce a protocol to perform these swaps on top of a private ledger.

Our protocol to execute swaps adheres to the following design criteria:

- *private*: we do not want other other parties to learn who the participants are in a swap,
- *atomic*: either both transfers take place or no asset will be swapped,
- *safe*: if a party does not follow the protocol, the other parties will not end up losing assets,
- *non-blocking*: other transfers can be submitted on the private ledger during the time between a swap is proposed and accepted.

5.1 Protocol

There are three parties involved in the protocol: account holders A and B, who want to swap assets with each other, and the verifier V, who has to accept the transfers into the private ledger. The protocol consists of four steps:

- setup ($A \to \mathsf{V}, A \leftrightarrow B$)
- proposal ($A \to B$)
- assessment ($B \to \mathsf{V}$) or timeout ($A \to \mathsf{V}$)
- validation (V)

Setup We assume that A and B have already exchanged the asset types and amounts they want to swap.

An initial state of the ledger would contain m accounts and t transactions which are not displayed in the following table:

ID	**Time**	**Asset**	**Account A**	**Account B**	. . .	**Account M**
0. . .k	. . .	. . .	$t_0 \dots t_k$			

Both accounts A and B involved in a swap should be able to participate in other transfers or swaps during the time a swap takes place.

That B can participate in other transfers or swaps is obvious, as the proof of assets in $\mathsf{pt}_{B\to A}$ will be based on an up-to-date ledger state.

To make A's participation in other transfers during the course of a swap possible, another temporary account e_A must be created in the ledger before the swap takes place. By submitting a transfer to e_A with the agreed amount X, eA will have enough of the asset to be swapped by the time B accepts or rejects it. Because e_A is created only for this swap and is not participating in any other swap or transfer, e_A holds the same balance during the swap and the verifier V can verify its non-participation checking the proof of spender. Hence, A is free to participate in other transfers without compromising the swap.

A has knowledge of e_A's private key.

ID	**Time**	**Asset**	**Account** A	**Account** B	...	**Account** M	**Account** e_A
0...k	...	...	$t_0 \ldots t_k$				———

Proposal Account A wants to swap X holdings of an asset α in exchange of Y of an asset β with account B.

First, A submits a private transfer of asset α to the newly created e_A account with the proposing amount. e_A has now X amount of α to spend in the ledger.

ID	**Time**	**Asset**	**Account** A	**Account** B	...	**Account** M	**Account** e_A
0...k	...	...	$t_0 \ldots t_k$				———
k+1	...	α	Com(-X)	Com(0)	...	Com(0)	Com(+X)

e_A creates two conflicting private transfers of amount X of asset α, $\mathsf{pt}_{e_A\to B}$ and $\mathsf{pt}_{e_A\to A}$, for the cases where B accepts and rejects the swap, respectively. Both private transfers are encrypted with the verifier's public key, so that B won't be able to see or modify them when she receives the swap proposal. Only one of these two private transfers will finally be submitted to the ledger, depending on whether B accepts or rejects the proposed swap.

By the time e_A proposes the swap, he also knows what amounts B will have to send to A in case she accepts the exchange. Therefore, e_A creates the commitments of the transfer $\mathsf{pt}_{B\to A}$ that B will later submit, i.e. a set of commitments of value Y in entry A and 0 otherwise. e_A signs the commitments of this transfer so that B cannot forge them. This way we ensure that B will submit her agreed amount in the swap.

e_A will then send the following data to B:

– $\mathsf{enc}_{\mathsf{pt}_{e_A\to A}}$: $\mathsf{pt}_{e_A\to A}$ encrypted with V's private key in case the swap is rejected.

- $\mathsf{enc}_{\mathsf{r}_{eA\to A}}$: The blinders of the commitments of $\mathsf{pt}_{e_A\to A}$ encrypted with V's private key.
- $\mathsf{enc}_{\mathsf{pt}_{e_A\to B}}$: $\mathsf{pt}_{e_A\to B}$ encrypted with V's private key in case the swap is accepted
- $(\mathsf{v}_{e_A\to B}, \mathsf{r}_{e_A\to B})$: The openings of the commitments of $\mathsf{pt}_{e_A\to B}$
- $(\mathsf{v}_{B\to A}, \mathsf{r}_{B\to A})$: The openings of the commitments of $\mathsf{pt}_{B\to A}$
- $\sigma^{eA}_{\mathsf{cms}_{B\to A}}$: The signature of the commitments of $\mathsf{pt}_{B\to A}$

Assessment Once B receives the proposed swap from e_A, she first checks that openings $(v_0, \ldots, v_m)$ of $\mathsf{pt}_{e_A\to B}$ and $(v'_0, \ldots, v'_m)$ of $\mathsf{pt}_{B\to A}$ correspond to the agreed amounts to exchange.

