MERCAT:
Mediated, Encrypted, Reversible, SeCure Asset Transfers

Aram Jivanyan and Jesse Lancaster, with the assistance of Arash Afshar and Parnian Alimi

Polymath
aram@polymath.network, jesse@polymath.network, arash@polymath.netowrk, parnian@polymath.network
https://polymath.network
15 Sep, 2020
Version 1.0.1

Abstract. For security token adoption by financial institutions and industry players on blockchain, there is a need for a secure asset management protocol which enables confidential asset issuance and transfers by concealing from the public the transfer amounts and asset types, while on a public blockchain. Flexibly supporting arbitrary restrictions on financial transactions, only some of which need be supported by zero-knowledge proofs. This paper proposes leveraging a hybrid design approach, by using zero-knowledge proofs, supported by restrictions enforced by trusted mediators. As part of our protocol, we also describe a novel transaction ordering mechanism which can support flexible transaction workflow without putting any timing constraints on when the transactions should be generated by the users or processed by the network validators. This technique is likely to be of independent interest.
Keywords: Confidential asset management, one-out-of-many proofs

1 Introduction

Financial instruments have many different desirable and required traits when compared to currency (and currency equivalent) digital representations in the context of privacy. This is important since it drives many of the design trade-offs when dealing with different types of assets.

Perhaps the two foremost differences are:

- While there are many arguments to be made for varying levels of anonymity and identity when it comes to currency, with securities identity needs to be strongly established, however equity positions are private. This is a different set of requirements than many other blockchain privacy use cases.
- Ownership of securities is mediated by third parties, and reporting of securities ownership by the issuer requires ongoing transparency of ownership.

  Securities require the issuer to be able to report on the distribution and holdings of the asset to varying degrees. This is very different from currency.

To satisfy those requirements, in this paper we provide a brief technical overview of a zero-knowledge proof (ZKP) based proposal for securely managing assets on a public blockchain in an auditable way. Asset issuance and transfers enabled by cryptographic proofs could be verified by all network participants rather than only selected parties. MERCAT represents a novel protocol specially designed for security token use cases, and even more specifically for the Polymath Blockchain, Polymesh. Its design is inspired by different ideas used in the Zether [1], AZTEC [8], PGC [2], ZkVM [17] and Lelantus [16] protocols. Several of the cryptographic ideas could be of interest for other use cases as well.

The asset data structure in abstract, as well as the mediation and transfer restriction check process will be described in brief prior to describing the technicalities of the confidential transfer itself.

1.1 Contribution

We describe a zero-knowledge proof protocol enabling confidential asset management which hides the transfers amounts and asset. Our design could be extended to further obfuscate a transaction's nature and ensure a transaction's anonymity (concealing the participants).

The protocol provides:

- Transfer Amount Confidentiality: Transfer amounts are always kept obfuscated and the user provides ZKP to initiate the transfers. This enables the confidentiality of the exact amount of security tokens exchanged between trading parties.
- Reversible Transfers: The transfers can be easily reversed by trusted mediators without any need of interaction with the transfer originator or the receiver. In this case 2 transactions would be recorded on-chain, and confidentiality in terms of the amount and asset transferred is maintained.
- Asset Confidentiality: The asset traded can also be kept confidential. This will enable investors to maintain private portfolios. To the best of our knowledge, ZkVM is the only cryptographic construction which supports the confidentiality of the asset. Unlike ZkVM, which is based solely on zero-knowledge cryptography without trusted intermediaries, our design requires the existence of trusted mediators, which help to initially issue the assets.
- Audit-ability: Users could provide audit rights to the trusted parties which could view transaction details without removing the privacy of those transactions or learning any other user secrets such as the cryptographic private keys or asset spending keys.

We highlight the following important differentiation of our protocol to Zether and PGC. (i) Unlike Zether, we do not use the Σ-bullets construction to provide range proofs for ElGamal encrypted values but instead use the Twisted ElGamal encryption introduced by [2], for its performance and compatibility with Bulletproofs[22]. (ii) We discuss how to transact over multiple asset types and keep the transaction asset types confidential (iii) We discuss transactions' audit-ability by trusted mediators.

Some key points of our protocol are:

- **Zero-knowledge proofs:** Different zero-knowledge primitives (such as Range proofs [22], One-out-of-Many proofs [19], proofs of ciphertext equivalency [13], [4], etc.) are used as standalone building blocks to support the confidential asset transfers, issuance, and spending processes. This eliminates the risk of double-spending attacks without any trusted third-party involvement.
- **No trusted setup:** We use standard-based cryptography which does not rely on any trusted setup procedures like Nightfall or AZTEC. Importantly, our construction does not require building and compiling complex arithmetic circuits like Zerocash [11], AZTEC or Nightfall protocols require. This makes it significantly easier to implement, test and cryptanalyze all cryptographic building blocks.
- **Users are Stateless:** There is no need to maintain a state – information on demand can always be computed from data on the blockchain, for example to view balances or review transaction history.

1.2 Design Concepts

We designed our scheme in the account-based model, assuming that for each financial transaction there is an associated trusted mediator having some control over the transaction's validation. This is not a new concept when dealing with securities, as (generally speaking) all securities transfers involve the oversight satisfaction of various transfer restrictions via legal representatives, exchanges, brokers, etc. In most jurisdictions, it is not viable for two holders of equity to trade that equity without the involvement of a third party (effectively a trusted mediator).

Additionally, the issuer of the asset must be able to report on and validate ownership of their assets at any time.

We want to strike a balance between enabling progressive use of ZKPs to check restrictions (as development of ZKPs for these use cases progresses), while still allowing restrictions that cannot be handled currently by ZKPs to be validated by the mediator.

UTXO Model vs Account-based Model Many proposals for making confidential transactions use homomorphic Pedersen commitments (Confidential Transactions, Lelantus, Zcash, etc.) to encode the asset value and its ownership. When the asset is transferred, the original commitment is invalidated, and a new commitment is created for the

recipient. Though this approach is quite simple and efficient, the opening of these commitments must be transferred to the receiver so that they can spend the asset later. This randomness could be stored on-chain in some encrypted manner or sent directly through a separate out-of-band channel, but either scenario requires the asset owner to manage access for all his UTXO secrets.

Account-based model takes a simpler approach by using homomorphic public key encryption scheme, where each user account is associated with a special ciphertext encrypted under the user public key. The value encoded by this ciphertext will correspond to the account balance and this ciphertext can be updated upon new transfers as the encryption scheme has linear encoding properties. With the exception of the account balance, each account can be associated with an asset type, which in turn can be represented through the second ciphertext. We will utilize a Twisted ElGamal algorithm and its homomorphic properties to create efficient ZKPs of correct value encryption and transaction validity.

A similar account-based approach is used in several protocols [5] including the Zether[1] and PGC [2].

Asset Identifiers We assume each asset/token has a unique identifier and the list of all asset identifiers is public. The users can refer to this list immediately upon creating new accounts. Every time a new asset is issued, the list of identifiers is updated.

This is similar to the idea of a stock ticker today.

In order to support asset confidentiality, we assume that the asset registration step (when the user registers the asset name and the ID is created on-chain) and its initial issuance step (when the tokens are created and assigned to an identity) is logically separated because the asset ID need to be available to create the confidential balance.

Trusted Mediators One of the complex areas of confidential transfer design when it comes to security tokens is how to ensure data privacy, while also revealing sufficient information to satisfy transfer restrictions. Securities transfer restrictions often require data outside the particular transaction occurring to determine if the transactions can proceed, for example restrictions on maximum percentage of ownership, time-based volume restrictions, etc. While there are many common restrictions, it is also desirable to allow issuers to enforce more uncommon restrictions. Additionally there may be restrictions based on off-chain events, such as receiving fiat payments. Since ZKPs require some design and development work for each case they want to validate, and some data is not available on-chain, any approach relying solely on ZKPs will end up with a very limited set of supported restrictions.

To improve this situation, we have capitalized on the fact that for securities trading, current regulation requires that there are always third parties involved. For a private company this third party may simply be the asset issuer for some assets. However it could also be an exchange, a broker dealer, transfer agent, registrar, etc. Today these parties perform some verification of data and approval of transactions each time a trade takes place, even if some or all of those checks are automated.

For MERCAT we use the same model, where to initiate a trade, a third party must participate in the initiation of the proceeding and sign off that all restrictions have been met, though this may be accomplished automatically in the case of an exchange for example.

If the trade restrictions specified by the issuer are all fulfilled with ZKPs, nothing further needs to be done. However in the future, we expect that most assets will require some validation that cannot be satisfied through ZKPs, and must be satisfied by the mediator.

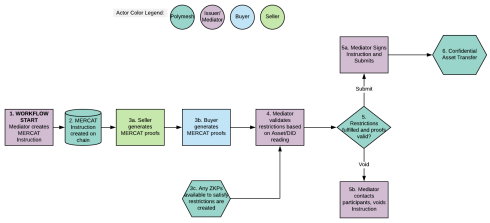
Transfer Restrictions The cryptography described in section 3 is related to the final step in the trade workflow, the actual confidential transfer of assets.

While we have mentioned in the previous section that some transfer restrictions may be satisfied by ZKPs, we will not discuss which restrictions will be satisfied in that way or the specific proofs to satisfy particular restrictions here. Since transfer restrictions are an open ended topic and each restriction must be handled differently depending on data input required, transfer restriction proofs will be discussed in other documents.

2 Transaction Workflow

The full workflow of the ideas we have discussed is best visualized below.

The cryptography discussed in section 3 onward is related to steps 3, 5, and 6.



Confidential Asset Data Structure There are many data points that may be stored in relation to a confidential asset on-chain, however outlining those potential values is not in the scope of this paper.

To secure asset data outside of the trading workflow, we can make use of simple public key cryptography.

