MantaPay Protocol Specification
v1.0.0

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

August 10, 2022

Abstract

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

Contents

*ordered alphabetically

1 Introduction

$\mathsf{MantaPay}$ aims to solve the long-standing privacy problems facing cryptocurrencies in the Web3 age. At its heart, it uses various cryptographic constructions including NIZK (non-interactive zero knowledge proof) systems to ensure user privacy from *first principles*.

Protocol	Cryptographic Primitives	Consensus	Layer	Multi-Asset
ZCash (Sapling)	NIZK	PoW	1	✗
Monero	RingCT/NIZK	PoW	1	✗
Tornado Cash (Nova)	NIZK	✗	2	✓
MantaPay 1.0.0	NIZK	PoS	1	✓

Table 1: Comparison of $\mathsf{MantaPay}$ with previous constructions

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 $[\ldots]$ 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, $y \in \big\{x \,|\, x \sim \mathfrak{D}\langle T\rangle\big\} \leftrightarrow y \in \mathfrak{D}\langle T\rangle$.
- 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.

3 Concepts

3.1 Assets

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

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

[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.

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 Asset objects.

We use PublicAsset and SecretAsset to explicitly describe whether an Asset is visible to public. More specifically, whenever an Asset is being used in a public setting, we refer to it as a PublicAsset, but when the AssetId and/or AssetValue of a particular Asset is meant to be hidden from public view, we refer to the Asset as either *secret*, *private*, *hidden*, or *shielded* with SecretAsset type.

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

3.2 Addresses

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

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

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:

- **Shielded Address** (send): Access to the shielded address $(\mathsf{D}, \mathsf{pk}_\mathsf{D})$ gives the user the right to send Assets to the owner of the associated sk. The diversifier D allows the owner of a given sk key to generate many shielded addresses with the same backing spend authority.
- **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.
- **Proof Authorization Key** (prove): Proof authorization key ak gives the user the right to build the transaction proof on behalf of the owner of sk. In the cases of delegating proof generation, i.e. using hardware wallet to control the *sk* or signing associated data in transparent UTXOs, the owner of the secret key generates a randomizer α and sends it to the prover which generates the proof. The owner then signs the transaction against ak_α with their randomized key sk_α which proves that they have knowledge of sk.
- **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 addresses, but they are not unique representations, since one participant may have access to more than one secret key. See § 4.3 for more information on how these keys are constructed and used for spending, proving, viewing, and receiving.

3.3 Ledger

We model a blockchain as a byzantine fault tolerance [1] replicated state machine [2], a.k.a ledger. When interacting with the blockchain, we call the entity who initiates the interaction (e.g., sending a transfer request) the user; the entity who verifies the interaction and logs it into the blockchain the validator (also known as miner in other contents). Users interact with the blockchain by sending amendment requests to the ledger. The amendment is appended to the database once validators approve the request. For simplicity, we assume

[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*.

1) the ledger is synchronized, and the block finality is instant; 2) the validators are trusted for liveness and completeness. The underlying consensus protocol that validators employ is indeed orthogonal to this paper. What is also of out the scope of this paper is the governance token for the underlying blockchain. We nonetheless assume that the senders of our protocol holds enough governance tokens to send the transactions.

More specifically, MantaPay's ledger state Ledger consists of two parts: the public ledger as PublicAssetLedger, and the shielded asset pool as ShieldedAssetPool.

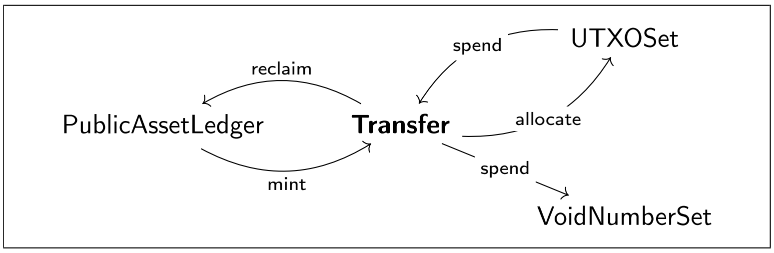

Figure 2: Life cycle of an Asset.

The ShieldedAssetPool is made up of three parts that are used to enforce the balanced transfer of SecretAssets among anonymous participants:

1. § 3.3.1 UTXOSet: The UTXOSet is a collection of ownership claims to subsets of the ShieldedAssetPool (called UTXOs), each one refering to an allocated SecretAsset transfered to a participant of the protocol.
2. § 3.3.2 EncryptedNotes: For every UTXO there is a matching EncryptedNote which contains information necessary to spend the SecretAsset, which can be used to *provably reconstruct* the UTXO convincing the Ledger of unique ownership. The EncryptedNote can only be decrypted by the recipient of the SecretAsset or the designated viewer of the UTXO, specifically, the correct viewing key vk. See § 3.2 for more.
3. § 3.3.3 VoidNumberSet: The VoidNumberSet is a collection of commitments, like UTXOs, but which track the *spent state* of a SecretAsset and are used to prove to the Ledger that a SecretAsset is spent *exactly one time*.

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

3.3.1 UTXOs and the UTXOSet

An *unspent transaction output*, or UTXO for short, represents a claim to the output of a balanced transfer which has *not yet been spent*. A UTXO is a tuple

$$\mathsf{utxo} = (\mathsf{version}, \mathsf{transparent}, \mathsf{pa}, \mathsf{sutxo}) : \mathsf{VersionTag} \times \mathsf{Bool} \times \mathsf{PublicAsset} \times \mathsf{sUTXO}$$

where

- version: VersionTag keeps track of the MantaPay protocol version in which the UTXO was minted.
- transparent: Bool is True if and only if the Asset encoded by the UTXO is public.
- pa: PublicAsset represents the *public outputs* of a transfer.
- sutxo: sUTXO represents the *private outputs* of a transfer.

