Shielded Transaction Protocol

CONTENTS

1. Overview

TRONZ team from TRON community has implemented shielded transaction. This specification is intended to give a concise summary of the protocol.

Value in transaction is either transparent or shielded. Shielded value is carried by notes, which specify an amount and (indirectly) a shielded payment address, which is a destination to which notes can be sent. Note is associated with a private key named spending key that can be used to spend notes sent to the address.

To each note there is cryptographically associated a note commitment. Once the transaction creating the note has been mined, it is associated with a fixed note position in MerkleTree that is a tree of note commitments, and with a nullifier unique to that note. Computing the nullifier requires the nullifier deriving key. It is infeasible to correlate the note commitment or note position with the corresponding nullifier without knowledge of at least this key. An unspent valid note, at a given point on the block chain, is one for which the note commitment has been publically revealed on the block chain prior to that point, but the nullifier has not.

A transaction can contain transparent inputs and outputs. It also includes Spend descriptions, and Receive descriptions. Together these describe shielded transfers which take in shielded input notes, and/or produce shielded output notes. We support one input, and two outputs at most(Technically, we could support many inputs and outputs). Each shielded input or shielded output has its own description. It is also possible for value to be transferred between the transparent and shielded domains.

The nullifiers of the input notes are revealed (preventing them from double-spending) and the commitments of the output notes are revealed (allowing them to be spent in future). A transaction also includes computationally sound zk-SNARK proofs and signatures, which prove that all of the following hold except with insignificant probability:

For shielded input,
• there is a revealed value commitment to the same value as the input note;
• if the value is non-zero, some revealed note commitment exists for this note;
• the prover knew the proof authorizing key of the note;
• the nullifier and note commitment are computed correctly.
and for each shielded output,
• there is a revealed value commitment to the same value as the output note;
• the note commitment is computed correctly;
• it is infeasible to cause the nullifier of the output note to collide with the nullifier of any other note.

In addition, various measures are used to ensure that the transaction cannot be modified by a party not authorized to do so.

Outside the zk-SNARK, it is checked that the nullifiers for the input notes had not already been revealed (i.e. they had not already been spent).

A shielded payment address includes a transmission key for a key-private asymmetric encryption scheme. “Key- private” means that ciphertexts do not reveal information about which key they were encrypted to, except to a holder of the corresponding private key, which in this context is called the receiving key. This facility is used to communicate encrypted output notes on the block chain to their intended recipient, who can use the receiving key to scan the block chain for notes addressed to them and then decrypt those notes.

For each spending key there is a full viewing key that allows recognizing both incoming and outgoing notes without having spend authority. This is implemented by an additional ciphertext in each Output description.

The basis of the privacy properties is that when a note is spent, the spender only proves that some commitment for it had been revealed, without revealing which one. This implies that a spent note cannot be linked to the transaction in which it was created. That is, from an adversary's point of view the set of possibilities for a given note input to a transaction—its note traceability set— includes *all* previous notes that the adversary does not control or know to have been spent.

The nullifiers are necessary to prevent double-spending: each note on the block chain only has one valid nullifier, and so attempting to spend a note twice would reveal the nullifier twice, which would cause the second transaction to be rejected.

2. Notation

$\mathbb{B}$ means the type of bit values, i.e. {0, 1}. $\mathbb{B}^{\mathbb{Y}}$ means the type of byte values, i.e. {0 ... 255}. $\mathbb{N}$ means the type of nonnegative integers. $\mathbb{N}^{+}$ means the type of positive integers. $\mathbb{Z}$ means the type of integers. $\mathbb{Q}$ means the type of rationals.

$x:T$ is used to specify that x has type T. A cartesian product type is denoted by S × T, and a function type by $S \to T$. An argument to a function can determine other argument or result types.

The type of a randomized algorithm is denoted by $S \xrightarrow{R} T$. The domain of a randomized algorithm may be (), indicating that it requires no arguments. Given $f\colon S \xrightarrow{R} T$ and $s:S$, sampling a variable $x:T$ from the output of f applied to s is denoted by $x \xleftarrow{R} f\ (s)$.

Initial arguments to a function or randomized algorithm may be written as subscripts, e.g. if $x:X$, $y:Y$, and $f:X \times Y \to Z$, then an invocation of $f(x,y)$ can also be written $f_x(y)$.

$\{x:T \mid p_x\}$ means the subset of x from T for which p_x (a Boolean expression depending on x) holds.

T ⊆ U indicates that T is an inclusive subset or subtype of U. S ∪ T means the set union of S and T.

S∩T means the set intersection of S and T, i.e. $\{x:S \mid x \in T\}$.

S \ T means the set difference obtained by removing elements in T from S, i.e. $\{x:S \mid x \notin T\}$.

$x:T \mapsto e_x:U$ means the function of type T → U mapping formal parameter x to e_x (an expression depending on x). The types T and U are always explicit.

$x:T \mapsto_{\neq y} e_x:U$ means $x:T \mapsto e_x:U \cup \{y\}$restricted to the domain$\{x:\mathrm{T} \mid e_x \neq y\}$and range U.

$\mathcal{P}(T)$ means the powerset of T.

$T^{[\ell]}$ where T is a type and ℓ is an integer, means the type of sequences of length ℓ with elements in T. For example, $\mathbb{B}^{[\ell]}$ means the set of sequences of ℓ bits, and $\mathbb{B}^{\mathbb{Y}[k]}$ means the set of sequences of k bytes.

$\mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$means the type of byte sequences of arbitrary length.

length(S) means the length of number of elements in S.

$\mathrm{truncate}_k$ (S) means the sequence formed from the first k elements of S.

0x followed by a string of monospace hexadecimal digits means the corresponding integer converted from hexadecimal. $[0x00]^l$ means the sequence of l zero bytes.

"..." means the given string represented as a sequence of bytes in US-ASCII.

$[0]^l$ means the sequence of l zero bits. $[1]^l$ means the sequence of l one bits.

$\{a \ldots b\}$ means the set or type of integers from a through b inclusive.

$[\ f\ (x)\ for\ x\ from\ a\ up\ to\ b\ ]$ means the sequence formed by evaluating f on each integer from a to b inclusive, in ascending order. Similarly, $[\ f\ (x)\ for\ x\ from\ a\ down\ to\ b\ ]$ means the sequence formed by evaluating f on each integer from a to b inclusive, in descending order.

a || b means the concatenation of sequences a then b.

concat_{B}(S) means the sequence of bits obtained by concatenating the elements of S viewed as bit sequences. If the elements of S are byte sequences, they are converted to bit sequences with the *most significant* bit of each byte first.

sorted(S) means the sequence formed by sorting the elements of S.

$\mathbb{F}_n$ means the finite field with n elements, and $\mathbb{F}_n^*$ means its group under multiplication (which excludes 0).

Where there is a need to make the distinction, we denote the unique representative of $a\colon \mathbb{F}_n$ in the range {0 ...n − 1} (or the unique representative of $a\colon \mathbb{F}_n^*$ in the range {1 ... n − 1}) as a mod n. Conversely, we denote the element of $\mathbb{F}_n$ corresponding to an integer $k\colon \mathbb{Z}$ $as\ k\ (mod\ n)$. We also use the latter notation in the context of an equality $k = k' \pmod n$ as shorthand for $k \bmod n = k' \bmod n$, and similarly $k \neq k' \pmod n$ as shorthand for $k \bmod n \neq k' \bmod n$. (When referring to constants such as 0 and 1 it is usually not necessary to make the distinction between field elements and their representatives, since the meaning is normally clear from context.)

$\mathbb{F}_n[x]$means the ring of polynomials over x with coefficients in $\mathbb{F}_n$.

a + b means the sum of a and b. This may refer to addition of integers, rationals, finite field elements, or group elements according to context.

−a means the value of the appropriate integer, rational, finite field, or group type such that (−a) + a = 0 (or when a is an element of a group G, (−a) + a = O_G), and a − b means a + (−b).

a · b means the product of multiplying a and b. This may refer to multiplication of integers, rationals, or finite field elements according to context (this notation is not used for group elements).

a/b, also written $\frac{a}{b}$, means the value of the appropriate integer, rational, or finite field type such that (a/b) · b = a.

a mod q, for $a\colon \mathbb{N}$ and $q\colon \mathbb{N}^+$, means the remainder on dividing a by q. (This usage does not conflict with the notation above for the unique representative of a field element.)

a ⊕ b means the bitwise-exclusive-or of a and b, and a & b means the bitwise-and of a and b. These are defined on integers or (equal-length) bit sequences according to context.

$\sum_{i=1}^{N} a_i$means the sum of $\text{a}_{1..\text{N}}$. $\prod_{i=1}^{N} a_i$means the product of $\text{a}_{1..\text{N}}$. $\bigoplus_{i=1}^{N}$means the bitwise exclusive-or of $\text{a}_{1..\text{N}}$. When N = 0 these yield the appropriate neutral element, i.e. $\sum_{i=1}^{0} a_i = 0$, $\prod_{i=1}^{0} a_i = 1$, and $\bigoplus_{i=1}^{N} = 0$ or the all-zero bit sequence of the appropriate length given by the type of a.

$\sqrt{a}$, where a: $\mathbb{F}_q$ means the positive (i.e. in the range $\{0 \ldots \frac{q-1}{2}\}$) square root of a in $\mathbb{F}_q$. It is only used in cases where the square root must exist.

b?x:y means x when b = 1, or y when b = 0.

a^b, for a: an integer or finite field element and b: $\mathbb{Z}$, means the result of raising a to the exponent b, i.e.

$$a^b := \begin{cases} \prod_{i=1}^{b} a, \ if \ b \geq 0. \\ \prod_{i=1}^{-b} \frac{1}{a}, \ otherwise \end{cases}$$

The convention of affixing ⋆ to a variable name is used for variables that denote bit-sequence representations of group elements.