The transaction data may optionally be written into various state representations, for example a current cap table so that state does not need to be calculated from raw transactions every time. This intermediate data is encrypted using a symmetrical encryption key by the issuer. Since this data could be used for non-ZKP transfer restrictions, the issuer provides copies of the symmetric key for each mediator, encrypting the symmetric key with the mediators public key (a similar approach to PGP).

2.1 Issuer Reporting

It's assumed that report generation software is off-chain (a dApp or other software). Since the issuer can see the details of all transactions, they can construct any kind of reporting they see fit.

2.2 Buyer and Seller Workflow

In the majority of cases, the buyers and sellers of confidential securities would not be aware of the mechanics of this protocol.

For example, an asset listed on an exchange. The seller would indicate on the exchanges platform that they have security tokens of type *acme* to sell. Once the exchange has a buyer, it can create the instruction and validate the transfer restrictions without any interaction between the buyer and seller except perhaps digitally signing the transaction and proof generation.

The workflow is similar for fundraising (where the entity initiating the trade contract would be a broker dealer, custodian or even the asset issuer) or trade between asset holders of assets not listed on an exchange (where likely the asset issuer would initiate the trade contract). The main differences lie in the mediators level of automation; an exchange is likely to be a very automated process, where a small issuer might manually perform the required actions using client software.

2.3 Dealing with Delays and Transaction Ordering

The primary issue created by doing off-chain review of encrypted assertions then writing a completion transaction is one of ordering. If a smart contract is performing all validations of all on-chain data, the state at the time of the checks and the state at the time of the transfer of value is the same. However with off-chain validation, you can have the following situation:

- An asset exists with an asset wide volume restriction of 1000 units every 24 hours.
- 15 trade contracts are created, trading 100 units each.

- All 15 contracts are valid in and of themselves, however 5 of them should be rejected.

So ordering of transactions even in a block is relevant to validity.

We go into more detail on transaction ordering in section 7.

2.4 Evolution of the Design

One advantage of the design is that in the course of technical evolution, as ZKPs become available, the basic workflow and structure still functions, just some (or all) of the assertions are checked on-chain. At the point where none of the assertions require mediator validation, the process becomes self serve since the trade contract could be created and fulfilled without the mediator.

So if we look forward to how this design might be evolved to remove some of the current trade-offs, the primary answer is in designing ZKPs so that as much of the trade contract assertions as possible can be validated on-chain without needing certification or even exposure (in the case of an exchange) to the mediator.

3 Cryptographic Background

In this section we describe the cryptographic building blocks and the base security assumptions we are relying on to initiate our confidential asset transfer system.

3.1 Security Assumptions

Let Gen be a Probabilistic Polynomial Time (PPT) algorithm that on input a security parameter 1^λ, outputs description of a cyclic group G of prime order $p = O(2^\lambda)$, and a random generator g for G. We describe the discrete-logarithm based assumptions related to $(G, g, p) \leftarrow Gen(1^\lambda)$.

Definition 3.1 (Discrete Logarithm Assumption). The discrete logarithm assumption holds if for any PPT adversary A, we have:

$$Pr[A(g, h) = a \quad s.t. \quad g^a = h] \leq negl(\lambda),$$

where the probability is over the randomness of $Gen(1^\lambda)$, A's random tape, and the random choice of $h \xleftarrow{R} G$.

Definition 3.2 (Decisional Diffie-Hellman Assumption). The DDH assumption holds if for any PPT adversary, we have:

$$Pr[A(g, g^a, g^b, g^{ab}) = 1] - Pr[A(g, g^a, g^b, g^c) = 1] \leq negl(\lambda)$$

where the probability is over the randomness of $Gen(1^\lambda)$, A's random tape, and the random choice of $a, b, c \xleftarrow{R} Z_p$.

Definition 3.3 (Divisible Decisional Diffie-Hellman Assumption). The divisible DDH assumption holds if for any PPT adversary, we have:

$$Pr[A(g, g^a, g^b, g^{\frac{a}{b}}) = 1] - Pr[A(g, g^a, g^b, g^c) = 1] \leq negl(\lambda)$$

where the probability is over the randomness of $Gen(1^\lambda)$, A's random tape, and the random choice of $a, b, c \xleftarrow{R} Z_p$. As proved in [12], the divisible DDH assumption is equivalent to the standard DDH assumption.

The security of our asset transfer system is based on these security assumptions.

3.2 Commitments

A non-interactive commitment scheme is a pair of probabilistic polynomial time algorithms $(G; Com)$. All algorithms in our schemes get a security parameter λ as input written in 1^λ. The setup algorithm $ck \leftarrow G(1^\lambda)$ generates a commitment key ck which specifies the message space M_{ck}, a randomness space R_{ck} and a commitment space C_{ck}. The commitment algorithm combined with the commitment key specifies a function $Com_{ck} : M_{ck}$ x $R_{ck} \rightarrow C_{ck}$. Given a message $m \in M_{ck}$ the sender picks uniformly at random $r \xleftarrow{R} R_{ck}$ and computes the commitment $C = Com_{ck}(m; r)$. We require the commitment scheme to be both hiding and binding. Informally, a non-interactive commitment scheme $(G; Com)$ is hiding if a commitment does not reveal the committed value. Formally, we require for all probabilistic polynomial time state full adversaries A

$$Pr[ck \leftarrow G(1^\lambda); (m_0, m_1) \leftarrow A(ck); b \leftarrow \{0, 1\}; c \leftarrow Com_{ck}(m_b) : A(c) = b] \approx \frac{1}{2}$$

where A outputs $(m_0, m_1) \leftarrow M_{ck}$. If the probability is exactly $\frac{1}{2}$ we say the commitment scheme is perfectly hiding.

A non-interactive commitment scheme $(G; Com)$ is strongly binding if a commitment can only be opened to one value. Formally we require

$$Pr[ck \leftarrow G(1^\lambda); (m_0, r_0, m_1, r_1) \leftarrow A(ck) : (m_0, r_0) \neq (m_1, r_1) \wedge Com_{ck}(m_0; r_0) = Com_{ck}(m_1; r_1)] \approx 0$$

where A outputs $(m_0, m_1) \leftarrow M_{ck}$ and $(r_0, r_1) \leftarrow R_{ck}$. If the probability is exactly 0 we say the commitment scheme is perfectly binding.

The Pedersen commitment scheme [25] is an additively homomorphic commitment scheme, which is perfectly hiding and computationally strongly binding under the discrete logarithm assumption. The key generation algorithm G outputs a description of a cyclic group G of prime order p and random generators g and h. The commitment key is $ck = (G, p, g, h)$. To commit to $m \in Z_p$ the committer picks randomness $r \in Z_p$ and computes $Com_{ck}(m; r) = h^m \cdot g^r$.

Note: We will henceforth refer to the Pedersen commitment for value m using randomness r as $Com(m; r)$.

3.3 Σ-Protocols

Σ-protocols are 3-move interactive protocols that allow a prover to convince a verifier that a statement is true. The prover sends an initial message, the verifier responds with a random challenge, and the prover sends a response to the verifier. At the end of the interaction, the verifier looks at the transcript and decides whether to accept or reject the proof that the statement is true. A Σ-protocol should be complete, sound and have special soundness in the following sense:

- **Perfect Completeness**: If the prover knows a witness w for the statement s then they should be able to convince the verifier. Formally, for any $(s, w) \in R$ we have $\Pr[\text{Verify}(ck, s, \text{Prove}(ck, s, w))] = 1$ meaning that the verifier will accept all valid transcripts.
- **Special honest verifier zero-knowledge (SHVZK)**: The Σ-protocol should not reveal anything about the prover's witness. This is formalized as saying that given any verifier challenge x it is possible to simulate a protocol transcript.
- **n-Special Soundness**: If the prover does not know a witness w for the statement, they should not be able to convince the verifier. This is formalized as saying that if the prover can answer n different challenges satisfactorily, then it is possible to extract a witness from the accepting transcripts. For any statement s and from n accepting transcripts $\{a, x_i, z_i\}_{i=1}^n$ for the $s \in L_R$ with distinct challenges x_i, the witness w can be extracted s.t. $(s; w) \in R$.

An advantage of Σ-protocols is that they are easy to make non-interactive by using the Fiat-Shamir heuristic [7] where a cryptographic hash-function is used to compute the challenge instead of having an online verifier. It can be argued in the random oracle model where the hash-function is modeled as a truly random function that this gives us secure non-interactive zero-knowledge proofs.

We will use numerous non-interactive zero-knowledge proofs as building blocks for constructing our payment system as described below.

3.4 Schnorr Signatures

Like in [2], we choose Schnorr signature [4] as the signature component as its setup and key generation algorithms are almost identical to those of twisted ElGamal public key encryption scheme which yields to efficient instantiation of integrated signature and encryption scheme.

The Schnorr digital signature scheme consists of four polynomial time algorithms as follows.

- $Setup(1^\lambda)$: Run the $Gen(1^\lambda)$, pick a cryptographic hash function $H : M \text{ x } G \to Z_p$ where M denotes the message space. Output the public parameters as $pp = (G, g, \mathbf{H}, p)$.
- $KeyGen(pp)$: on input pp, chose a secret key $sk \xleftarrow{R} Z_p$, and set the public key pk as $pk = g^{sk}$.
- $Sign(sk, m)$: on input the secret key sk and a message $m \in M$: pick random $r \xleftarrow{R} Z_p$, set $A = g^r$, compute $e = H(m, A)$, $z = r + sk \cdot e \bmod p$ and output the signature $\sigma = (A, z)$.
- $Ver(pk, m, \sigma)$: parse $\sigma = (A, z)$, compute $e = H(m, A)$, output '1' if $g^z = A \cdot pk^e$ and '0' otherwise.