Every balanced transfer generates some number of UTXOs and these UTXOs are stored in the UTXOSet of the ShieldedAssetPool. A UTXO can only be claimed by the participant who owns the underlying Asset, where ownership means *knowledge of the correct spending key* and the Transfer protocol requires that all inputs to a balanced transfer *prove* that they own a UTXO which the ShieldedAssetPool has already seen in the past. The UTXOSet is *append-only* since it represents the past state of *unspent* Assets. UTXOs can only be added to the UTXOSet as outputs in the execution of a Transfer which the Ledger checks for correctness.

3.3.2 EncryptedNotes

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

3.3.3 VoidNumbers and the VoidNumberSet

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

4 Abstract Protocol

4.1 Algebra

In the following section, we define the algebraic objects that are used throughout the paper and the MantaPay protocol.

Definition 4.1.1 (Ring of scalars)**.** A *ring (of scalars)* consists of a set R together with two binary operations:

$$\begin{aligned} +_R &\colon R \times R \longrightarrow R \\ \cdot &\colon R \times R \longrightarrow R \end{aligned}$$

satisfying the following properties:

- $+_R$ is associative, commutative, has an identity element 0_R and every element $r \in R$ has an inverse, denoted by $-r$.
- $\cdot$ is bilinear w.r.t. $+_R$, associative, commutative and has an identity element 1.

Definition 4.1.2 (Group)**.** A *group* over the ring of scalars R[3] consists of a set G together with two maps

$$\begin{aligned} +_G &\colon G \times G \longrightarrow G \\ \cdot &\colon R \times G \longrightarrow G \end{aligned}$$

satisfying the following properties:

- $+_G$ is associative, commutative, has an identity element 0_G and every element $g \in G$ has an inverse, denoted by $-g$.
- $\cdot_G$ is bilinear w.r.t. both $+_G$ and $+_R$. For all $r_1, r_2 \in R$, $g \in G$, we have $(r_1 \cdot r_2) \cdot g = r_1 \cdot (r_2 \cdot g)$ and $1 \cdot g = g$.

Example: For most practical applications, $G = E$ will be an elliptic curve defined over a finite field $R = \mathbb{F}$.

Notation: From now on, we will denote both $+_G$ and $+_R$ by $+$, and 0_G and 0_R by 0, without it leading to confusion.

Notation: Sometimes, instead of additive notation as in Definition 4.1.2, we will use multiplicative notation for groups. This means we will denote $g_1 + g_2$ as $g_1 g_2$, and rg as g^r.

The following properties of groups will be referred to throughout the text.

Definition 4.1.3. We say that a group G over a finite finite field $\mathbb{F}$ satisfies the

1. *discrete logarithm hardness assumption* if, given $g, h = g^a \in G$, there is no efficient algorithm that can compute $a = \log_g(h) \in \mathbb{F}$.
2. *computational Diffie-Hellman hardness assumption* if, given $g, g_1 = g^a, g_2 = g^b$, there is no efficient algorithm that can compute g^{ab}.
3. *decisional Diffie-Hellman hardness assumption* if there is no efficient algorithm that can distinguish the triples (g^a, g^b, g^{ab}) and (g^a, g^b, g^c).

4.2 Abstract Cryptographic Schemes

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

[3] Technically speaking, this is an R-module. When $R = \mathbb{Z}$, then this definition is indeed a group in the mathematical sense.

Definition 4.2.1 (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 below *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 4.2.2 (Commitment Scheme)**.** A *commitment scheme* COM is defined by the schema:

$$\begin{aligned} \mathsf{Input} &: \mathsf{Type} \\ \mathsf{Output} &: \mathsf{Type} \\ \mathsf{Randomness} &: \mathsf{Type} \\ \mathsf{RandomnessDistribution} &: \mathfrak{D}\langle \mathsf{Randomness} \rangle \\ \mathsf{commit} &: \mathsf{Randomness} \times \mathsf{Input} \to \mathsf{Output} \end{aligned}$$

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 $\{\mathsf{commit}(r, x) \mid r \sim \mathsf{RandomnessDistribution}\}$ and $\{\mathsf{commit}(r, y) \mid r \sim \mathsf{RandomnessDistribution}\}$ are *computationally indistinguishable*.

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

Definition 4.2.3 (Key-Derivation Function)**.** A *key-derivation function* KDF is defined by the schema:

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

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

Note: This abstract definition covers many different cases of key related functions. The security properties of a specific KDF are outlined wherever it's used.

Definition 4.2.4 (Key-Agreement Scheme)**.** A *key-agreement scheme* KA is defined by the schema:

$$\begin{aligned} \mathsf{SecretKey} &: \mathsf{Type} \\ \mathsf{PublicKey} &: \mathsf{Type} \\ \mathsf{SharedSecret} &: \mathsf{Type} \\ \mathsf{SecretKeyDistribution} &: \mathfrak{D}\langle \mathsf{SecretKey} \rangle \\ \mathsf{derive} &: \mathsf{SecretKey} \to \mathsf{PublicKey} \\ \mathsf{agree} &: \mathsf{SecretKey} \times \mathsf{PublicKey} \to \mathsf{SharedSecret} \end{aligned}$$

with the following properties:

- **Agreement**: For all $\mathsf{sk}_1, \mathsf{sk}_2 : \mathsf{SecretKey}$, $\mathsf{agree}(\mathsf{sk}_1, \mathsf{derive}(\mathsf{sk}_2)) = \mathsf{agree}(\mathsf{sk}_2, \mathsf{derive}(\mathsf{sk}_1))$
- **Passive Security**: Even if an adversary eavesdrops on the network communication, she cannot forge the agreed secret unless she knows how to find a preimage for derive which should be as hard as a known hard cryptography problem like the Diffie-Hellman Problem.