The binary relations <, ≤, =, ≥, and > have their conventional meanings on integers and rationals, and are defined lexicographically on sequences of integers.

$floor(x)$ means the largest integer ≤ x. ceiling (x) means the smallest integer ≥ x.

$bitlength(x)$, for x: $\mathbb{N}$, means the smallest integer l such that $2^l > x$.

The symbol ⊥ is used to indicate unavailable information, or a failed decryption or validity check.

3. Cryptographic Primitives

3.1. Encoding rules

All integers encodings are unsigned, have a fixed bit length, and are encoded in little-endian byte order unless otherwise specified.

The following functions convert between sequences of bits, sequences of bytes, and integers:

• $I2LEBSP$: $(l: \mathbb{N}) \times \{0 \ldots 2^l - 1\} \rightarrow \mathbb{B}^l$, such that $I2LEBSP_l(x)$ is the sequence of l bits representing x in little-endian order;

• $I2BEBSP$: $(l: \mathbb{N}) \times \{0 \ldots 2^l - 1\} \rightarrow \mathbb{B}^l$ such that $I2BEBSP_l(x)$ is the sequence of l bits representing x in big-endian order.

• $LEOS2IP$: $(l: \mathbb{N} \mid l \bmod 8 = 0) \times \mathbb{B}^{\mathbb{Y}[l/8]} \rightarrow \{0 \ldots 2^l - 1\}$ such that $LEOS2IP_l(S)$ is the integer represented in little-endian order by the byte sequence S of length $l/8$.

• $LEBS2OSP$: $(l: \mathbb{N}) \times \mathbb{B}^{[l]} \rightarrow \mathbb{B}^{\mathbb{Y}\left[ceiling\left(\frac{l}{8}\right)\right]}$ defined as follows: pad the input on the right with $8 \cdot ceiling\left(\frac{l}{8}\right) - l$ zero bits so that its length is a multiple of 8 bits. Then convert each group of 8 bits to a byte value with the *least* significant bit first, and concatenate the resulting bytes in the same order as the groups.

•$LEOS2BSP$: $(l: \mathbb{N} | l \bmod 8 = 0) \times \mathbb{B}^{\mathbb{Y}\left[ceiling\left(\frac{l}{8}\right)\right]} \rightarrow \mathbb{B}^{[l]}$ defined as follows: convert each byte to a group of 8 bits with the *least* significant bit first, and concatenate the resulting groups in the same order as the bytes.

In bit layout diagrams, each box of the diagram represents a sequence of bits. Diagrams are read from left-to- right, with lines read from top-to-bottom; the breaking of boxes across lines has no significance. The bit length l is given explicitly in each box, except when it is obvious (e.g. for a single bit, or for the notation $[0]^l$ representing the sequence of l zero bits, or for the output of $LEBS2OSP_l$).

The entire diagram represents the sequence of *bytes* formed by first concatenating these bit sequences, and then treating each subsequence of 8 bits as a byte with the bits ordered from *most significant* to *least significant*. Thus, the *most significant* bit in each byte is toward the left of a diagram. Where bit fields are used, the text will clarify their position in each case.

3.2. Constants

$MerkelDepth: \mathbb{N} = 32$
$N^{old}: \mathbb{N} = 2$
$N^{new}: \mathbb{N} = 2$
$l_{value}: \mathbb{N} = 64$
$l_{merkle}: \mathbb{N} = 256$
$l_{PRFexpand}: \mathbb{N} = 256$
$l_{PRFnf}: \mathbb{N} = 256$
$l_{rcm}: \mathbb{N} = 256$
$l_{sk}: \mathbb{N} = 256$
$l_d: \mathbb{N} = 88$
$l_{ivk}: \mathbb{N} = 251$
$l_{ovk}: \mathbb{N} = 256$
$l_{scalar}: \mathbb{N} = 252$
$Uncommited: \mathbb{B}^{l_{merkle}} = I2LEBSP_{l_{merkle}}(1)$

3.3. Hash Functions

3.3.1. BLAKE2 Hash Function

BLAKE2 is a hash defined in [ANWW2013], specifically, BLAKE2b and BLAKE2s variants are used in shielded transaction.

$BLAKE2b_l(p, x)$ refers to unkeyed $BLAKE2b_l$ in sequential mode, with an output digest length of l/8 bytes, 16-byte personalization string p, and input x.

$BLAKE2s_l(p, x)$ refers to unkeyed $BLAKE2s_l$ in sequential mode, with an output digest length of l/8 bytes, 8-byte personalization string p, and input x.

3.3.2. $\mathrm{CRH}_{\mathrm{ivk}}$ Hash Function

CRH^{ivk} is used to derive the incoming viewing key ivk for a shielded payment address. It is defined as follow:

$$CRH^{ivk}(ak\star, nk\star) := LEOS2IP_{256}(BLAKE2s_{256}(\text{"Zcashivk"}, crhInput)) \bmod 2^{l_{ivk}}$$

Where

$$crhInput = (LEBS2OSP_{256}(ak\star) \,||\, LEBS2OSP_{256}(nk\star))$$

3.3.3. DiversifyHash Function

$DiversifyHash$ is used to derive a diversified base from a diversifier.

Let $GroupHash^{\mathbb{J}^{(r)*}}$ and U be defined in §3.8.

Define $DiversifyHash_U^{\mathbb{J}^{(r)*}} = Grouphash_U^{\mathbb{J}^{(r)*}}\left(\text{"Zcash\_gd"},\ LEBS2OSP_{l_d}(d)\right)$

3.3.4. Pedersen Hash Function

PedersenHash is an algebraic hash function with collision resistance (for fixed input length) derived from assumed hardness of the Discrete Logarithm Problem on the Jubjub curve.

PedersenHash is used in the incremental Merkle tree over note commitments and in the definition of Pedersen commitments

Let $\mathbb{J}$, $\mathbb{J}^{(r)}, \mathcal{O}_{\mathbb{J}}, r_{\mathbb{J}}, a_{\mathbb{J}}, and\ d_{\mathbb{J}}$ be as defined in §3.7.

Let $Extract_{\mathbb{J}^{(r)}}: \mathbb{J}^{(r)} \to \mathbb{B}^{l_{merkle}}$ *be as defined in* §3.7.

Let $FindGroupHash^{\mathbb{J}^{(r)}*}$ be as defined in §3.7.

Let $c = 63$

Define $\mathcal{J}: \mathbb{B}^{\mathbb{Y}[8]} \times \mathbb{N} \to \mathbb{J}^{(r)*}$ by:

$$\mathcal{J}_i^D = FindGroupHash^{\mathbb{J}^{(r)}*}(D, 32bit(i-1))$$

Define $PedersenHashToPoint\left(D: \mathbb{B}^{\mathbb{Y}[8]}, M: \mathbb{B}^{\mathbb{N}+}\right) \to \mathbb{J}^{(r)}$ as follows:

Pad M to a multiple of 3 bits by appending zero bits, giving M′.

Let $n = ceiling(\frac{length(M\prime)}{3 \cdot c})$

Split M′ into n "segments" $M_{1..n}$ so that $M' = concat_B(M_{1..n})$, and each of $M_{1..n-1}$ is of length 3·c bits. (M_n may be shorter.)

Return $\sum_{i=1}^{n}[< M_i >]\,\mathcal{J}_i^D: \mathbb{J}^{(r)}$

Where $<\cdot>: \mathbb{B}^{[3\cdot\{1..c\}]} \to \{-\frac{r_{\mathbb{J}}-1}{2}..\frac{r_{\mathbb{J}}+1}{2}\}\{0\}$ is defined as:

Let $k_i = length(M_i)/3$

Split M_i into 3-bit "chunks" $m_{1..k_i}$ so that $M_i = concat_B(m_{1..k_i})$.

Write each m as $[s_0^j, s_1^j, s_2^j]$, and let $enc(m) = \left(1 - 2 \cdot s_2^j\right) \cdot \left(1 + s_0^j + 2 \cdot s_1^j\right): \mathbb{Z}$

Let $< M_i > = \sum_{j=1}^{k_i} enc(m_j) \cdot 2^{4(j-1)}$.

Finally, define $PedersenHash: \mathbb{B}^{\mathbb{Y}[8]} \times \mathbb{B}^{\mathbb{N}+} \to \mathbb{B}^{l_{merkle}}$ by:

$$PedersenHash(D, M) = Extract_{\mathbb{J}^{(r)}}(PedersenHashToPoint(D, M))$$

3.3.5. Mixing Pedersen Hash Function

A mixing Pedersen hash is used to compute ρ from cm and pos. It takes as input a Pedersen commitment P, and hashes it with another input x.

Define $\mathcal{J} = FindGroupHash^{\mathbb{J}^{(r)*}}$("Zcash_J_", "")

Define $MixingPedersenHash: \mathbb{J} \times \{0..r_{\mathbb{J}} - 1\} \to \mathbb{J}$ by:

$$MixingPedersenHash(P, x) = P + [x]\,\mathcal{J}$$

3.4. Pseudo Random Function

PRF^{expand} is used in to derive the spend authorizing key ask and the proof authorizing key nsk.