Correctness: For all $pp \leftarrow Setup(1^\lambda)$, all $(pk, sk) \leftarrow KeyGen(pp)$ and all $m \in M$ it holds that $Ver(pk, m, Sign(sk, m)) = 1$.

The security of Schnorr signature is based on the discrete logarithm assumption[13].

3.5 Twisted and Const-Time ElGamal Algorithms

We will use a modified version of ElGamal encryption algorithm as is discussed in [2] and referred to as twisted ElGamal. Twisted ElGamal ensures the second component of the ciphertext is free of obvious trapdoor. In addition to g, we pick another generator h which is orthogonal to g. The global public parameters of the encryption systems are (G, g, h, p).

The twisted ElGamal public key encryption scheme consists of four polynomial time algorithms as follows.

- $Setup(1^\lambda)$: Run the $Gen(1^\lambda)$, chose another generator element $h \xleftarrow{R} G$. Output $pp = (G, g, h, p)$ as global public parameters.
- $KeyGen(pp)$: on input pp, chose a secret key $sk \xleftarrow{R} Z_p$, and set the public key pk as $pk = g^{sk}$.
- $Enc(pk, m)$: on input the public key pk and a message $m \in M$: pick random $r \xleftarrow{R} Z_p$, compute $X = pk^r$, $Y = g^r \cdot h^m$, and output the ciphertext $C = (X, Y)$.
- $Dec(sk, C)$: parse $C = (X, Y)$ and compute $h^m = \frac{Y}{X^{sk^{-1}}}$. Next the value h^m should be brute-forced to recover m.

Correctness. For all $pp \leftarrow Setup(1^\lambda)$, all $(pk, sk) \leftarrow KeyGen(pp)$ and all $m \in M$ it holds that $Dec(sk, Enc(pk, m)) = m$ for any $r \xleftarrow{R} Z_p$.

Let's denote $\circ$ as the multiplication operation of two ciphertexts defined as

$$(X_1, Y_1) \circ (X_2, Y_2) = (X_1 \cdot X_2, Y_1 \cdot Y_2)$$

Twisted ElGamal encryption retains the additive homomorphic properties of the ciphertexts encrypted under the same public key. Assuming $Enc(Y, m_1) = (Y^{r_1}, h^{m_1} g^{r-1})$ and $Enc(Y, m_2) = (Y^{r_2}, h^{m_2} g^{r_2})$, then

$$Enc(Y, m_1) \circ Enc(Y, m_2) = (Y^{r_1+r_2}, h^{m_1+m_2} g^{r_1+r_2}) = Enc(Y, m_1 + m_2)$$

In our system design, the advantage of Twisted ElGamal over the regular ElGamal scheme is the fact that the second component of the ciphertext can be viewed as a Pedersen commitment for the message m under the randomness r(which is secure, as there is no known discrete logarithm relationship between the public generators g and h). Thus, we can safely invoke Bulletproofs over the value $g^r h^m$ to produce a range proof for the value m.

As shown in [2], the twisted ElGamal encryption is "chosen-plaintext" secure under the divisible DDH (Decisional Diffie-Hellman) assumption, which is equivalent to the standard DDH assumption.

Enabling Constant-Dime Decryption. Twisted ElGamal decryption process requires brute-forcing the message value m. It would be highly inefficient scheme for secret sharing where the additive homomorphic properties of the ciphertexts are dispensable. For such applications, we can build a constant time decryption process by encoding a random pad through the Twisted ElGamal scheme and then use it for concealing the actual message. These changes require addition of an extra element to the ciphertext but will result in fast and constant time decryption procedure. In our protocol we will use both the Twisted ElGamal and its modified version which ensures constant-time decryption let's name the constant time encryption and decryption algorithms as *cEnc* and *cDec*. The *Setup* and *KeyGen* functions for this modified version remain the same however the *cEnc* and *cDec* works as follow:

- $cEnc(pk, m)$: on input the public key pk and a message $m \in M$: pick two random values $r, s \xleftarrow{R} Z_p$, compute $X = pk^r$, $Y = g^r \cdot h^s$, $Z = \mathcal{H}(h^s) \oplus m$ and output the ciphertext $C = (X, Y, Z)$.
- $cDec(sk, C)$: parse $C = (X, Y, Z)$ and compute $h^s = \frac{Y}{X^{sk^{-1}}}$. Next output the plaintext message as $m = \mathcal{H}(h^s) \oplus Z$.

3.6 Ciphertext Refreshment Method

For generating certain zero-knowledge proofs over the given ciphertext, one may need the randomnesses used in its cryptographic computation. As we aim to keep the users stateless, these random factors are not necessary stored by the users. This method of ciphertext refreshment was first discussed in the PGC design [2]. The idea of this method is that user can create a new ciphertext which provably encodes the same message under the same public key as the original ciphertext, but uses different randomnesses. The knowledge of this freshly generated randomness can be used further to prove certain properties of the ciphertext in a zero-knowledge manner.

Refresh

- Inputs: A twisted ElGamal ciphertext $(C, D) = (y^r, h^m g^r)$, the private key x and the public key $y = g^x$
- Output: A pair of new ciphertext and proof of ciphertexts equality: $((C', D'), \pi_{equal})$

1. Decrypts the ciphertext(C, D) and reveals the encoded value $h^m = \frac{D}{C^{x^{-1}}}$
2. Picks a new randomness $r' \xleftarrow{R} Z_p$ and encodes the value h^m as $(C', D') = (y^{r'}, h^m g^{r'})$
3. Generates a non-interactive zero-knowledge proof π_{equal} of knowledge of two ciphertexts (C, D) and (C', D') are encrypting the same value h^m under the same public key y: The π_{equal} generation algorithm is detailed in the section 5.3

3.7 One-out-of-Many Proofs for a Commitment Opening to 0

In [19], a Σ-protocol is provided for knowledge of one out of N Pedersen commitments $C_0, \ldots, C_{N-1}$ is opening to 0. More precisely a Σ-protocol for the relation

$$R = \{(ck, (C_0; \ldots; C_{N-1}); (l, r) \quad | \quad \forall i : C_i \in C_{ck} \wedge l \in \{0, \ldots, N-1\} \wedge r \in Z_p \wedge C_l = Com_{ck}(0, r))\}$$

The protocol described in [19] was further optimized in [22] to reduce the proof sizes and hasten the proof generation process. The one-out-of-many proofs are perfectly complete, special honest verifier zero-knowledge, and special sound as is proven in[19] We are leveraging this construction to build membership proof to show a given ciphertext is encrypting some element from the specified set without revealing which one. We refer to the papers [19] and [22] for more details of protocol.

4 Confidential Asset Management System with Accountability

Each step in the asset creation or transfer workflow should be supported by a specified cryptographic algorithm. Below we provide a high level description of all cryptographic algorithms which are used to empower the whole protocol.

4.1 Transparent Setup

Benefiting from our dedicated design, the global parameters of MERCAT are exactly the public parameters of the Bulletproofs and 1-out-of-Many proofs, a.k.a. simply random generators $(g, h, \mathbf{g}, \mathbf{h})$ without special structure. We can thus use hash function to dismiss trusted setup. We refer to [19] and [23] for more details on the transparent setup parameters.

4.2 Overview of MERCAT Asset Transfer Functions

We present a general framework for accountable, confidential transactions hiding transaction amounts and types as below.

1. **CreateAddress**: CreateAddress enables the user to create a cryptographic address in the blockchain that further will uniquely identify the user's identity in the system. It takes the security parameter as input and outputs a secret key sk and a public key pk compatible with the twisted ElGamal encryption scheme. User public keys will be subsequently associated with his DID (Note, that DID can be associated with multiple keys used for different purposes such as encryption, digital signature etc). The user can create as many addresses as he wants.

2. **CreateAccount(Address, Asset)**: This algorithm enables the user creating a new account associated with the certain asset and encoding its corresponding balance, which initially can be 0. Users can create empty accounts for all assets. They can create a new account and prove it corresponds to a valid asset without revealing which asset. Each account is associated with

   - The user address.
   - Asset Identifier. This is represented through an ElGamal ciphertext C_A which is generated once during the account creation, is immutable and hides the asset identifier.
   - Account balance. This is represented through ElGamal ciphertext $C_B$which hides the account balance and is subject to change upon new transactions. The account remainder can not be negative.
   - A zero-knowledge proof that the encoded asset type actually corresponds to a valid asset identifier listed publicly. We assume the list of all asset identifiers is public.

   Thus each account will be represented through a tuple of *(Address, Encrypted Asset Identifier, Encrypted Account Balance)* and will be associated with augmentary cryptographic proofs ensuring the account validity. The user should have a separate account for different assets similar to regular bank accounts for different currencies.

3. **RevealBalance(Account, Amount, Asset Identifier)**: This algorithm is used to publicly reveal the account balance and its asset. It takes as input the secret key sk, the current encrypted balance C_B and the encrypted asset C_A and outputs the balance and account type as $b \leftarrow Dec(sk, C)$ and $aID \leftarrow Dec(sk, C_A)$. This algorithm may also provide a proof of valid decryption.

4. **IssueAsset(Issuer Address, Asset Identifier, Amount)**: This algorithm enables the Issuer to generate the cryptographic representation of the issued assets. Each asset is associated with a unique identifier aID. The issuer can issue more of an asset later increasing the overall accessible amount of that asset. Let's assume the issuer's public key is Y, the issued asset identifier is aID, and the issued amount is N. Similar to the **CreateAccount** algorithm, this algorithm will generate.

   - An ElGamal ciphertext which encodes the issued amount $(C^v, D^v) = (g^r, h^N g^r)$
   - A zero-knowledge proof that the ciphertext (C^v, D^v) correctly encodes the specified value N.