- **Known-key Security**: Suppose an adversary learned a shared secret from a past session, then, the adversary does not gain any additional information by combining the past key and public visible data for the purpose of deducing future shared secrets.
- **No Key Control**: The shared secrets are determined by both parties, neither party can control the outcome of the shared secret by restricting it to lie in some predetermined small set.

Notation: For convenience, we may refer to $\mathsf{KA.agree}(\mathsf{sk}, \mathsf{D})$ as $\mathsf{KA.agree}_{\mathsf{D}}(\mathsf{sk})$ for all $\mathsf{sk} : \mathsf{SecretKey}$ and $\mathsf{D} : \mathsf{PublicKey}$.

Definition 4.2.5 (G-Key-Agreement Scheme)**.** A *G-Key-Agreement Scheme G-KA* is a key-agreement scheme where

- G is a group over the ring of scalars R,
- $\mathsf{SecretKey} = R$,
- $\mathsf{PublicKey} = G$.

satisfying $\mathsf{derive}(r \cdot sk) = r \cdot \mathsf{derive}(sk)$ for all $r, sk \colon \mathsf{SecretKey}$. Equivalently, in multiplicative notation, it reads $\mathsf{derive}(r \cdot sk) = \mathsf{derive}(sk)^r$.

Notation: Often we omit the group G in the notation and refer to G-KA as KA.

Remark. One can also define G-Key-Derivation Functions and G-Signature Schemes in a completely analogous manner. More generally, any cryptographic primitive with a derive function admits this structure.

Definition 4.2.6 (Signature Scheme)**.** A *signature scheme* SIG is defined by the schema:

$$\begin{aligned}
\mathsf{SecretKey} &: \mathsf{Type} \\
\mathsf{PublicKey} &: \mathsf{Type} \\
\mathsf{Message} &: \mathsf{Type} \\
\mathsf{Signature} &: \mathsf{Type} \\
\mathsf{derive} &: \mathsf{SecretKey} \to \mathsf{PublicKey} \\
\mathsf{sign} &: \mathsf{SecretKey} \times \mathsf{Message} \to \mathfrak{D}\langle \mathsf{Signature} \rangle \\
\mathsf{verify} &: \mathsf{PublicKey} \times \mathsf{Message} \times \mathsf{Signature} \to \mathsf{Bool}
\end{aligned}$$

with the following properties:

- **Completeness**: For all $\mathsf{sk} : \mathsf{SecretKey}$, $m : \mathsf{Message}$, and any signature $\sigma \sim \mathsf{sign}(\mathsf{sk}, m)$, we have that $\mathsf{verify}(\mathsf{derive}(\mathsf{sk}), m, \sigma) = \mathsf{True}$.
- **Existencial unforgeability against CMA**: Fix $k : \mathsf{SecretKey}$, and denote $pk = \mathsf{derive}(k)$. An adversary who knows the public key pk and can make any number of queries $m_i : \mathsf{Message}$, obtaining valid signatures $\sigma_i \sim \mathsf{sign}(k, m_i)$ cannot forge a pair $(m, \sigma) \neq (m_i, \sigma_i)$ for all i such that $\mathsf{verify}(pk, m, \sigma) = \mathsf{True}$.

Definition 4.2.7 (Symmetric-Key Encryption Scheme)**.** An *authenticated one-time symmetric-key encryption scheme* SYM is defined by the schema:

$$\begin{aligned}
\mathsf{Key} &: \mathsf{Type} \\
\mathsf{Plaintext} &: \mathsf{Type} \\
\mathsf{Ciphertext} &: \mathsf{Type} \\
\mathsf{encrypt} &: \mathsf{Key} \times \mathsf{Plaintext} \to \mathsf{Ciphertext} \\
\mathsf{decrypt} &: \mathsf{Key} \times \mathsf{Ciphertext} \to \mathsf{Option}\langle \mathsf{Plaintext} \rangle
\end{aligned}$$

with the following properties:

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

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

- **Security Requirement**: The symmetric-key encryption scheme must be one-time (INT-CTXT $\wedge$ IND-CPA)-secure [5]. "One-time" means that an honest protocol participant will almost surely encrypt only one message with a given key; however, the adversary could make many adaptive chosen ciphertext queries for a given key.

Definition 4.2.8 (Hybrid Public Key Encryption Scheme)**.** A *hybrid public key encryption scheme* [3] HPKE is an encryption scheme made up of a symmetric-key encryption scheme SYM, a key-agreement scheme KA, and a key-derivation function KDF to convert from $\mathsf{KA.SharedSecret}$ to $\mathsf{SYM.Key}$. We can define the following encryption and decryption algorithms:

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

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

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

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

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

Notation: We denote the above *encrypted message* type as $\mathsf{Encrypted}\langle\mathsf{SYM.Plaintext}\rangle := \mathsf{KA.PublicKey} \times \mathsf{SYM.Ciphertext}$, and the above two algorithms by

$$\begin{aligned}
\mathsf{encrypt} &: \mathsf{KA.SecretKey} \times \mathsf{KA.PublicKey} \times \mathsf{SYM.Plaintext} \to \mathsf{Encrypted}\langle\mathsf{SYM.Plaintext}\rangle \\
\mathsf{decrypt} &: \mathsf{KA.SecretKey} \times \mathsf{KA.PublicKey} \times \mathsf{SYM.Ciphertext} \to \mathsf{Option}\langle\mathsf{SYM.Plaintext}\rangle
\end{aligned}$$

Security Properties: The HPKE constructed from KA, KDF, and SYM is required to be CCA2-secure and key-private [4].

