

MantaPay Protocol Specification

v0.4.0

Shumo Chu and Brandon H. Gomes

November 1, 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	1
2	Notation	1
3	Concepts	2
3.1	Assets	2
3.2	Addresses	2
3.3	Ledger	2
3.3.1	UTXOs and the UTXOSet	3
3.3.2	EncryptedNotes	3
3.3.3	VoidNumbers and the VoidNumberSet	3
4	Abstract Protocol	4
4.1	Abstract Cryptographic Schemes	4
4.2	Addresses and Key Components	7
4.3	Transfer Protocol	7
4.4	Semantic Transactions	11
5	Concrete Protocol	11
5.1	Constants	11
5.2	Concrete Cryptographic Schemes	11
6	Differences from MANTADAP	11
6.1	Reusable Addresses	11
6.2	Transfer Circuit Unification	11
7	Acknowledgements	11
8	References	11

1 Introduction

TODO: add introductory remarks

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 $(-, -) : T \rightarrow (S \rightarrow T \times S)$.
- 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 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 vector, 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.

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

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.

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*.

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. Enforcing this second invariant can be solved by using 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**.

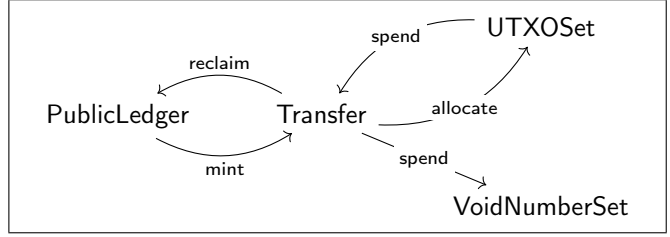


Figure 1: Lifecycle of an Asset.

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.
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 a key diversifier, 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,

³ A 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 *consensus mechanism* that secures the MantaPay ledger is beyond the scope of this paper.

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 primitives* used in the MantaPay protocol.

Definition 4.1.1. 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 $\text{COM}_r(x)$.

Definition 4.1.2. A *hash function* CRH is defined by the schema:

$$\begin{aligned} \text{Input} &: \text{Type} \\ \text{Output} &: \text{Type} \\ \text{hash} &: \text{Input} \rightarrow \text{Output} \end{aligned}$$

with the following properties:

- **Pre-Image Resistance:** For a given $y : \text{Output}$, it is infeasible to find $x : \text{Input}$ such that $\text{hash}(x) = y$.
- **Collision Resistance:** It is infeasible to find an $x_1, x_2 : \text{Input}$ such that $x_1 \neq x_2$ and $\text{hash}(x_1) = \text{hash}(x_2)$.

Notation: For convenience we refer to $\text{CRH.hash}(x)$ by $\text{CRH}(x)$.

Definition 4.1.3. 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:

- **Validity:** 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. A *key-agreement scheme* KA is defined by the schema:

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

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 4.1.5. A *key-derivation function* KDF defined over a symmetric-key encryption scheme SYM and a key-agreement scheme KA is a function of type:

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

Definition 4.1.6. An *integrated encryption scheme* IES is a hybrid encryption scheme made of up a symmetric-key encryption scheme SYM, a key-agreement scheme KA, and a KDF to convert from KA.SharedSecret to SYM.Key. We can define the following encryption/decryption algorithms:

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

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

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

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

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 $\text{Message} := \text{KA.PublicKey} \times \text{SYM.Ciphertext}$, and the above two algorithms by

$$\text{encrypt} : \text{KA.SecretKey} \times \text{KA.PublicKey} \times \text{SYM.Plaintext} \rightarrow \text{Message}$$

$$\text{decrypt} : \text{KA.SecretKey} \times \text{SYM.Ciphertext} \rightarrow \text{Option}(\text{SYM.Plaintext})$$

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

TODO: add explicit message authentication

Definition 4.1.7. A *key-diversification scheme* KDIV over a key-agreement scheme KA is defined by the schema:

$$\text{public} : \text{KA.SecretKey} \times \text{KA.PublicKey} \times \text{KA.PublicKey} \rightarrow \text{KA.PublicKey}$$

$$\text{secret} : \text{KA.PublicKey} \times \text{KA.SecretKey} \times \text{KA.SecretKey} \rightarrow \text{KA.SecretKey}$$

Notation: We refer to the first argument to a KDIV function as the *diversifier* and we write it as a subscript

$$\text{public}_d(x, y) := \text{public}(d, x, y) \text{ and } \text{secret}_d(x, y) := \text{secret}(d, x, y)$$

For convenience we also write KDIV_d to mean $\text{KDIV}.\text{public}_d$ or $\text{KDIV}.\text{secret}_d$, when the context is clear.

