

MantaPay Protocol Specification

v0.4.0

Shumo Chu and Brandon H. Gomes

December 11, 2021

Abstract

MantaPay is an implementation of a *decentralized anonymous payment* scheme based on the MANTADAP protocol outlined in the original [MANTA whitepaper](#).

Contents

1	Introduction	2
2	Notation	2
3	Concepts	2
3.1	Assets	2
3.2	Addresses	3
3.3	Ledger	3
3.3.1	UTXOs and the UTXOSet	4
3.3.2	EncryptedNotes	4
3.3.3	VoidNumbers and the VoidNumberSet	4
4	Abstract Protocol	4
4.1	Abstract Cryptographic Schemes	4
4.1.1	Commitment Scheme	4
4.1.2	Key-Agreement Scheme	4
4.1.3	Symmetric-Key Encryption Scheme	5
4.1.4	Key-Derivation Function	5
4.1.5	Hybrid Public Key Encryption Scheme	5
4.1.6	Dynamic Cryptographic Accumulator	5
4.1.7	Non-Interactive Zero-Knowledge Proving System	6
4.2	Addresses and Key Components	7
4.3	Transfer Protocol	8
4.4	Semantic Transactions	12
5	Concrete Protocol	12
5.1	Concrete Cryptographic Schemes	13
6	Acknowledgements	13
7	References	13

1 Introduction

MantaPay aims to solve the long standing privacy problem of crypto asset in Web3 age. At its heart, it uses various cryptographic constructions including NIZK (non-interactive zero knowledge proof) systems to ensures users privacy from the *first principle*.

Protocol	Cryptographic Primitive	Consensus	Arbitrary Nomination	Multi-Asset
ZCash (Sampling)	NIZK	PoW	✓	✗
Monero	RingCT	PoW	✓	✗
Tornado.Cash	NIZK	PoW	✗	✗
MantaPay (0.40)	NIZK	PoS	✓	✓

Table 1: Comparison of MantaPay with previous construction

Table 1 shows comparison of MantaPay with previous constructions.

2 Notation

The following notation is used throughout this specification:

- **Type** is the type of types¹.
- If $x : T$ then x is a value and T is a type, denoted $T : \text{Type}$, and we say that x *has type* T .
- **Bool** is the type of booleans with values **True** and **False**.
- For any types $A : \text{Type}$ and $B : \text{Type}$ we denote the *type of functions* from A to B as $A \rightarrow B : \text{Type}$.
- For any types $A : \text{Type}$ and $B : \text{Type}$ we denote the *product type* over A and B as $A \times B : \text{Type}$ with constructor $(-, -) : A \rightarrow (B \rightarrow A \times B)$.
- For any type $T : \text{Type}$, we define $\text{Option}(T) : \text{Type}$ as the inductive type with constructors:

$$\begin{aligned} \text{None} &: \text{Option}(T) \\ \text{Some} &: T \rightarrow \text{Option}(T) \end{aligned}$$

- We denote the *type of finite sets* over a type $T : \text{Type}$ as $\text{FinSet}(T) : \text{Type}$. The membership predicate for a value $x : T$ in a finite set $S : \text{FinSet}(T)$ is denoted $x \in S$.
- We denote the *type of finite ordered sets* over a type $T : \text{Type}$ as $\text{List}(T) : \text{Type}$. This can either be defined by an inductive type or as a $\text{FinSet}(T)$ with a fixed ordering. We denote the constructor for a list as $[\dots]$ for an arbitrary set of elements.
- We denote the *type of distributions* over a type $T : \text{Type}$ as $\mathfrak{D}(T) : \text{Type}$. A value x sampled from $\mathfrak{D}(T)$ is denoted $x \sim \mathfrak{D}(T)$ and the fact that the value x belongs to the range of $\mathfrak{D}(T)$ is denoted $x \in \mathfrak{D}(T)$. So namely, $y \in \{x \mid x \sim \mathfrak{D}(T)\} \leftrightarrow y \in \mathfrak{D}(T)$.
- Depending on the context, the notation $|\cdot|$ denotes either the absolute value of a quantity, the length of a list, the number of characters in a string, or the cardinality of a set.

3 Concepts

3.1 Assets

The **Asset** is the fundamental currency object in the MantaPay protocol. An asset $a : \text{Asset}$ is a tuple

$$a = (a.\text{id}, a.\text{value}) : \text{AssetId} \times \text{AssetValue}$$

where the **AssetId** encodes the type of currency stored in a and the **AssetValue** encodes how many units of that currency are stored in a . **MantaPay** is a *decentralized anonymous payment* protocol which facilitates the private ownership and private transfer of **Asset** objects.

¹By *type of types*, we mean the type of *first-level* types in some family of type universes. Discussion of the type theory necessary to make these notions rigorous is beyond the scope of this paper.

Whenever an **Asset** is being used in a public setting, we simply refer to it as an **Asset**, but when the **AssetId** and/or **AssetValue** of a particular **Asset** is meant to be hidden from public view, we refer to the **Asset** as either, *secret*, *private*, *hidden*, or *shielded*.

Assets form the basic units of *transactions* which consume **Assets** on input, transform them, and return **Assets** on output. To preserve the economic value stored in **Assets**, the sum of the input **AssetValues** must balance the sum of the output **AssetValues**, and all assets in a single transaction must have the same **AssetId**². This is called a *balanced transfer*: no **AssetValue** is created or destroyed in the process. The MantaPay protocol uses a distributed algorithm called **Transfer** to perform balanced transfers and ensure that they are valid.