Definition 4.2.9 (Authenticated Hybrid Public Key Encryption Scheme)**.** An *authenticated hybrid public encryption scheme* aHPKE is an authenticated encryption scheme built off of an HPKE and a MAC used in the following way:

$$\mathsf{aHPKE.encrypt} : \mathsf{KA.SecretKey} \times \mathsf{KA.PublicKey} \times \mathsf{SYM.Plaintext} \to \mathsf{AuthEncrypted}\langle\mathsf{SYM.Plaintext}\rangle$$

where $\mathsf{AuthEncrypted}$ is the encrypted note type:

$$\mathsf{AuthEncrypted}\langle\mathsf{SYM.Plaintext}\rangle := \mathsf{MAC.Tag} \times \mathsf{Encrypted}\langle\mathsf{SYM.Plaintext}\rangle$$

and the tag is computed by applying the MAC onto the encrypted note:

$$\mathsf{tag} := \mathsf{MAC}(\mathsf{sk}, \mathsf{HPKE.encrypt}(\mathsf{esk}, \mathsf{pk}))$$

Definition 4.2.10 (Dynamic Cryptographic Accumulator)**.** A *dynamic cryptographic accumulator* DCA is defined by the schema:

$$\begin{aligned}
\mathsf{Item} &: \mathsf{Type} \\
\mathsf{Output} &: \mathsf{Type} \\
\mathsf{Witness} &: \mathsf{Type} \\
\mathsf{State} &: \mathsf{Type} \\
\mathsf{current} &: \mathsf{State} \to \mathsf{Output} \\
\mathsf{insert} &: \mathsf{Item} \times \mathsf{State} \to \mathsf{State} \\
\mathsf{prove} &: \mathsf{Item} \times \mathsf{State} \to \mathsf{Option}\langle\mathsf{Output} \times \mathsf{Witness}\rangle \\
\mathsf{verify} &: \mathsf{Item} \times \mathsf{Output} \times \mathsf{Witness} \to \mathsf{Bool}
\end{aligned}$$

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 : State and any list of items I : $\mathsf{List}\langle\mathsf{Item}\rangle$ 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 $t \in I$,

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

Definition 4.2.11 (Duplex Sponge)**.** A *duplex sponge* DS is defined by the schema:

$$\begin{aligned} \mathsf{Item} &: \mathsf{Type} \\ \mathsf{State} &: \mathsf{Type} \\ \mathsf{absorb} &: \mathsf{List}\langle\mathsf{Item}\rangle \times \mathsf{State} \to \mathsf{State} \\ \mathsf{squeeze} &: \mathbb{Z} \times \mathsf{State} \to \mathsf{List}\langle\mathsf{Item}\rangle \end{aligned}$$

Definition 4.2.12 (Non-Interactive Zero-Knowledge Proving System)**.** A *non-interactive zero-knowledge proving system* NIZK is defined by the schema:

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

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,

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

- Given P: $\mathsf{Statement}$ and pk: $\mathsf{ProvingKey}$, we define the function

$$\mathsf{satisfying}^P_{\mathsf{pk}} : \mathsf{PublicInput} \times \mathsf{SecretInput} \longrightarrow \mathsf{Bool},$$

which is true if $\exists \pi : \mathsf{Proof}$ such that $\mathsf{Some}(\pi) \in \mathsf{prove}^P_{\mathsf{pk}}(x, w)$ and false otherwise. If $\mathsf{satisfying}^P_{\mathsf{pk}}(x, w) = \mathsf{True}$, we call the pair (x, w) a *satisfying input.*

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 $\mathcal{A}$ such that the probability

$$\Pr\left[\mathsf{verify}_{\mathsf{vk}}(x, \pi) = \mathsf{True} \;\middle|\; \begin{matrix} (\mathsf{pk}, \mathsf{vk}) \sim \mathsf{keys}(P) \\ (x, \pi) \sim \mathcal{A}(\mathsf{pk}, \mathsf{vk}) \end{matrix}\right]$$

is non-negligible, there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$

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

such that the difference

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

is negligible.

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

4.3 Addresses and Key Components

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

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

Figure 3: Detailed Key Schedule for MantaPay.

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

Definition 4.3.1 (Spending Key)**.** Given a G-key-agreement scheme KA we define:

$$\mathsf{SpendingKey} := \mathsf{KA.SecretKey}$$

Definition 4.3.2 (Proof Authorizing Key)**.** Given a G-key-agreement scheme KA and a signature scheme SIG we have:

$$\mathsf{ProofAuthorizingKey} := \mathsf{KA.PublicKey}$$

where a given $\mathsf{sk} : \mathsf{SpendingKey}$ derives the proof-authorizing key by

$$\mathsf{ak} := \mathsf{KA.derive}(\mathsf{sk})$$

This key can be twisted by $\alpha : \mathsf{KA.SecretKey}$ to get

$$\mathsf{ak}_\alpha := \alpha \cdot \mathsf{ak}$$

which is also equal to the key-agreement derivation of the twisted secret key

$$\mathsf{ak}_\alpha = \mathsf{KA.derive}(\alpha \cdot \mathsf{sk})$$

To authorize a message m, the owner of sk can perform the following signature algorithm:

$$\sigma \sim \mathsf{SIG.sign}(\alpha \cdot \mathsf{sk}, m)$$

which can then be verified against ak_α with

$$\mathsf{SIG.verify}(\mathsf{ak}_\alpha, m, \sigma)$$

Note: For the Transfer protocol, the message will be the zero-knowlege proof of a valid transfer and any additional associated data and Ledger payload.

Definition 4.3.3 (Viewing Key)**.** Given a proof-authorizing key, we require a KDF of type

$$\mathsf{KDF} : \mathsf{ProofAuthorizingKey} \to \mathsf{ViewingKey}$$

where $\mathsf{ViewingKey}$ is of type $\mathsf{KA.SecretKey}$ so that it can be available for the KA key-agreement scheme so that we have

$$\mathsf{vk} := \mathsf{KDF}(\mathsf{ak})$$

Definition 4.3.4 (Shielded Address)**.** Given a viewing key which is the secret key for the key-agreement scheme KA, the shielded address is given by randomly selecting a public key $\mathsf{D} : \mathsf{KA.PublicKey}$ and then performing $\mathsf{KA.agree}$ against it:

$$\mathsf{pk_D} := \mathsf{KA.agree_D}(\mathsf{vk})$$

We return the pair as the shielded address: $\mathsf{addr} := (\mathsf{D}, \mathsf{pk_D})$. We call the random element D the $\mathsf{Diversifier}$ for the shielded address addr.

