MantaPay Protocol Specification
v1.0.0

Shumo Chu, Boyuan Feng, Brandon H. Gomes, Francisco Hernández Iglesias, and Todd Norton *

September 23, 2022

Abstract

MantaPay is an implementation of a *decentralized anonymous payment* scheme based on the MANTA$_{\texttt{DAP}}$ protocol outlined in the original MANTA whitepaper.

Contents

*ordered alphabetically

1 Introduction

MantaPay aims to solve the long-standing privacy problems facing cryptocurrencies. At its heart, it uses various cryptographic constructions including NIZK (non-interactive zero knowledge proof) systems to ensure user privacy from *first principles* and to build the foundational layer for programmable private money. The MantaPay protocol provides the following features:

1. Elastic Multi-Asset Shielded Pool: A shielded pool for every kind of asset with elastic annonymity set resizing
2. Verifiable Viewing Keys: Opt-in transaction transparency with audit correctness assurance
3. Programmable zkAssets: New Transparent UTXO model allowing programmability layers to be built on top of the shielded pool
4. Delegated Proof Generation: Decoupling the spending access from the proof generation access gives hardware wallets native support for zkAssets

2 Notation

The following notation is used throughout this specification:

- Type is the type of types[1].
- If $x : T$ then x is a value and T is a type, denoted $T : \mathsf{Type}$, and we say that x *has type* T.
- Bool is the type of booleans with values True and False.
- For any types $A : \mathsf{Type}$ and $B : \mathsf{Type}$ we denote the *type of functions* from A to B as $A \to B : \mathsf{Type}$.
- For any types $A : \mathsf{Type}$ and $B : \mathsf{Type}$ we denote the *product type* over A and B as $A \times B : \mathsf{Type}$ with constructor $(-,-) : A \to (B \to A \times B)$. Depending on context, we may omit the constructor and inline the pair into another constructor/destructor. For example, if $f : A \times B \to C$ we can denote $f((a,b))$ as $f(a,b)$ to reduce the number of parentheses.
- For any type $T : \mathsf{Type}$, we define $\mathsf{Option}\langle T\rangle : \mathsf{Type}$ as the inductive type with constructors:

$$\begin{aligned}\mathsf{None} &: \mathsf{Option}\langle T\rangle \\ \mathsf{Some} &: T \to \mathsf{Option}\langle T\rangle\end{aligned}$$

- We denote the *type of finite sets* over a type $T : \mathsf{Type}$ as $\mathsf{FinSet}\langle T\rangle : \mathsf{Type}$. The membership predicate for a value $x : T$ in a finite set $S : \mathsf{FinSet}(T)$ is denoted $x \in S$.
- We denote the *type of finite ordered sets* over a type $T : \mathsf{Type}$ as $\mathsf{List}\langle T\rangle : \mathsf{Type}$. This can either be defined by an inductive type or as a $\mathsf{FinSet}(T)$ 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 $T : \mathsf{Type}$ as $\mathfrak{D}\langle T\rangle : \mathsf{Type}$. A value x sampled from $\mathfrak{D}\langle T\rangle$ is denoted $x \sim \mathfrak{D}\langle T\rangle$ and the fact that the value x belongs to the range of $\mathfrak{D}\langle T\rangle$ is denoted $x \in \mathfrak{D}\langle T\rangle$. So namely, $y \in \{x \mid x \sim \mathfrak{D}\langle T\rangle\} \leftrightarrow y \in \mathfrak{D}\langle T\rangle$.
- We denote the equality predicate as $(- = -) : T \times T \to \mathsf{Type}$ and the equality function as $\mathsf{eq} : T \times T \to \mathsf{Bool}$ whenever they exist.
- We denote the selection function as $\mathsf{select} : \mathsf{Bool} \times T \times T \to T$. For a boolean $b : \mathsf{Bool}$ and two values $t_1, t_2 : T$, $\mathsf{select}(b, t_1, t_2)$ returns t_1 when $b = \mathsf{True}$ and returns t_2 when $b = \mathsf{False}$.
- 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.

[1] By *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.

3 Concepts

3.1 zkAssets

The zkAsset is the fundamental currency object in the MantaPay protocol. An asset a : zkAsset is a tuple

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

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 facilitiates the private ownership and private transfer of zkAssets.

zkAssets are the basic building-blocks of *transactions* which consume a set of input zkAssets and produce a set of transformed output zkAssets. To preserve the economic value stored in zkAssets, 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 value 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 UTXOs

But zkAssets are not private on their own. A UTXO is a container for a zkAsset that hides its value and its owner and is the main object that MantaPay uses to transfer the spending power of zkAssets between different protocol participants. A UTXO is a cryptographic commitment along with some associated data that represents a spendable subset of an account stored in the protocol. In the MantaPay protocol, UTXOs come in two flavors, *opaque* and *transparent*. The *opaque* UTXOs are completely private and they do not reveal the owner or underlying asset contained in them, whereas *transparent* UTXOs reveal the underlying asset but not the owner. The *opaque* UTXO is used for the private transfer of zkAssets and the *transparent* UTXO is used to give programability to zkAssets whenever the MantaPay protocol lives in the same environment as other smart contracts by allowing contracts to control the AssetId and AssetValue stored in the *transparent* UTXO.