Asset issuance process should be overseen by one of trusted mediators who will be able to verify the legitimacy of the issuance process and check the process against different legal requirements. The issuer shares the asset type, the number of issued assets as well as auxiliary compliance rules with the mediator privately. Note that apart of the mediator, one or more auditors also could be associated with the asset issuance or transfer operation for auditing the process. The mediator has the power to approve or reject the transaction when it violates some compliance rule, while the auditors can only passively observe the transaction for auditing reasons.

5. **CreateCTx**(Sender Account, Recipient Account, Asset Identifier, Amount): This algorithm is used to transfer a certain amount of a given asset from the Sender's account to the Recipient's account without revealing any information about the transferred amounts or asset type. Suppose a user wants to transfer an amount b of asset aID from an account associated with his public key X to an account associated with the public key X'. The transfer operation should deduct the amount b from X's balance and add the same amount to X''s balance. Since we need to hide the transaction amount b in this process, sender will encrypt b under both the public keys X and X'
   - New created ciphertexts are well-formed and both encrypt the same value.
   - The ciphertexts encode a positive value.
   - After deduction, the sender account still holds a non-negative balance.
   - The transfer has happened between accounts of the same asset type, meaning the recipient's account type is identical to the transferred asset type.

   It is important to mention that the **CreateCTx** transaction is initiated by the sender but should be finalized by the recipient, as the proof of asset type equivalency is generated through two steps by both the sender and the recipient. After this transaction is finalized, the recipient's balance will be increased and the sender balance will be deducted of the same amount. The output of this algorithm is an initialized confidential transaction data ctx_{init} comprised of all necessary zero-knowledge proofs, a transaction memo which includes the sender and recipient account data along with other cryptographic operands, the transaction serial number or its ordering state and the sender's signature as will be discussed in detail later. Confidential transaction creation process also should be overseen by one of trusted mediators who will be able to verify the legitimacy of the transfer and check the transaction against different legal requirements. The sender shares the asset type, the number of transferred assets as well as auxiliary compliance rules with the mediator privately. Like in the issuance transaction, one or more auditors also could be associated with the confidential transfer for auditing the process. When auditors are assigned, the sender again shares the confidential transfer secret data with the auditors privately by encrypting the asset type, the transferred amount and relevant compliance rules with the auditor's public key. Along with the private data, the sender should also generate specific zero-knowledge proofs that the transaction payload data is actually matching with the data shared privately with the mediator or the auditors.

6. **FinalizeCTx**(Recipient Private Key, Initialized Transaction data ctx_{init}). This algorithm is used by the recipient to finalize the confidential transaction generation process which has been initiated by the sender. The recipient approves the transaction by signing it with his own private key but he also includes the proof of transaction validity by computing the last component of the zero-knowledge proof which proves that sender and recipient's accounts types are matching.

7. **VerifyCTx**(A Finalized Confidential Transaction Data): This algorithm is used to check if a finalized confidential transaction ctx_{final} is valid. It takes as input $ctx_{final} = (sn, memo, \pi_{valid}, \sigma_s, \sigma_r)$ and checks its validity. If the validity checks passes, the algorithm outputs "1". Validators confirm that ctx is valid and records it on the blockchain by engaging some consensus protocol after which both the sender and recipient balances are updated. If any validity test fails, the algorithm outputs "0" and validators discard this transaction.

8. **ReverseCTx**(A Finalized Confidential Transaction Data): In some scenarios like identity theft or court order it might be required to reverse certain confidential transactions in a zero-knowledge manner. Due to the homomorphic properties of the underlying public key encryption scheme, the transaction can be reversed and the balances of the sender and recipient can be updated without any interaction from the transaction recipient.

9. **JustifyCTx**(A Finalized Confidential Transaction Data) Transaction audit-ability can be achieved using zero-knowledge techniques. This algorithm is used by the auditors or mediators to justify that the value hidden in

transaction is indeed the claimed value v or the transaction relates to the claimed asset type. Here, we assume that the confidential transaction is valid and recorded on the blockchain. The auditor has to learn the exact number of the transferred shares and the asset type for a certain transaction but the sender may not want to reveal his private keys to the auditor for privacy reasons. Indeed the sender can do the following

- Share the encoded values v and aID with the auditor.
- Prove in a zero-knowledge way that the corresponding ciphertexts in the transaction are actually encrypting these claimed values.

MERCAT enables the specified mediator and third-party auditors to justify the validity and correctness of the submitted transactions. Different transactions can be assigned to different mediators and auditors and many auditors can be associated with a single transaction. In a broader context, the mediator's role is verifying that all submitted transactions stay compliant with the assigned transfer restrictions and he reserves the right to terminate any transaction which does not satisfy some restriction. The auditor role is mostly limited by having a view-only access to the transactions details.

5 Cryptographic Details of Zero-Knowledge Proofs

The sigma protocol is comprised of three algorithms $\Sigma_{enc} = (Setup, Prove, Verify)$.
The *Setup* algorithm specifies the cryptographic group and generators. Note that the same cryptographic group and generators are also used for the twisted ElGamal algorithm. In this section we detail all sigma protocols which will serve as building blocks in our payment system.

5.1 Proof of knowledge of the encrypted value

The non-interactive zero-knowledge proof for value encryption proves the knowledge of value v encrypted by a twisted ElGamal ciphertext (X, Y) under the public key pk and a secret randomness r. More formally, we define the language L_{enc} as

$$L_{\text{enc}} = \{(pk, (X, Y))|\exists r, v \text{ s.t. } X = pk^r \wedge Y = h^v g^r\}$$

For a given instance $(pk, (X, Y))$ of public key and a twisted ElGamal ciphertext, the protocol Σ_{equal} proves that $(pk, (X, Y)) \in L_{\text{enc}}$

Sigma-protocol for L_{enc}. On the given statement $(pk, (X, Y))$, the prover P and verifier V engage into the following steps:

1. P selects random values $a, b \xleftarrow{R} Z_p$
2. P computes $A = pk^a$ and $B = h^b g^a$ and sends to V
3. V sends a random challenge value $x \leftarrow (0, 1)^\lambda$
4. P computes $z_1 = a + xr$ and $z_2 = b + xv$ and sends z_1, z_2 to V
5. V accepts the final proof if and only if
   - $(pk)^{z_1} = A \cdot X^x$
   - $h^{z_2} g^{z_1} = B \cdot Y^x$

Let's denote the NIZKPoK for L_{enc} as π_{enc}. π_{enc} can be obtained by applying the Fiat-Shamir transform to the Σ_{enc}.

5.2 Proof of correct encryption of the given value

The non-interactive zero-knowledge proof for the value encryption proves the twisted ElGamal ciphertext (X, Y) correctly encrypts the public value v under the public key pk without revealing the randomness or the corresponding private key. More formally, we define the language L_{correct} as

$$L_{\text{correct}} = \{(pk, v, (X, Y)) \quad | \quad \exists r \text{ s.t. } X = pk^r \wedge Y = h^v g^r\}$$

For a given instance $(pk, v, (X, Y))$ of a public key, message and a twisted ElGamal ciphertext, the protocol $\Sigma_{correct}$ proves that $(pk, v, (X, Y)) \in L_{\text{correct}}$.

Sigma-protocol for L_{correct}. On the given statement $x = (pk, v, (X, Y))$, the prover P and verifier V engage into the following steps:

- Both P and V homomorphically subtract the value v from the value $Y = h^v g^r$ and compute the value $Y' = \frac{Y}{h^v}$
- Prover proves the discrete logarithm equality of elements X and Y' with respect to the generators pk and g through the following steps.
  1. P selects u $\xleftarrow{R} Z_P$
  2. P computes $A = pk^u$ and $B = g^u$ and sends A, B to the V
  3. V sends a random challenge value $x \leftarrow (0, 1)^\lambda$
  4. P computes $z = u + x \cdot r$ and sends it to V
  5. V accepts the final proof if and only if
     - $(pk)^z = A \cdot X^x$
     - $g^z = B \cdot Y'^x$

Let's denote the NIZKPoK for L_{correct} as π_{correct}. π_{correct} can be obtained by applying the Fiat-Shamir transform to the Σ_{correct}.

5.3 Proof of two ciphertext encrypting the same value under the same public key

This non-interactive zero-knowledge proof proves two ElGamal ciphertexts (X_1, Y_1) and (X_2, Y_2) are encoding the same value under the given public key pk. More formally, we define the language L_{equal} as

$$L_{\text{equal}} = \{(pk, (X_1, Y_1), (X_2, Y_2)) | \exists v, r_1, r_2 \text{ s.t. } (X_1 = pk^{r_1}, Y_1 = h^v g^{r_1}) \wedge (X_2 = pk^{r_2}, Y_2 = h^v g^{r_2})\}$$

Note that in this context the prover does not necessarily know the randomnesses r_1 and/or r_2 used to comprise these ciphertexts. This algorithm uses the secret key sk as a witness based on the observation that proving that ciphertexts (X_1, Y_1) and (X_2, Y_2) encode the same message under the same public key is equivalent to proving that

$$log_{\frac{Y_1}{Y_2}} \frac{X_1}{X_2} = log_g pk = sk. \tag{1}$$

For a given instance $(pk, (X_1, Y_1), (X_2, Y_2))$ of a public key and two twisted ElGamal ciphertexts, the protocol Σ_{equal} proves that $(pk, (X_1, Y_1), (X_2, Y_2)) \in L_{\text{equal}}$.

Sigma-protocol for L_{equal}. On the given statement $(pk, (X_1, Y_1), (X_2, Y_2))$, the prover P and verifier V engage into the following steps:

- Both P and V compute the values $X = \frac{X_1}{X_2} = pk^{r_1 - r_2}$ and $Y = \frac{Y_1}{Y_2} = g^{r_1 - r_2}$.
- Prover proves the discrete logarithm equality of elements X and pk with respect to the generators Y and g through the following steps.