It is instantiated using the BLAKE2b hash function

$$PRF^{expand}_{sk}(t) := BLAKE2b_{512}(\text{"}Zcash\_ExpandSeed\text{"}, LEBS2OSP(sk) \,||\, t)$$

PRF^{ock} is used to derive the outgoing cipher key ock used to encrypt an output ciphertext .
It is instantiated using the BLAKE2b hash function.

$$PRF^{ock}_{ovk}(cv, cm_u, ephemeralKey) := BLAKE2b_{256}(\text{“}\boldsymbol{Ztron\_Derive\_ock}\text{”}, ockInput)$$

where

$$ockInput = LEBS2OSP_{256}(ovk)\big|\big|32byte\ cv\big|\big|32byte\ cm_u||32byte\ ephemeralKey$$

PRF^{nf} is used to derive the nullifier for a note. It is instantiated using the BLAKE2s hash function.

$$PRF^{nf}_{nk\star}(\rho\star) = BLAKE2s_{256}(\text{“}Zcash\_nf\text{”}, LEBS2OSP_{256}(nk\star)||LEBS2OSP_{256}(\rho\star))$$

3.5. Authenticated One-Time Symmetric Encryption

Let $Sym.K := \mathbb{B}^{[256]}, Sym.P := \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}, and\ Sym.C := \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$.

Let $Sym.Encrypt_K(P)$ be authenticated encryption using $AEAD\_CHACHA20\_POLY1305$ [RFC-7539] encryption of plaintext $P \in Sym.P$, with empty “associated data", all-zero nonce $[0]^{96}$, and 256-bit key $K \in Sym.K$.

Similarly, let $Sym.Decrypt_K(C)$ be $AEAD\_CHACHA20\_POLY1305$ decryption of ciphertext $C \in Sym.C$, with empty “associated data", all-zero nonce $[0]^{96}$, and 256-bit key $K \in Sym.K$. The result is either the plaintext byte sequence, or $\perp$ indicating failure to decrypt.

3.6. Key Agreement and Derivation

KA is a key agreement scheme. It is instantiated as Diffie-Hellman with cofactor multiplication on Jubjub as follows:
Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and the cofactor $h_{\mathbb{J}}$ be as defined in §3.7.
Define $KA.Public := \mathbb{J}$
Define $KA.PublicPrimeOrder := \mathbb{J}^{(r)*}$
Define $KA.SharedSecret := \mathbb{J}^{(r)}$
Define $KA.Private := \mathbb{F}_{r_{\mathbb{J}}}$
Define $KA.DerivePublic(sk, B) := [sk]B$
Define $KA.Agree(sk, P) := [h_{\mathbb{J}} \cdot sk]\,P$

KDF is a Key Derivation Function. It is instantiated using $BLAKE2b_{256}$ as follows:

$$KDF(sharedSecret, epk) := BLAKE2b_{256}(\text{"}Ztron\_SaplingKDF\text{"}, kdfinput)$$

Where

$$kdfinput = LEBS2OSP_{256}(repr_{\mathbb{J}}(sharedSecret)||LEBS2OSP_{256}(repr_{\mathbb{J}}(epk))$$

3.7. Jubjub and RedJubjub

We use an elliptic curve designed to be efficiently implementable in zk-SNARK circuits, called "Jubjub".

Let $\mathbb{J}$ be the group of points (u, v) on a twisted Edwards curve $E_{\mathbb{J}}$ over $F_{r_{\mathbb{J}}}$.

The equation is $a_{\mathbb{J}} \cdot u^2 + v^2 = 1 + d_{\mathbb{J}} \cdot u^2 \cdot v^2 \ mod \ q_{\mathbb{J}}$. The parameters are defined as follows.

Let
$q_{\mathbb{J}}$ =52435875175126190479447740508185965837690552500527637822603658699938581184513

Let
$r_{\mathbb{J}}$ =65544843968907738099309675635232457297059212658723172813653591623921832 54199

($q_{\mathbb{J}}$ and $r_{\mathbb{J}}$ are prime.)

Let $h_{\mathbb{J}} = 8$

Let $a_{\mathbb{J}} = -1$

Let $d_{\mathbb{J}} = -\frac{10240}{10241} mod \ q_{\mathbb{J}}$

The zero point with coordinates (0, 1) is denoted $\mathcal{O}_{\mathbb{J}}$. $\mathbb{J}$ has order $h_{\mathbb{J}} \cdot r_{\mathbb{J}}$.

Let $l_{\mathbb{J}} = 256$.

Define $repr_{\mathbb{J}}: \mathbb{J} \to \mathbb{B}^{l_{\mathbb{J}}}$ such that $repr_{\mathbb{J}}(u, v) = I2LEBSP_{256}(v + 2^{255} \cdot \tilde{u})$, where $\tilde{u} = u \ mod \ 2$.

Let $abst_{\mathbb{J}}: \mathbb{B}^{l_{\mathbb{J}}} \to \mathbb{J} \cup \{\perp\}$ be the left inverse of $repr_{\mathbb{J}}$ such that if S is not in the range of $repr_{\mathbb{J}}$, then $abst_{\mathbb{J}}(S) = \perp$.

Define $\mathbb{J}^{(r)}$ as the order-$r_{\mathbb{J}}$ subgroup of $\mathbb{J}$. Note that this includes $\mathcal{O}_{\mathbb{J}}$. For the set of points of order $r_{\mathbb{J}}$ (which excludes $\mathcal{O}_{\mathbb{J}}$), we write $\mathbb{J}^{(r)*}$.

Define $\mathbb{J}^r_* := \{repr_{\mathbb{J}}(P):)\mathbb{B}^{l_{\mathbb{J}}} | P \in \mathbb{J}^{(r)}\}$.

When computing square roots in $F_{q_{\mathbb{J}}}$ in order to decompress a point encoding, the implementation must not assume that the square root exists, or that the encoding represents a point on the curve.

Let $\mathcal{U}((u, v)) = u$ and let $\mathcal{V}((u, v)) = v$.

Define $Extract_{\mathbb{J}^{(r)}}: \mathbb{J}^{(r)} \rightarrow \mathbb{B}^{l_{merkle}}$ by

$$Extract_{\mathbb{J}^{(r)}}(P) = I2LEBSP_{l_{merkle}}(\mathcal{U}(P)).$$

RedJubjub is Schnorr-based signature scheme to the Jubjub curve.

Let define $\mathcal{P}$ as the generator of $\mathbb{J}^{(r)}$.

Define $l_H = 512$

Its associated types are defined as follows:

$RedJubjub.Message = \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$

$RedJubjub.Signature = \mathbb{B}^{\mathbb{Y}[ceiling\left(\frac{l_{\mathbb{J}}}{8}\right)+ceiling(bitlength(r_{\mathbb{J}})/8)]}$

$RedJubjub.Public = \mathbb{J}$

$RedJubjub.Private = F_{r_{\mathbb{J}}}$

$RedJubjub.Random = F_{r_{\mathbb{J}}}$

Define$RedJubjub.GenPrivate: () \xrightarrow{R} RedJubjub.Private$ as:

Return $sk \xleftarrow{R} F_{r_{\mathbb{J}}}$.

Define $RedJubjub.DerivePublic: RedJubjub.Private \rightarrow RedJubjub.Public$ by:

$RedJubjub.DerivePublic(sk) = [sk]\,\mathcal{P}$.

Define $RedJubjub.GenRandom: () \xrightarrow{R} RedJubjub.Random$ as:

Choose a byte sequence T uniformly at random on $\mathbb{B}^{\mathbb{Y}[(l_H+128)/8]}$.

Return $BLAKE2b_{512}("Zcash\_RedJubjubH", T)$

Define $\mathcal{O}_{RedJubjub.random} = 0\ (mod\ r_{\mathbb{J}})$.

Define $RedJubjub.RandomizePrivate: RedJubjub.Random \times RedJubjub.Private \rightarrow RedJubjub.Private$ by:

$RedJubjub.RandomizePrivate(\alpha, sk) := sk + \alpha\ (mod\ r_{\mathbb{J}})$.

Define $RedJubjub.RandomizePublic: RedJubjub.Random \times RedJubjub.Public \rightarrow RedJubjub.Public$ as: $RedJubjub.RandomizePublic(\alpha, vk) := vk + [\alpha]\ \mathcal{P}$.

Define $RedJubjub.Sign$:
$(sk: RedJubjub.Private) \times (M: RedJubjub.Message) \xrightarrow{R} RedJubjub.Signature$
as:
Choose a byte sequence T uniformly at random on $\mathbb{B}^{\mathbb{Y}[(l_H+128)/8]}$.

Let $r = BLAKE2b_{512}$("Zcash_RedJubjubH", $T||M$).
Let $R = [r]\,\mathcal{P}$.
Let $\underline{R} = LEBS2OSP_{l_{\mathbb{J}}}(repr_{\mathbb{J}}(R))$
Let $\underline{vk} = LEBS2OSP_{l_{\mathbb{J}}}(repr_{\mathbb{J}}(RedJubjub.DerivrPublic(sk)))$
Let $S = (r + BLAKE2b_{512}$("Zcash_RedJubjubH", $\underline{R}||\underline{vk}||M) \cdot sk)\ mod\ r_{\mathbb{J}}$
Let $\underline{S} = LEBS2OSP_{bitlength(r_{\mathbb{J}})}(I2LEBSP_{bitlength(r_{\mathbb{J}})}(S))$
Return $\underline{R}||\underline{S}$

Define $RedJubjub.Verify$:
$(vk: RedJubjub.Public) \times (M: RedJubjub.Message) \times (\sigma: RedJubjub.Signature) \rightarrow \mathbb{B}$
as:
Let $\underline{R}$ be the first ceiling $l_{\mathbb{J}}/8$ bytes of σ, and let $\underline{S}$ be the remaining ceiling $ceiling(bitlength(r_{\mathbb{J}})/8)$ bytes.
Let $R = abst_{\mathbb{J}}(LEOS2BSP_{l_{\mathbb{J}}}(\underline{R}),$ and $S = LEOS2IP_{bitlength(r_{\mathbb{J}})}(\underline{S})$
Let $\underline{vk} = LEBS2OSP_{l_{\mathbb{J}}}(repr_{\mathbb{J}}(vk))$
Let $c = BLAKE2b_{512}$("Zcash_RedJubjubH", $\underline{R}||\underline{vk}||M$)
Return 1 if $R \neq \perp$ and $S < r_{\mathbb{J}}$ and $[h_{\mathbb{J}}](-[S]\mathcal{P} + R + [c]vk) = \mathcal{O}_{\mathbb{J}}$, otherwise 0

3.7.1. Spend Authorization Signature

SpendAuthSig is instantiated as RedJubjub with key re-randomization.