3.3 Nullifiers

One of the important ways that privacy is preserved for zkAssets across many transactions is that the exact transaction where a UTXO is spent is not known to the public. Instead, only the owner of the zkAsset, or anyone with the appropriate viewing key, can know this information. The Nullifier is another cryptographic commitment that takes the place of the UTXO when it is spent and it is cryptographically hard for any particular UTXO to be derived from its Nullifier.

3.4 zkAddresses

In order for MantaPay participants to receive zkAssets via the Transfer protocol, they create *zk-addresses* which they use as identifiers to represent them on the ledger.

$$\begin{tikzcd}
 & \mathsf{sk} \arrow[dl] \arrow[dr] & & & \\
\mathsf{sk}_\alpha \arrow[dr] & & \mathsf{ak} \arrow[dl] \arrow[r] & \mathsf{vk} \arrow[r] & \mathsf{pk} \\
 & \mathsf{ak}_\alpha & & &
\end{tikzcd}$$

Figure 1: Key Schedule for MantaPay.

MantaPay uses four kinds of keys all derived from a base secret, spending key sk, which give the following kinds of privileged access in the protocol:

- **zkAddress** (send): Access to the zk-address pk gives the user the right to send zkAssets to the owner of the associated sk.
- **Viewing Key** (view): Access to the viewing key vk gives the user the right to view all transactions for the owner of the associated sk.

[2] It 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*.

- **Proof Authorization Key** (prove): Proof authorization key ak gives the user the right to build the Transfer proof on behalf of the owner of sk. This key is used when delegating proof generation to a semi-trusted entity while still protecting the spending rights associated to the sk, for example, if a hardware wallet holds sk it can ask a more capable computer to produce the Transfer proof for it without sending the spending rights off of the hardware wallet.
- **Spending Key** (spend): Access to the spending key sk gives total control over the assets owned by this secret, including spending, proof generation, and viewing.

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

3.5 Notes

The encrypted Note is the primary means of communication in the MantaPay protocol. For a zkAddress owner to know that they have received a zkAsset and can now spend it they decrypt Notes with their viewing key to discover how much of an asset they have received and what information they need to spend it. The Note is also used to keep track of the balances of an entire account over its transaction history.

There are two kinds of Notes in the MantaPay protocol, *incoming* Notes and *outgoing* Notes. The IncomingNote is attached to every new UTXO and contains the same zkAsset as the UTXO and also a secret randomizer used to hide the UTXO commitment. The OutgoingNote is attached to every new Nullifier and contains the same zkAsset as the UTXO that the Nullifier is marking. When performing accounting over a zkAddress to measure how much of a particular AssetId that address controls, the AssetValue stored in the IncomingNotes should be *added* to the running total whereas the AssetValue stored in the OutgoingNotes should be *subtracted* from the running total as they represent inflows and outflows respectively.

3.6 ShieldedPool

The ShieldedPool is a data structure that contains the necessary data to enable the MantaPay Transfer protocol. The ShieldedPool is made up of the following three general storage groups:

- UTXO Storage: Contains all of the UTXOs that have ever been created along with their IncomingNotes
- Nullifier Storage: Contains all of the Nullifiers that have ever been created along with their OutgoingNotes
- Public Pool Account: The public account of the pool itself that holds a backing of all the zkAssets held in the UTXOs in the pool. Depositing into or withdrawing out of the pool has to go through this account.

There are two general requirements on the UTXO and Nullifier storage items:

1. Fast non-membership query for UTXOs and Nullifiers
2. Fast insertion and insertion-order iteration over $(\mathsf{UTXO}, \mathsf{IncomingNote})$ and $(\mathsf{Nullifier}, \mathsf{OutgoingNote})$ pairs

In order to satisfy both of these requirements we have the following breakdown of the storage:

- UTXO Storage:
  - $\mathsf{UTXOSet} : \mathsf{UTXO} \to \mathsf{Bool}$
  - $\mathsf{UTXOStorageInsertionOrder} : \mathbb{N} \to (\mathsf{UTXO}, \mathsf{IncomingNote})$
- Nullifier Storage:
  - $\mathsf{NullifierSet} : \mathsf{Nullifier} \to \mathsf{Bool}$
  - $\mathsf{NullifierStorageInsertionOrder} : \mathbb{N} \to (\mathsf{Nullifier}, \mathsf{OutgoingNote})$

where we use the sets for fast non-membership checks and the insertion order maps for insertion-order preserving insertion and iteration.