1. P selects u $\xleftarrow{R} Z_P$
2. P computes $A = Y^u$ and $B = g^u$ and sends A, B to the V
3. V sends a random challenge value $x \leftarrow (0,1)^\lambda$
4. P computes $z = u + x \cdot sk$
5. V accepts the final proof if and only if
   - $Y^z = A \cdot X^x$
   - $g^z = B \cdot pk^x$

Let's denote the NIZKPoK for L_{equal} as π_{equal} which can be obtained by applying the Fiat-Shamir transform to the Σ_{correct}.

5.4 Proof of two ciphertexts encrypting the same value under different public keys

This non-interactive zero-knowledge proof proves two ElGamal ciphertexts (X_1, Y_1) and (X_2, Y_2) are encoding the same value under two given public keys pk_1 and pk_2. More formally, we define the language L_{cipher} as

$$L_{\text{cipher}} = \{(pk_1, pk_2, (X_1, Y_1), (X_2, Y_2)) | \exists v, r_1, r_2 \text{ s.t. } (X_1 = pk_1^{r_1}, Y_1 = h^v g^{r_1}) \wedge (X_2 = pk_2^{r_2}, Y_2 = h^v g^{r_2})\}$$

As discussed in [2], the same randomness r can be re-used when encrypting the same message for different recipients. In most cases of our payment system context the language L_{cipher} can be defined as

$$L_{\text{cipher}} = \{(pk_1, pk_2, X_1, X_2, Y) | \exists v, r \text{ s.t. } X_1 = pk_1^r, X_2 = pk_2^r \wedge Y = h^v g^r)\}$$

For a given instance $(pk_1, pk_2, X_1, X_2, Y)$ the protocol Σ_{cipher} proves that $(pk_1, pk_2, X_1, X_2, Y) \in L_{\text{cipher}}$.

Sigma-protocol for L_{cipher}. The sigma protocol is comprised of three algorithms $\Sigma_{correct} = (Setup, Prove, Verify)$. The *Setup* algorithm is the same discussed in the context of $\Sigma_{correct}$. On the given statement $(pk_1, pk_2, X_1, X_2, Y)$, the prover P and verifier V engage into the following steps:

- P selects a, b $\xleftarrow{R} Z_P$
- P computes $A_1 = pk_1^a$, $A_2 = pk_2^a$ and $B = h^b g^a$ and sends the values A_1, A_2, B to the V
- V sends a random challenge value $x \leftarrow (0,1)^\lambda$
- P computes $z_1 = a + x \cdot r$ and $z_2 = b + x \cdot v$ and sends z_1, z_2 to V.
- V accepts the final proof if and only if
  - $pk_1^{z_1} = A_1 X_1^x$
  - $pk_2^{z_1} = A_2 X_2^x$
  - $g^{z_1} h^{z_2} = B Y^x$

5.5 Bulletproofs

Bulletproofs are a powerful scheme for providing short and aggregatable range proofs.

Formally, let $v \in Z_p$ and let $V \in G$ be a Pedersen commitment to v using randomness r. Then the proof system will convince the verifier that $v \in [0, 2^n - 1]$. In other words, the proof system proves the following relation

$$L_{\text{range}} = \{C, n; \quad v, r \in Z_p \quad | \quad C = h^v g^r \wedge v \in [0, 2^n - 1]\}$$

For the original protocol details, we refer to the paper [23]. Bulletproofs are interactive protocols which can be made non-interactive by using the Fiat-Shamir heuristic in the random oracle model. Let's denote the NIZKPoK for L_{range} as π_{range}.

5.6 Membership Proofs Based on 1-out-of-Many Proofs

This proof enables to convince that the provided Pedersen commitment (can be modified to work with ElGamal ciphertexts as well) encodes a value from the given set without revealing which one.

- Inputs: A Pedersen commitment $C = h^v g^r$ and its witness (v, r), a public vector of values $(v_1, v_2, \cdots, v_N))$

1. Computes a new vector of commitments by homomorphically subtracting the values $(v_1, v_2, \cdots, v_N)$ from the commitment C

$$\frac{C}{Com(v_1, 0)}, \cdots, \frac{C}{Com(v_N, 0))} \tag{2}$$

2. Prove the knowledge of one out of N commitments $(C \cdot Com^{-1}(v_1, 0), \cdots, C \cdot Com^{-1}(v_N, 0))$ opening to 0 by using the 1-out-of-Many proofs [19]. Obviously, if the commitment C is opening to one of the given values $(v_1, v_2, \cdots, v_N)$, then one of the commitments $(C \cdot Com^{-1}(v_1, 0), \cdots, C \cdot Com^{-1}(v_N, 0))$ will be opening to 0.

6 Algorithm Implementations

We assume each user can be associated with multiple addresses, and each address can be associated with one or more accounts. Hence we describe these two algorithms separately.

CreateAddress

- Inputs: Security parameter λ
- Outputs: $sk = x \in Z_p, pk = y \in G$

1. $x \xleftarrow{R} Z_P$
2. $y = g^x$

This key pair is compatible both with Twisted ElGamal public key encryption algorithm and the Shnorr's signature scheme.

CreateAccount

- Inputs: User private key x, user public key $y = g^x$, asset identifier aID
- Outputs: $acct = [(C_y^a, D_y^a), (C_y^b, D_y^b), \pi_{account}, \sigma_{account}, memo]$

1. Generates $r, r' \leftarrow Z_p$
2. Computes the first ciphertext encoding the asset identifer: $(C_y^a, D_y^a) = Enc(y, aID) = (y^r, h^{aID} g^r)$
3. Generates a proof of knowledge of the encrypted value $\pi_1 = Prove(L_{\text{enc}}(y, (C_y^a, D_y^a))$
4. Computes the second ciphertext encoding the balance: $(C_y^b, D_y^b) = Enc(y, 0) = (y^{r'}, h^0 g^{r'})$. Note that new created account balance is initially 0.
5. Generates a proof of correct encryption $\pi_2 = Prove(L_{\text{correct}}(y, 0, (C_y^a, D_y^a))$
6. Generates a proof of valid asset encoding $\pi_{member}(D_y^a, (aID_1, aID_2, \ldots, aID_N))$.
7. Set $\pi_{account} = \pi_1 || \pi_2 || \pi_{member}$
8. Set memo $= y||$timestamp
9. $\sigma_{account} = Sign(x, (C_y^a, D_y^a), (C_y^v, D_y^v), \pi_{account}, \text{memo})$.
10. Return $acct = [(C_y^a, D_y^a), (C_y^b, D_y^b), \pi_{account}, \sigma_{account}, \text{memo}]$

CreateAssetIssuanceTx

- Inputs: user private key x, user public key y, issued amount b, user account acct, trusted mediator's public key y_M,
- Output: $tx_{\text{issued}} = (C_A, C_B, (C_y^{*b}, D_y^{*b}), \pi_{issue}, \sigma_{issue})$

1. The user encrypts the asset identifier aID with the mediator public key using the modified version of Twisted ElGamal system ensuring constant-time decryption: $C_A = cEnc(y_M, aID)$
2. The user encrypts the number of issued assets b with the mediator public key: $C_B = cEnc(y_M, b)$
3. Set $memo = (C_y^{*b}, D_y^{*b}) = (y^r, h^b g^r)$
4. Generates a proof $\pi_{cipher} = \text{Prove}(L_{\text{cipher}}(y, y_M, acct[a], C_A)$. to show the user account type matches with the new issued asset type.
5. Generates a proof $\pi_{enc} = \text{Prove}(L_{\text{enc}}(y, (C_y^{*b}, D_y^{*b}))$ to prove the ciphertext (C_y^{*b}, D_y^{*b}) is well formed.
6. Generates a proof $\pi_{correct} = \text{Prove}(L_{\text{correct}}(y, b, (C_y^{*b}, D_y^{*b}))$. This proof will convince the mediator that the provided ciphertext encodes the claimed amount b.
7. Set $\pi_{issue} = \pi_{cipher} || \pi_{correct} || \pi_{enc}$
8. Set $\sigma_{issue} = Sign(x, (C_A, C_B, (C_y^b, D_y^b), \pi_{issue})$
9. Return $tx_{\text{issued}} = (C_A, C_B, (C_y^{*b}, D_y^{*b}), \pi_{issue}, \sigma_{issue})$

JustifyAssetTx

- **Inputs:** a mediator private key x_M, user public key y, asset issuance transaction tx_{issued}, user account $acct$,
- **Outputs:** $tx_{justified} = (tx_{issue}, \text{state}, \sigma_{justify})$

1. Parses the tx_{issue} to $(C_A, C_B, (C_y^b, D_y^b), \pi_{issue}, \sigma_{issue})$
2. Parses the user account acct to $(C_A, C_B, (C_y^b, D_y^b), \pi_{issue}, \sigma_{issue})$
3. Decrypt the value C_B to reveal **b** by using the constant-time decryption procedure of the modified Twisted ElGamal system.
4. Set state = $\text{Verify}(L_{cipher}(y, y_M, acct[a], C_A), \pi_{cipher})$
5. Set state = **state** $\wedge \text{Verify}(L_{correct}(y, b, (C_y^b, D_y^b)), \pi_{correct}$
6. Set $\sigma_{justify} = Sign(x_M, (tx_{issued}, \text{state}))$
7. Return $tx_{justified} = (tx_{issue}, \text{state}, \sigma_{justify})$

CreateCTX

- **Inputs:** sender's private key x_s, sender public key y_s, receiver public key y_r, sender account acct_s, receiver account acct_r, asset identifier aID, transfer amount $b*$, the mediator public key y_M, the auditors public keys $Y_{A_1}, ..., Y_{A_l}$
- **Outputs:** Confidential transaction data $ctx_{\text{init}} = (memo, \pi_{valid}, \sigma_s)$