3.2 Addresses

In order for MantaPay participants to send and receive **Assets** via the **Transfer** protocol, they create *addresses* which represent their participation in the protocol. MantaPay has a 3-address system consisting of a *spending key* **sk**, a *viewing key* **vk**, and a *receiving key* **rk**. The keys have the following uses/properties:

- Access to a receiving key **rk** represents the ability to send **Assets** to the owner of the associated **sk**.
- Access to a viewing key **vk** represents the ability to reveal shielded **Asset** information for **Assets** belonging to the owner of the associated **sk**.
- Access to a spending key **sk** represents the ability to spend **Assets** that were received under the associated receiving key **rk**.

Participants in MantaPay are represented by their addresses, but they are not unique representations, since one participant may have access to more than one triple of keys. See § 4.2 for more information on how these keys are constructed and used for spending, viewing, and receiving **Assets**.

3.3 Ledger

Preserving the economic value of **Assets** requires more than just balanced transfers. It also requires that **Assets** are owned by exactly one address at a time, namely, that the ability to spend an **Asset** can be proved before a transfer and revoked after a transfer. It is not simply the *information-content* of an **Asset** that should be transferred, but the *ability to spend the asset in the future*, which should be transferred. To enforce this second invariant we can use a public ledger³ that keeps track of the movement of **Assets** from one participant to another. Unfortunately, using a public ledger alone does not allow participants to remain anonymous, so MantaPay extends the public ledger by adding a special account called the *shielded asset pool* which is responsible for keeping track of the **Assets** which have been anonymized by the protocol. We denote the three ledger types in the protocol as follows: the public ledger as **PublicLedger**, the shielded asset pool as **ShieldedAssetPool**, and the combined ledger we denote **Ledger**.

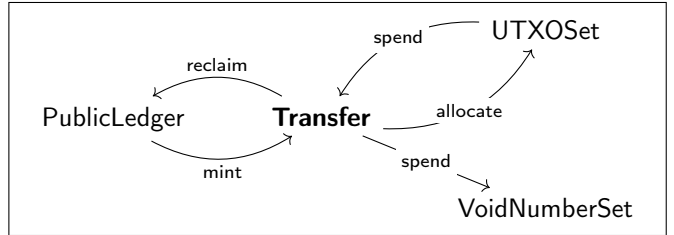


Figure 1: Lifecycle of an Asset.

The **ShieldedAssetPool** is made up of four parts which serve to enforce the balanced transfer of **Assets** among anonymous participants:

1. **ShieldedAssetPool Balance**: The **Ledger** contains a collection of **Assets** which encode the combined economic value of the **ShieldedAssetPool** and the **PublicLedger**. The **ShieldedAssetPool** balance is the subset of this total collection that has been anonymized by the MantaPay protocol. This balance is stored as a finite set of non-zero **Assets**.
2. § 3.3.1 **UTXOSet**: The **UTXOSet** is a collection of ownership claims to subsets of the **ShieldedAssetPool** (called **UTXOs**), each one referring to an allocated **Asset** transferred to a participant of the protocol.
3. § 3.3.2 **EncryptedNotes**: For every **UTXO** there is a matching **EncryptedNote** which contains information necessary to spend the **Asset**, which can be used to *provably reconstruct* the **UTXO** convincing the **Ledger** of unique ownership. The **EncryptedNote** can only be decrypted by the recipient of the **Asset**, specifically, the correct viewing key **vk**. See § 3.2 for more.

²It is beyond the scope of this paper to discuss transactions with inputs and outputs that feature different **AssetIds**, like those that would be featured in a *decentralized anonymous exchange*.

4. § 3.3.3 VoidNumberSet: The VoidNumberSet is a collection of commitments, like UTXOs, but which track the *spent state* of an Asset and are used to prove to the Ledger that an Asset is spent *exactly one time*.

The operation of these different parts of the ShieldedAssetPool is elaborated in the following subsections.

3.3.1 UTXOs and the UTXOSet

An *unspent transaction output*, or UTXO for short, represents a claim to the output of a balanced transfer which has otherwise *not yet been spent*. Every balanced transfer produces *public outputs*, just publicly visible Assets, and *private outputs*, represented by UTXOs, and these UTXOs are stored in the UTXOSet of the ShieldedAssetPool. A UTXO can only be claimed by the participant who owns the underlying Asset, where ownership means *knowledge of the correct spending key* and the Transfer protocol requires that all inputs to a balanced transfer *prove* that they own a UTXO which the ShieldedAssetPool has already seen in the past. The UTXOSet is *append-only* since it represents the past state of *unspent* Assets. UTXOs can only be added to the UTXOSet as outputs in the execution of a Transfer which the Ledger checks for correctness.

3.3.2 EncryptedNotes

In order to find out what Asset a UTXO is connected to, every UTXO comes with an associated EncryptedNote which stores two pieces of information, the underlying Asset, and an ephemeral public key, a value which allows the new owner of the Asset to reconstruct the UTXO. Being able to *provably reconstruct* a correct UTXO is a prerequisite to ownership and the ability to spend the Asset in the future. Once a participant spends an Asset that they can decrypt, they build a new EncryptedNote for the next participant that they sent their Assets to, so that they can then spend it, and so on. This is called the *in-band secret distribution*.