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:

$$\begin{aligned} \mathsf{Randomness} &: \mathsf{Type} \\ \mathsf{Input} &: \mathsf{Type} \\ \mathsf{Output} &: \mathsf{Type} \\ \mathsf{commit} &: \mathsf{Randomness} \times \mathsf{Input} \to \mathsf{Output} \end{aligned}$$

with the following properties:

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

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

Definition 4.1.2 (Hash Function)**.** A *hash function* HASH is defined by the schema:

$$\begin{aligned} \mathsf{Input} &: \mathsf{Type} \\ \mathsf{Output} &: \mathsf{Type} \\ \mathsf{hash} &: \mathsf{Input} \to \mathsf{Output} \end{aligned}$$

with the following properties:

- **Collision Resistance**: It is infeasible to find $a, b : \mathsf{Input}$ such that $a \neq b$ and $\mathsf{hash}(a) = \mathsf{hash}(b)$.
- **Pre-Image Resistance**: Given $y : \mathsf{Output}$, it is infeasible to find an $x : \mathsf{Input}$ such that $\mathsf{hash}(x) = y$.
- **Second Pre-Image Resistance**: Given $a : \mathsf{Input}$, it is infeasible to find another $b : \mathsf{Input}$ such that $a \neq b$ and $\mathsf{hash}(a) = \mathsf{hash}(b)$.

We can also ask that a hash function be *binding* or *hiding* as in the above *Commitment Scheme* definition if we partition the Input space into a separate $\mathsf{Randomness}$ and Input space.

Notation: For convenience, we may refer to $\mathsf{HASH.hash}(x)$ by $\mathsf{HASH}(x)$.

Definition 4.1.3 (Signature Scheme)**.** A *signature scheme* SIG is defined by the schema:

$$\begin{aligned} \mathsf{SigningKey} &: \mathsf{Type} \\ \mathsf{VerifyingKey} &: \mathsf{Type} \\ \mathsf{Randomness} &: \mathsf{Type} \\ \mathsf{Message} &: \mathsf{Type} \\ \mathsf{Signature} &: \mathsf{Type} \\ \mathsf{derive} &: \mathsf{SigningKey} \to \mathsf{VerifyingKey} \\ \mathsf{sign} &: \mathsf{SigningKey} \times \mathsf{Randomness} \times \mathsf{Message} \to \mathsf{Signature} \\ \mathsf{verify} &: \mathsf{VerifyingKey} \times \mathsf{Signature} \times \mathsf{Message} \to \mathsf{Bool} \end{aligned}$$

with the following properties:

- **Correctness**: For a given $\mathsf{sk} : \mathsf{SigningKey}$, $r : \mathsf{Randomness}$, and $m : \mathsf{Message}$, we have that

$$\mathsf{verify}(\mathsf{derive}(\mathsf{sk}), \mathsf{sign}(\mathsf{sk}, r, m), m) = \mathsf{True}$$

- **TODO**:

Definition 4.1.4 (Authenticated Encryption Scheme)**.** An *authenticated encryption* scheme AUTH is defined by the schema:

$$\begin{aligned} \mathsf{Key} &: \mathsf{Type} \\ \mathsf{Plaintext} &: \mathsf{Type} \\ \mathsf{Ciphertext} &: \mathsf{Type} \\ \mathsf{Tag} &: \mathsf{Type} \\ \mathsf{encrypt} &: \mathsf{Key} \times \mathsf{Plaintext} \to \mathsf{Tag} \times \mathsf{Ciphertext} \\ \mathsf{decrypt} &: \mathsf{Key} \times \mathsf{Tag} \times \mathsf{Ciphertext} \to \mathsf{Option}(\mathsf{Plaintext}) \end{aligned}$$

with the following properties:

- **Correctness**: For a given $k : \mathsf{Key}$, $p : \mathsf{Plaintext}$, we have that $\mathsf{decrypt}(k, \mathsf{encrypt}(k, p)) = \mathsf{Some}(p)$.
- **TODO**: ...

Definition 4.1.5 (Dynamic Cryptographic Accumulator)**.** A *dynamic cryptographic accumulator* DCA is defined by the schema:

$$\begin{aligned}
\mathsf{Item} &: \mathsf{Type} \\
\mathsf{Output} &: \mathsf{Type} \\
\mathsf{Witness} &: \mathsf{Type} \\
\mathsf{State} &: \mathsf{Type} \\
\mathsf{current} &: \mathsf{State} \to \mathsf{Output} \\
\mathsf{insert} &: \mathsf{Item} \times \mathsf{State} \to \mathsf{State} \\
\mathsf{contains} &: \mathsf{Item} \times \mathsf{State} \to \mathsf{Option}(\mathsf{Output} \times \mathsf{Witness}) \\
\mathsf{verify} &: \mathsf{Item} \times \mathsf{Output} \times \mathsf{Witness} \to \mathsf{Bool}
\end{aligned}$$

with the following properties:

- **Unique Accumulated Values**: For any initial state $s : \mathsf{State}$ and any list of items $I : \mathsf{List}(\mathsf{Item})$ we can generate the sequence of states:
$$s_0 := s, \quad s_{i+1} := \mathsf{insert}(I_i, s_i)$$
Then, if we collect the accumulated values for these states, $z_i := \mathsf{current}(s_i)$, there should be exactly $|I|$-many unique values, one for each state update.
- **Provable Membership**: For any initial state $s : \mathsf{State}$ and any list of items $I : \mathsf{List}(\mathsf{Item})$ we can generate the sequences of states:
$$s_0 := s, \quad s_{i+1} := \mathsf{insert}(I_i, s_i)$$
Then, if we collect the states s_i into a set S, we have the following property for all $s \in S$ and $t \in I$,
$$\mathsf{Some}(z, w) := \mathsf{contains}(t, s), \quad \mathsf{verify}(t, z, w) = \mathsf{True}$$

Definition 4.1.6 (Non-Interactive Zero-Knowledge Proving System)**.** A *non-interactive zero-knowledge proving system* NIZK is defined by the schema:

$$\begin{aligned}
\mathsf{Statement} &: \mathsf{Type} \\
\mathsf{ProvingKey} &: \mathsf{Type} \\
\mathsf{VerifyingKey} &: \mathsf{Type} \\
\mathsf{PublicInput} &: \mathsf{Type} \\
\mathsf{SecretInput} &: \mathsf{Type} \\
\mathsf{Proof} &: \mathsf{Type} \\
\mathsf{keys} &: \mathsf{Statement} \to \mathfrak{D}(\mathsf{ProvingKey} \times \mathsf{VerifyingKey}) \\
\mathsf{prove} &: \mathsf{Statement} \times \mathsf{ProvingKey} \times \mathsf{PublicInput} \times \mathsf{SecretInput} \to \mathfrak{D}(\mathsf{Option}(\mathsf{Proof})) \\
\mathsf{verify} &: \mathsf{VerifyingKey} \times \mathsf{PublicInput} \times \mathsf{Proof} \to \mathsf{Bool}
\end{aligned}$$

Notation: We use the following notation for a NIZK:

- We write the $\mathsf{Statement}$ and $\mathsf{ProvingKey}$ arguments of prove in the superscript and subscript respectively,
$$\mathsf{prove}^P_{\mathsf{pk}}(x, w) := \mathsf{prove}(P, \mathsf{pk}, x, w)$$
- We write the $\mathsf{VerifyingKey}$ argument of verify in the subscript,
$$\mathsf{verify}_{\mathsf{vk}}(x, \pi) := \mathsf{verify}(\mathsf{vk}, x, \pi)$$
- We say that $(x, w) : \mathsf{PublicInput} \times \mathsf{SecretInput}$ has the property of being a $\mathsf{satisfying}$ input whenever
$$\mathsf{satisfying}^P_{\mathsf{pk}}(x, w) := \exists \pi : \mathsf{Proof}, \mathsf{Some}(\pi) \in \mathsf{prove}^P_{\mathsf{pk}}(x, w)$$

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

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

$$\mathcal{A} : \mathsf{ProvingKey} \times \mathsf{VerifyingKey} \to \mathfrak{D}(\mathsf{PublicInput} \times \mathsf{Proof})$$

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

$$\mathcal{E}_{\mathcal{A}} : \mathsf{ProvingKey} \times \mathsf{VerifyingKey} \to \mathfrak{D}(\mathsf{SecretInput})$$

  such that the following probability is negligible:

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

- **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:

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

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

Definition 4.1.7 (Cryptographic Group)**.** We define a *cryptographic group* $(\mathbb{G}, p, g)$ as some finite cyclic group $\mathbb{G}$, of prime order p with generator g where the discrete logarithm problem is hard, namely, given $X \in \mathbb{G}$ it is infeasible to find x such that $X = g^x$. We may omit the prime p when convenient.

4.2 Addresses and Key Components

For the $\mathsf{Transfer}$ protocol we use a multi-layered system of keys:

$$\begin{tikzcd}
& \mathsf{sk} \arrow[dl, "(-)\cdot\alpha"'] \arrow[dr, "g\cdot(-)"] & & & \\
\mathsf{sk}_\alpha \arrow[dr, "g\cdot(-)"'] & & \mathsf{ak} \arrow[dl, "(-)\cdot\alpha"] \arrow[r, "\mathsf{KDF}^{\mathsf{vk}}"] & \mathsf{vk} \arrow[r, "g\cdot(-)"] & \mathsf{pk} \\
& \mathsf{ak}_\alpha & & &
\end{tikzcd}$$

Figure 2: Detailed Key Schedule for MantaPay where α is a random scalar and g is a generator.