1. Samples $r \leftarrow Z_p$ and computes the values $C_s^* = y_s^r$, $C_r^* = y_r^r$ and $D^* = h^{b*} g^r$.
2. Generates $\pi_1 = \text{Prove}(L_{\text{cipher}}(y_s, y_r, (C_s^*, D^*), (C_r^*, D^*)))$
3. Generates a range proof $\pi_2 = \text{Prove}(L_{\text{range}}(D^*))$ to prove that the committed value $b*$ is not negative.
4. Parses the sender account $acct_s = [(C_{y_s}^a, D_{y_s}^a), (C_{y_s}^b, D_{y_s}^b), \pi_{asset_s}, \sigma_{account_s}, \text{memo}_s]$
5. Parses the recipient account $acct_r = [(C_{y_r}^a, D_{y_r}^a), (C_{y_r}^b, D_{y_r}^b), \pi_{asset_r}, \sigma_{account_r}, \text{memo}_r]$
6. Proves sender has enough funds to transfer by generating a range proof for the sender's updated balance account.
   - Compute $[\pi_3, (C_{y_s}^{b\prime}, D_{y_s}^{b\prime})] = \text{Refresh}((C_{y_s}^b, D_{y_s}^b))$

- Compute $(X, Y) = (\frac{C^{b\prime}_{y_s}}{C^*_{y_s}}, \frac{D^{b\prime}_{y_s}}{D^*})$.
- Generate $\pi_4 = \text{Prove}(L_{\text{range}}(Y = h^{b-b*}g'^r))$.

7. Computes a new ciphertext $(C^{a*}_{y_r}, D^{a*}_{y_r}) = Enc(\text{aID}, y_r)$
8. Encrypts the asset identifier aID with the mediator and auditor's public keys using the modified version of ElGamal, which ensures constant time decryption process: $C^M_A = cEnc(y_M, aID)$, $C^{A_1}_A = cEnc(y_{A_1}, aID), \ldots, C^{A_l}_A = cEnc(y_{A_l}, aID)$.
9. Encrypts the transfer amount b^* with the mediator and auditors public keys using the modified version of ElGamal, which ensures constant time decryption process: $C^M_B = cEnc(y_M, b^*)$, $C^{A_1}_B = cEnc(y_{A_1}, b^*), \ldots, C^{A_l}_B = cEnc(y_{A_l}, b^*)$.
10. Set $memo = ((C^*_s, D^*), y_s, (C^*_r, D^*), y_r)||(C^{b\prime}_{y_s}, D^{b\prime}_{y_s})||(C^{a*}, D^{a*})||\{C^M_B, C^{A_1}_B, \ldots, C^{A_l}_B\}||\{C^M_A, C^{A_1}_A, \ldots, C^{A_l}_A\}$
11. Generate $[\pi_5, (C^{a*}_{y_s}, D^{a*}_{y_s})] = \text{Refresh}(((C^a_{y_s}, D^a_{y_s}))$. The same randomness used to encrypt the asset identifier with the recipient's public key at Step 7 is re-used for creating the fresh ciphertext $(C^{a*}_{y_s}, D^{a*}_{y_s})$.
12. Generate $\pi_6 = \text{Prove}(L_{\text{cipher}}(y_s, y_r, (C^{a*}_{y_s}, D^{a*}_{y_s}), (C^{a*}_{y_r}, D^{a*}_{y_r}))$.
13. Generate $\pi_7 = \text{Prove}(L_{\text{correct}}(y_s, b^*, (C^*_s, D^*))$. This proof will convince the mediator and auditors that the operand ciphertext encodes the claimed number b^*.
14. Generate $\pi_8 = \text{Prove}(L_{\text{correct}}(y_r, aID, (C^{a*}_{y_r}, D^{a*}_{y_r}))$. This proof will convince the mediator and auditors that the operand ciphertext encodes the claimed asset identifier aID.
15. Set $\pi_{valid} = \pi_1||\pi_2||\pi_3||\pi_4||\pi_5||\pi_6||\pi_7||\pi_8$.
16. Set $\sigma_s = Sign(x_s; (memo, \pi_{valid}))$
17. Output $ctx_{\text{init}} = (memo, \pi_{valid}, \sigma_s)$

Certain information, such as a notification of the transaction or even the randomness data used to compute the ciphertext (C^{a*}, D^{a*}), can be sent to the recipient through some out-of-band encrypted channel. Nevertheless, the recipient can always detect targeted transactions by simply scanning events on the blockchain.

After checking if the received amount matches the agreed amount, the receiver can finalize the transaction by proving the incoming transaction asset corresponds to its account asset. He can finalize all necessary zero-knowledge proofs by leveraging the ciphertext refreshment method and without knowing the ciphertext randomness data.

FinalizeCTx

- **Inputs:** Initialized confidential transaction ctx_{init}, user private key x_r, user public key y_r, sender public key y_s,sender account acct_s, receiver account acct_r, asset identifier aID, transfer amount $b*$
- **Outputs:** Confidential transaction data $ctx_{\text{final}} = (memo, \pi_{valid}, \sigma_s, \sigma_r)$

1. Parses $ctx_{\text{init}} = (((C^*_s, D^*), y_s, (C^*_r, D^*), y_r)||(C^{b\prime}_{y_s}, D^{b\prime}_{y_s})||(C^{a*}, D^{a*}), \pi_1||\pi_2||\pi_3||\pi_4||\pi_5, \sigma_s)$
2. Generates $\pi_6 = \text{Prove}(L_{equal}(y_r, (C^{a*}, D^{a*}), (C^a_{y_r}, D^a_{y_r}))$.
3. Set $memo$ to be the same as ctx_{init} memo.
4. Set $\pi_{\text{valid}} = \pi_1||\pi_2||\pi_3||\pi_4||\pi_5||\pi_6$.
5. Set $\sigma_r = Sign(x_r; memo, \pi_{\text{valid}})$
6. Output $ctx_{\text{final}} = (memo, \pi_{valid}, \sigma_s, \sigma_r)$

JustifyCTx

- **Inputs:** a mediator or auditor private key x, initialized confidential transaction ctx_{init}.

- **Outputs:** $ctx_{justified} = \{ctx_{\text{init}} = (memo, \pi_{valid}, \sigma_s), \text{state}, \sigma_{justify}\}$

1. Parses the *memo* and extracts the corresponding ciphertexts C_A^+, C_B^+ which are encrypted under the public key corresponding to the the private key x. It also extracts the ciphertexts (C_s^*, D^*) and $(C_{y_r}^{a*}, D_{y_r}^{a*})$ and the proofs π_7 π_8.
2. Decrypts the ciphertext C_A^+ with the private key x to reveal aID.
3. Decrypts the ciphertext C_B^+ with the private key x to reveal b^*.
4. Set state = Verify($L_{correct}(y_s, b^*, (C_s^*, D^*), \pi_7)$.
5. Set state = **state** $\wedge$ Verify($L_{correct}(y_r, aID, (C_{y_r}^{a*}, D_{y_r}^{a*}), \pi_8)$.
6. Set $\sigma_{justify} = Sign(x_M, (tx_{issued}, \text{state}))$
7. Return $tx_{justified} = (ctx_{\text{init}}, \text{state}, \sigma_{justify})$.

VerifyCTX. This algorithm is used by validators and other system participants to check if the finalized confidential transaction is valid.

- **Inputs:** The sender account acct_s, the receiver account acct_r, a finalized confidential transaction data $ctx_{\text{final}} = (memo, \pi_{valid}, \sigma_s, \sigma_r)$
- **Outputs:** A bit value b indicating the transaction validness.

1. Parses the transaction data ctx_{final} to get $memo = ((C_s^*, D^*), y_s, (C_r^*, D^*), y_r)||(C^{a*}, D^{a*})$ and $\pi_{valid} = \pi_1||\pi_2||\pi_3||\pi_4||\pi_5$.
2. Parses the sender account to get $\text{acct}_s = [(C_{y_s}^a, D_{y_s}^a), (C_{y_s}^b, D_{y_s}^b), ...]$.
3. Parses the recipient account to get $\text{acct}_r = [(C_{y_r}^a, D_{y_r}^a), (C_{y_r}^b, D_{y_r}^b), ...]$.
4. Check if $\text{Verify}_{\text{NIZK}}((L_{\text{cipher}}(y_s, y_r, (C_s^*, D^*), (C_r^*, D^*))), \pi_1)$=1
5. Check if $\text{Verify}_{\text{NIZK}}((L_{\text{range}}(D^*), \pi_2)$=1
6. Check if $\text{Verify}_{\text{NIZK}}((L_{\text{equal}}((C_{y_s}^{a\prime}, D_{y_s}^{b\prime}), (C_{y_s}^b, D_{y_s}^b), \pi_3)$=1
7. Check if $\text{Verify}_{\text{NIZK}}((L_{\text{range}}(\frac{D_{y_s}^{b\prime}}{D^*}, \pi_4)$=1.
8. Check if $\text{Verify}_{\text{NIZK}}((L_{\text{equal}}((C_{y_s}^{a*}, D_{y_s}^{a*}), (C_{y_s}^a, D_{y_s}^a), \pi_5)$=1
9. Check if $\text{Verify}_{\text{NIZK}}(L_{\text{cipher}}(y_s, y_r, (C_{y_s}^{a*}, D_{y_s}^{a*}), (C_{y_r}^{a*}, D_{y_r}^{a*})), \pi_6)$=1
10. Check if $\text{Ver}(y_s; (memo, \pi_{\text{valid}}), \sigma_s) = 1$
11. Check if $\text{Ver}(y_r; (memo, \pi_{\text{valid}}, \sigma_s), \sigma_r) = 1$

In some scenarios it may be required to reverse the confidential transaction without any interaction with the sender and/or the recipient. The need of reversing the transaction may arise due to some mutual agreement between the sender and the recipient or in case of somewhat fraudulent scenarios. The reverse transaction should be initiated and executed by any trusted mediators.