(Accept) If B accepts the swap, she creates and signs the commitments from the openings of $\mathsf{pt}_{e_A\to B}$ sent by e_A, proving that B agrees with the committed values in the transfer.

B also creates another set of commitments from the openings of $\mathsf{pt}_{B\to A}$ sent by e_A and compute the corresponding proofs. B submits the private transfer $\mathsf{pt}_{B\to A}$ along with:

- $\sigma^{B}_{\mathsf{cms}_{e_A\to B}}$: The signature of the commitments of $\mathsf{pt}_{e_A\to B}$
- $\mathsf{enc}_{\mathsf{pt}_{eA\to B}}$: The already encrypted transfer $\mathsf{pt}_{e_A\to B}$ by e_A
- $\mathsf{pt}_{B\to A}$ computed from the commitments received from e_A and the necessary proofs
- $\sigma^{eA}_{\mathsf{cms}_{B\to A}}$: The signature of the commitments of $\mathsf{pt}_{B\to A}$ signed by e_A

(Reject) If B rejects the swap, she just submits the encrypted private transfer $\mathsf{pt}_{e_A\to A}$ received from e_A and the blinders of its commitments.

Verification

(Accept) If B accepted the exchange, the verifier V performs the following actions:

- $\mathsf{Dec}(\mathsf{enc}_{\mathsf{pt}_{e_A\to B}})$: Decrypts the accepted transfer $\mathsf{pt}_{e_A\to B}$
- Verify$(\sigma^{B}_{\mathsf{cms}_{e_A\to B}}, \mathsf{pt}_{e_A\to B})$: Verifies the signature produced by B against the commitments of $\mathsf{pt}_{e_A\to B}$
- Verify$(\sigma^{e_A}_{\mathsf{cms}_{B\to A}}, \mathsf{pt}_{B\to A})$: Verifies the signature produced by e_A against the commitments of $\mathsf{pt}_{B\to A}$
- Checks both $\mathsf{pt}_{e_A\to B}$ and $\mathsf{pt}_{B\to A}$ proofs.

If signatures and other NIZKproofs are verified, V submits both $\mathsf{pt}_{e_A\to B}$ and $\mathsf{pt}_{B\to A}$ at the same time. Otherwise, no private transfer is submitted. In any case, atomicity is preserved.

ID	**Time**	**Asset**	**Account** A	**Account** B	...	**Account** M	**Account** e_A
0...k	...	...	$t_0 \dots t_k$				—
k+1	...	α	cm(-X)	cm(0)	...	cm(0)	cm(+X)
k+2...n	...	...	$t_{k+2} \dots t_n$				
n+1	...	α	cm(0)	cm(+X)	...	cm(0)	cm(-X)
n+2	...	β	cm(+Y)	cm(-Y)	...	cm(0)	cm(0)

(Reject) If B rejected the exchange, V:

- $\mathsf{Dec}(\mathsf{enc}_{\mathsf{pt}_{e_A \to A}})$: Decrypts $\mathsf{pt}_{e_A \to A}$ encrypted by A
- $\mathsf{Dec}(\mathsf{enc}_{\mathsf{r}_{e_A \to A}})$: Decrypts blinders of the commitments of $\mathsf{pt}_{e_A \to A}$ encrypted by A with the verifier's public key
- Validate all proofs in $\mathsf{pt}_{e_A \to A}$

Only $\mathsf{pt}_{e_A \to A}$ is submitted on the private ledger, returning funds to A.

ID	**Time**	**Asset**	**Account** A	**Account** B	...	**Account** M	**Account** e_A
0...k	...	...	$t_0 \dots t_k$				—
k+1	...	α	cm(-X)	cm(0)	...	cm(0)	cm(+X)
k+2...n	...	...	$t_{k+2} \dots t_n$				
n+1	...	α	cm(+X)	cm(0)	...	cm(0)	cm(-X)

Finally, e_A is removed from the set of active accounts on the ledger and she will not participate in following transfers:

ID	**Time**	**Asset**	**Account** A	**Account** B	...	**Account** M	**Account** e_A
0...k	...	...	$t_0 \dots t_k$				—
k+1...n+l	...	...	$t_{k+1} \dots t_{n+l}$				
n+l+1...n+l+r	...	...	$t_{n+l+1} \dots t_{n+l+r}$				—

Certain invariants need to hold before one can remove an escrow account e_i or any other account in the private ledger:

- The issuer must hold a private key of the account to remove
- The account must have 0 holdings of every asset

Once an account is removed, it becomes an inactive account. No account can send any holdings to an inactive account, but the ledger needs to still be able to point to it so that other previous transfers' proofs are verified.