The generator is $\mathcal{P} = FindGrouphash^{\mathbb{J}^{(r)*}}$("Zcash_G", "").

3.7.2. Binding Signature

BindingSig is instantiated as RedJubjub, without use of key re-randomization.

The generator is $\mathcal{P} = FindGrouphash^{\mathbb{J}^{(r)*}}$("Zcash_cv", "$r$").

3.8. Group Hash into Jubjub

Let $GroupHash.Input = \mathbb{B}^{\mathbb{Y}[8]} \times \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$, and let $GroupHash.URSType = \mathbb{B}^{\mathbb{Y}[64]}$
(The input element with type $\mathbb{B}^{\mathbb{Y}[8]}$ is intended to act as a "personalization" parameter to distinguish uses of the group hash for different purposes.)

Let URS be the MPC randomness beacon,
URS=" 096b36a5804bfacef1691e173c366a47ff5ba84a44f26ddd7e8d9f79d5b42df0 "
Let $D: \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ be a 8-byte domain separator, and let $M: \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ be the hash input.
The hash $GroupHash^{\mathbb{J}^{(r)*}}_{URS}(D, M): \mathbb{J}^{(r)*}$ *is* calculated as follows:
let $\underline{H} = BLAKE2s_{256}(D, URS\ ||M)$
let $P = abst_{\mathbb{J}}(LEOS2BSP_{256}(\underline{H})$
if $P = \perp$, then return⊥
let $Q = [h_{\mathbb{J}}]\,P$
if $Q = \mathcal{O}_{\mathbb{J}}$ then return ⊥, else return Q.

Define $first: (\mathbb{B}^{\mathbb{Y}} \rightarrow T \cup \{\perp\}) \rightarrow T \cup \{\perp\}$ so that $first(f) = f(i)$ where I is the least integer in $\mathbb{B}^{\mathbb{Y}}$ such that $f(i) \neq \perp$, or $\perp$ if no such i exists.
Define

$$FindGrouphash^{\mathbb{J}^{(r)*}} = first(i: \mathbb{B}^{\mathbb{Y}} \rightarrow GroupHash^{\mathbb{J}^{(r)*}}_{URS}(D, M\,||[i]): \mathbb{J}^{(r)*} \cup \{\perp\}).$$

3.9. Commitment Schemes

3.9.1. Note Commitments

Let define WindowedPedersenCommit as follows:

$$\begin{aligned} &WindowedPedersenCommit_r(s) \\ &\quad = PedersenHashToPoint(\text{“}Zcash\_PH\text{”}, s) \\ &\quad + [r]\, FindGrouphash^{\mathbb{J}^{(r)*}}(\text{"}Zcash\_PH\text{”}, \text{“}r\text{"}) \end{aligned}$$

Using WindowedPedersenCommit, the commitment scheme NoteCommit is instantiated as follows:

$$\begin{aligned} &NoteCommit_{rcm}(g_d\star, pk_d\star, v): \\ &\quad = WindowedPedersenCommit_{rcm}([1]^6 \,||\, I2LEBSP_{64}(v) \,||\, g_d\star \,||\, pk_d\star) \end{aligned}$$

$NoteCommit.GenTrapdoor()$ generates the uniform distribution on $F_{r_{\mathbb{J}}}$.

3.9.2. Value Commitments

In order to support homomorphic property, we define “homomorphic” Pedersen commitments as follows:

$$\begin{aligned} &HomomorphicPedersenCommit_{rcv}(D, v) \\ &\quad = [v]\, FindGrouphash^{\mathbb{J}^{(r)*}}(D, \text{“}\boldsymbol{v}\text{"}) + [rcv]\, FindGrouphash^{\mathbb{J}^{(r)*}}(D, \text{“}\boldsymbol{r}\text{"}) \end{aligned}$$

$ValueCommit.GenTrapdoor()$ generates the uniform distribution on $F_{r_{\mathbb{J}}}$.

Define:

$\mathcal{V} = FindGrouphash^{\mathbb{J}^{(r)*}}(\text{“}Zcash\_cv\text{”}, \text{“}v\text{”})$

$\mathcal{R} = FindGrouphash^{\mathbb{J}^{(r)*}}(\text{“}Zcash\_cv\text{”}, \text{“}r\text{”})$

Value commitment scheme is instantiated as follows using HomomorphicPedersenCommit:

$$ValueCommit_{rcv}(v) = HomomorphicPedersenCommit_{rcv}(\text{“}Zcash\_cv\text{”}, v).$$

Which is equivalent to:

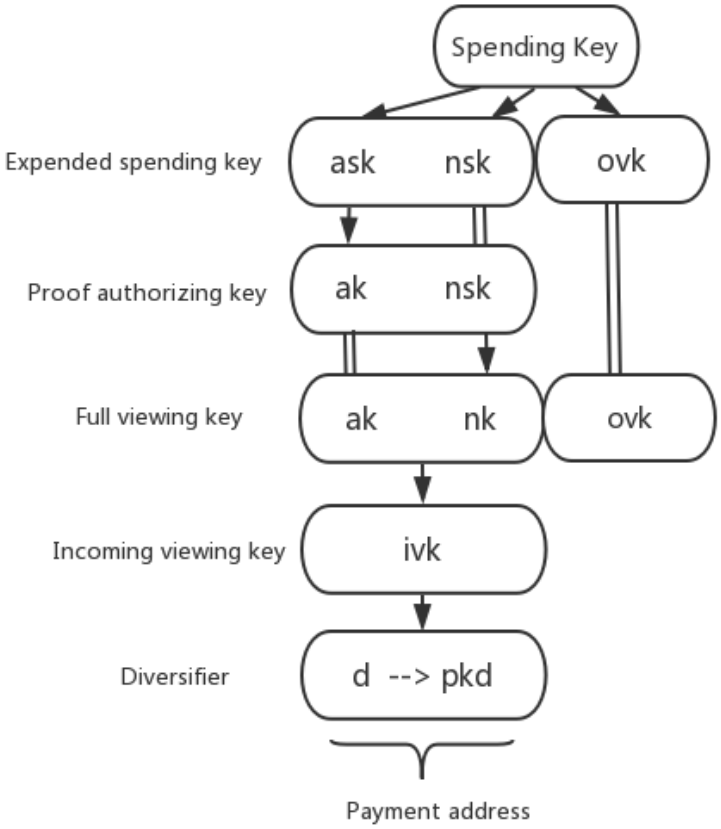
$$ValueCommit_{rcv}(v) = [v]\mathcal{V} + [rcv]\mathcal{R}$$

4. Concepts

4.1. Payment Addresses and Keys

Users who wish to receive payments under this scheme first generate a random spending key **sk**.

The following diagram depicts the relations between key components. Arrows point from a component to any other component(s) that can be derived from it. Double lines indicate that the same component is used in multiple abstractions.



For each spending key, there is also a default diversified payment address with a "random-looking" diversifier. This allows an implementation that does not expose diversified addresses as a user-visible feature, to use a default address that cannot be distinguished (without knowledge of the spending key) from one with a random diversifier as above.

Define:

$$CheckDiversifier(d) = \begin{cases} \bot, \text{ if DiversifyHash(d)} = \bot \\ d, \qquad\qquad otherwise \end{cases}$$

$$DefaultDiversifier(sk) = first(i: \mathbb{B}^Y \to CheckDicersifier\left(truncate_{\frac{\ell_d}{8}}\left(PRF_{sk}^{expand}(3,i)\right)\right): \mathbb{J}^{(r)*} \cup \{\bot\})$$

For a random spending key, if $DefaultDiversifier$ returns ⊥, then discard the key and repeat with a different sk.

The composition of shielded payment addresses, incoming viewing keys, full viewing keys, and spending keys is a cryptographic protocol detail that should not normally be exposed to users. However, user-visible operations should be provided to obtain a shielded payment address or incoming viewing key or full viewing key from a spending key.

Users can accept payment from multiple parties with a single shielded payment address and the fact that these payments are destined to the same payee is not revealed on the block chain, even to the paying parties. However, if two parties collude to compare a shielded payment address they can trivially determine they are the same. In the case that a payee wishes to prevent this they should create a distinct shielded payment address for each payer.

4.2. Notes

A note represents that a value v is spendable by the recipient who holds the spending key corresponding to a given shielded payment address.

A note is a tuple(d, pk_d, v, rcm), where:

- d: $\mathbb{B}^{l_d}$ is the diversifier of the recipient's shielded payment address;
- pk_d is the diversified transmission key of the recipient's shielded payment address ;
- v is an integer representing the value of the note;
- rcm is a random commitment trapdoor which is a random number, indeed.

Where notes are created and send, only a commitment to the above values is disclosed publically, and added to a data structure called note commitment tree. This allows the value and recipient be kept private, while the commitment is used by the zero-knowledge proof when the note is spent, to check that it exists on the block chain.

Let $DiversifyHash$ be as defined in § 3.3.3.

A note commitment on a note(d, pk_d, v, rcm) is computed as:

$$g_d = DiversifyHash(d)$$

$$NoteCommitment(n) = \begin{cases} \bot \\ NoteCommitment_{rcm}\left(repr_{\mathbb{J}}(g_d), repr_{\mathbb{J}}(pk_d), v\right), otherwise \end{cases}$$

$NoteCommitment_{rcm}$ is instantiated in §3.8.

A nullifier (denoted nf) is derived from nullifier deriving key nk. This computation uses a Pseudo Random Function.