Here we define each key and its function in the $\mathsf{Transfer}$ protocol:

Definition 4.2.1 (Key Schedule)**.** A $\mathsf{KeySchedule}$ is a collection of implementations of the following abstract cryptographic primitives as described in the above definitions:

- **Cryptographic Group**: $(\mathbb{G}, p, g)$
- **Viewing Key Derivation Function**: $\mathsf{KDF}^{\mathsf{vk}}$
- **Proof Authorization Signature**: SIG

with the following notational conventions:

$$\begin{aligned}
\mathsf{SpendingKey} &:= Z_p \\
\mathsf{ProofAuthorizingKey} &:= \mathbb{G} \\
\mathsf{ViewingKey} &:= Z_p \\
\mathsf{zkAddress} &:= \mathbb{G}
\end{aligned}$$

with the following constraints:

$$\begin{aligned} \mathsf{SIG.SecretKey} &= \mathbb{Z}_p \\ \mathsf{SIG.PublicKey} &= \mathbb{G} \\ \mathsf{SIG.derive} &= g \cdot (-) \end{aligned}$$

To derive the zkAddress, pk, we use the following:

$$\mathsf{sk} \quad \mapsto \quad \mathsf{ak} := g \cdot \mathsf{sk} \quad \mapsto \quad \mathsf{vk} := \mathsf{KDF}^{\mathsf{vk}}(\mathsf{ak}) \quad \mapsto \quad \mathsf{pk} := g \cdot \mathsf{vk}$$

For signing a message m with a randomized key, the owner of the SpendingKey, sk, and owner of the ProofAuthorizingKey, ak, perform the following protocol:

1. Spender samples α randomly and sends it to prover.
2. Prover computes $\mathsf{ak}_\alpha := \mathsf{ak} \cdot \alpha$ and binds it to the message m and sends the message to spender.
3. Spender computes $\mathsf{sk}_\alpha := \mathsf{sk} \cdot \alpha$ and checks that $\mathsf{ak}_\alpha = g \cdot \mathsf{sk}_\alpha$ and signs the message m with sk_α.

4.3 Transfer Protocol

The Transfer protocol is the fundamental abstraction in MantaPay and facilitiates the valid transfer of zkAssets 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 zkAssets, 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:

- **Key Schedule**: KeySchedule
- **Incoming Authenticated Encryption Scheme**: $\mathsf{AUTH}^{\mathsf{in}}$
- **Outgoing Authenticated Encryption Scheme**: $\mathsf{AUTH}^{\mathsf{out}}$
- **UTXO Commitment Scheme**: $\mathsf{COM}^{\mathsf{UTXO}}$
- **Void Number Commitment Scheme**: $\mathsf{COM}^{\mathsf{VN}}$
- **UTXO Dynamic Cryptographic Accumulator**: UTXOSet
- **Zero-Knowledge Proving System**: NIZK

with the following notational conventions:

$$\begin{aligned} \mathsf{UTXO} &:= \mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Output} \\ \mathsf{VoidNumber} &:= \mathsf{COM}^{\mathsf{VN}}.\mathsf{Output} \end{aligned}$$

and the following constraints:

$$\begin{aligned} \mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Input} &= \mathbb{G} \times \mathsf{Asset} \\ \mathsf{COM}^{\mathsf{VN}}.\mathsf{Randomness} &= \mathbb{G} \\ \mathsf{COM}^{\mathsf{VN}}.\mathsf{Input} &= \mathbb{F} \\ \mathsf{UTXOSet.Item} &= \mathbb{F} \\ \mathsf{ValidTransfer} &: \mathsf{NIZK.Statement} \end{aligned}$$

where ValidTransfer is defined below.

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

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

$$\begin{aligned} r &: \mathbb{F} \\ \mathsf{sa} &: \mathsf{Asset} \\ \mathsf{pa} &: \mathsf{Asset} \\ t &: \mathsf{Bool} \\ \mathsf{asset} &: \mathsf{Asset} \\ \mathsf{cm} &: \mathbb{F} \\ \mathsf{utxo} &: \mathsf{UTXO} \\ h &: \mathbb{F} \\ h_z &: \mathsf{UTXOSet.Output} \\ h_w &: \mathsf{UTXOSet.Witness} \\ \mathsf{vn} &: \mathsf{VoidNumber} \\ \mathsf{esk} &: \mathbb{Z}_p \\ \mathsf{epk} &: \mathbb{G} \\ \mathsf{C}_{\mathsf{out}} &: \mathsf{AUTH}^{\mathsf{out}}.\mathsf{Ciphertext} \end{aligned}$$