Private Swap Protocol

Input: $g, h \in \mathbb{G}, \alpha, \beta, \mathsf{v}_{eA\to A}, \mathsf{v}_{eA\to B}, \mathsf{v}_{B\to A} \in \mathbb{Z}_q^m$

Proposal $(A \to B)$ **Assessment** $(B \to \mathsf{V})$ **Validation** (V)

$\mathsf{r}_{e_A\to A}, \mathsf{r}_{e_A\to B}, \mathsf{r}_{B\to A} \leftarrow_{\$} \mathbb{Z}_q^m$

A computes:

$\mathsf{pt}_{e_A\to A} = pt(\mathsf{v}_{eA\to A}, \mathsf{r}_{eA\to A})$

$\mathsf{pt}_{e_A\to B} = pt(\mathsf{v}_{eA\to B}, \mathsf{r}_{eA\to B})$

$\mathsf{cms}_{B\to A} = \mathsf{Com}(\mathsf{v}_{B\to A}, \mathsf{r}_{B\to A})$

$\mathsf{enc}_{\mathsf{pt}_{e_A\to A}} = \mathsf{Enc}_{pk_{\mathsf{V}}}(\mathsf{pt}_{e_A\to A})$

$\mathsf{enc}_{\mathsf{r}_{eA\to A}} = \mathsf{Enc}_{pk_{\mathsf{V}}}(\mathsf{r}_{eA\to A})$

$\mathsf{enc}_{\mathsf{pt}_{e_A\to B}} = \mathsf{Enc}_{pk_{\mathsf{V}}}(\mathsf{pt}_{e_A\to B})$

$\sigma^{e_A}_{\mathsf{cms}_{B\to A}} = \mathsf{Sig}_{eA}(\mathsf{cms}_{B\to A})$

A sends:

$$\xrightarrow[]{\substack{\mathsf{enc}_{\mathsf{pt}_{e_A\to A}},\ \mathsf{enc}_{\mathsf{r}_{eA\to A}} \\ \mathsf{enc}_{\mathsf{pt}_{e_A\to B}},\ \sigma^{e_A}_{\mathsf{cms}_{B\to A}} \\ (\mathsf{v}_{e_A\to B}, \mathsf{r}_{e_A\to B}),\ (\mathsf{v}_{B\to A}, \mathsf{r}_{B\to A})}}$$

If B *accepts* the swap:

B computes:

$\mathsf{cms}_{e_A\to B} = \mathsf{Com}(\mathsf{v}_{e_A\to B}, \mathsf{r}_{e_A\to B})$
$\mathsf{pt}_{B\to A} = pt(\mathsf{v}_{B\to A}, \mathsf{r}_{B\to A})$
$\sigma^{B}_{\mathsf{cms}_{e_A\to B}} = \mathsf{Sig}_{eA}(\mathsf{cms}_{B\to A})$

B sends:

$$\xrightarrow[]{\substack{\mathsf{enc}_{\mathsf{pt}_{e_A\to B}},\ \mathsf{pt}_{B\to A} \\ \sigma^{B}_{\mathsf{cms}_{e_A\to B}},\ \sigma^{e_A}_{\mathsf{cms}_{B\to A}}}}$$

V computes:

$\mathsf{pt}_{e_A\to B} = \mathsf{Dec}(\mathsf{enc}_{\mathsf{pt}_{e_A\to B}})$
$\text{Verify}(\sigma^{B}_{\mathsf{cms}_{e_A\to B}}, \mathsf{pt}_{e_A\to B})$
$\text{Verify}(\sigma^{e_A}_{\mathsf{cms}_{B\to A}}, \mathsf{pt}_{B\to A})$

If B *rejects* the swap:

B sends:

$$\xrightarrow[]{\mathsf{enc}_{\mathsf{pt}_{e_A\to A}},\ \mathsf{enc}_{\mathsf{r}_{e_A\to A}}}$$

V computes:

$\mathsf{pt}_{e_A\to B} = \mathsf{Dec}(\mathsf{enc}_{\mathsf{pt}_{e_A\to A}})$
$\mathsf{r}_{e_A\to A} = \mathsf{Dec}(\mathsf{enc}_{\mathsf{r}_{e_A\to A}})$

Completeness Given two honest parties, we consider the protocol to be complete if both parties receive their agreed amount of the corresponding asset when a swap is accepted.

