

# MantaPay Protocol Specification

## v1.0.0

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

September 26, 2022

### Abstract

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Notation</b>	<b>2</b>
<b>3</b>	<b>Concepts</b>	<b>3</b>
3.1	zkAssets . . . . .	3
3.2	UTXOs . . . . .	3
3.3	Nullifiers . . . . .	3
3.4	zkAddresses . . . . .	3
3.5	Notes . . . . .	4
3.6	ShieldedPool . . . . .	4
<b>4</b>	<b>Abstract Protocol</b>	<b>4</b>
4.1	Abstract Cryptographic Schemes . . . . .	4
4.1.1	Commitment Scheme . . . . .	4
4.1.2	Hash Function . . . . .	5
4.1.3	Signature Scheme . . . . .	5
4.1.4	Authenticated Encryption Scheme . . . . .	5
4.1.5	Dynamic Cryptographic Accumulator . . . . .	6
4.1.6	Non-Interactive Zero-Knowledge Proving System . . . . .	6
4.1.7	Cryptographic Group . . . . .	7
4.2	Addresses and Key Components . . . . .	7
4.3	Transfer Protocol . . . . .	8
4.4	Batched Transactions . . . . .	12
<b>5</b>	<b>Concrete Protocol</b>	<b>13</b>
5.1	Poseidon Permutation and Poseidon Hash . . . . .	13
5.2	Elliptic Curve Cryptography . . . . .	14
5.3	Concrete Cryptographic Schemes . . . . .	14
5.4	AssetValue Bounds Check . . . . .	14
<b>6</b>	<b>Acknowledgements</b>	<b>14</b>

---

\*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 anonymity 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<sup>1</sup>.
- 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  $(-, -) : A \rightarrow (B \rightarrow A \times B)$ . Depending on context, we may omit the constructor and inline the pair into another constructor/destructor. For example, if  $f : A \times B \rightarrow C$  we can denote  $f((a, b))$  as  $f(a, b)$  to reduce the number of parentheses.
- For any type  $T : \text{Type}$ , we define  $\text{Option}\langle T \rangle : \text{Type}$  as the inductive type with constructors:

$$\begin{aligned} \text{None} &: \text{Option}\langle T \rangle \\ \text{Some} &: T \rightarrow \text{Option}\langle T \rangle \end{aligned}$$

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

---

<sup>1</sup>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  $a : \text{zkAsset}$  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 `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`<sup>2</sup>. 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

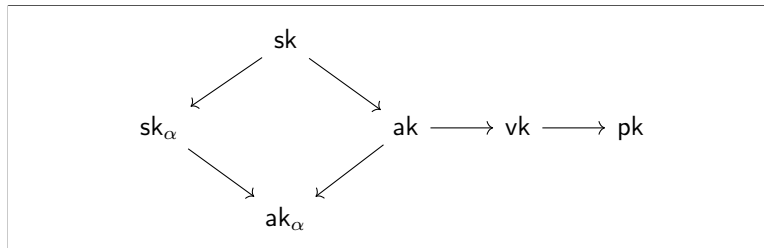
But `zkAssets` are not private on their own. A `UTXO` is a container for a `zkAsset` that hides its value and its owner and is the main object that `MantaPay` uses to transfer the spending power of `zkAssets` between different protocol participants. A `UTXO` is a cryptographic commitment along with some 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.



**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`.

<sup>2</sup>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 `(UTXO, IncomingNote)` and `(Nullifier, OutgoingNote)` pairs

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

- **UTXO Storage:**
  - `UTXOSet : UTXO → Bool`
  - `UTXOStorageInsertionOrder : ℕ → (UTXO, IncomingNote)`
- **Nullifier Storage:**
  - `NullifierSet : Nullifier → Bool`
  - `NullifierStorageInsertionOrder : ℕ → (Nullifier, 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:

Randomness : Type  
Input : Type  
Output : Type  
commit : Randomness  $\times$  Input  $\rightarrow$  Output

with the following properties:

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

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

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

Input : Type  
Output : Type  
hash : Input  $\rightarrow$  Output

with the following properties:

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

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 **Randomness** and **Input** space.