A Sender, S, is constructed from a public key $\mathsf{pk} : \mathsf{zkAddress}$ with the following algorithm:

$$\begin{aligned} t &:= \mathsf{iszero}(\mathsf{sa}) \\ \mathsf{asset} &:= \mathsf{select}(t, \mathsf{sa}, \mathsf{pa}) \\ \mathsf{cm} &:= \mathsf{COM}^{\mathsf{UTXO}}(r, \mathsf{pk}, \mathsf{sa}) \\ \mathsf{utxo} &:= (t, \mathsf{pa}, \mathsf{cm}) \\ h &:= H(\mathsf{utxo}) \\ \mathsf{Some}\,(h_z, h_w) &:= \mathsf{UTXOSet.contains}(h, \mathsf{Ledger.utxos}()) \\ \mathsf{vn} &:= \mathsf{COM}^{\mathsf{VN}}(\mathsf{ak}, h) \\ \mathsf{epk} &:= g \cdot \mathsf{esk} \\ \mathsf{C}_{\mathsf{out}} &:= \mathsf{AUTH}^{\mathsf{out}}.\mathsf{encrypt}(\mathsf{pk} \cdot \mathsf{esk}, \mathsf{select}(t, \mathsf{sa}, \mathsf{pa})) \end{aligned}$$

Definition 4.3.3 (Transfer Sender Post)**.** A $\mathsf{SenderPost}$ is the following tuple extracted from a Sender:

$$\begin{aligned} h_z &: \mathsf{UTXOSet.Output} \\ \mathsf{vn} &: \mathsf{VoidNumber} \\ \mathsf{epk} &: \mathbb{G} \\ \mathsf{C}_{\mathsf{out}} &: \mathsf{AUTH}^{\mathsf{out}}.\mathsf{Ciphertext} \end{aligned}$$

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

Definition 4.3.4 (Transfer Receiver)**.** A $\mathsf{Receiver}$ is the following tuple:

$$\begin{aligned} \mathsf{pk} &: \mathsf{zkAddress} \\ r &: \mathbb{F} \\ \mathsf{sa} &: \mathsf{Asset} \\ \mathsf{pa} &: \mathsf{Asset} \\ t &: \mathsf{Bool} \\ \mathsf{asset} &: \mathsf{Asset} \\ \mathsf{cm} &: \mathbb{F} \\ \mathsf{utxo} &: \mathsf{UTXO} \\ \mathsf{esk} &: \mathbb{Z}_p \\ \mathsf{epk} &: \mathbb{G} \\ \mathsf{C}_{\mathsf{in}} &: \mathsf{AUTH}^{\mathsf{in}}.\mathsf{Ciphertext} \end{aligned}$$

A $\mathsf{Receiver}$, R, is constructed in the following way:

$$\begin{aligned} t &:= \mathsf{iszero}(\mathsf{sa}) \\ \mathsf{asset} &:= \mathsf{select}(t, \mathsf{sa}, \mathsf{pa}) \\ \mathsf{cm} &:= \mathsf{COM}^{\mathsf{UTXO}}(r, \mathsf{pk}, \mathsf{sa}) \\ \mathsf{utxo} &:= (t, \mathsf{pa}, \mathsf{cm}) \\ \mathsf{epk} &:= g \cdot \mathsf{esk} \\ \mathsf{C_{in}} &:= \mathsf{AUTH}^{\mathsf{in}}.\mathsf{encrypt}(\mathsf{pk} \cdot \mathsf{esk}, (r, \mathsf{sa})) \end{aligned}$$

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

$$\begin{aligned} \mathsf{utxo} &: \mathsf{UTXO} \\ \mathsf{epk} &: \mathbb{G} \\ \mathsf{C_{in}} &: \mathsf{AUTH}^{\mathsf{in}}.\mathsf{Ciphertext} \end{aligned}$$

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.

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

$$\begin{aligned} \mathsf{id} &: \mathsf{Option}(\mathsf{AssetId}) \\ \mathsf{sources} &: \mathsf{List}(\mathsf{AssetValue}) \\ \mathsf{senders} &: \mathsf{List}(\mathsf{Sender}) \\ \mathsf{receivers} &: \mathsf{List}(\mathsf{Receiver}) \\ \mathsf{sinks} &: \mathsf{List}(\mathsf{AssetValue}) \end{aligned}$$

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

$$\left(|T.\mathsf{sources}|, |T.\mathsf{senders}|, |T.\mathsf{receivers}|, |T.\mathsf{sinks}|\right)$$

Also, note that the id value is optional. This is inhabited whenever there are sources or sinks, but if the shape of the transaction is $(0, m, n, 0)$ then $\mathsf{id} = \mathsf{None}$.

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

- **Correct Key Signing**: The keys used to construct Senders and Receivers are valid and can be signed by a unique SpendingKey.
- **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.