ReverseCTX

- **Inputs:** A finalized confidential transaction data $ctx_{\text{final}} = (memo, \pi_{valid}, \sigma_s, \sigma_r)$,mediator private key x_m
- **Outputs:** A reverse transaction ctx_{reverse}.

1. The trusted mediator parses the confidential transaction data ctx_{final} and extracts the

$$\text{memo} = ((C_s^*, D^*), y_s, (C_r^*, D^*), y_r)||(C_{y_s}^{b\prime}, D_{y_s}^{b\prime})||(C^{a*}, D^{a*})$$

2. Set $\text{memo} = ((C_s^*, D^*), y_s, (C_r^*, D^*), y_r))$.
3. Set $\sigma_m = Sign(x_m; \text{memo}, \text{ctx}_{\text{final}})$

4. Output $ctx_{\text{reverse}} = (memo, \sigma_m, \text{ctx}_{\text{final}})$

System algorithms specify how the user requests are processed and how their accounts are managed. System algorithms can be implemented in the platform smart contract. **ProcessCTX**

- **Inputs:** A confidential transaction data ctx of the type ctx_{final}, ctx_{reverse} or ctx_{issue}

1. Extracts the sender and receiver account balances $acct_{y_s}[b] = (C^b_{y_s}, D^b_{y_s})$ and $acct_{y_r}[b] = (C^b_{y_r}, D^b_{y_r})$
2. If $ctx = ctx_{\text{final}}$
   - Parses the confidential transaction data ctx_{final} and extracts the
   $$\text{memo} = ((C^*_s, D^*), y_s, (C^*_r, D^*), y_r)||(C^{b\prime}_{y_s}, D^{b\prime}_{y_s})||(C^{a*}, D^{a*})$$
   - Set $acct_{y_s}[b] = (C^b_{y_s}, D^b_{y_s}) \circ (C^{*-1}_s, D^{*-1})$
   - Set $acct_{y_r}[b] = (C^b_{y_r}, D^b_{y_r}) \circ (C^*_r, D^*)$
3. If $ctx = ctx_{\text{reverse}}$
   - Parses the confidential transaction data ctx_{reverse} and extracts the
   $$\text{memo} = ((C^*_r, D^*), y_r, (C^*_s, D^*), y_s))$$
   - Set $acct_{y_s}[b] = (C^b_{y_s}, D^b_{y_s}) \circ (C^*_s, D^*)$
   - Set $acct_{y_r}[b] = (C^b_{y_r}, D^b_{y_r}) \circ (C^{*-1}_r, D^{*-1})$
4. If $ctx = ctx_{\text{issue}}$
   - Parses the confidential transaction data ctx_{issue} and extracts the $memo = (C^{*b}_{y_s}, D^{*b}_{y_s})$
   - Set $acct_{y_s}[b] = (C^b_{y_s}, D^b_{y_s}) \circ (C^{*b}_{y_s}, D^{*b}_{y_s})$

After the transactions are verified by validators, they can be processed and result to the account updates.

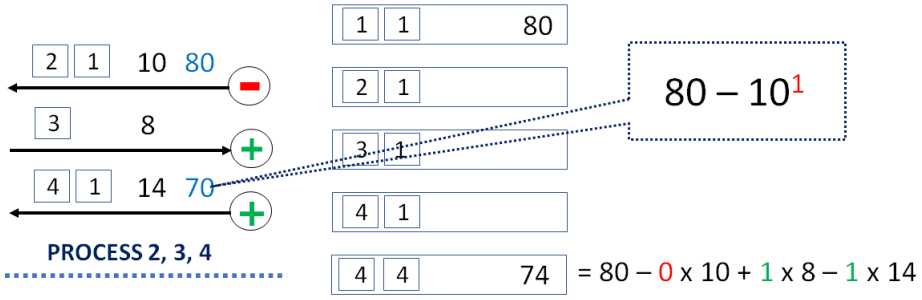
7 Transaction Ordering Mechanism

In a privacy payment protocol, cryptographic proofs are generated with respect to a certain state of the account balance. For example, the zero-knowledge proof in the initialized transaction needs to show that the sender's remaining balance is positive. A user Alice generates her transaction proofs with respect to her current account balance, which is stored encrypted on the blockchain. But a proof that a ciphertext encrypts certain value gets invalidated as soon as the ciphertext changes. If Alice gets another transfer from Bob before her transaction gets accepted and processed by the network, then this incoming transaction will update Alice's balance and invalidate the proceeding transaction's proofs. This requires all transactions to be generated and processed according to some specific ordering mechanism which takes into account the possible update of the user balance. For example in Zether, the time is divided into epochs and the users can generate transactions any time during the epoch. The sender's account balance is updated immediately upon each new transaction creation, but the submitted transactions are finalized and the recipient account are updated only at the end of each epoch. This approach will not work in the MERCAT case, as the sender account can not be updated anytime earlier before the transaction is finalized by the recipient and also approved by the mediator. But these phases may take unexpectedly longer resulting to a situation where a single pending transaction blocks the user of creating new transactions. We discuss a potential candidate for special ordering mechanism aiming to support smooth workflow of transaction generation and processing.

Ordering States: The proposal below describes a sequential transaction ordering design without relying on the time epochs. Transactions can be safely generated by the users regardless of the processing ordering of the prior pending transactions, and they can be securely processed by network validators in a sequential manner through the following manner. We associate two state variables with each account. The first variable is the account-specific

transaction counter which increments upon each incoming or outgoing transaction. For outgoing transactions, the counter is incremented and assigned to the new transaction at the transaction initialization phase. For incoming transactions, this counter is incremented and assigned to the incoming transaction during its finalization phase by the user. The second variable indicates the last processed transaction number for the origin account. Let's refer to this pair of counters as the account's "ordering state". For example, the account's "ordering state" $(7, 4)$ shows there are 7 transactions associated with that account where the first 4 transactions are already processed (they can be either accepted or rejected), and the transactions number 5, 6, and 7 are still in a pending state (they can be waiting for the finalization by respective recipients, approval from a mediator, or validation by the network validators). Each outgoing transaction is assigned two counters as well. The first one is the transaction id, the second counter indicates the last processed transaction id for the origin account at the moment of transaction generation (we can refer to this as a reference counter). E.g. the transaction counter (9, 5) indicates the transaction counter is 9, but at the moment of its generation, only the first 5 transactions associated with its origin account have been processed, and the transactions 6,7 and 8 had been yet in the pending state. These ordering states helps to maintain which state the transaction proofs are generated with respect to and re-compute that state in order to verify the cryptography proofs. The zero-knowledge proofs for each transaction are generated with respect to an account's pending balance which is basically the account balance after all the proceeding pending transactions get processed. For each transaction, the corresponding account pending balance can be computed by using the account on-chain balance corresponding to the last processed state and then virtually 'processing' all pending transactions up to the certain transaction. Pending balance calculation should consider only the outgoing transactions and skip the incoming transactions which can only increase the balance. Below we describe two different scenarios of generating transactions to show how the pending balance computation works for enabling transaction processing at arbitrary times.

- **Scenario 1:** The user Alice initiates a new transaction when there are already other pending transactions in the queue. The newly issued transaction is added to the validation/processing queue before the network validators start the processing of pending transactions.Let's assume that Alice's account state is (1,1) and her current on-chain balance is 80 tokens. This implies that there is only one associated transaction with this account and that transaction has been successfully processed at some time. Alice next has initiated the second outgoing transaction which is sending 10 tokens to Bob, then finalizes one incoming transaction which transfers 8 token to Alice. This incoming transaction gets the number 3. Last, Alice initiates new outgoing transaction number 4 to send 14 tokens to Carol. The following diagram illustrates the situation.



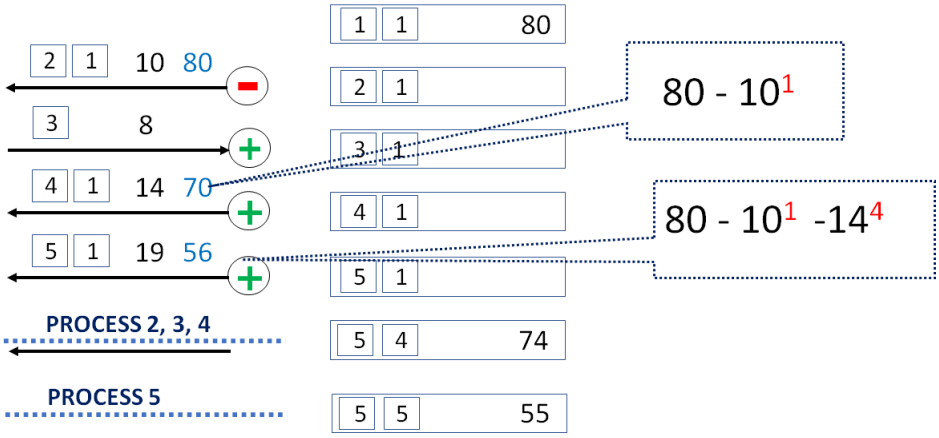
Computation of pending balances

The zero-knowledge proofs of the tx 2 are generated with respect to the on-chain balance which is 80, as there are no pending transactions prior to the transaction 2. The zero-knowledge proofs of the transaction 4 are generated with respect to a pending balance, which is computed by subtracting the amounts of all preceding outgoing transactions from the on-chain balance. There is only one pending transaction preceding tx 2 and the zero-knowledge proofs of tx 4 are computed with respect to the pending balance 70, which is computed by simply subtracting the tx 2's transferred amount from the on-chain balance. Note that the pending balance is computed through homomorphic operations over the ciphertext and it never appears in the plain-text form.