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

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

SigningKey : Type  
VerifyingKey : Type  
Randomness : Type  
Message : Type  
Signature : Type  
derive : SigningKey  $\rightarrow$  VerifyingKey  
sign : SigningKey  $\times$  Randomness  $\times$  Message  $\rightarrow$  Signature  
verify : VerifyingKey  $\times$  Signature  $\times$  Message  $\rightarrow$  Bool

with the following properties:

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

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

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

Key : Type  
Plaintext : Type  
Ciphertext : Type  
Tag : Type  
encrypt : Key  $\times$  Plaintext  $\rightarrow$  Tag  $\times$  Ciphertext  
decrypt : Key  $\times$  Tag  $\times$  Ciphertext  $\rightarrow$  Option(Plaintext)

with the following properties:

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

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

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

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  $t \in I$ ,

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

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

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

**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  $\pi$ , 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:

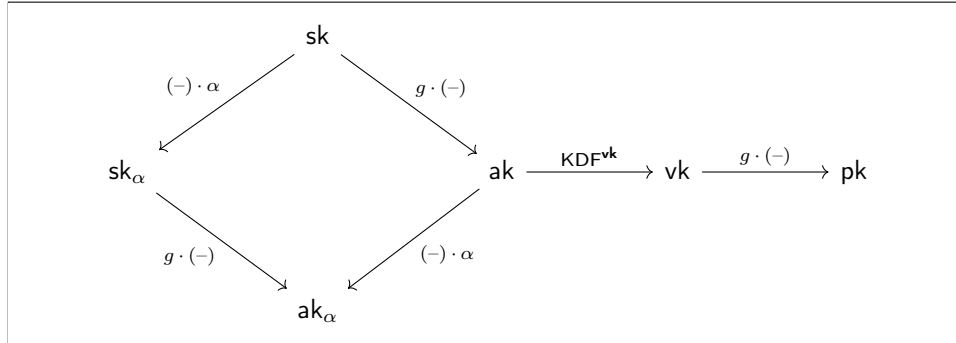
$$\Pr \left[ \begin{array}{l} \text{satisfying}_{\text{pk}}^P(x, w) = \text{True} \\ \mathcal{D}(\pi) = \text{True} \end{array} \middle| \begin{array}{l} (\text{pk}, \text{vk}) \sim \text{keys}(P) \\ (x, w) \sim \mathcal{D}(\text{pk}, \text{vk}) \\ \text{Some}(\pi) \sim \text{prove}_{\text{pk}}^P(x,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

- **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  $\mathcal{O}(|x|)$ .

**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 Transfer protocol we use a multi-layered system of keys:



**Figure 2:** Detailed Key Schedule for MantaPay where  $\alpha$  is a random scalar and  $g$  is a generator.

Here we define each key and its function in the Transfer protocol:

**Definition 4.2.1** (Key Schedule). A 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:**  $\text{KDF}^{\text{vk}}$
- **Proof Authorization Signature:**  $\text{SIG}$

with the following notational conventions:

$$\begin{aligned} \text{SpendingKey} &:= \mathbb{Z}_p \\ \text{ProofAuthorizingKey} &:= \mathbb{G} \\ \text{ViewingKey} &:= \mathbb{Z}_p \\ \text{zkAddress} &:= \mathbb{G} \end{aligned}$$

with the following constraints:

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

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

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

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

### 4.3 Transfer Protocol

The `Transfer` protocol is the fundamental abstraction in `MantaPay` and facilitates 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:**  $\text{AUTH}^{\text{in}}$
- **Outgoing Authenticated Encryption Scheme:**  $\text{AUTH}^{\text{out}}$
- **UTXO Commitment Scheme:**  $\text{COM}^{\text{UTXO}} : \mathbb{F} \times \mathbb{G} \times \text{Asset} \rightarrow \mathbb{F}$
- **Nullifier Commitment Scheme:**  $\text{COM}^{\text{N}} : \mathbb{G} \times \mathbb{F} \rightarrow \mathbb{F}$
- **UTXO Hash Function:**  $H^{\text{UTXO}} : \text{Bool} \times \text{Asset} \times \mathbb{F} \rightarrow \mathbb{F}$
- **UTXO Dynamic Cryptographic Accumulator:** `UTXOSet`
- **Zero-Knowledge Proving System:** `NIZK`

where  $\mathbb{F}$  is a prime field and  $\mathbb{G}$  is a cryptographic group. The `Nullifier` type is the output of  $\text{COM}^{\text{N}}$  and the `UTXO` type is the tuple  $\text{Bool} \times \text{Asset} \times \mathbb{F}$  where the last field element is the output of  $\text{COM}^{\text{UTXO}}$ .

For the rest of this section, we assume the existence of a `TransferConfiguration` and use the primitives outlined 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:

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

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

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

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

$$\begin{aligned}
& h_z : \text{UTXOSet.Output} \\
& n : \text{Nullifier} \\
& epk : \mathbb{G} \\
& C_{\text{out}} : \text{AUTH}^{\text{out}}.\text{Ciphertext}
\end{aligned}$$

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:

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

A Receiver,  $R$ , is constructed in the following way:

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

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

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

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:

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

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

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

Also, note that the  $\text{id}$  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  $\text{id} = \text{None}$ .

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  $T : \text{Transfer}$  is considered *valid* if and only if

1. The signing authority is correctly constructed:

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

2. All the AssetIds in  $T$  are equal:

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

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

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

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

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

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

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

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

To finish the transfer, the `SpendingKey` for the `Transfer.ak : 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 `TransferPostBody` is the following tuple:

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

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

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

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 we can sign this body with  $sk_\alpha : \text{SpendingKey} := sk \cdot \alpha$  where the signature scheme has `TransferPostBody` as the `SIG.Message` type and we use  $ak_\alpha$  as the verifying key:

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

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

Note that the  $\sigma$  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  $P : \text{TransferPost}$  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  $\text{SIG.verify}(P.\sigma_0, P.\sigma_1, P.\text{body}) = \text{True}$ . 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  $P.\text{body.sources}$  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  $P.\text{body.sinks}$  exist.
- **Current Accumulated State:** The `UTXOSet.Output` stored in each  $P.\text{body.senders}$  is equal to current accumulated value,  $\text{UTXOSet.current}(\text{Ledger.utxos}())$ , for the current state of the `Ledger`.
- **New VoidNumbers:** All the `VoidNumbers` in  $P.\text{body.senders}$  are unique, and no `VoidNumber` in  $P.\text{body.senders}$  has already been stored in the `Ledger.VoidNumberSet`.
- **New UTXOs:** All the `UTXOs` in  $P.\text{body.receivers}$  are unique, and no `UTXO` in  $P.\text{body.receivers}$  has already been stored on the ledger.
- **Verify Transfer:** Check that the following relation holds:

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

where  $P.\sigma_0$  is included whenever the transfer shape includes at least one `Sender` and  $P.\text{body.id}$  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 withdrawal from their balance. To describe such a *batched transaction*, we assume the existence of two transfer shapes<sup>3</sup>: `Mint` with shape  $(1, 0, 1, 0)$  and `PrivateTransfer` with shape  $(0, N, N, 0)$  for some natural number  $N > 1$ .

<sup>3</sup>Other `Transfer` accumulation algorithms are possible with different starting shapes.