Every KDIV also has the following properties:

- **Derivation Invariance:** For any diversifier $d : \text{KA.SecretKey}$ and pair of secret keys (sk_1, sk_2) we have

$$\text{KDIV}_d(\text{KA.derive}(sk_1), \text{KA.derive}(sk_2)) = \text{KA.derive}(\text{KDIV}_{\text{KA.derive}(d)}(sk_1, sk_2))$$

- **TODO:** security properties?

Definition 4.1.8. A *dynamic cryptographic accumulator* DCA is defined by the schema:

$$\text{Item} : \text{Type}$$

$$\text{State} : \text{Type}$$

$$\text{Checkpoint} : \text{Type}$$

$$\text{Proof} : \text{Type}$$

$$\text{checkpoint} : \text{State} \rightarrow \text{Checkpoint}$$

$$\text{update} : \text{Item} \times \text{State} \rightarrow \text{State}$$

$$\text{contains} : \text{Item} \times \text{State} \rightarrow \text{Option}(\text{Checkpoint} \times \text{Proof})$$

$$\text{verify} : \text{Item} \times \text{Checkpoint} \times \text{Proof} \rightarrow \text{Bool}$$

with the following properties:

- **TODO:** add checkpoint, update, contains, and verify properties
- **TODO:** security properties

Definition 4.1.9. A *non-interactive zero-knowledge proving system* ZKPS is defined by the schema:

```

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

```

Notation: We use the following notation for a ZKPS:

- 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

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

Every ZKPS 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 there exists a proof $\pi : \text{Proof}$, such that $\text{Some}(\pi) \in \text{prove}_{\text{pk}}^P(x, w)$, 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:

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

- **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:

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

- **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 IES we have the following definitions:

Definition 4.2.1. A `SpendingKey` is the following pair of keys:

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

The first secret key, `view`, is called the `ViewingKey`.

Definition 4.2.2. A `ReceivingKey` is the following pair of keys:

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

which is derived from a spending key `sk : SpendingKey` by deriving each component:

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

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 `sk : SpendingKey` we have an associated viewing key `vk : ViewingKey := 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.

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. The `TransferConfiguration` is a collection of implementations of the following abstract cryptographic primitives:

- **Integrated Encryption Scheme:** IES
- **Commitment Scheme:** COM
- **Hash Function:** CRH
- **Dynamic Cryptographic Accumulator:** DCA
- **Zero-Knowledge Proving System:** ZKPS

with the following notational conventions:

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

and the following constraints:

$$\begin{aligned}
\text{COM.Trapdoor} &= \text{KA.PublicKey} \\
\text{ZKPS.Statement} &= \text{ValidTransfer}
\end{aligned}$$

where `ValidTransfer` is defined below.

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

TODO: Add the fact that we have a conversion from `CRH.Output` to `KA.SecretKey`, notably for diversifiers.

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. A `Sender` is the following tuple:

$$\begin{aligned}
sk &: \text{SpendingKey} \\
\tilde{d} &: \text{KA.PublicKey} \\
\text{trapdoor} &: \text{KA.PublicKey} \\
\text{asset} &: \text{Asset} \\
cm &: \text{UTXO} \\
cm_c &: \text{UTXOSet.Checkpoint} \\
cm_\pi &: \text{UTXOSet.Proof} \\
vn &: \text{VoidNumber}
\end{aligned}$$

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

$$\begin{aligned}
S.sk &:= sk \\
\tilde{d}, c &:= \text{note} \\
\text{Some}(\text{asset}) &:= \text{IES.decrypt}(S.sk.view, c) \\
S.asset &:= \text{asset} \\
S.\tilde{d} &:= \tilde{d} \\
S.\text{trapdoor} &:= \text{KA.derive}(KDIV_{S.\tilde{d}}(S.sk.view, S.sk.spend)) \\
S.cm &:= \text{COM}_{S.\text{trapdoor}}(S.asset) \\
\text{Some}(cm_c, cm_\pi) &:= \text{UTXOSet.contains}(S.cm, \text{Ledger.utxos}()) \\
S.cm_c &:= cm_c \\
S.cm_\pi &:= cm_\pi \\
S.vn &:= \text{COM}_{S.\tilde{d}}(S.sk.view || S.sk.spend)
\end{aligned}$$

Definition 4.3.3. A `SenderPost` is the following tuple extracted from a `Sender`:

$$\begin{aligned}
cm_c &: \text{UTXOSet.Checkpoint} \\
vn &: \text{VoidNumber}
\end{aligned}$$