A note is spent by proving knowledge of (ρ, ak, nsk) in zero knowledge while publically disclosing its nullifier nf, allowing nf to be used to prevent double-spending. A spend authorization signature is also required, in order to demonstrate knowledge of ask.

4.3. Transactions and Treestates

To each transaction there are associated initial treestates. Each treestate consists of a note commitment tree and a nullifier set.

An anchor is a Merkle tree root of a note commitment tree. It uniquely identifies a note commitment tree state given the assumed security properties of the Merkle tree's hash function. Since the nullifier set is always updated together with the note commitment tree, this also identifies a particular state of the associated nullifier set.

In a given block chain, treestates are chained as follows:

- The input treestate of the first block is the empty treestate.
- The input treestate of the first transaction of a block is the final treestate of the immediately preceding block.
- The input treestate of each subsequent transaction in a block is the output treestate of the immediately preceding transaction.
- The final treestate of a block is the output treestate of its last transaction.

4.4. Spend Descriptions and Receive Descriptions

Spend descriptions and Receive descriptions are data included in a transaction that describe Spend transfers and Receive transfers, respectively.

A Spend transfer spends a note $\boldsymbol{n}^{old}$. Its Spend description includes a Pedersen value commitment to the value of the note. It is associated with an instance of a Spend proof for which it provides a zk-SNARK proof.

An Receive transfer creates a note $\boldsymbol{n}^{new}$. Its Receive description includes a Pedersen value commitment to the note value. It is associated with an instance of an Output proof (§6.2.5) for which it provides a zk-SNARK proof.

Each transaction has a sequence of Spend descriptions and a sequence of Receive descriptions.

To ensure balance, we use a homomorphic property of Pedersen commitments that allows them to be added and subtracted, as elliptic curve points. The result of adding two Pedersen value commitments, committing to values v_1 and v_2, is a new Pedersen value commitment that commits to $v_1 + v_2$. Subtraction works similarly.

Therefore, balance can be enforced by adding all of the value commitments for shielded inputs, subtracting all of the value commitments for shielded outputs, and proving by use of a binding signature that the result commits to a value consistent with the net transparent value change. This approach allows all of the zk-SNARK statements to be independent of each other, potentially increasing opportunities for precomputation.

A Spend description includes an anchor, which refers to the output treestate of a previous block. It also reveals a nullifier, which allows detection of double-spends.

4.5. Nullifier Sets

Each full validator maintains a nullifier set logically associated with each treestate. As valid transactions are processed, the nullifiers revealed in Spend descriptions are inserted into the nullifier set associated with the new treestate. Nullifiers are enforced to be unique within a valid block chain, in order to prevent double-spends.

5. zk-SNARK

Zero-knowledge proving system is a cryptographic protocol that allows proving a particular statement, dependent on primary and auxiliary inputs, in zero knowledge — that is, without revealing information about the auxiliary inputs other than that implied by the statement.

5.1. Zero-Knowledge Proof Model

We use zk-SNARK with the proving system describe in [Groth2016]. These are used for proofs in spend descriptions and output descriptions.

A preprocessing zk-SNARK instance ZK defines:

• a type of zero-knowledge proving keys, $ZK.ProvingKey$;
• a type of zero-knowledge verifying keys, $ZK.VerifyingKey$;
• a type of primary inputs $ZK.PrimaryInput$;
• a type of auxiliary inputs $ZK.AuxiliaryInput$;
• a type of proofs $ZK.Proof$;
• a type $ZK.SatisfyingInputs \subseteq ZK.PrimaryInput \times ZK.AuxiliaryInput$ of inputs satisfying the statement;
• a randomized key pair generation algorithm $ZK.Gen : () \xrightarrow{R} ZK.ProvingKey \times ZK.VerifyingKey$;
• a proving algorithm $ZK.Prove: ZK.ProvingKey \times ZK.SatisfyingInputs \rightarrow ZK.Proof$;
• a verifying algorithm $ZK.Verify: ZK.VerifyingKey \times ZK.PrimaryInput \times ZK.Proof \rightarrow \mathbb{B}$;

A proof consists of $(\pi_A: \mathbb{S}_1^{(r)*}, \pi_B: \mathbb{S}_2^{(r)*}, \pi_C: \mathbb{S}_1^{(r)*})$. It is computed as described using the pairing parameters in curve *BLS12-381*[Bowe2017].

A proof is encoded by concatenating the encodings of its elements; for the BLS12-381 pairing this is:

$$384bit\ \pi_A\ ||768bit\ \pi_B||384bit\ \pi_C$$

The resulting proof size is 192 bytes.

Zk-SNARK protocol is detailed as follows.

5.2. Construct zk-SNARK

This section will introduce how to construct zk-SNARK in shielded transaction. we will give a brief overview of how the rules for determining a valid transaction get transformed into equations that can then be evaluated on a candidate solution without revealing any sensitive information to the parties verifying the equations.

The main step is:

$$Computation \rightarrow Arithmetic\ Circuit \rightarrow R1CS \rightarrow QAP \rightarrow zk-SANRK$$

5.2.1. Generate Arithmetic Circuit

The first step, we convert the original code, which may contain arbitrarily complex statements and expressions, into a sequence of statements that are of two forms: $x = y$ (where y can be a variable or a number)and $x = y\ (op)\ z$ (where op can be $+, -, *, /$ and y and z can be variables, numbers or themselves sub-expressions). Each of these statements is kind of like logic gates in a circuit.

To take an example, prove that we know the solution to equation:$x^3 + x + 5 = 35$ The result of the process for the above equation is as follows:

$$sym\_1 = x * x$$
$$y = sym\_1 * x$$
$$sym\_2 = y + x$$
$$\sim out = sym\_2 + 5$$

5.2.2. R1CS

Now, we convert this into something called a rank-1 constraint system (R1CS). An R1CS is a sequence of groups of three vectors (a, b, c), and the solution to an R1CS is a vector s, where s must satisfy the equation $s.a * s.b - s.c = 0$, where . represents the dot product in simpler terms, if we "zip together" a and s, multiplying the two values in the same positions, and then take the sum of these products, then do the same to b and s and then c and s, then the third result equals the product of the first two results. For example, this is a satisfied R1CS:

$$s = (1,3,35,9,27,30)$$

$$a = (5,0,0,0,0,1)$$

$$b = (1,0,0,0,0,0)$$

$$c = (0,0,1,0,0,0)$$

Instead of having just one constraint, we are going to have many constraints: one for each logic gate. There is a standard way of converting a logic gate into a (a, b, c) triple depending on what the operation is $(+, -, *\ or\ /)$ and whether the arguments are variables or numbers. The length of each vector is equal to the total number of variables in the system, including a dummy variable $\sim one$ at the first index representing the number 1, the input variables, a dummy variable $\sim out$ representing the output, and then all of the intermediate variables ($sym1$ and $sym2$ above); the vectors are generally going to be very sparse, only filling in the slots corresponding to the variables that are affected by some particular logic gate.

First, we'll provide the variable mapping that we'll use:

$$'\sim one', 'x', '\sim out', 'sym\_1', 'y', 'sym\_2'$$

The solution vector will consist of assignments for all of these variables, in that order.

Now, we'll give the (a, b, c) triple for the first gate:

$$\begin{aligned} a &= [0, 1, 0, 0, 0, 0] \\ b &= [0, 1, 0, 0, 0, 0] \\ c &= [0, 0, 0, 1, 0, 0] \end{aligned}$$

You can see that if the solution vector contains 3 in the second position, and 9 in the fourth position, then regardless of the rest of the contents of the solution vector, the dot product check will boil down to $3 * 3 = 9$, and so it will pass. If the solution vector has -3 in the second position and 9 in the fourth position, the check will also pass; in fact, if the solution vector has 7 in the second position and 49 in the fourth position then that check will still pass—the purpose of this first check is to verify the consistency of the inputs and outputs of the first gate only.

Now, let's go on to the second gate:

$$\begin{aligned} a &= [0, 0, 0, 1, 0, 0] \\ b &= [0, 1, 0, 0, 0, 0] \\ c &= [0, 0, 0, 0, 1, 0] \end{aligned}$$

In a similar style to the first dot product check, here we're checking that $sym\_1 * x = y$.

Now, the third gate:

$$\begin{aligned} a &= [0, 1, 0, 0, 1, 0] \\ b &= [1, 0, 0, 0, 0, 0] \\ c &= [0, 0, 0, 0, 0, 1] \end{aligned}$$

Here, the pattern is somewhat different: it's multiplying the first element in the solution vector by the second element, then by the fifth element, adding the two results, and checking if the sum equals the sixth element. Because the first element in the solution vector is always one, this is just an addition check, checking that the output equals the sum of the two inputs.

Finally, the fourth gate:

$$\begin{aligned} a &= [5, 0, 0, 0, 0, 1] \\ b &= [1, 0, 0, 0, 0, 0] \\ c &= [0, 0, 1, 0, 0, 0] \end{aligned}$$

Here, we're evaluating the last check, $\sim out = sym\_2 + 5$. The dot product check works by taking the sixth element in the solution vector, adding five times the first element (reminder: the first element is 1, so this effectively means adding 5), and checking it against the third element, which is where we store the output variable.

And there we have our R1CS with four constraints. The witness is simply the assignment to all the variables, including input, output and internal variables:

$$[1, 3, 35, 9, 27, 30]$$

We can simply compute this by "executing" the code above, starting off with the input variable assignment x=3, and putting in the values of all the intermediate variables and the output as you compute them.