Now the validators take on processing this account. Let's assume the tx 2 was rejected by the recipient(this will not invalidate the subsequent transactions), but the txs 3 and 4 were accepted and successfully processed. No matter that tx 2 was rejected, the validators can consider it while computing the pending balance for the tx 4 and still succesfully validate the cryptographic proofs of tx 4. The account's ordering state is updated to be (4,4)

which indicates that all four transactions associated with this account are successfully processed and there are not pending transactions at this moment.

- **Scenario 2:** Now let's discuss more complex scenario when the user initiates a new transaction and then part of the previous pending transfers get processed right after this new tx has been initialized, but before it was finalized by the recipient. Let's assume the tx 5 is initiated in the previous workflow before the first processing happens.The transaction ordering state will be $(5, 1)$ which shows that this transaction proofs are generated with respect to the account balance corresponding to the moment when only the first transaction is processed. The diagram below depicts this situation.



Computation of pending balances

After the transaction 5 is initialized, the validators process the pending transactions 2, 3, and 4, but not 5, as the later still is waiting to be finalized by the recipient. The account's ordering state is updated to $(5, 4)$ which shows that the 4 out of 5 transaction are processed and tx 5 is still in the pending state. After the account state is updated to $(5, 4)$, the tx 5 gets finalized by the recipient and ready to be processed. In this case, the validator should compute the pending balance state for tx 5 when it was generated, by virtually rolling back the previously processed transactions up to the tx 1. Rolling back implies subtracting the amounts of all *rejected outgoing transactions* and the amounts of all *accepted incoming transactions* processed between the transaction's ordering state's second counter(reference counter) and the on-chain account second counter(processed counter).

8 Performance Insights

To evaluate the performance of our protocol, we created a reference implementation in RUST using the popular Dalek cryptography library which implements the Ristretto group over the curve25519 elliptic curves[24]. Dalek's library is one of the most popular cryptographic libraries used by numerous blockchain and cryptocurrency projects. All experiments were performed on an [2XLarge Server HARDWARE SPECIFICATION].

Table 1 shows the zero-knowledge proof size estimates for the MERCAT transactions. In Table 2 we provide basic storage and computational characteristics of all utilized zero-knowledge building blocks. In the compressed form, each group element(Ristretto point) is stored as 32 bytes. As can be seen from the table, the most storage expensive operation is the **CreatAccount** transaction. It contains an one-out-of-many proof over a set of 2^16 asset tickers and takes approximately 1.8kb space on the ledger. Not that although being quite storage expensive, this is a one-time operation per each new MERCAT account and is the only MERCAT operation using one-out-of-many proofs.

Table 1. MERCAT Transaction Payload Sizes

	CreateAccount	**CreateAssetIssuanceTx**	**CreateCtx**	**FinalizeCtx**	**ReverseCTx**
Size	1.8KB	600B	1.2KB	180B	100B

The **CreateCtx** transaction payload, which is generated upon each new transaction, after final optimizations will take approximately 1.2KB storage on the ledger. This storage consumption is mostly due to the zero-knowledge range proofs. Worth to mention that a significant storage optimization for this transaction comes through the usage of the batch proving methods of bulletproofs which will enable to combine two different proofs into a single proof of a smaller size. Table 3 shows time-measured performance characteristics for MERCAT transactions based on our RUST implementation.

Table 2. The performance characteristics and storage requirements for cryptographic building blocks

	Proof Size	Num. Exp (Prove)	Num. Exp. (Verify)
π_{enc}	130B	3	5
$\pi_{correct}$	98B	2	4
π_{equal}	98B	2	4
π_{cipher}	195B	4	6
$\pi_{refresh}$	164B	8	4
π_{range}	670B	29ms	4ms
π_{member}	1.5KB	1.8s	28.4ms (with batch ver.)

Table 3. MERCAT Operations Performance

	Transaction Generation	Mediator Verification	Validator Verification
Account Creation	651.92 ms	n/a	120.79 ms
Asset Issuance	1.6036 ms	1.8519 ms	1.0904 ms
Create Transaction (Initial Balance: 1M)	1.5038 s	10.592 ms	
n/a heightFinalize Transaction	416.11 us	n/a	9.1680 ms

The prove and verification processes for the $\pi_{enc}, \pi_{correct}, \pi_{value}$ and π_{cipher} zero-knowledge proofs require only a few exponentiation operations. Their speed is compatible with ECDSA signing and verification processes. The performance data for range proofs is taken from the corresponding paper[23]. Note that all sender's computation for generating the **CreateCTx** involves Twisted ElGamal decryption which is a linear operation on the encrypted value. The provided benchmark is with respect to an initial account balance of 1M tokens, and the transaction generation time may vary depending on the account balance. All other operations are constant time, including the mediator operations which require constant-time ElGamal decryption operations.

The validation of a single finalized transaction takes approximately 9ms which ensures 110 TPS for the network. As each transaction validation requires the verification of separate zero-knowledge proofs which all boil down to multi-exponentiation operations, this throughput could be increased by implementing fast multi-exponentiation and batch verification techniques.

9 Further Work

The construction presented here enables a confidential asset management system allowing users to transact confidentially by hiding the transaction balances and asset types. However, it requires a trusted mediator to intervene in the asset issuance process and does not fully obfuscate the transaction graph. Enabling fully anonymous transactions remains a subject for the future work.

This document also describes a special transaction ordering mechanism which supports flexible transaction generation and validation process without putting any strict timing constraints on when the validation process should take place. Future implementation details of this ordering mechanism will be provided after its full implementation and integration with the blockchain network.

We also discuss initial implementation and protocol performance details for the provided construction, although leave certain protocol optimizations for the future work. Note that the transaction validation process is subject of significant optimizations through the usage of fast multi-exponentiation and joint verification techniques which are not fully implemented yet.

Providing formal theoretical proofs of the protocol security properties also remains a subject for the future work.

References

1. Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. https://eprint.iacr. org/2019/191
2. Yu Chen, Xuecheng Ma, Cong Tang. PGC: Pretty Good Confidential Transaction System with Accountability. Cryptology ePrint Archive, Report 2019/319, 2019. https://eprint.iacr. org/2019/319.pdf
3. Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Advances in Cryptology - EUROCRYPT 2000, volume 1807 of Lecture Notes in Computer Science, pages 259–274. Springer, 2000.
4. Claus-Peter Schnorr. Efficient identification and signatures for smart cards (abstract) (rump session). In Jean-Jacques Quisquater and Joos Vandewalle, editors, EUROCRYPT'89, volume 434 of LNCS, pages 688–689. Springer, April 1990.
5. Ma, Shunli, Yi Deng, Debiao He, Jiang Zhang and Xiang Xie. "An Efficient NIZK Scheme for Privacy-Preserving Transactions over Account-Model Blockchain." IACR Cryptology ePrint Archive 2017 (2017): 1239.
6. Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, 28 PKC 2002, volume 2274 of Lecture Notes in Computer Science, pages 48–63. Springer.
7. Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, CRYPTO'86,volume 263 of LNCS, pages 186–194. Springer, August 1987.
8. AZTEC Protocol — Ethereum's Privacy Engine. https://www.aztecprotocol.com/
9. Nightfall: A private-public blockchain. https://github.com/EYBlockchain/nightfall
10. Ronald Cramer and Ivan Damgard. Zero-knowledge proofs for finite field arithmetic; or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, CRYPTO98, volume 1462 of LNCS, pages 424–441. Springer, August 1998.
11. Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In 2014 IEEE Symposium on Security and Privacy, pages 459–474. IEEE Computer Society Press, May 2014.
12. Feng Bao, Robert H. Deng, and Huafei Zhu. Variations of diffie-hellman problem. In Information and Communications Security, 5th International Conference, ICICS 2003, volume 2836 of Lecture Notes in Computer Science, pages 301–312. Springer, 2003.
13. David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. J. Cryptology, 13(3):361–396, 2000.
14. J. Camenisch, A. Kiayias, M. Yung. "On the Portability of Generalized Schnorr Proofs". https://eprint.iacr.org/2009/050.pdf
15. Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In 2013 IEEE Symposium on Security and Privacy, pages 397–411. IEEE Computer Society Press, May 2013.
16. A. Jivanyan. Lelantus: Towards Confidentiality and Anonymity of Blockchain Transactions From Standard Assumptions, https://eprint.iacr.org/2019/373
17. ZkVM: A new design for fast and confidential smart contracts. https://medium.com/stellar-developers-blog/zkvm-a-new-design-for-fast-confidential-smart-contracts-d1122890d9ae
18. Claus-Peter Schnorr. Efficient signature generation by smart cards. Journal of Cryptology, 4(3):161–174.
19. J. Groth and M. Kohlweiss. One-out-of-many proofs: Or how to leak a secret and spend a coin. In EUROCRYPT, vol. 9057 of LNCS. Springer, 2015.
20. Nicolas van Saberhagen. CryptoNote v 2.0, 2013. https://cryptonote.org/whitepaper.pdf.
21. Shen Noether. Ring signature confidential transactions for monero. Cryptology ePrint Archive, Report 2015/1098, 2015. http://eprint.iacr.org/2015/1098.
22. Bootle, J., Cerulli, A., Chaidos, P., Ghadafi, E., Groth, J., Petit, C.: "Short accountable ring signatures based on DDH". In: Pernul, G., et al. (eds.) ESORICS. LNCS, vol. 9326, pp. 243–265. Springer, Heidelberg (2015).
23. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, and Greg Maxwell. "Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Report 2017/1066, 2017. https://eprint.iacr. org/2017/1066"
24. A pure-Rust implementation of group operations on Ristretto and Curve25519. https://github.com/dalek-cryptography/curve25519-dalek
25. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In CRYPTO, volume 576 of Lecture Notes in Computer Science, pages 129 -140,1991.