3.3.3 VoidNumbers and the VoidNumberSet

Once the ability to spend an Asset is extracted from a (UTXO, EncryptedNote) pair, the ShieldedAssetPool requires another commitment in order to spend the Asset, transferring it to another participant. This commitment, called the VoidNumber, represents the revocation of the right to spend the Asset in the future, and ensures that the same Asset cannot be spent twice. Like the UTXOSet, the VoidNumberSet is *append-only* since it represents the past state of *spent* Assets. VoidNumbers can only be added to the VoidNumberSet as inputs in the execution of a Transfer which the Ledger checks for correctness.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

In the following section, we outline the formal specifications for all of the *cryptographic schemes* used in the MantaPay protocol.

Definition 4.1.1 (Commitment Scheme). A *commitment scheme* COM is defined by the schema:

$$\begin{aligned}
&\text{Trapdoor} : \text{Type} \\
&\text{Input} : \text{Type} \\
&\text{Output} : \text{Type} \\
&\text{TrapdoorDistribution} : \mathcal{D}(\text{Trapdoor}) \\
&\text{commit} : \text{Trapdoor} \times \text{Input} \rightarrow \text{Output}
\end{aligned}$$

with the following properties:

- **Binding:** It is infeasible to find an $x, y : \text{Input}$ and $r, s : \text{Trapdoor}$ such that $x \neq y$ and $\text{commit}(r, x) = \text{commit}(s, y)$.
- **Hiding:** For all $x, y : \text{Input}$, the distributions $\{\text{commit}(r, x) \mid r \sim \text{TrapdoorDistribution}\}$ and $\{\text{commit}(r, y) \mid r \sim \text{TrapdoorDistribution}\}$ are *computationally indistinguishable*.

Notation: For convenience, we refer to $\text{COM.commit}(r, x)$ by $\text{COM}_r(x)$.

³A public (or private) ledger is not enough to solve the *provable-ownership problem* or the *double-spending problem*. A *consensus mechanism* is also required to ensure that all participants agree on the current state of the ledger. The design and specification of the consensus mechanism that secures the MantaPay ledger is beyond the scope of this paper.

Definition 4.1.2 (Key-Agreement Scheme). A *key-agreement scheme* KA is defined by the schema:

$$\begin{aligned} \text{SecretKey} &: \text{Type} \\ \text{PublicKey} &: \text{Type} \\ \text{SharedSecret} &: \text{Type} \\ \text{derive} &: \text{SecretKey} \rightarrow \text{PublicKey} \\ \text{agree} &: \text{SecretKey} \times \text{PublicKey} \rightarrow \text{SharedSecret} \end{aligned}$$

with the following properties:

- **Agreement:** For all $sk_1, sk_2 : \text{SecretKey}$, $\text{agree}(sk_1, \text{derive}(sk_2)) = \text{agree}(sk_2, \text{derive}(sk_1))$
- **TODO:** security properties

Definition 4.1.3 (Symmetric-Key Encryption Scheme). A *symmetric-key encryption scheme* SYM is defined by the schema:

$$\begin{aligned} \text{Key} &: \text{Type} \\ \text{Plaintext} &: \text{Type} \\ \text{Ciphertext} &: \text{Type} \\ \text{encrypt} &: \text{Key} \times \text{Plaintext} \rightarrow \text{Ciphertext} \\ \text{decrypt} &: \text{Key} \times \text{Ciphertext} \rightarrow \text{Option}(\text{Plaintext}) \end{aligned}$$

with the following properties:

- **Invertibility:** For all keys $k : \text{Key}$ and plaintexts $p : \text{Plaintext}$, we have that

$$\text{decrypt}(k, \text{encrypt}(k, p)) = \text{Some}(p)$$

- **TODO:** hiding, one-time encryption security?

Definition 4.1.4 (Key-Derivation Function). A *key-derivation function* KDF defined over a symmetric-key encryption scheme SYM and a key-agreement scheme KA is a function of type:

$$\text{KDF} : \text{KA.SharedSecret} \rightarrow \text{SYM.Key}$$

with the following properties:

- **TODO:** security properties

Definition 4.1.5 (Hybrid Public Key Encryption Scheme). A *hybrid public key encryption scheme* [1] HPKE is an encryption scheme made up of a symmetric-key encryption scheme SYM, a key-agreement scheme KA, and a key-derivation function KDF to convert from KA.SharedSecret to SYM.Key. We can define the following encryption and decryption algorithms:

- **Encryption:** Given an ephemeral secret key $esk : \text{KA.SecretKey}$, a public key $pk : \text{KA.PublicKey}$, and plaintext $p : \text{SYM.Plaintext}$, we produce the pair

$$m : \text{SYM.Ciphertext} \times \text{KA.PublicKey} := (\text{SYM.encrypt}(\text{KDF}(\text{KA.agree}(esk, pk)), p), \text{KA.derive}(esk))$$

- **Decryption:** Given a secret key $sk : \text{KA.SecretKey}$, and an encrypted message, as above, $m := (c, epk)$, we can decrypt m , producing the plaintext,

$$p : \text{Option}(\text{SYM.Plaintext}) := \text{SYM.decrypt}(\text{KDF}(\text{KA.agree}(sk, epk)), c)$$

which should decrypt successfully if the KA.PublicKey that m was encrypted with is the derived key of $sk : \text{KA.SecretKey}$.

Notation: We denote the above *encrypted message* type as $\text{Message} := \text{SYM.Ciphertext} \times \text{KA.PublicKey}$, and the above two algorithms by

$$\begin{aligned} \text{encrypt} &: \text{KA.PublicKey} \times \text{KA.SecretKey} \times \text{SYM.Plaintext} \rightarrow \text{Message} \\ \text{decrypt} &: \text{KA.SecretKey} \times \text{KA.PublicKey} \times \text{SYM.Ciphertext} \rightarrow \text{Option}(\text{SYM.Plaintext}) \end{aligned}$$

TODO: security properties, combine with SYM and KA properties, like the fact that some of these keys are ephemeral, etc.