The complete R1CS put together is:

$$A$$

$$\begin{matrix}[0, 1, 0, 0, 0, 0] \\ [0, 0, 0, 1, 0, 0] \\ [0, 1, 0, 0, 1, 0] \\ [5, 0, 0, 0, 0, 1]\end{matrix}$$

$$B$$

$$\begin{matrix}[0, 1, 0, 0, 0, 0] \\ [0, 1, 0, 0, 0, 0] \\ [1, 0, 0, 0, 0, 0] \\ [1, 0, 0, 0, 0, 0]\end{matrix}$$

$$C$$

$$\begin{matrix}[0, 0, 0, 1, 0, 0] \\ [0, 0, 0, 0, 1, 0] \\ [0, 0, 0, 0, 0, 1] \\ [0, 0, 1, 0, 0, 0]\end{matrix}$$

5.2.3. QAP

The next step is taking this R1CS and converting it into QAP form, which implements the exact same logic except using polynomials instead of dot products. We do this as follows. We go from four groups of three vectors of length six to six groups of three degree-3 polynomials, where evaluating the polynomials at each *x coordinate* represents one of the constraints. That is, if we evaluate the polynomials at $x = 1$, then we get our first set of vectors, if we evaluate the polynomials at $x = 2$, then we get our second set of vectors, and so on.

We can make this transformation using something called a *Lagrange interpolation*. The problem that a Lagrange interpolation solves is this: if you have a set of points (ie. (x, y) coordinate pairs), then doing a Lagrange interpolation on those points gives you a polynomial that passes through all of those points. We do this by decomposing the problem: for each x coordinate, we create a polynomial that has the desired y coordinate at that x coordinate and a y coordinate of 0 at all the other x coordinates we are interested in, and then to get the final result we add all of the polynomials together.

Now, let's use Lagrange interpolation to transform our R1CS. What we are going to do is take the first value out of every a vector, use Lagrange interpolation to make a polynomial out of that (where evaluating the polynomial at i gets you the first value of the i^{th} a vector), repeat the process for the first value of every b and c vector, and then repeat that process for the second values, the third, values, and so on.

The reason of above transformation is that instead of checking the constraints in the R1CS individually, we can now check all of the constraints at the same time by doing the dot product check on the polynomials.

$$A(x) = s.a = (1,3,35,9,27,30).(A_1(x), A_2(x), A_3(x), A_4(x), A_5(x), A_6(x))$$

$$B(x) = s.b = (1,3,35,9,27,30).(B_1(x), B_2(x), B_3(x), B_4(x), B_5(x), B_6(x))$$

$$C(x) = s.c = (1,3,35,9,27,30).(C_1(x), C_2(x), C_3(x), C_4(x), C_5(x), C_6(x))$$

$$A(x) * B(x) - C(x) = H * Z(x)$$

Because in this case the dot product check is a series of additions and multiplications of polynomials, the result is itself going to be a polynomial. If the resulting polynomial, evaluated at every x coordinate that we used above to represent a logic gate, is equal to zero, then that means that all of the checks pass; if the resulting polynomial evaluated at least one of the $x\ coordinate$ representing a logic gate gives a nonzero value, then that means that the values going into and out of that logic gate are inconsistent (i.e. the gate is $y = x * sym\_1$ but the provided values might be $x = 2, sym_1 = 2, y = 5$).

Note that the resulting polynomial does not itself have to be zero, and in fact in most cases won't be; it could have any behavior at the points that don't correspond to any logic gates, as long as the result is zero at all the points that *do* correspond to some gate. To check correctness, we don't actually evaluate the polynomial $t = A.s * B.s - C.s$ at every point corresponding to a gate; instead, we divide t by another polynomial, Z, and check that Z evenly divides t - that is, the division t / Z leaves no remainder.

Z is defined as $(x - 1) * (x - 2) * (x - 3)$ … - the simplest polynomial that is equal to zero at all points that correspond to logic gates. It is an elementary fact of algebra that *any* polynomial that is equal to zero at all of these points has to be a multiple of this minimal polynomial, and if a polynomial is a multiple of Z then its evaluation at any of those points will be zero; this equivalence makes our job much easier.

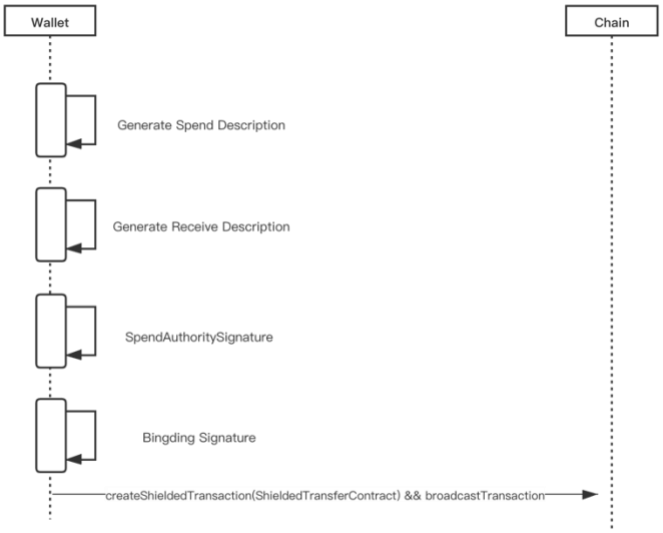
In order to accelerate polynomial A(x), B(x) and C(x) with FFT, we often define $Z = (x - w^0)(x - w^1)(x - w^3)$ …, where w is the n^{th} roots of unity, n is the smallest power of 2 that is bigger than number of constraints.

5.2.4. zk-SNARK

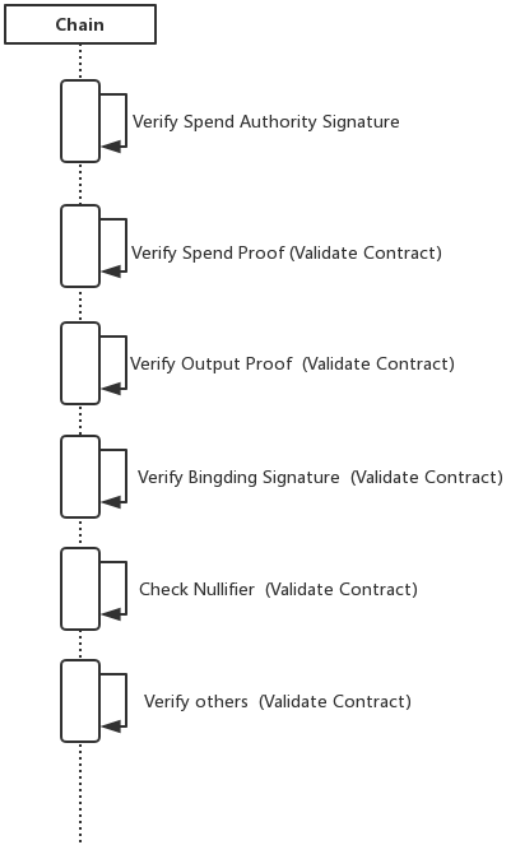
Based on QAP, we construct a Non-Interactive Zero Knowledge argument for arithmetic satisfiability where a proof consists of 3 group elements. The proof is easy to verify. The verifier just needs to compute a number of exponentiations proportional to the statement size and check a single pairing product equation, which only has 3 pairings. If more details are needed, please refer to [Groth2016].

6. Shielded Transaction

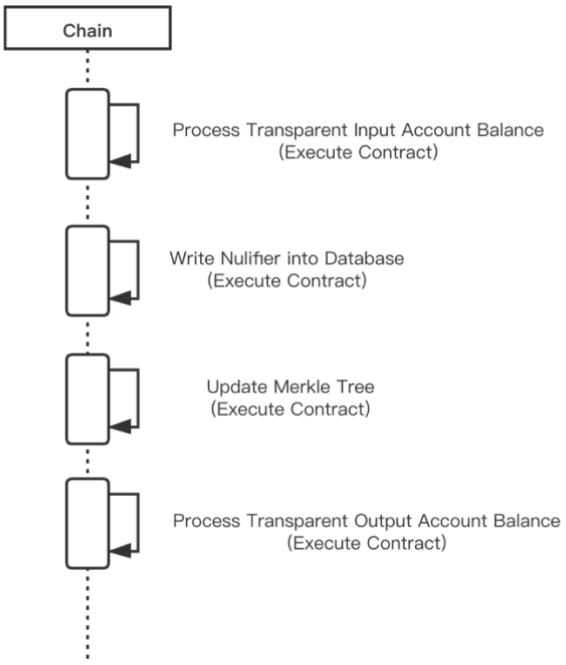
Flow of building a transaction in a wallet is as following.



Flow of verifying a transaction on the chain is as following.



Flow of Executing a transaction on the chain is as following.



6.1. Trusted Setup

The target of Trusted Setup phase is to generate common reference string(CRS) for partial zk-SNARK parameters.

We construct CRS based on multi-party computation based on MPC ceremony of *Zcash*. The security of trusted setup relies on that one of the participants is honest, that is, discard the toxic waste after generating corresponding parameters.