Definition 4.3.8 (Transfer Validity Statement)**.** A transfer T : Transfer is considered *valid* if and only if

1. The signing authority is correctly constructed:

$$\begin{aligned} \mathsf{ak}_\alpha &:= \mathsf{ak} \cdot \alpha \\ \mathsf{vk} &:= \mathsf{KDF}^{\mathsf{vk}}(\mathsf{ak}) \\ \mathsf{pk} &:= g \cdot \mathsf{vk} \end{aligned}$$

2. All the AssetIds in T are equal:

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

3. The sum of input $\mathsf{AssetValue}$s is equal to the sum of output $\mathsf{AssetValue}$s:

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

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

$$\begin{aligned}
S.t &:= \mathsf{iszero}(\mathsf{sa}) \\
S.\mathsf{asset} &:= \mathsf{select}(S.t, S.\mathsf{sa}, S.\mathsf{pa}) \\
S.\mathsf{cm} &:= \mathsf{COM}^{\mathsf{UTXO}}(S.r, S.\mathsf{pk}, S.\mathsf{sa}) \\
S.\mathsf{utxo} &:= (S.t, S.\mathsf{pa}, S.\mathsf{cm}) \\
S.h &:= H(S.\mathsf{utxo}) \\
\mathsf{UTXOSet.verify}(S.h, S.h_z, S.h_w) &= \mathsf{True} \\
S.\mathsf{vn} &:= \mathsf{COM}^{\mathsf{VN}}(\mathsf{ak}, S.h) \\
S.\mathsf{epk} &:= g \cdot S.\mathsf{esk} \\
S.\mathsf{C_{out}} &:= \mathsf{AUTH}^{\mathsf{out}}.\mathsf{encrypt}(\mathsf{pk} \cdot S.\mathsf{esk}, S.\mathsf{asset})
\end{aligned}$$

5. For all $R \in T.\mathsf{receivers}$, the $\mathsf{Receiver}$ R is well-formed:

$$\begin{aligned}
R.t &:= \mathsf{iszero}(R.\mathsf{sa}) \\
R.\mathsf{asset} &:= \mathsf{select}(R.t, R.\mathsf{sa}, R.\mathsf{pa}) \\
R.\mathsf{cm} &:= \mathsf{COM}^{\mathsf{UTXO}}(R.r, R.\mathsf{pk}, R.\mathsf{sa}) \\
R.\mathsf{utxo} &:= (R.t, R.\mathsf{pa}, R.\mathsf{cm}) \\
R.\mathsf{epk} &:= g \cdot R.\mathsf{esk} \\
R.\mathsf{C_{in}} &:= \mathsf{AUTH}^{\mathsf{in}}.\mathsf{encrypt}(R.\mathsf{pk} \cdot R.\mathsf{esk}, (R.r, R.\mathsf{sa}))
\end{aligned}$$

Notation: This statement is denoted $\mathsf{ValidTransfer}$ and is assumed to be expressible as a $\mathsf{Statement}$ of NIZK.

To finish the transfer, the $\mathsf{SpendingKey}$ for the $\mathsf{Transfer.ak} : \mathsf{ProofAuthorizingKey}$ needs to sign the public side of the transaction. The public part of the transaction is the following post body:

Definition 4.3.9 (Transfer Post Body). A $\mathsf{TransferPostBody}$ is the following tuple:

$$\begin{aligned}
\mathsf{id} &: \mathsf{Option}(\mathsf{AssetId}) \\
\mathsf{sources} &: \mathsf{List}(\mathsf{Source}) \\
\mathsf{senders} &: \mathsf{List}(\mathsf{SenderPost}) \\
\mathsf{receivers} &: \mathsf{List}(\mathsf{ReceiverPost}) \\
\mathsf{sinks} &: \mathsf{List}(\mathsf{Sink}) \\
\pi &: \mathsf{NIZK.Proof}
\end{aligned}$$

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

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

where $\mathsf{Transfer.public}$ returns $\mathsf{SenderPost}$s for each Sender in T and $\mathsf{ReceiverPost}$s for each $\mathsf{Receiver}$ in T, keeping Sources and Sinks as they are, and $\mathsf{Transfer.secret}$ returns all the rest of T which is not part of the output of $\mathsf{Transfer.public}$.

Now we can sign this body with $\mathsf{sk}_\alpha : \mathsf{SpendingKey} := \mathsf{sk} \cdot \alpha$ where the signature scheme has TransferPostBody as the SIG.Message type and we use ak_α as the verifying key:

Definition 4.3.10 (Transfer Post)**.** A TransferPost is the following tuple:

$$\begin{aligned} \sigma &: \mathsf{Option}(\mathsf{SIG.VerifyingKey} \times \mathsf{SIG.Signature}) \\ \mathsf{body} &: \mathsf{TransferPostBody} \end{aligned}$$

Note that the σ value is optional. This is inhabited whenever the number of Senders in a transaction is positive.

Now that a participant has constructed a transfer post P : TransferPost they can send it to the Ledger for verification.