Definition 4.1.6 (Dynamic Cryptographic Accumulator). A *dynamic cryptographic accumulator* DCA is defined by the schema:

Item : Type
 Output : Type
 Witness : Type
 State : Type
 current : State \rightarrow Output
 insert : Item \times State \rightarrow State
 contains : Item \times State \rightarrow Option(Output \times Witness)
 verify : Item \times Output \times Witness \rightarrow Bool

with the following properties:

- **Unique Accumulated Values:** For any initial state $s : \text{State}$ and any list of items $I : \text{List}(\text{Item})$ we can generate the sequence of states:

$$s_0 := s, \quad s_{i+1} := \text{insert}(I_i, s_i)$$

Then, if we collect the accumulated values for these states, $z_i := \text{current}(s_i)$, there should be exactly $|I|$ -many unique values, one for each state update.

- **Provable Membership:** For any initial state $s : \text{State}$ and any list of items $I : \text{List}(\text{Item})$ we can generate the sequences of states:

$$s_0 := s, \quad s_{i+1} := \text{insert}(I_i, s_i)$$

Then, if we collect the states s_i into a set S , we have the following property for all $s \in S$ and $t \in I$,

$$\text{Some}(z, w) := \text{contains}(t, s), \quad \text{verify}(t, z, w) = \text{True}$$

- **TODO:** security properties

TODO: add finite capacity constraint, something like $\text{insert} : \text{Item} \times \text{State} \rightarrow \text{Option}(\text{State})$ where it fails when capacity is reached

Definition 4.1.7 (Non-Interactive Zero-Knowledge Proving System). A *non-interactive zero-knowledge proving system* NIZK is defined by the schema:

Statement : Type
 ProvingKey : Type
 VerifyingKey : Type
 PublicInput : Type
 SecretInput : Type
 Proof : Type
 keys : Statement $\rightarrow \mathcal{D}(\text{ProvingKey} \times \text{VerifyingKey})$
 prove : Statement \times ProvingKey \times PublicInput \times SecretInput $\rightarrow \mathcal{D}(\text{Option}(\text{Proof}))$
 verify : VerifyingKey \times PublicInput \times Proof \rightarrow Bool

Notation: We use the following notation for a NIZK:

- We write the Statement and ProvingKey arguments of prove in the superscript and subscript respectively,

$$\text{prove}_{\text{pk}}^P(x, w) := \text{prove}(P, \text{pk}, x, w)$$

- We write the VerifyingKey argument of verify in the subscript,

$$\text{verify}_{\text{vk}}(x, \pi) := \text{verify}(\text{vk}, x, \pi)$$

- We say that $(x, w) : \text{PublicInput} \times \text{SecretInput}$ has the property of being a satisfying input whenever

$$\text{satisfying}_{\text{pk}}^P(x, w) := \exists \pi : \text{Proof}, \text{Some}(\pi) \in \text{prove}_{\text{pk}}^P(x, w)$$

Every NIZK has the following properties for a fixed statement $P : \text{Statement}$ and keys $(\text{pk}, \text{vk}) \sim \text{keys}(P)$:

- **Completeness:** For all $(x, w) : \text{PublicInput} \times \text{SecretInput}$, if $\text{satisfying}_{\text{pk}}^P(x, w) = \text{True}$ with proof witness π , then $\text{verify}_{\text{vk}}(x, \pi) = \text{True}$.
- **Knowledge Soundness:** For any polynomial-size adversary \mathcal{A} ,

$$\mathcal{A} : \text{ProvingKey} \times \text{VerifyingKey} \rightarrow \mathcal{D}(\text{PublicInput} \times \text{Proof})$$

there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$

$$\mathcal{E}_{\mathcal{A}} : \text{ProvingKey} \times \text{VerifyingKey} \rightarrow \mathcal{D}(\text{SecretInput})$$

such that the following probability is negligible:

$$\Pr \left[\begin{array}{l} \text{satisfying}_{\text{pk}}^P(x, w) = \text{False} \\ \text{verify}_{\text{vk}}(x, w) = \text{True} \end{array} \middle| \begin{array}{l} (\text{pk}, \text{vk}) \sim \text{keys}(P) \\ (x, \pi) \sim \mathcal{A}(\text{pk}, \text{vk}) \\ w \sim \mathcal{E}_{\mathcal{A}}(\text{pk}, \text{vk}) \end{array} \right]$$

- **Statistical Zero-Knowledge:** There exists a stateful simulator \mathcal{S} , such that for all stateful distinguishers \mathcal{D} , the difference between the following two probabilities is negligible:

$$\Pr \left[\begin{array}{l} \text{satisfying}_{\text{pk}}^P(x, w) = \text{True} \\ \mathcal{D}(\pi) = \text{True} \end{array} \middle| \begin{array}{l} (\text{pk}, \text{vk}) \sim \text{keys}(P) \\ (x, w) \sim \mathcal{D}(\text{pk}, \text{vk}) \\ \text{Some}(\pi) \sim \text{prove}_{\text{pk}}^P(x, w) \end{array} \right] \text{ and } \Pr \left[\begin{array}{l} \text{satisfying}_{\text{pk}}^P(x, w) = \text{True} \\ \mathcal{D}(\pi) = \text{True} \end{array} \middle| \begin{array}{l} (\text{pk}, \text{vk}) \sim \mathcal{S}(P) \\ (x, w) \sim \mathcal{D}(\text{pk}, \text{vk}) \\ \pi \sim \mathcal{S}(x) \end{array} \right]$$

- **Succinctness:** For all $(x, w) : \text{PublicInput} \times \text{SecretInput}$, if $\text{Some}(\pi) \sim \text{prove}(P, \text{pk}, x, w)$, then $|\pi| = \mathcal{O}(1)$, and $\text{verify}(\text{vk}, x, \pi)$ runs in time $\mathcal{O}(|x|)$.