---

**Algorithm 1** Batched Transaction Algorithm

---

```
procedure BUILD BATCH(sk,  $\mathcal{B}$ , total, pk)
   $B \leftarrow \text{Sample}(\text{total}, \mathcal{B})$  ▷ Samples coins 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  $B$ 
       $S \leftarrow \text{BuildSenders}_{\text{sk}}(b)$ 
       $[acc, zs...] \leftarrow \text{BuildAccumulatorAndZeroes}_{\text{sk}}(S)$  ▷ Build a new accumulator and zeroes
       $P \leftarrow P + \text{TransferPost}(\text{Transfer}([], S, [acc, zs...], []))$ 
       $(A, Z) \leftarrow (A + acc, Z + zs)$  ▷ Save  $acc$  for the next loop,  $zs$  for the end
    end for
     $B \leftarrow A + \text{remainder}(B, N)$ 
  end while
   $S \leftarrow \text{PrepareZeroes}_{\text{sk}}(N, B, Z, P)$  ▷ Use  $Z$  and Mints to make  $B$  go up to  $N$  in size.
   $R \leftarrow \text{BuildReceiver}_{\text{sk}}(\text{pk}, S)$ 
   $[c, zs...] \leftarrow \text{BuildAccumulatorAndZeroes}_{\text{sk}}(S)$ 
  return  $P + \text{TransferPost}(\text{Transfer}([], S, [R, c, zs...], []))$ 
end procedure
```

---

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

$$\text{BUILD BATCH}(\text{sk}, \mathcal{B}, \text{total}, \text{pk})$$

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

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  $\text{zkAssets}$  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 **BUILD BATCH** algorithm.

## 5 Concrete Protocol

We define the instantiation of the abstract protocol in this section, but first some preliminary notes.

### 5.1 Poseidon Permutation and Poseidon Hash

The **Poseidon** Permutation ( $\text{Poseidon}^\pi$ ) [2] is a finite field cryptographic primitive that can be used to build many cryptographic primitives, like hash functions, commitment schemes, and symmetric encryption schemes. **Poseidon** plays a fundamental role in simplifying the **Transfer** protocol and reducing the overall cost of the Zero-Knowledge circuits.  $\text{Poseidon}^\pi$  is a family of permutation functions with the following type:

$$\text{Poseidon}_k^\pi : \mathbb{F} \times \mathbb{F}^k \rightarrow \mathbb{F}^k$$

over some sufficiently large finite field  $\mathbb{F}$ . The first distinguished field element is used as a domain separation element. For this purpose, we use the following hashing function to generate domain strings:

$$\text{HashToScalar}(m) := \mathbb{F}.\text{truncate}(\text{Blake2s}(m))$$

The **Poseidon** hash function (without sponges) with the following type:

$$\text{Poseidon}_k : \mathbb{F} \times \mathbb{F}^k \rightarrow \mathbb{F}$$

is defined as extracting the first finite field element out of  $\text{Poseidon}_k^\pi$ .

We make use of **Poseidon** for a few values of  $k$  in the concrete protocol below.

## 5.2 Elliptic Curve Cryptography

Because our protocol relies on a cryptographic group which should be efficient in a Zero-Knowledge Proving System we choose an elliptic curve defined over the finite field  $\mathbb{F}$  of the proving system. To use group elements in affine form we also define the projections:

$$\mathcal{X} : \mathbb{G} \rightarrow \mathbb{F} \text{ and } \mathcal{Y} : \mathbb{G} \rightarrow \mathbb{F}$$

which we use below to insert group elements into field-only hash functions.

For this protocol, we use BN254 as our outer (pairing-friendly) curve with scalar field  $\mathbb{F}$  and BabyJubJub [4] as our inner curve with scalar field  $\mathbb{S}$ . For this protocol, we call the inner curve  $\mathbb{G}$ .