Definition 4.3.5 (Key Schedule)**.** A $\mathsf{KeySchedule}$ is a collection of implementations of the following abstract cryptographic primitives as described in the above definitions:

- ***G*-Key-Agreement Scheme**: KA
- **Viewing Key Derivation Function**: KDF
- **Proof Authorization Signature**: SIG

with the following notational conventions:

$$\begin{aligned}
\mathsf{SpendingKey} &:= \mathsf{KA.SecretKey} \\
\mathsf{ProofAuthorizingKey} &:= \mathsf{KA.PublicKey} \\
\mathsf{ViewingKey} &:= \mathsf{KA.SecretKey} \\
\mathsf{Diversifier} &:= \mathsf{KA.PublicKey} \\
\mathsf{ShieldedAddress} &:= \mathsf{KA.Diversifier} \times \mathsf{KA.PublicKey}
\end{aligned}$$

with the following constraints:

$$\begin{aligned}
\mathsf{KA.PublicKey} &= \mathsf{KA.SharedSecret} \\
\mathsf{SIG.SecretKey} &= \mathsf{KA.SecretKey} \\
\mathsf{SIG.PublicKey} &= \mathsf{KA.PublicKey} \\
\mathsf{SIG.derive} &= \mathsf{KA.derive} \\
\mathsf{KDF.SecretKey} &= \mathsf{KA.PublicKey} \\
\mathsf{KDF.PublicKey} &= \mathsf{KA.SecretKey}
\end{aligned}$$

4.4 Transfer Protocol

The Transfer protocol is the fundamental abstraction in MantaPay and facilitiates the valid transfer of Assets among participants while preserving their privacy. The Transfer is made up of sub-components called Senders and Receivers which represent the private input and the private output of a transaction.[4] To perform a Transfer, a protocol participant gathers the SpendingKeys they own, selects a subset of the tUTXOs they have still not spent (with a fixed AssetId), collects ShieldedAddresses from other participants for the outputs of the Transfer, assigning each key a subset of the input Assets, and then builds a Transfer object representing that transaction. From this Transfer object, they construct a TransferPost which they then send to the Ledger to be validated, representing a completed state transition in the Ledger, updating the tUTXOSet and VoidNumberSet. 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 *should 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 4.4.1 (Transfer Configuration)**.** A TransferConfiguration is a collection of implementations of the following abstract cryptographic primitives:

[4] Note that they do not represent actual individual participants in a transaction, but instead just the data involved in the transaction.

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

with the following notational conventions:

$$\begin{aligned}
\mathsf{sUTXO} &:= \mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Output} \\
\mathsf{UTXO} &:= \mathsf{VersionTag} \times \mathsf{Bool} \times \mathsf{PublicAsset} \times \mathsf{sUTXO} \\
\mathsf{VoidNumber} &:= \mathsf{COM}^{\mathsf{VN}}.\mathsf{Output} \\
\mathsf{IncomingNote} &:= \mathsf{KeySchedule}.\mathsf{ShieldedAddress} \times \mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Randomness} \times \mathsf{Asset} \\
\mathsf{OutgoingNote} &:= \mathsf{Asset} \\
\mathsf{UTXOSet} &:= \mathsf{DCA}
\end{aligned}$$

and the following constraints:

$$\begin{aligned}
\mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Input} &= \mathsf{KeySchedule}.\mathsf{ShieldedAddress} \times \mathsf{Asset} \\
\mathsf{COM}^{\mathsf{VN}}.\mathsf{Randomness} &= \mathsf{KeySchedule}.\mathsf{ProofAuthorizingKey} \\
\mathsf{COM}^{\mathsf{VN}}.\mathsf{Input} &= \mathsf{UTXO} \\
\mathsf{UTXOSet}.\mathsf{Item} &= \mathsf{UTXO} \\
\mathsf{aHPKE}^{\mathsf{in}}.\mathsf{KA} &= \mathsf{KeySchedule}.\mathsf{KA} \\
\mathsf{aHPKE}^{\mathsf{out}}.\mathsf{KA} &= \mathsf{KeySchedule}.\mathsf{KA} \\
\mathsf{HASH}.\mathsf{Input} &= \mathsf{UTXO} \\
\mathsf{ValidTransfer} &: \mathsf{NIZK}.\mathsf{Statement}
\end{aligned}$$

where $\mathsf{ValidTransfer}$ 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 also implicitly use the $\mathsf{KeySchedule}$ and drop its prefix when refering to its members. We continue by defining the Sender and $\mathsf{Receiver}$ constructions as well as their public counterparts, the $\mathsf{SenderPost}$ and $\mathsf{ReceiverPost}$.