4.2 Addresses and Key Components

Given a choice of HPKE we have the following definitions:

Definition 4.2.1 (Spending Key). A **SpendingKey** is the following pair of keys:

$$\begin{aligned} \text{spend} &: \text{HPKE.KA.SecretKey} \\ \text{view} &: \text{HPKE.KA.SecretKey} \end{aligned}$$

The second secret key, **view**, is called the **ViewingKey**.

Definition 4.2.2 (Receiving Key). A **ReceivingKey** is the following pair of keys:

$$\begin{aligned} \text{spend} &: \text{HPKE.KA.PublicKey} \\ \text{view} &: \text{HPKE.KA.PublicKey} \end{aligned}$$

which is derived from a spending key $\text{sk} : \text{SpendingKey}$ with the following algorithm:

$$\begin{aligned} \text{rk.spend} &:= \text{KA.derive}(\text{sk.spend}) \\ \text{rk.view} &:= \text{KA.derive}(\text{sk.view}) \end{aligned}$$

A keypair $(\text{sk}, \text{rk}) : \text{SpendingKey} \times \text{ReceivingKey}$, represents the ability to spend and receive **Assets** as a unique *representative participant* on the **Ledger**. Any user of the **MantaPay** protocol can create many such keypairs, but each one represents a different participant and **Assets** must be transferred between them using the **Transfer** protocol as if they were independently owned by different users. A **ReceivingKey** can be used to receive any number of **Assets** and the **SpendingKey** can be used to spend any number of those **Assets**. See § 4.4 for the protocol used to spend a subset of **Assets** owned by a single user.

Important: To every spending key $\text{sk} : \text{SpendingKey}$ we have an associated viewing key $\text{vk} : \text{ViewingKey} := \text{sk.view}$ which allows the owner to decrypt the encrypted messages associated to sk , but does not contain enough information to perform a spend with those **Assets**. This can be used for account auditing purposes, and for removing anonymity, but sharing this key should be done with caution.

In general, one may have a collection of viewing keys which can be used to separate the encrypted notes into different sets, by key. This way only certain transactions can be de-anonymized by certain parties.

4.3 Transfer Protocol

The **Transfer** protocol is the fundamental abstraction in **MantaPay** and facilitates the valid transfer of **Assets** among participants while preserving their anonymity. The **Transfer** is made up of special cryptographic constructions called **Senders** and **Receivers** which represent the private input and the private output of a transaction. To perform a **Transfer**, a protocol participant gathers the **SpendingKeys** they own, selects a subset of the **UTXOs** they have still not spent (with a fixed **AssetId**), collects **ReceivingKeys** from other participants for the outputs, assigning each key a subset of the input **Assets**, and then builds a **Transfer** object representing the transfer they want to build. From this **Transfer** object, they construct a **TransferPost** which they then send to the **Ledger** to be validated and stored, representing a completed state transition in the **Ledger**. The transformation from **Transfer** to **TransferPost** involves keeping the parts of the **Transfer** that *must* be known to the **Ledger** and for the parts that *must* not be known, substituting them for a *zero-knowledge proof* representing the validity of the secret information known to the participant, and the **Transfer** as a whole.

We begin by defining the cryptographic primitives involved in the **Transfer** protocol:

Definition 4.3.1 (Transfer Configuration). A **TransferConfiguration** is a collection of implementations of the following abstract cryptographic primitives:

- **Hybrid Public Key Encryption:** HPKE
- **Ephemeral-Key Commitment Scheme:** COM^{EK}
- **Commitment Scheme:** COM
- **Trapdoor Derivation Function:** $\text{KDF}^{\text{Trapdoor}}$
- **Dynamic Cryptographic Accumulator:** DCA
- **Zero-Knowledge Proving System:** NIZK

with the following notational conventions:

$$\begin{aligned} \text{KA} &:= \text{HPKE.KA} \\ \text{UTXO} &:= \text{COM.Output} \\ \text{VoidNumber} &:= \text{COM.Output} \\ \text{EncryptedNote} &:= \text{HPKE.Message} \\ \text{UTXOSet} &:= \text{DCA} \end{aligned}$$

and the following constraints:

$$\begin{aligned} \text{KDF}^{\text{Trapdoor}} &: \text{KA.SharedSecret} \rightarrow \text{COM.Trapdoor} \\ \text{UTXOSet.Item} &: \text{UTXO} \\ \text{ValidTransfer} &: \text{NIZK.Statement} \end{aligned}$$

where **ValidTransfer** is defined below.

TODO: Add the fact that **COM.Input** has a concatenation property. In general, add (de)serialization to the spec.

For the rest of this section, we assume the existence of a **TransferConfiguration** and use the primitives outlines above explicitly. We continue by defining the **Sender** and **Receiver** constructions as well as their public counterparts, the **SenderPost** and **ReceiverPost**.