6.2. Wallet

6.2.1. Create Payment Address

Let PRF^{expand} and PRF^{ock} be Pseudo Random Functions instantiated in § 3.3.
Let CRH^{ivk} be a hash function, instantiated in § 3.3.2.
Let $DiversifyHash$ be as defined in § 3.3.3.
Let $SpendAuthSig$, instantiated in §3.7.1, be a signature scheme with re-randomizable keys.
Let $repr_{\mathbb{J}}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and$\mathbb{J}_{*}^{(r)}$ be as defined in §3.7, and let $FindGroupHash^{\mathbb{J}^{(r)*}}$ be as defined in §3.3
Define $\mathcal{H} = FindGroupHash^{\mathbb{J}^{(r)*}}$(Zcash_H, "")
Define $toScalar\left(x: \mathbb{B}^{Y}\left[\frac{l^{PRFexpend}}{8}\right]\right) = LEOS2IP_{l_{PRFexpend}}(x)\ mod\ r_{\mathbb{J}}$
A new spending key sk is generated by choosing a bit sequence uniformly at random from$\mathbb{B}^{l_{sk}}$. From this spending key, the spend authorizing key ask, the proof authorizing key nsk, and the outgoing key ovk are devived as following.

$$ask = ToScalar\ (PRF_{sk}^{expand}(0))$$
$$nsk = ToScalar\ (PRF_{sk}^{expand}(1))$$
$$ovk = ToScalar\ (PRF_{sk}^{expand}(2))$$

If $ask\ =\ 0$, discard this key and repeat with a new sk.

ak, nk and the incoming viewing key ivk are then derived as:

$$ak = SpendAuthSig.DerivePublic(ask)$$

$$nk = [nsk]\mathcal{H}$$

$$ivk = CRH^{ivk}(repr_{\mathbb{J}}(ak), repr_{\mathbb{J}}(nk))$$

If $ivk = 0$, discard this key and repeat with a new key.

Multiple diversified payment addresses with the same spending authority could be created efficiently. A group of such addresses shares the same full viewing key and incoming viewing key.

To create a new diversified payment address given an incoming viewing key ivk, repeatedly pick a diversifier d uniformly at random from $\mathbb{B}^{l_d}$ until $g_d = DiversifyHash(d)$ is not $\perp$. Then calculate:

$$pk_d = KA.DerivePublic(ivk, g_d)$$

(d, pk_d) is the resulting diversified payment address.

In addition, for each spending key, there is also a default diversified payment address with a "random-looking" diversifier. This allows an implementation that does not expose diversified addresses as a user-visible feature, to use a default address that cannot be distinguished (without knowledge of the spending key) from one with a random diversifier as above.

Define:

$$CheckDiversifier(d) = \begin{cases} \perp, \text{ if DiversifyHash(d)} = \perp \\ d, \qquad\qquad otherwise \end{cases}$$

$$DefaultDiversifier(sk) = first(i: \mathbb{B}^Y \to CheckDicersifier\left(truncate_{\frac{\ell_d}{8}}\left(PRF_{sk}^{expand}(3, i)\right)\right): \mathbb{J}^{(r)*} \cup \{\perp\})$$

For a random spending key, if $DefaultDiversifier$ returns $\perp$, then discard the key and repeat with a different sk.

6.2.2. Scan Blockchain

Block chain scanning requires only the nk and ivk key components.

Giving the block chain, and (nk, ivk), the following algorithm can be used to obtain each note send to the corresponding shield payment address, its memo field, and its final status.

Initialize $ReceivedSet: \mathcal{P}\left(Note \times \mathbb{B}^{\mathbb{Y}[512]}\right) = \{\}$

Initialize $SpendSet: \mathcal{P}(Note) = \{\}$

Initialize $NullifierMap: \mathbb{B}^{l_{PRFnf}} \to Note$ to the empty mapping.

For each transaction tx,
    For each output description in tx with note position pos,
        Attempt to decrypt the transmitted note ciphertext components epk and C^{enc} using ivk. If this succeeds giving **np**:
        Extract **n** and $memo: \mathbb{B}^{\mathbb{Y}[512]}$ from **np**.
        Add (**n**, memo) to ReceivedSet.
        Calculate the nullifier nf of n using nk and pos.
        Add the mapping nf → n to NullifierMap.
    For each Spend description in tx,
        Let nf be the nullifier of the Spend description.
        If nf is present in NullifierMap, add NullifierMap(nf) to SpentSet.
Return (ReceivedSet, SpentSet).

6.2.3. Create Spend Proof

Zero-Knowledge proof protocol zk-SNARK is used in shielded transaction.
A valid instance of $\pi_{ZKspend}$ assures that given a primary input:

$$(rt: \mathbb{B}^{l_{Merkle}},$$
$$cv^{old}: ValueCommit.output,$$
$$nf^{old}: \mathbb{B}^{l_{PRFnf}},$$
$$rk: SpendAuthSig.Public\ )$$

The prover knows an auxiliary input:

$$(voucherPath: \mathbb{B}^{[l_{Merkle}][l_{MerkleDepth}]},$$
$$pos: \{0,1,\dots,2^{MerkleDepth}-1\},$$
$$g_d: \mathbb{J},$$
$$pk_d: \mathbb{J},$$
$$v^{old}: \{0,\dots 2^{l_{value}}-1\},$$
$$rcv^{old}: \{0,\dots 2^{l_{scalar}}-1\},$$
$$cm^{old}: \mathbb{J},$$
$$rcm^{old}: \{0,\dots 2^{l_{scalar}}-1\},$$
$$\alpha: \{0,\dots 2^{l_{scalar}}-1\},$$
$$ak: SpendAuthSig,$$
$$nsk: \{0,\dots 2^{l_{scalar}}-1\})$$

Such that the following conditions hold:
Note commitment integrity: $cm^{old} = NoteCommit_{rcm^{old}}(repr_{\mathbb{J}}(g_d), repr_{\mathbb{J}}(pk_d), v^{old})$.

Merkle path validity: Either $v^{old} = 0$; or (voucherpath, pos) is a valid Merkle path of depth MerkleDepth, from $cm_u = Extract_{\mathbb{J}^{(r)}}(cm^{old})$ to the anchor rt.

Value commitment integrity:

$$cv^{old} = NoteCommit_{rcv^{old}}(v^{old})$$

Small order checks: g_d and ak are not of small order. i.e. $[h_{\mathbb{J}}]\ g_d \neq \mathcal{O}_{\mathbb{J}}$, $[h_{\mathbb{J}}]\ ak \neq \mathcal{O}_{\mathbb{J}}$

Nullifier integrity $nf^{old} = PRF^{nf}_{nk\star}(\rho\star)$, where

$$nk\star = repr_{\mathbb{J}}([nsk]\mathcal{H})$$

$$\rho\star = repr_{\mathbb{J}}(MixingPedersenHash(cm^{old}, pos))$$

Spend authority $rk = spendAuthSig.RandomizePubnlic(\alpha, ak)$

Diversified address integrity $pk_d = [ivk]\ g_d$ where

$$ivk = CRH^{ivk}(ak\star, nk\star)$$

$$ak\star = repr_{\mathbb{J}}(ak).$$

Given $ak, nsk, d, rcm, \alpha, value, rt\ and\ voucherPath$, we generate the proof. simultaneously, we generate $cv\ and\ rk$ as primary input, used to verify the proof.

6.2.4. Signature with Re-randomizable Keys

$SpendAuthSig$ is used to prove knowledge of the spending key authorizing spending of an input note.

Knowledge of the spending key could have been proven directly in the Spend proof.

The verifying key of the signature must be revealed in the Spend description so that the signature can be checked by validators. To ensure that the verifying key cannot be linked to the shielded payment address or spending key from which the note was spent, we use a signature scheme with re-randomizable keys. The Spend statement proves that this verifying key is a re-randomization of the spend authorization address key ak with a randomizer known to the signer.

The spend authorization signature is over the transaction hash, so that it cannot be replayed in other transactions.

The hash algorithm that we use is SHA256, that is,

$$signature.message = Sha256Hash(SHA256Hash(tokenId)||transactionRawData).$$

For each Spend description, the signer uses a fresh spend authorization randomizer α:

1. Choose $\alpha \xleftarrow{R} SpendAuthSig.GenRandom()$.
2. Let $rsk = SpendAuthSig.RandomizePrivate(\alpha, ask)$
3. Let $rk = SpendAuthSig.derivePublic(rsk)$
4. Generate a proof $\pi_{ZKSpend}$ of the spend statement with α in the auxiliary input and rk in the primary input.
5. Let $spendAuthSig = SpendAuthSig.Sign_{rsk}(signature.message)$

The resulting $spendAuthSig$ and $\pi_{ZKSpend}$ are included in the spend description.

$SpendAuthSig$ is instantiation of a kind of signature with re-randomizable keys Sig, which defines:

• a type of randomizers $Sig.Random$;
• a randomizer generator $Sig.GenRandom: () \xleftarrow{R} Sig.Random$;
• a private key randomization algorithm $Sig.RandomizePrivate: Sig.Random \times Sig.Private \rightarrow Sig.Private$;
• a public key randomization algorithm $Sig.RandomizePublic: Sig.Random \times Sig.Public \rightarrow Sig.Public$;
• a distinguished "identity" randomizer $\mathcal{O}_{Sig.Random}: Sig.Random$
Such that

• for any $\alpha: Sig.Random, Sig.RandomizePrivate_{\alpha}: Sig.Private \rightarrow Sig.Private$ is injective and easily invertible;

• $Sig.RandomizePrivate_{\mathcal{O}_{Sig.Random}}$ is the identity function on $Sig.Private$.

• for any $sk: Sig.Private, Sig.RandomizePrivate(\alpha, sk) : \alpha \xleftarrow{R} Sig.GenRandom()$

is identically distributed to $Sig.GenPrivate()$.

• for any $sk: Sig.Private$ and $\alpha: Sig.Random$,

$$Sig.RandomizePublic(\alpha, Sig.DerivePublic(sk)) = Sig.DerivePublic(Sig.RandomizePrivate(\alpha, sk)).$$

6.2.5. Create Output Proof

A valid instance of $\pi_{ZKOutput}$ assures that given a primary input :

$$(cv^{new}: ValueCommit.output, cm_u: \mathbb{B}^{l_{merkle}}, epk: \mathbb{J})$$

The prover knows an auxiliary input:

$$(g_d: \mathbb{J},$$
$$pk\star_d: \mathbb{B}^{l_{\mathbb{J}}},$$
$$v^{new}: \{0, \dots 2^{l_{value}} - 1\},$$
$$rcv^{new}: \{0, \dots 2^{l_{scalar}} - 1\},$$
$$cm^{old}: \mathbb{J},$$
$$rcm^{rcm}: \{0, \dots 2^{l_{scalar}} - 1\},$$
$$esk: \{0, \dots 2^{l_{scalar}} - 1\})$$

Such that the following conditions hold:
Note commitment integrity: $cm_u = Extract_{\mathbb{J}^{(r)}}(NoteCommit_{rcm^{new}}(g\star_d, pk\star_d,\ v^{new}))$, where $g\star_d = repr_{\mathbb{J}}(g_d)$.