Definition 4.4.2 ($\mathsf{Transfer}$ Sender)**.** A Sender is the following tuple:

$$\begin{aligned}
\mathsf{ak} &: \mathsf{ProofAuthorizingKey} \\
\alpha &: \mathsf{rKDF}.\mathsf{Randomness} \\
\mathsf{ak}_\alpha &: \mathsf{KA}.\mathsf{PublicKey} \\
\mathsf{ViewKey} &: \mathsf{ViewingKey} \\
(\mathsf{tag}_{\mathsf{in}}, \mathsf{epk}_{\mathsf{in}}, \mathsf{C}_{\mathsf{in}}) &: \mathsf{AuthEncrypted}\langle\mathsf{IncomingNote}\rangle \\
\mathsf{D} &: \mathsf{Diversifier} \\
r &: \mathsf{COM}^{\mathsf{UTXO}}.\mathsf{Randomness} \\
\mathsf{sa} &: \mathsf{SecretAsset} \\
\mathsf{pk}_{\mathsf{D}} &: \mathsf{KA}.\mathsf{PublicKey} \\
\mathsf{esk}_{\mathsf{out}} &: \mathsf{KA}.\mathsf{SecretKey} \\
(\mathsf{tag}_{\mathsf{out}}, \mathsf{epk}_{\mathsf{out}}, \mathsf{C}_{\mathsf{out}}) &: \mathsf{AuthEncrypted}\langle\mathsf{OutgoingNote}\rangle \\
\mathsf{utxo} &: \mathsf{UTXO} \\
\mathsf{tutxo} &: \mathsf{tUTXO} \\
(z_{\mathsf{tutxo}}, \pi_{\mathsf{tutxo}}) &: \mathsf{tUTXOSet}.\mathsf{MembershipProof} \\
\mathsf{vn} &: \mathsf{VoidNumber}
\end{aligned}$$

A Sender, S, is constructed from a proof authorizing key $\mathsf{ak} : \mathsf{ProofAuthorizingKey}$, a randomizer $\alpha : \mathsf{rKDF.Randomness}$, a tutxo identification $\mathsf{id} : \mathsf{tUTXOID}$, a public asset $\mathsf{pa} : \mathsf{PublicAsset}$, and an encrypted message $(\mathsf{tag}_\mathsf{in}, \mathsf{epk}_\mathsf{in}, \mathsf{C}_\mathsf{in}) : \mathsf{AuthEncrypted}(\mathsf{IncomingNote})$ with the following algorithm:

$$\begin{aligned}
\mathsf{ak}_\alpha &:= \mathsf{rKDF.rand}_\alpha(\mathsf{ak}) \\
\mathsf{vk} &:= \mathsf{KDF}^\mathsf{vk}(\mathsf{ak}) \\
\mathsf{Some}(\mathsf{D}, r, \mathsf{sa}) &:= \mathsf{aHPKE}^\mathsf{in}.\mathsf{decrypt}(\mathsf{vk}, \mathsf{tag}_\mathsf{in}, \mathsf{epk}_\mathsf{in}, \mathsf{C}_\mathsf{in}) \\
\mathsf{saiz} &:= \mathsf{eq}(\mathsf{sa.value}, 0) \\
\mathsf{pk}_\mathsf{D} &:= \mathsf{KA.agree}_\mathsf{D}(\mathsf{vk}) \\
\mathsf{esk}_\mathsf{out} &:\sim \mathsf{KA.SecretKeyDistribution} \\
(\mathsf{tag}_\mathsf{out}, \mathsf{epk}_\mathsf{out}, \mathsf{C}_\mathsf{out}) &:= \mathsf{aHPKE}^\mathsf{out}.\mathsf{encrypt}_\mathsf{D}(\mathsf{esk}_\mathsf{out}, \mathsf{pk}_\mathsf{D}, \mathsf{sa}) \\
\mathsf{utxo} &:= \mathsf{COM}^\mathsf{UTXO}_r(\mathsf{D}, \mathsf{pk}_\mathsf{D}, \mathsf{sa}) \\
h &:= \mathsf{HASH.hash}(\mathsf{pa}, \mathsf{utxo}, \mathsf{saiz}, \mathsf{id}) \\
\mathsf{Some}(z_\mathsf{tutxo}, \pi_\mathsf{tutxo}) &:= \mathsf{tUTXOSet.prove}(h) \\
\mathsf{vn} &:= \mathsf{COM}^\mathsf{VN}_\mathsf{ak}(h)
\end{aligned}$$

Definition 4.4.3 (Transfer Sender Post)**.** A $\mathsf{SenderPost}$ is the following tuple extracted from a Sender:

$$\begin{aligned}
\mathsf{ak}_\alpha &: \mathsf{KA.PublicKey} \\
(\mathsf{tag}_\mathsf{out}, \mathsf{epk}_\mathsf{out}, \mathsf{C}_\mathsf{out}) &: \mathsf{AuthEncrypted}\langle\mathsf{OutgoingNote}\rangle \\
z_\mathsf{tutxo} &: \mathsf{tUTXOSet.Output} \\
\mathsf{vn} &: \mathsf{VoidNumber}
\end{aligned}$$

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

Definition 4.4.4 (Transfer Receiver)**.** A $\mathsf{Receiver}$ is the following tuple:

$$\begin{aligned}
(\mathsf{D}, \mathsf{pk}_\mathsf{D}) &: \mathsf{ShieldedAddress} \\
r &: \mathsf{COM}^\mathsf{UTXO}.\mathsf{Randomness} \\
\mathsf{pa} &: \mathsf{PublicAsset} \\
\mathsf{sa} &: \mathsf{SecretAsset} \\
\mathsf{saiz} &: \mathsf{Bool} \\
\mathsf{utxo} &: \mathsf{UTXO} \\
\mathsf{tutxo} &: \mathsf{tUTXO} \\
\mathsf{esk}_\mathsf{in} &: \mathsf{KA.SecretKey} \\
(\mathsf{tag}_\mathsf{in}, \mathsf{epk}_\mathsf{in}, \mathsf{C}_\mathsf{in}) &: \mathsf{AuthEncrypted}\langle\mathsf{IncomingNote}\rangle
\end{aligned}$$