Definition 4.3.2 (Transfer Sender). A **Sender** is the following tuple:

$$\begin{aligned} \text{sk} &: \text{SpendingKey} \\ \text{epk} &: \text{KA.PublicKey} \\ \text{trapdoor} &: \text{COM.Trapdoor} \\ \text{asset} &: \text{Asset} \\ \text{cm} &: \text{UTXO} \\ \text{cm}_z &: \text{UTXOSet.Output} \\ \text{cm}_w &: \text{UTXOSet.Witness} \\ \text{vn} &: \text{VoidNumber} \end{aligned}$$

A Sender, S , is constructed from a spending key $sk : \text{SpendingKey}$ and an encrypted message note : EncryptedNote with the following algorithm:

```

 $S.sk := sk$ 
 $c, epk := note$ 
 $\text{Some}(asset) := \text{HPKE.decrypt}(S.sk.view, epk, c)$ 
 $S.asset := asset$ 
 $S.epk := epk$ 
 $S.trapdoor := \text{KDF}^{\text{Trapdoor}}(\text{KA.agree}(S.sk.spend, S.epk))$ 
 $S.cm := \text{COM}_{S.trapdoor}(S.asset.id || S.asset.value)$ 
 $\text{Some}(cm_z, cm_w) := \text{UTXOSet.contains}(S.cm, \text{Ledger.utxos}())$ 
 $S.cm_z := cm_z$ 
 $S.cm_w := cm_w$ 
 $S.vn := \text{COM}_{S.trapdoor}(S.sk.spend)$ 

```

Definition 4.3.3 (Transfer Sender Post). A SenderPost is the following tuple extracted from a Sender :

```

 $cm_z : \text{UTXOSet.Output}$ 
 $vn : \text{VoidNumber}$ 

```

which are the parts of a Sender which should be *posted* to the Ledger .

Definition 4.3.4 (Transfer Receiver). A Receiver is the following tuple:

```

 $rk : \text{ReceivingKey}$ 
 $esk : \text{KA.SecretKey}$ 
 $trapdoor : \text{COM.Trapdoor}$ 
 $asset : \text{Asset}$ 
 $cm : \text{UTXO}$ 
 $note : \text{EncryptedNote}$ 

```

A Receiver , R , is constructed from a receiving key $rk : \text{ReceivingKey}$, an asset $asset : \text{Asset}$, and a given⁴ ephemeral secret key $esk : \text{HPKE.KA.SecretKey}$ with the following algorithm:

```

 $R.rk := rk$ 
 $R.esk := esk$ 
 $R.trapdoor := \text{KDF}^{\text{Trapdoor}}(\text{KA.agree}(R.esk, R.rk.spend))$ 
 $R.asset := asset$ 
 $R.cm := \text{COM}_{R.trapdoor}(R.asset.id || R.asset.value)$ 
 $R.note := \text{HPKE.encrypt}(R.rk.view, R.esk, R.asset)$ 

```

Definition 4.3.5 (Transfer Receiver Post). A ReceiverPost is the following tuple extracted from a Receiver :

```

 $cm : \text{UTXO}$ 
 $note : \text{EncryptedNote}$ 

```

which are the parts of a Receiver which should be *posted* to the Ledger .

Definition 4.3.6 (Transfer Sources and Sinks). A Source (or a Sink) is an Asset representing a public input (or output) of a Transfer .

⁴The ephemeral secret key is not directly chosen by the ledger participants building the Transfer . Instead, it is derived from other Transfer data and the current state of the ledger. See [Def 4.3.8](#) for more.

Definition 4.3.7 (Transfer Object). A Transfer is the following tuple:

$$\begin{aligned} \text{sources} &: \text{List}(\text{Asset}) \\ \text{senders} &: \text{List}(\text{Sender}) \\ \text{receivers} &: \text{List}(\text{Receiver}) \\ \text{sinks} &: \text{List}(\text{Asset}) \end{aligned}$$

The *shape* of a Transfer is the following 4-tuple of cardinalities of those sets

$$(|T.\text{sources}|, |T.\text{senders}|, |T.\text{receivers}|, |T.\text{sinks}|)$$

In order for a Transfer to be considered *valid*, it must adhere to the following constraints:

- **Same Id:** All the AssetIds in the Transfer must be equal.
- **Balanced:** The sum of input AssetValues must be equal to the sum of output AssetValues.
- **Well-formed Senders:** All of the Senders in the Transfer must be constructed according to the above Sender definition.
- **Well-formed Receivers:** All of the Receivers in the Transfer must be constructed according to the above Receiver definition.

In order to prove that these constraints are satisfied for a given Transfer, we build a zero-knowledge proof which will witness that the Transfer is valid and should be accepted by the Ledger. It is not necessary to prove that the encryption of Receiver.note and the decryption of a note from the Ledger are valid. Deviation from the protocol in encryption or decryption stages does not reduce the security of the protocol for honest participants.

Definition 4.3.8 (Transfer Validity Statement). A transfer $T : \text{Transfer}$ is considered *valid* if and only if

1. All the AssetIds in T are equal:

$$\left| \left(\bigcup_{a \in T.\text{sources}} a.\text{id} \right) \cup \left(\bigcup_{S \in T.\text{senders}} S.\text{asset.id} \right) \cup \left(\bigcup_{R \in T.\text{receivers}} R.\text{asset.id} \right) \cup \left(\bigcup_{a \in T.\text{sinks}} a.\text{id} \right) \right| = 1$$

2. The sum of input AssetValues is equal to the sum of output AssetValues:

$$\left(\sum_{a \in T.\text{sources}} a.\text{value} \right) + \left(\sum_{S \in T.\text{senders}} S.\text{asset.value} \right) = \left(\sum_{R \in T.\text{receivers}} R.\text{asset.value} \right) + \left(\sum_{a \in T.\text{sinks}} a.\text{value} \right)$$