(B → A) Account A is certain to receive the agreed amount Y of asset β from B for two the following two actions that e_A performs during the proposal stage (beware that e_A acts on behalf of A):

- e_A signs the commitments of $\mathsf{pt}_{B \to A}$ and sends the openings to B. B will only find different values for the same commitments if she breaks the discrete log problem, which is not computationally feasible, so she can only use the blinders of the commitments sent by A to recompute the commitments of $\mathsf{pt}_{B \to A}$. The verifier's account V validates the private transfer $\mathsf{pt}_{B \to A}$ only if its commitments are signed by e_A.
- e_A encrypts the first half of the swap ($\mathsf{pt}_{e_A \to B}$ and $\mathsf{pt}_{e_A \to A}$) with V's public key, guaranteeing that B cannot see the proofs and therefore, cannot submit a private transfer $\mathsf{pt}_{e_A \to B}$ separately without submitting her part of the swap. Only V can decrypt these encrypted private transfers, check the swap is valid and submit them afterwards.

($A \to B$) If both parties act honestly, account B is also certain to receive the agreed amount X of asset α from A if B accepts the swap. In assessment step of the protocol, B receives from e_A the openings of the commitments of $\mathsf{pt}_{e_A \to B}$. As B creates new commitments with the openings sent by A and signs these commitments, $\mathsf{pt}_{e_A \to B}$ is valid only if its commitments match the ones signed by B, once $\mathsf{pt}_{e_A \to B}$ is decrypted.

Soundness If A or B are malicious, an atomic swap cannot take place.

(A Malicious) Let e_A be a malicious account that encrypts $\mathsf{pt}_{e_A \to B}$ with values that differ from the agreed values. Despite not being honest, e_A needs to send the openings of the commitments to B. If the openings $(v_1, \ldots, v_m)$ don't match with the values agreed, B rejects the swap. If they do match, B commits and signs the commitments of $\mathsf{pt}_{e_A \to B}$. Finally, V decrypts $\mathsf{pt}_{e_A \to B}$ and checks that its commitments don't match the commitments signed by B and the swap is dismissed.

In a different scenario, if e_A tries to submit a transfer to return funds to A while the swap is on going, the first part of the swap will fail and both transfers in the swap will be rejected, due to its atomicity.

(B Malicious) Let B be malicious now. She commits to different values in $\mathsf{pt}_{B \to A}$. As these commitments are signed by e_A separately, the swap fails.

B cannot replicate $\mathsf{pt}_{e_A \to B}$ as she does not have the means to create the necessary proofs on behalf of e_A and cannot decrypt the encrypted private transfer $\mathsf{pt}_{e_A \to B}$ sent by e_A, so $\mathsf{pt}_{e_A \to B}$ cannot be forged.

6 Implementation

We have implemented a prototype in Haskell to evaluate the design proposed in this paper. We use the elliptic curve `secp256k1`, a 256-bit Koblitz curve with verifiably random parameters over a finite prime field $\mathbb{F}_p$. The points on this curve form a cyclic group of prime order q, so every element in the group is a generator. That every point on the curve is a generator is important to our setting. Let g be the base generator of the curve and $h = g^x$ another generator. Our setting requires that no participant in the ledger knows the discrete log (x) of h and that h is drawn uniformly at random. Every account keeps a public key $\mathsf{pk}_i = h^{\mathsf{sk}_i}$ on the private ledger.

A $\mathsf{token} = h^{\mathsf{sk} \cdot r}$ is a value stored with each commitment in a private transfer that allows every account to reliably answer queries to an auditor without revealing openings v and r. As mentioned above, a Pedersen commitment to a value v is formed as $\mathsf{Com}(v, r) = g^v h^r; v, r \in \mathbb{Z}_p$. They perfectly hide values and can be homomorphically combined. This homomorphic property of commitments is key to the auditing capabilities the private ledger offers, such as sums, averages, variance, standard deviation and ratios.

We use the Schnorr NIZKprotocol [14,9] as it is implemented in Adjoint's open source Schnorr library [2] to generate non-interactive zero-knowledge proofs of knowledge and the signatures involved in the atomic swap protocol. For the Fiat-Shamir transformation, we use SHA256 as the random oracle.