A $\mathsf{Receiver}$, R, is constructed from a shielded address $(\mathsf{D}, \mathsf{pk}_\mathsf{D}) : \mathsf{ShieldedAddress}$, a secret asset $\mathsf{sa} : \mathsf{SecretAsset}$, a public asset $\mathsf{pa} : \mathsf{PublicAsset}$, a tutxo identification $\mathsf{id} : \mathsf{tUTXOID}$, and a UTXO-commitment randomness $r : \mathsf{COM}^\mathsf{UTXO}.\mathsf{Randomness}$ with the following algorithm:

$$\begin{aligned}
\mathsf{utxo} &:= \mathsf{COM}^\mathsf{UTXO}_r(\mathsf{D}, \mathsf{pk}_\mathsf{D}, \mathsf{sa}) \\
\mathsf{saiz} &:= \mathsf{eq}(\mathsf{sa.value}, 0) \\
\mathsf{tutxo} &:= (\mathsf{pa}, \mathsf{utxo}, \mathsf{saiz}, \mathsf{id}) \\
\mathsf{esk}_\mathsf{in} &:\sim \mathsf{KA.SecretKeyDistribution} \\
(\mathsf{tag}_\mathsf{in}, \mathsf{epk}_\mathsf{in}, \mathsf{C}_\mathsf{in}) &:= \mathsf{aHPKE}^\mathsf{in}.\mathsf{encrypt}_\mathsf{D}(\mathsf{esk}_\mathsf{in}, \mathsf{pk}_\mathsf{D}, (\mathsf{D}, r, \mathsf{sa}))
\end{aligned}$$

Definition 4.4.5 (Transfer Receiver Post)**.** A $\mathsf{ReceiverPost}$ is the following tuple extracted from a $\mathsf{Receiver}$:

$$\begin{aligned}
\mathsf{tutxo} &: \mathsf{tUTXO} \\
(\mathsf{tag}_\mathsf{in}, \mathsf{epk}_\mathsf{in}, \mathsf{C}_\mathsf{in}) &: \mathsf{AuthEncrypted}\langle\mathsf{IncomingNote}\rangle
\end{aligned}$$

which are the parts of a $\mathsf{Receiver}$ which should be *posted* to the Ledger.

Definition 4.4.6 (Transfer Sources and Sinks)**.** A Source (or a Sink) is a $\mathsf{PublicAsset}$ representing a public input (or output) of a $\mathsf{Transfer}$.

Definition 4.4.7 (Transfer Object)**.** A Transfer is the following tuple:

$$\begin{aligned}\mathsf{sources} &: \mathsf{List}\langle\mathsf{PublicAsset}\rangle\\ \mathsf{senders} &: \mathsf{List}\langle\mathsf{Sender}\rangle\\ \mathsf{receivers} &: \mathsf{List}\langle\mathsf{Receiver}\rangle\\ \mathsf{sinks} &: \mathsf{List}\langle\mathsf{PublicAsset}\rangle\end{aligned}$$

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

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

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

- **Correct Zero Asset Value Indicator**: All saizs in the Transfer must correctly indicate whether the underlying secret asset value is zero.
- **Same Id**: All non-zero-value Assets in the Transfer must have the same AssetId.
- **Balanced**: For all non-zero-value Assets, 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 4.4.8 (Transfer Validity Statement)**.** A transfer T : Transfer is considered *valid* if and only if

1. For all $S \in T.\mathsf{senders}$ and $R \in T.\mathsf{receivers}$, $\mathsf{saiz}s : \mathsf{Bool}$ and $\mathsf{nza}s : \mathsf{Asset}$ are set correctly:

$$\begin{aligned}\mathsf{saiz} &= \mathsf{eq}(\mathsf{sa.value}, 0)\\ \mathsf{nza} &= \mathsf{select}(\mathsf{saiz}, \mathsf{pa}, \mathsf{sa})\end{aligned}$$

2. All the non-zero-value Assets in T has the same AssetIds:

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

3. For all the non-zero-value Assets, the sum of input AssetValues is equal to the sum of output AssetValues:

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

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

$$\begin{aligned}\mathsf{ak}_\alpha &= \mathsf{rKDF.rand}_\alpha(\mathsf{ak})\\ \mathsf{vk} &= \mathsf{KDF}^{\mathsf{vk}}(\mathsf{ak})\\ \mathsf{pk}_{\mathsf{D}} &= \mathsf{KA.derive}_{\mathsf{D}}(\mathsf{vk})\\ (\mathsf{tag}_{\mathsf{out}}, \mathsf{epk}_{\mathsf{out}}, \mathsf{C}_{\mathsf{out}}) &= \mathsf{aHPKE}^{\mathsf{out}}.\mathsf{encrypt}_{\mathsf{D}}(\mathsf{esk}_{\mathsf{out}}, \mathsf{pk}_{\mathsf{D}}, \mathsf{sa})\\ \mathsf{utxo} &= \mathsf{COM}_r^{\mathsf{UTXO}}(\mathsf{D}, \mathsf{pk}_{\mathsf{D}}, \mathsf{sa})\\ h &= \mathsf{HASH.hash}(\mathsf{pa}, \mathsf{utxo}, \mathsf{saiz}, \mathsf{id})\\ \mathsf{True} &= \mathsf{tUTXOSet.verify}(h, z_{\mathsf{tutxo}}, \pi_{\mathsf{tutxo}})\\ \mathsf{vn} &= \mathsf{COM}_{\mathsf{ak}}^{\mathsf{VN}}(h)\end{aligned}$$

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

$$\begin{aligned}\mathsf{utxo} &= \mathsf{COM}_r^{\mathsf{UTXO}}(\mathsf{D}, \mathsf{pk}_{\mathsf{D}}, \mathsf{sa})\\ (\mathsf{tag}_{\mathsf{in}}, \mathsf{epk}_{\mathsf{in}}, \mathsf{C}_{\mathsf{in}}) &= \mathsf{aHPKE}^{\mathsf{in}}.\mathsf{encrypt}_{\mathsf{D}}(\mathsf{esk}_{\mathsf{in}}, (\mathsf{D}, r, \mathsf{sa}))\end{aligned}$$

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

Definition 4.4.9 (Transfer Post)**.** A $\mathsf{TransferPost}$ is the following tuple:

$$\begin{aligned}
\mathsf{sources} &: \mathsf{List}\langle \mathsf{Source} \rangle \\
\mathsf{senders} &: \mathsf{List}\langle \mathsf{SenderPost} \rangle \\
\mathsf{receivers} &: \mathsf{List}\langle \mathsf{ReceiverPost} \rangle \\
\mathsf{sinks} &: \mathsf{List}\langle \mathsf{Sink} \rangle \\
\pi &: \mathsf{NIZK.Proof}
\end{aligned}$$

A $\mathsf{TransferPost}$, P, 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 $T : \mathsf{Transfer}$:

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

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 that the prover has constructed the proof, the underlying spending keys need to authorize the transaction before it can be sent to the Ledger.