3. For all $S \in T.\text{senders}$, the Sender S is well-formed:

$$\begin{aligned} S.\text{trapdoor} &= \text{KDF}^{\text{Trapdoor}}(\text{KA}.\text{agree}(S.\text{sk}.\text{spend}, S.\text{epk})) \\ S.\text{cm} &= \text{COM}_{S.\text{trapdoor}}(S.\text{asset.id} \parallel S.\text{asset.value}) \\ S.\text{vn} &= \text{COM}_{S.\text{trapdoor}}(S.\text{sk}.\text{spend}) \\ \text{UTXOSet.verify}(S.\text{cm}, S.\text{cm}_z, S.\text{cm}_w) &= \text{True} \end{aligned}$$

4. For all $(i, R) \in \text{enumerate}(T.\text{receivers})$, the Receiver R is well-formed at index i with respect to FAIR:

$$\begin{aligned} R.\text{esk} &= \text{COM}_{\tau}^{\text{EK}}(\mathcal{L} \parallel i \parallel R.\text{rk}.\text{spend}) \\ R.\text{trapdoor} &= \text{KDF}^{\text{Trapdoor}}(\text{KA}.\text{agree}(R.\text{esk}, R.\text{rk}.\text{spend})) \\ R.\text{cm} &= \text{COM}_{R.\text{trapdoor}}(R.\text{asset.id} \parallel R.\text{asset.value}) \end{aligned}$$

where $\tau \sim \text{COM}^{\text{EK}}.\text{TrapdoorDistribution}$ is a randomly chosen trapdoor and \mathcal{L} is the ledger checkpoint, a constant determined by the current state of the ledger.

Notation: This statement is denoted **ValidTransfer** and is assumed to be expressible as a Statement of NIZK.

Definition 4.3.9 (Transfer Post). A `TransferPost` is the following tuple:

$$\begin{aligned} \mathcal{L} &: \text{Ledger.Checkpoint} \\ \text{sources} &: \text{List}(\text{Source}) \\ \text{senders} &: \text{List}(\text{SenderPost}) \\ \text{receivers} &: \text{List}(\text{ReceiverPost}) \\ \text{sinks} &: \text{List}(\text{Sink}) \\ \pi &: \text{NIZK.Proof} \end{aligned}$$

A `TransferPost`, P , is constructed by assembling the zero-knowledge proof of `Transfer` validity from a known proving key $\text{pk} : \text{NIZK.ProvingKey}$ and a given $T : \text{Transfer}$:

$$\begin{aligned} x &:= \text{Transfer.public}(T) \\ w &:= \text{Transfer.secret}(T) \\ \text{Some}(\pi) &\sim \text{NIZK.prove}_{\text{pk}}^{\text{ValidTransfer}}(x, w) \\ P.\mathcal{L} &:= x.\mathcal{L} \\ P.\text{sources} &:= x.\text{sources} \\ P.\text{senders} &:= x.\text{senders} \\ P.\text{receivers} &:= x.\text{receivers} \\ P.\text{sinks} &:= x.\text{sinks} \\ P.\pi &:= \pi \end{aligned}$$

where `Transfer.public` returns `SenderPosts` for each `Sender` in T and `ReceiverPosts` for each `Receiver` in T , keeping `Sources` and `Sinks` as they are, and `Transfer.secret` returns all the rest of T which is not part of the output of `Transfer.public`.

Now that a participant has constructed a transfer post $P : \text{TransferPost}$ they can send it to the `Ledger` for verification.

Definition 4.3.10 (Ledger-side Transfer Validity). To check that P represents a valid `Transfer`, the ledger checks the following:

- **Valid Checkpoint:** The checkpoint $P.\mathcal{L}$ is valid and matches the checkpoint of the current ledger state.
- **Public Withdraw:** All the public addresses corresponding to the `Assets` in $P.\text{sources}$ have enough public balance (i.e. in the `PublicLedger`) to withdraw the given `Asset`.
- **Public Deposit:** All the public addresses corresponding to the `Assets` in $P.\text{sinks}$ exist.
- **Shielded Withdraw:** The total balance in $P.\text{sinks}$ does not exceed the amount in the `ShieldedAssetPool` balance.
- **Current Accumulated State:** The `UTXOSet.Output` stored in each $P.\text{senders}$ is equal to current accumulated value, `UTXOSet.current(Ledger.utxos())`, for the current state of the `Ledger`.
- **New VoidNumbers:** All the `VoidNumbers` in $P.\text{senders}$ are unique, and no `VoidNumber` in $P.\text{senders}$ has already been stored in the `Ledger.VoidNumberSet`.
- **New UTXOs:** All the `UTXOs` in $P.\text{receivers}$ are unique, and no `UTXO` in $P.\text{receivers}$ has already been stored on the ledger.
- **Verify Transfer:** Check that $\text{NIZK.verify}_{\text{vk}}(P.\text{sources} \parallel P.\text{senders} \parallel P.\text{receivers} \parallel P.\text{sinks}, P.\pi) = \text{True}$.

Definition 4.3.11 (Ledger Transfer Update). After checking that a given `TransferPost` P is valid, the `Ledger` updates its state by performing the following changes:

- **Public Updates:** All the relevant public accounts on the `PublicLedger` are updated to reflect their new balances using the `Sources` and `Sinks` present in P .
- **Pool Update:** The `ShieldedAssetPool` balance is updated to reflect the new shielded balances, increasing by the amount:

$$\left(\sum_{a \in P.\text{sources}} a.\text{value} \right) - \left(\sum_{a \in P.\text{sinks}} a.\text{value} \right)$$

- **UTXOSet Update:** The new UTXOs are appended to the UTXOSet.
- **VoidNumberSet Update:** The new VoidNumbers are appended to the VoidNumberSet.