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

Definition 4.3.4. A Receiver is the following tuple:

$rk : \text{ReceivingKey}$
 $d : \text{KA.SecretKey}$
 $\text{trapdoor} : \text{KA.PublicKey}$
 $\text{asset} : \text{Asset}$
 $\text{cm} : \text{UTXO}$
 $\text{note} : \text{EncryptedNote}$

A Receiver, R , is constructed from a receiving key $rk : \text{ReceivingKey}$, an asset $\text{asset} : \text{Asset}$, and a chosen diversifier $d : \text{IES.KA.SecretKey}$ with the following algorithm:

$R.rk := rk$
 $R.d := d$
 $R.\text{trapdoor} := \text{KDIV}_{R.d}(R.rk.\text{view}, R.rk.\text{spend})$
 $R.\text{asset} := \text{asset}$
 $R.\text{cm} := \text{COM}_{R.\text{trapdoor}}(R.\text{asset})$
 $R.\text{note} := \text{IES.encrypt}(R.d, R.rk.\text{view}, \text{asset})$

Definition 4.3.5. A ReceiverPost is the following tuple extracted from a Receiver:

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

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

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

Definition 4.3.7. A Transfer is the following tuple:

$\text{sources} : \text{FinSet}(\text{Asset})$
 $\text{senders} : \text{FinSet}(\text{Sender})$
 $\text{receivers} : \text{FinSet}(\text{Receiver})$
 $\text{sinks} : \text{FinSet}(\text{Asset})$

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:

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

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

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

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

$$\begin{aligned} S.trapdoor &= KA.derive(KDIV_{S.d}(S.sk.view, S.sk.spend)) \\ S.cm &= COM_{S.trapdoor}(S.asset) \\ S.vn &= COM_{S.d}(S.sk.view || S.sk.spend) \\ UTXOSet.verify(S.cm, S.cm_c, S.cm_\pi) &= \text{True} \end{aligned}$$

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

$$\begin{aligned} R.d &= CRH(i || R.rk.view || R.rk.spend || \text{FAIR}) \\ R.trapdoor &= KDIV_{R.d}(R.rk.view, R.rk.spend) \\ R.cm &= COM_{R.trapdoor}(R.asset) \end{aligned}$$

where FAIR is the following constant, using the ledger as a randomness oracle:

$$\text{FAIR} := CRH(\text{UTXOSet.checkpoint}(\text{Ledger.utxos}()) || \text{Concat}_{S \in T.senders}(S.sk.spend))$$

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

Definition 4.3.9. A TransferPost is the following tuple:

$$\begin{aligned} \text{sources} &: \text{FinSet}(\text{Source}) \\ \text{senders} &: \text{FinSet}(\text{SenderPost}) \\ \text{receivers} &: \text{FinSet}(\text{ReceiverPost}) \\ \text{sinks} &: \text{FinSet}(\text{Sink}) \\ \pi &: \text{ZKPS.Proof} \end{aligned}$$

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

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

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. To check that P represents a valid Transfer, the ledger checks the following:

- **Public Withdraw:** All the public addresses corresponding to the Assets in $P.sources$ 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 $P.sinks$ exist.
- **Current Checkpoint:** The UTXOSet.Checkpoint stored in each $P.senders$ is equal to current checkpoint, $\text{UTXOSet.checkpoint}(\text{Ledger.utxos}())$, for the current state of the Ledger.

- **New VoidNumbers:** All the VoidNumbers in $P.senders$ are unique, and no VoidNumber in $P.senders$ has already been stored in the `Ledger.VoidNumberSet`.
- **New UTXOs:** All the UTXOs in $P.receivers$ are unique, and no UTXO in $P.receivers$ has already been stored on the ledger.
- **Verify Transfer:** Check that $ZKPS.verify_{vk}(P.sources || P.senders || P.receivers || P.sinks, P.\pi) = \text{True}$.

4.4 Semantic Transactions

TODO: add semantic transactions

5 Concrete Protocol

5.1 Constants

TODO: add constants for the protocol

5.2 Concrete Cryptographic Schemes

TODO: add names of cryptographic scheme implementations

6 Differences from MANTA_{DAP}

6.1 Reusable Addresses

TODO: compare old one-time address protocol to reusable addresses and why reusable is better

6.2 Transfer Circuit Unification

TODO: compare new single transfer circuit to the many old circuits

7 Acknowledgements

TODO: add acknowledgements

8 References

TODO: add references