Definition 4.4.10 (Proof Authorization)**.** Given a transfer post $P : \mathsf{TransferPost}$ and a set of spending keys $S = \{(\mathsf{sk}_i, \alpha_i)\}$ where $(\mathsf{sk}_i, \alpha_i)$ come from the ith spender associated to the $P.\mathsf{senders}_i$, we have the following signature:

$$\Sigma := \{\mathsf{SIG.sign}(\mathsf{rKDF.rand}_\alpha(\mathsf{sk}), P) | (\mathsf{sk}, \alpha) \in S\}$$

which can be verified by the ledger with

$$\forall_i \mathsf{SIG.verify}(P.\mathsf{senders}_i.\mathsf{ak}_\alpha, P, \Sigma_i) = \mathsf{True}$$

Now that the transfer post has been signed by the owners of the spending keys, it can be sent up to the Ledger.

Definition 4.4.11 (Ledger-side Transfer Validity)**.** To check that the transfer post P represents a valid $\mathsf{Transfer}$, the ledger checks the following:

- **Signature Check**: All the signatures associated to the transactions are valid.
- **Public Withdraw**: All the public addresses corresponding to the Assets in $P.\mathsf{sources}$ have enough public balance (i.e. in the $\mathsf{PublicAssetLedger}$) to withdraw the given Asset.
- **Public Deposit**: All the public addresses corresponding to the Assets in $P.\mathsf{sinks}$ exist.
- **Current Accumulated State**: The $\mathsf{tUTXOSet.Output}$ stored in each $P.\mathsf{senders}$ is equal to current accumulated value, $\mathsf{tUTXOSet.current}(\mathsf{Ledger.utxo}s())$, for the current state of the Ledger.
- **New VoidNumbers**: All the $\mathsf{VoidNumber}$s in $P.\mathsf{senders}$ are unique, and no $\mathsf{VoidNumber}$ in $P.\mathsf{senders}$ has already been stored in the $\mathsf{Ledger.VoidNumberSet}$.
- **New UTXOs**: All the UTXOs in $P.\mathsf{receivers}$ are unique, and no UTXO in $P.\mathsf{receivers}$ has already been stored on the ledger.
- **Verify Transfer**: Check that $\mathsf{NIZK.verify}_{\mathsf{vk}}(P.\mathsf{sources} \,||\, P.\mathsf{senders} \,||\, P.\mathsf{receivers} \,||\, P.\mathsf{sinks}, P.\pi) = \mathsf{True}$. Here, $\mathsf{vk} : \mathsf{NIZK.VerifyingKey}$ is a known verifying key.

Definition 4.4.12 (Ledger Transfer Update)**.** After checking that a given $\mathsf{TransferPost}$ 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.
- **tUTXOSet Update**: The new tUTXOs are appended to the tUTXOSet.
- **VoidNumberSet Update**: The new VoidNumbers are appended to the VoidNumberSet.

Note: Ledger only accepts a tutxo : tUTXO for smart contract if tutxo.saiz = True.

4.5 Batched Transactions

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

Algorithm 1 Batch Transaction Algorithm

```
procedure BuildTransaction(sk, 𝓑, total, addr)
    B ← Sample(total, 𝓑)        ▷ Samples key-asset pairs from 𝓑 whose asset 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.d̃, acc.asset.value), 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(addr, S)
    [c, zs...] ← BuildAccumulatorAndZeroes_sk(S)
    return P + TransferPost(Transfer([], S, [R, c, zs...], []))
end procedure
```

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 shielded balance. To describe such a *semantic transaction*, we assume the existence of two transfer shapes[5]: Mint with shape $(1, 0, 1, 0)$ and PrivateTransfer with shape $(0, N, N, 0)$ for some natural number $N > 1$.

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

$$\textsc{BuildTransaction}(\mathsf{sk}, \mathcal{B}, \mathsf{total}, \mathsf{addr})$$

to receive a List(TransferPost) to send to the ledger, representing the transfer of total to addr.

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 Assets 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 BuildTransaction algorithm.

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

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**$^\pi$) [7] 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:

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

over some sufficiency 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:

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

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

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` [9] 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:

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

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:

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

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:

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

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 $\mathsf{OutgoingNote}$ and $\mathsf{IncomingNote}$ 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 $\textbf{Poseidon}_2$ 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 $\textbf{Poseidon}_2$ 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 [7] and/or PLONK/PLONKUP [6, 8].

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 $\mathsf{AssetValue}$ 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 `BLS12-381` 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] Byzantine Fault Tolerance. https://en.wikipedia.org/wiki/Byzantine_fault.

[2] State Machine Replication. https://en.wikipedia.org/wiki/State_machine_replication.

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

[4] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT*, volume 2248 of *Lecture Notes in Computer Science*, pages 566–582. Springer, 2001.

[5] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[6] 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.

[7] 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.

[8] 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.

[9] Barry WhiteHat, Marta Bellés, and Jordi Baylina. EIP-2494: Baby Jubjub Elliptic Curve . Eip, Ethereum Foundation, 2020.