Definition 4.3.11 (Ledger-side Transfer Validity)**.** To check that P represents a valid Transfer, the ledger checks the following:

- **Verify Signature**: Check that $\mathsf{SIG.verify}(P.\sigma_0, P.\sigma_1, P.\mathsf{body}) = \mathsf{True}$. This check is only performed if the transfer shape includes at least one Sender.
- **Public Withdraw**: All the public addresses corresponding to the Assets in P.body.sources have enough public balance (i.e. in the PublicAssetLedger) to withdraw the given Asset.
- **Public Deposit**: All the public addresses corresponding to the Assets in P.body.sinks exist.
- **Current Accumulated State**: The UTXOSet.Output stored in each P.body.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.body.senders are unique, and no VoidNumber in P.body.senders has already been stored in the Ledger.VoidNumberSet.
- **New UTXOs**: All the UTXOs in P.body.receivers are unique, and no UTXO in P.body.receivers has already been stored on the ledger.
- **Verify Transfer**: Check that the following relation holds:

$$\begin{aligned} &\mathsf{NIZK.verify}_{\mathsf{vk}}(\\ &\qquad P.\sigma_0 \,||\, P.\mathsf{body.id} \,||\, P.\mathsf{body.sources} \,||\, P.\mathsf{body.senders} \,||\, P.\mathsf{body.receivers} \,||\, P.\mathsf{body.sinks}, \\ &\qquad P.\mathsf{body}.\pi \\ &\quad) = \mathsf{True} \end{aligned}$$

  where $P.\sigma_0$ is included whenever the transfer shape includes at least one Sender and P.body.id is included whenever the transfer shape includes at least one of Sources or Sinks.

Definition 4.3.12 (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 PublicAssetLedger are updated to reflect their new balances using the Sources and Sinks present in P.
- **UTXOSet Update**: The new UTXOs are appended to the UTXOSet.
- **VoidNumberSet Update**: The new VoidNumbers are appended to the VoidNumberSet.

4.4 Batched Transactions

For MantaPay participants to use the Transfer protocol, they will need to keep track of the current state of their zkAssets and use them to build TransferPosts to send to the Ledger. The balance of any participant is the sum of the balances of their zkAssets, 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 balance, the participant would need to accumulate all of the relevant fragments into a large enough zkAsset 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 atomic action that the user can take, performing a withdrawl from their balance. To describe such a *batched transaction*, we assume the existence of two transfer shapes[3]: Mint with shape $(1, 0, 1, 0)$ and PrivateTransfer with shape $(0, N, N, 0)$ for some natural number $N > 1$.

[3]Other Transfer accumulation algorithms are possible with different starting shapes.

Algorithm 1 Batched Transaction Algorithm

```
procedure BuildBatch(sk, 𝓑, total, pk)
    B ← Sample(total, 𝓑)                                    ▷ Samples coins from 𝓑 that total at least total
    if len(B) = 0 then
        return [ ]                                           ▷ Insufficient Balance
    end if
    P ← [ ]                                                  ▷ Allocate a new list for TransferPosts
    while len(B) > N do                                      ▷ While there are enough pairs to make another Transfer
        A ← [ ]
        for b ∈ (B, N) do                                    ▷ Get the next N pairs from B
            S ← BuildSenders_sk(b)
            [acc, zs...] ← BuildAccumulatorAndZeroes_sk(S)   ▷ Build a new accumulator and zeroes
            P ← P + TransferPost(Transfer([], S, [acc, zs...], []))
            (A, Z) ← (A + acc, Z + zs)                       ▷ Save acc for the next loop, zs for the end
        end for
        B ← A + remainder(B, N)
    end while
    S ← PrepareZeroes_sk(N, B, Z, P)                         ▷ Use Z and Mints to make B go up to N in size.
    R ← BuildReceiver_sk(pk, S)
    [c, zs...] ← BuildAccumulatorAndZeroes_sk(S)
    return P + TransferPost(Transfer([], S, [R, c, zs...], []))
end procedure
```

For a fixed spending key, $\mathsf{sk} : \mathsf{SpendingKey}$, and asset id, $\mathsf{id} : \mathsf{AssetId}$, we are given a balance state, $\mathcal{B} : \mathsf{FinSet}\,(\mathsf{Bool} \times \mathbb{F} \times \mathsf{AssetValue})$, a set of transparenct-blinder-balance triples for unspent assets, a total balance to withdraw, $\mathsf{total} : \mathsf{AssetValue}$, and a receiving key $\mathsf{pk} : \mathsf{zkAddress}$. We can then compute

$$\textsc{BuildBatch}(\mathsf{sk}, \mathcal{B}, \mathsf{total}, \mathsf{pk})$$

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

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. Wallets should also handle the more complex case when only some of the Transfers succeed in which case they need to be able to continue retrying the transaction until they are finally resolved. Since the only Transfer which sends zkAssets out of the control of the user is the last one (and it recursively depends on the previous Transfers), then it is safe to continue from a partially resolved state with a simple retry of the BuildBatch algorithm.

5 Concrete Protocol

6 Acknowledgements

We would like to thank Luke Pearson and Toghrul Maharramov for our insightful discussions on reusable shielded addresses.

References