## 5.3 Concrete Cryptographic Schemes

**Definition 5.3.1** (Commitment Schemes). The protocol features two different commitment schemes:  $\text{COM}^{\text{UTXO}}$  the UTXO Commitment Scheme and  $\text{COM}^{\text{VN}}$  the Void Number Commitment Scheme. Both commitment schemes use **Poseidon** as the underlying cryptographic primitive. The UTXO uses an arity-5 **Poseidon** with the following mapping:

$$\text{COM}^{\text{UTXO}}(r, \text{pk}, \text{asset}) := \text{Poseidon}_5(d, r, \mathcal{X}(\text{pk}), \mathcal{Y}(\text{pk}), \text{asset.id}, \text{asset.value})$$

where  $d = \text{HashToScalar}(\text{"manta-pay/1.0.0/com-utxo"})$  the domain separation element. For the Void Number Commitment Scheme we use an arity-3 **Poseidon** with the following mapping:

$$\text{COM}^{\text{VN}}(\text{ak}, h) := \text{Poseidon}_3(d, \mathcal{X}(\text{ak}), \mathcal{Y}(\text{ak}), h)$$

where  $d = \text{HashToScalar}(\text{"manta-pay/1.0.0/com-vn"})$  the domain separation element.

**Definition 5.3.2** (Hash Functions). The protocol features two additional hash functions:  $\text{KDF}^{\text{vk}}$  the viewing key derivation function and  $H^{\text{UTXO}}$  the UTXO hash function. Both hash functions use **Poseidon** as the underlying cryptographic primitive. The viewing key derivation function uses an arity-2 **Poseidon**

$$\text{KDF}^{\text{vk}}(\text{ak}) := \text{Poseidon}_2(d, \mathcal{X}(\text{ak}), \mathcal{Y}(\text{ak}))$$

where  $d = \text{HashToScalar}(\text{"manta-pay/1.0.0/kdf-vk"})$  the domain separation element. Since the target type of this hash function is the scalar field  $\mathbb{S}$  instead of  $\mathbb{F}$ , we reduce the result modulo the order of  $\mathbb{S}$ . For the UTXO hash function we use an arity-4 **Poseidon**

$$H^{\text{UTXO}}(t, \text{pa}, \text{cm}) := \text{Poseidon}_4(d, t, \text{pa.id}, \text{pa.value}, \text{cm})$$

where  $d = \text{HashToScalar}(\text{"manta-pay/1.0.0/utxo-hash"})$  the domain separation element.

**Definition 5.3.3** (Signature Scheme). For the signature scheme we use Schnorr signature over the inner curve,  $\mathbb{G}$ .

**Definition 5.3.4** (Authenticated Encryption Scheme). For  $\text{AUTH}^{\text{in}}$  and  $\text{AUTH}^{\text{out}}$  we use the **Poseidon** permutation as the permutation of a duplex sponge encryption protocol.

**Definition 5.3.5** (Dynamic Cryptographic Accumulator). For DCA, we use a Merkle Tree with **Poseidon**<sub>2</sub> as the inner node combining hash function and no leaf hash function. It is safe to omit the leaf hash function in this case because the leaf values are already the outputs of a hash function and cannot be directly controlled.

**Definition 5.3.6** (Non-Interactive Zero-Knowledge Proving System). For NIZK, the protocol can use any non-interactive zero-knowledge proving system like Groth16 [2] and/or PLONK/PLONKUP [1, 3].

## 5.4 AssetValue Bounds Check

In order to implement the balanced transfer relation one needs to ensure that the amount of input value is equal to the amount of output value. However, since we're working over finite fields, the naïve arithmetic wraps past zero and is vulnerable to range-based attacks. Instead we constrain every **AssetValue** to be less than some bound  $\mathcal{V}$  and that every sum over those values is also less than  $\mathcal{V}$ . Since we're using BN254 we are safe to use  $\mathcal{V} = 2^{128}$ .

## 6 Acknowledgements

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

## References

- [1] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.
- [2] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, pages 519–535. USENIX Association, 2021.
- [3] Luke Pearson, Joshua Fitzgerald, Héctor Masip, Marta Bellés-Muñoz, and Jose Luis Muñoz-Tapia. Plonkup: Reconciling plonk with plookup. *IACR Cryptol. ePrint Arch.*, page 86, 2022.
- [4] Barry WhiteHat, Marta Bellés, and Jordi Baylina. EIP-2494: Baby Jubjub Elliptic Curve . Eip, Ethereum Foundation, 2020.