Adjoint's implementation of the Bulletproofs protocol [1] is used to generate range proofs. The Bulletproofs protocol do not need a trusted setup and generate very short proofs, so they are a good fit for confidential transactions. The proof of spender described in section 4 also makes use of the efficient zero-knowledge argument for arithmetic circuits that the Bulletproofs paper presents and Adjoint's Bulletproofs library implements.

7 Conclusion

In this paper we have presented a way to asynchronously deal with private transfer submission and verification in a zkLedger setting. We achieve this by introducing a designated verifier who is told for every transfer who the spender is, but does not learn anything beyond that. This verifier then uses that information along with the other proofs to decide whether it can accept the transfer onto the ledger or not. The computations involved in this decision are done in a verifiable way, meaning that any participant can assess whether the verifier has acted correctly, without having access to the same information as the verifier.

Furthermore, as an example of a private multiparty workflow on the zkLedger platform, we have specified a protocol to perform swaps in this setting, making essential use of our asynchronous transfer submission and verification extension. This allows parties to swap assets without having other non-verifier participants learn who are the participants in the swap and which amounts are being transferred.

8 Future work

Obscuring assets At the moment, a private transfer exposes which asset it involves. When dealing with swaps, verifiers then see which assets are being exchanged. Instead, we could have a commitment to the id of an asset and work with that. This change would complicate computing the commitment to a certain account holder's balance with regards to a specific asset, as needed for the proof of assets.

Obscuring swap participants to the verifier Our protocol for private swaps does not expose the parties involved in a swap to any other non-verifier party. The verifier does learn who the participants are, as is needed for the signature checks to assess the authenticity of the swap.

Encrypting recipients in private transfers When a private transfer is accepted onto the ledger, the recipients of the transfer need to be notified and update their own balances accordingly. At the moment, this notification occurs "out of band", outside the scope of this system. A consequence of this is that if a recipient does not receive this notification and fails to update its balance, it will not be able to issue transfers, as the proof of assets will be incorrect. To solve this, we could attach the recipient information to the private transfer itself. Ideally this would be encrypted such that only the recipient can decrypt it, whilst any party should be able to assess the consistency of the encrypted piece of data with the rest of the private transfer. This is to prevent a validator accepting transfers to the ledger in which the issuers give the recipients incorrect information and thereby destroying the recipients' ability to issue new transfers.

Generalizing to arbitrary multiparty financial workflows In this paper we show how example of how a specific multiparty financial workflow can be implemented on top of zkLedger. We would like to be able to generalize this construction to arbitrary workflows. In the system described in this paper, which information each party has differs. While there is a shared state, namely the ledger, the access to the unencrypted information differs per party involved. A potential candidate for a language to reason about the transfer of information in these workflows is dynamic epistemic logic[7].

References

1. Adjoint Inc: Bulletproofs library. https://github.com/adjoint-io/bulletproofs (2018)
2. Adjoint Inc: Schnorr NIZK library. https://github.com/adjoint-io/schnorr-nizk (2018)
3. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: Proceedings of the 1st ACM conference on Computer and communications security. pp. 62–73. ACM (1993)
4. Boneh, D.: The Decision Diffie-Hellman problem. In: International Algorithmic Number Theory Symposium. pp. 48–63. Springer (1998)
5. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more (2018)
6. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Annual International Cryptology Conference. pp. 174–187. Springer (1994)

7. Dechesne, F., Wang, Y.: Dynamic epistemic verification of security protocols: Framework and case study. In: Proceedings of the Workshop on Logic, Rationality and Interaction, Texts in Computer Science. pp. 129–144 (2007)
8. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Official Journal of the European Union **L119**, 1–88 (May 2016)
9. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Advances in Cryptology – CRYPTO'86. pp. 186–194. Springer (1986)
10. Maurer, U.: Unifying zero-knowledge proofs of knowledge. In: International Conference on Cryptology in Africa. pp. 272–286. Springer (2009)
11. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
12. Narula, N., Vasquez, W., Virza, M.: zkLedger: Privacy-Preserving Auditing for Distributed Ledgers. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). USENIX (2018)
13. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Annual International Cryptology Conference. pp. 129–140. Springer (1991)
14. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of cryptology **4**(3), 161–174 (1991)
15. Setty, S.T., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: USENIX Security Symposium. pp. 253–268 (2012)