4.4 Semantic Transactions

For MantaPay participants to use the **Transfer** protocol, they will need to keep track of the current state of their shielded assets and use them to build **TransferPosts** to send to the **Ledger**. The *shielded balance* of any participant is the sum of the balances of their shielded assets, but this balance may be fragmented into arbitrarily many pieces, as each piece represents an independent asset that the participant received as the output of some **Transfer**. To then spend a subset of their shielded balance, the participant would need to accumulate all of the relevant fragments into a large enough *shielded asset* to spend all at once, building a collection of **TransferPosts** to send to the **Ledger**.

Any wallet implementation should see that their users need not keep track of this complexity themselves. Instead, like a public ledger, the notion of a *transaction* between one participant and another should be viewed as a single action that the user can take, performing a withdrawal from their shielded balance. To describe such a *semantic transaction*, we assume the existence of two transfer shapes⁵: **Mint** with shape $(1, 0, 1, 0)$ and **PrivateTransfer** with shape $(0, N, N, 0)$ for some natural number $N > 1$.

For a fixed spending key, $sk : \text{SpendingKey}$, and asset id, $id : \text{AssetId}$, we are given a balance state, $\mathcal{B} : \text{FinSet}(\text{KA.PublicKey} \times \text{AssetValue})$, a set of key-balance pairs for unspent assets, a total balance to withdraw, $total : \text{AssetValue}$, and a receiving key $rk : \text{ReceivingKey}$. We can then compute

$\text{BUILDTRANSACTION}(sk, \mathcal{B}, total, rk)$

to receive a $\text{List}(\text{TransferPost})$ to send to the ledger, representing the transfer of $total$ to rk .

Algorithm 1 Semantic Transaction Algorithm

```

procedure BUILDTRANSACTION( $sk, \mathcal{B}, total, rk$ )
   $B \leftarrow \text{Sample}(total, \mathcal{B})$  ▷ Samples pairs from  $\mathcal{B}$  that total at least  $total$ 
  if  $\text{len}(B) = 0$  then
    return [] ▷ Insufficient Balance
  end if
   $P \leftarrow []$  ▷ Allocate a new list for TransferPosts
  while  $\text{len}(B) > N$  do ▷ While there are enough pairs to make another Transfer
     $A \leftarrow []$ 
    for  $b \in (B, N)$  do ▷ Get the next  $N$  pairs from  $B$ 
       $S \leftarrow \text{BuildSenders}_{sk}(b)$ 
       $[acc, zs...] \leftarrow \text{BuildAccumulatorAndZeroes}_{sk}(S)$  ▷ Build a new accumulator and zeroes
       $P \leftarrow P + \text{TransferPost}(\text{Transfer}([], S, [acc, zs...], []))$ 
       $(A, Z) \leftarrow (A + (acc.\tilde{d}, acc.asset.value), Z + zs)$  ▷ Save  $acc$  for the next loop,  $zs$  for the end
    end for
     $B \leftarrow A + \text{remainder}(B, N)$ 
  end while
   $S \leftarrow \text{PrepareZeroes}_{sk}(N, B, Z, P)$  ▷ Use  $Z$  and Mints to make  $B$  go up to  $N$  in size.
   $R \leftarrow \text{BuildReceiver}_{sk}(rk, S)$ 
   $[c, zs...] \leftarrow \text{BuildAccumulatorAndZeroes}_{sk}(S)$ 
  return  $P + \text{TransferPost}(\text{Transfer}([], S, [R, c, zs...], []))$ 
end procedure

```

If all of the **Transfers** are accepted by the ledger, the balance state \mathcal{B} should be updated accordingly, removing all of the pairs which were used in the **Transfer**.

5 Concrete Protocol

TODO: other than cryptographic schemes, are there any implementation details we want to include here?

⁵Other **Transfer** accumulation algorithms are possible with different starting shapes.

5.1 Concrete Cryptographic Schemes

Definition 5.1.1 (Commitment Scheme). For COM, we use the *Pedersen Commitment Scheme* with 256 windows of size 4 each.

Definition 5.1.2 (Key-Agreement Scheme). For KA, we use

TODO: key-agreement scheme: x25519 elliptic curve Diffie-Hellman key exchange

Definition 5.1.3 (Symmetric-Key Encryption Scheme). For SYM, we use

TODO: symmetric-key encryption scheme: AES-GCM with magic-number nonce and no associated data

Definition 5.1.4 (Key-Derivation Function). For KDF, we use

TODO: key-derivation function: Blake2s with magic-number salt

Definition 5.1.5 (Hybrid Public Key Encryption). For HPKE, we use

TODO: integrated encryption scheme: combination of above encryption protocols

Definition 5.1.6 (Dynamic Cryptographic Accumulator). For DCA, we use

TODO: dynamic cryptographic accumulator: Merkle Tree with Pedersen hashes (incremental tree for the ledger is an optimization since it only needs to know enough to compute the accumulated value)

Definition 5.1.7 (Non-Interactive Zero-Knowledge Proofs). For NIZK, we use

TODO: non-interactive zero-knowledge proving system: Groth16 and/or PLONK

6 Acknowledgements

TODO: add acknowledgements

7 References

References

- [1] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. Hybrid Public Key Encryption. Internet-Draft draft-irtf-cfrg-hpke-12, Internet Engineering Task Force, September 2021. Work in Progress.