MantaPay Protocol Specification
v1.0.0

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

September 1, 2022

Abstract

MantaPay is an implementation of a *decentralized anonymous payment* scheme based on the MANTA$_{\mathsf{DAP}}$ 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:

$$\mathsf{None} : \mathsf{Option}\langle T\rangle$$
$$\mathsf{Some} : T \to \mathsf{Option}\langle T\rangle$$

- 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 $[\ldots]$ 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 $a : \mathsf{zkAsset}$ 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 $\{\mathsf{commit}(r, x) \,|\, r \sim \mathsf{Randomness}\}$ and $\{\mathsf{commit}(r, y) \,|\, r \sim \mathsf{Randomness}\}$ 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 (Authenticated Encryption)**.** 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 4.1.4 (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 4.1.5 (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 4.1.6 (Cryptographic Group)**.** We define a *cryptographic group* (G, p, g) as some finite cyclic group G, of prime order p with generator g where the discrete logarithm problem is hard, namely, given $X \in G$ it is infeasible to find x such that $X = g^x$.

4.2 Addresses and Key Components

4.3 Transfer Protocol

4.4 Batched Transactions

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

$$\textbf{Poseidon}_k^\pi : \mathbb{F} \times \mathbb{F}^k \to \mathbb{F}^k$$

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:

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

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

$$\textbf{Poseidon}_k : \mathbb{F} \times \mathbb{F}^k \to \mathbb{F}$$

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

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

5.2 Elliptic Curve Cryptography

Because we use a Zero-Knowledge Proving System, we want the cryptographic constructions that feature in our protocol to be *ZKP-friendly.* For a ZKP system defined over a finite field $\mathbb{F}$ we can look for elliptic curves that have a base field $\mathbb{F}$. These such curves are said to be “embeddable” or “embedded in” $\mathbb{F}$. For the constructions below, we use $\mathbb{F}$ as the proof system field and $\mathbb{G}$ as an embedded curve with scalar field $\mathbb{S}$. We also assume that $|\mathbb{S}| < |\mathbb{F}|$ so we can use the injection $\mathsf{lift} : \mathbb{S} \to \mathbb{F}$ to lift scalars to the proof system field.

To use group elements in affine form we also define the projections:

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

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 `Baby JubJub` [4] as our inner curve with scalar field $\mathbb{S}$.

5.3 Concrete Cryptographic Schemes

Definition 5.3.1 (Commitment Schemes)**.** The protocol features two different commitment schemes: $\mathsf{COM}^{\mathsf{UTXO}}$ the UTXO Commitment Scheme and $\mathsf{COM}^{\mathsf{VN}}$ the Void Number Commitment Scheme. Both commitment schemes use **Poseidon** as the underlying cryptographic primitive. The UTXO uses an arity-8 **Poseidon** with the following mapping:

$$\mathsf{COM}_r^{\mathsf{UTXO}}(\mathsf{D}, \mathsf{pk}_\mathsf{D}, \mathsf{asset}) := \textbf{Poseidon}_8(d, 0, r, \mathcal{X}(\mathsf{D}), \mathcal{Y}(\mathsf{D}), \mathcal{X}(\mathsf{pk}_\mathsf{D}), \mathcal{Y}(\mathsf{pk}_\mathsf{D}), \mathsf{asset.id}, \mathsf{asset.value})$$

where $d = \mathsf{HashToScalar}(\text{“}\texttt{manta-pay/1.0.0/com-utxo}\text{”})$. For the Void Number Commitment Scheme we use an arity-4 **Poseidon** with the following mapping:

$$\mathsf{COM}^{\mathsf{VN}}_{\mathsf{ak}}(\mathsf{cm}) := \textbf{Poseidon}_4(\mathsf{HashToScalar}(\text{“}\texttt{manta-pay/1.0.0/com-vn}\text{”}), 0, \mathcal{X}(\mathsf{ak}), \mathcal{Y}(\mathsf{ak}), \mathsf{cm})$$

Definition 5.3.2 (Key-Derivation Functions)**.** For the encryption scheme KDFs, we use the following which maps a group element $G : \mathbb{G}$ to a scalar:

$$\mathsf{KDF}(G) := \textbf{Poseidon}_2(\mathsf{HashToScalar}(\text{“}\texttt{manta-pay/1.0.0/encryption-kdf}\text{”}), \mathcal{X}(G), \mathcal{Y}(G))$$

Definition 5.3.3 (Randomizable Key-Derivation Function)**.** For rKDF, we use the following which uses a scalar $r : \mathbb{S}$ to randomize a scalar $x : \mathbb{S}$ to a scalar and a group element $G : \mathbb{G}$ to a group element:

$$\mathsf{rKDF.rand}^I(r, x) : \mathbb{S} \times \mathbb{S} \to \mathbb{S} := r * x$$
$$\mathsf{rKDF.rand}^O(r, G) : \mathbb{S} \times \mathbb{G} \to \mathbb{G} := r \cdot G$$

Definition 5.3.4 (Key-Agreement Scheme)**.** For KA, we use a Diffie-Hellman Key Exchange over $(\mathbb{G}, \mathbb{S})$:

$$\mathsf{KA.derive}(x) : \mathbb{S} \to \mathbb{G} := x \cdot G$$
$$\mathsf{KA.agree}(x, Y) : \mathbb{S} \times \mathbb{G} \to \mathbb{G} := x \cdot Y$$

where G is a fixed public point.

Definition 5.3.5 (Message Authentication Code)**.** For message authentication codes we use the following instantiation of **Poseidon**:

$$\mathsf{MAC}(\mathsf{sk}, m) := \textbf{Poseidon}_{|m|+1}(\mathsf{HashToScalar}(\text{“}\texttt{manta-pay/1.0.0/mac}\text{”}), \mathsf{sk}, m)$$

In this protocol, we use $|m| \in \{2, 6\}$ for OutgoingNote and IncomingNote respectively.

Definition 5.3.6 (Signature Scheme)**.** For the signature scheme we use Schnorr signature over $\mathbb{G}$.

Definition 5.3.7 (Symmetric-Key Encryption Scheme)**.** For SYM we use **Poseidon**$_2$ as the hash function in a message digest cipher with key-schedule given by the following:

$$K_i := \textbf{Poseidon}_2(\mathsf{HashToScalar}(\text{“}\texttt{manta-pay/1.0.0/mdc-key-schedule}\text{”}), K_0, K_{i-1})$$

Definition 5.3.8 (Dynamic Cryptographic Accumulator)**.** For DCA, we use a Merkle Tree with **Poseidon**$_2$ 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 5.3.9 (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.