Value commitment integrity: $cv^{new} = NoteCommit_{rcv^{new}}(v^{new})$

Small order checks: g_d is not of small order. i.e. $[h_{\mathbb{J}}]\ g_d \neq \mathcal{O}_{\mathbb{J}}$

Ephemeral public key integrity $epk = [esk]\ g_d$.

Given $esk,\ d, rcm, pk_d, value$, we generate the proof. simultaneously, we generate cv as primary input, used to verify the proof.

6.2.6. Binding Signature

Spend transfers and Output transfers are used in each transaction. The net value of Spend transfers minus Output transfers in a transaction is called the balancing value, measured in TRX as a signed integer $v^{balance}$.

$v^{balance}$ is encoded explicitly in a transaction as the field valueBalance.

A positive balancing value takes value from the shielded value pool and adds it to the transparent value pool. A negative balancing value does the reverse. As a result, positive $v^{balance}$ is treated like an *input* to the transparent value pool, whereas negative $v^{balance}$ is treated like an *output* from that pool.

Consistency of $v^{balance}$ with the value commitments in Spend descriptions and Output descriptions is enforced by the binding signature. This signature has a dual role in the protocol:

- To prove that the total value spent by Spend transfers, minus that produced by Output transfers, is consistent with the $v^{balance}$ field of the transaction;
- To prove that the signer knew the randomness used for the spend and output value commitments, in order to prevent Output descriptions from being replayed by an adversary in a different transaction. (A Spend description already cannot be replayed due to its spend authorization signature.)

Instead of generating a key pair at random, we generate it as a function of the value commitments in the Spend descriptions and Output descriptions of the transaction, and the balancing value.

Let $repr_{\mathbb{J}}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $\mathbb{J}_*^{(r)}$ be as defined in §3.7.

Let $ValueCommit, \mathcal{V}\ and\ \mathcal{R}$ be as defined in §3.8.

$$ValueCommit: ValueCommit.Trapdoor \times \{-\frac{r_{\mathbb{J}} - 1}{2} \dots -\frac{r_{\mathbb{J}} + 1}{2}\} \to ValueCommit.Output;$$

$\mathcal{V}$: $\mathbb{J}^{(r)*}$is the value base in $ValueCommit$.

$\mathcal{R}$: $\mathbb{J}^{(r)*}$is the randomness base in $ValueCommit$.

Suppose that the transaction has:
• n_1 Spend descriptions with value commitments $cv_{1...n_1}^{old_1}$, committing to values $v_{1...n_1}^{old_1}$ with randomness $rcv_{1...n_1}^{old_1}$;

• n_2 transparent inputs, value $v_{1...n_2}^{old_2}$;

• m_1 Output descriptions with value commitments $cv_{1...m_1}^{new_1}$, committing to values $v_{1...m_1}^{new_1}$ with randomness $rcv_{1...m_1}^{new_1}$;

• m_2 transparent outputs, value $v_{1...m_2}^{new_2}$;

• balancing value $v^{balance}$.

Obviously, the following equation is satisfied, fee is the transfer fee.

$$\sum_{i=1}^{n_1} v_i^{old_1} + \sum_{i=1}^{n_2} v_i^{old_2} = \sum_{j=1}^{m_1} v_j^{new_1} + \sum_{j=1}^{m_2} v_j^{new_2} + fee$$

In a correctly constructed transaction, $v^{balance} = \sum_{i=1}^{n_1} v_i^{old_1} - \sum_{j=1}^{m_1} v_j^{new_1} = \sum_{j=1}^{m_2} v_j^{new_2} - \sum_{i=1}^{n_2} v_i^{old_2} + fee$, but validators cannot check this directly, because the values are hidden by the commitments.

Instead, validators calculate the transaction binding verification key as:

$$bvk = (\circledast_{i=1}^{n} cv_i^{old}) \diamond (\circledast_{j=1}^{m} cv_j^{new}) \diamond valueCommit_0(v^{balance})$$

(This key is not encoded explicitly in the transaction and must be recalculated.)
The signer knows rcv^{old} and rcv^{new}, and so can calculate the corresponding signing key as:

$$bsk = (\boxplus_{i=1}^{n} rcv_i^{old}) \boxminus (\boxplus_{j=1}^{m} rcv_j^{old})$$

In order to check for implementation faults, the signer should also check that

$$bvk = BindingSig.DerivePublic(bsk).$$

$BindingSig$ is instantiated as $RedJubjub$, without use of key re-randomization.

6.2.7. Note Encryption

Let pk_d^{new}: $KA.PublicPrimeOrder$ be the diversified transmission key for the intended recipient address of a new note, and let g_d^{new}: $KA.PublicPrimeOrder$ be the corresponding diversified base computed as $DiversifyHash(d)$.

Since note encryption is used only in sending notes to receipt, we may assume that g^{new} has already been calculated and is not $\perp$.

Let ovk: $\mathbb{B}^{\mathbb{Y}[l_{ovk}/8]} \cup \{\perp\}$ be the outgoing viewing key of the shielded payment address from which the note is being spent, or an outgoing viewing key associated with a [ZIP-32] account, or $\perp$.

Let $\boldsymbol{np} = (d, v, rcm, memo)$ be the note plaintext, encoded when using.

Let cv^{new} be the value commitment for the new note, and let cm^{new} be the note commitment.

Then to encrypt:

Choose a uniformly random ephemeral private key $esk \xleftarrow{R} KA.Private \setminus \{0\}$.

let $epk = KA.DerivePublic(esk, g_d^{new})$

let P^{enc} be the raw encoding of **np**

let $sharedSecret = KA.Agree(esk, pk_d^{new})$

let $K^{enc} = KDF(sharedSecret, epk)$

let $C^{enc} = Sym.Encrypt_{K^{enc}}(P^{enc})$

if ovk = ⊥:

choose random $ock \xleftarrow{R} Sym.\boldsymbol{K}$ and $\boldsymbol{op} \xleftarrow{R} \mathbb{B}^{\mathbb{Y}[(l_{\mathbb{J}}+256)/8]}$

else:

let $cv = LEBS2OSP_{l_{\mathbb{J}}}(repr_{\mathbb{J}}(cv^{new}))$

let $cm_u = LEBS2OSP_{256}(Extract_{\mathbb{J}^{(r)}}(cm^{new}))$

let $ephemeralKey = LEBS2OSP_{l_{\mathbb{J}}}\left(repr_{\mathbb{J}}(epk)\right)$

let $ock = PRF_{ovk}^{ock}(cv, cm_u, ephemeralKey)$

let $\boldsymbol{op} = LEBS2OSP_{l_{\mathbb{J}}+256}\left(repr_{\mathbb{J}}(pk_d^{new}) || I2LEBSP_{256}(esk)\right)$

let $C^{out} = Sym.Encrypt_{ock}(\boldsymbol{op})$

The resulting transmitted note ciphertext is (epk, C^{enc}, C^{out}).

6.2.8. Note Decryption

The incoming viewing key (ivk) holder could decrypt the ciphertexts to get note, while the outgoing viewing key (ovk) holder could decrypt to get the note too.

6.2.8.1. Note Decryption with ivk

Given ciphertexts (epk, C^{enc}, C^{out}) from the output description, ivk holder could decrypt C^{enc} to get note as follows.

let $sharedSecret = KA.Agree(ivk, epk)$

let $K^{enc} = KDF^{Sapling}(sharedSecret, epk)$

let $P^{enc} = Sym.Decrypt_{K^{enc}}(C^{enc})$

if $P^{enc} = \perp$, return ⊥

extract $\boldsymbol{np} = \left(d: \mathbb{B}^{[l_d]}, v: \{0..2^{l_{value}} - 1\}, rcm: \mathbb{B}^{\mathbb{Y}32]}, memo: \mathbb{B}^{\mathbb{Y}32]}\right)$ from P^{enc}

let $rcm = LEOS2IP_{256}(rcm)$ and $g_d = DiversifyHash(d)$

if $rcm \geq r_J$ or $g_d = \perp$, return $\perp$

let $pk_d = KA.DerivePublic(ivk, g_d)$

let $cm'_u = Extract_{\mathbb{J}^{(r)}}(NoteCommit_{rcm^{new}}(repr_{\mathbb{J}}(g_d), repr_{\mathbb{J}}(pk_d), v))$.

if $LEBS2OSP_{256}(cm'_u \neq cm_u$, return $\perp$, else return **np**.

6.2.8.2. Note Decryption with ovk

Given ciphertexts (epk, C^{enc}, C^{out}) from the output description, ovk holder could decrypt C^{enc} *and* C^{enc} to get note as follows.

let $ock = PRF^{ock}_{ovk}(cv, cm_u, ephemeralKey)$

let $op = Sym.Decrypt_{ock}(C^{out})$

if op = $\perp$, return $\perp$

extract $(pk\star_d: \mathbb{B}^{[l_{\mathbb{J}}]}, \underline{esk}: \mathbb{B}^{\mathbb{Y}[32]})$ from op

let $esk = LEOS2IP_{256}(\underline{esk})$ *and* $pk_d = abst_J(pk\star_d)$

if $esk \geq r_{\mathbb{J}}$ *or* $pk_d \notin KA.PublicPrimeOrder$, return $\perp$

let $sharedSecret = KA.Agree(esk, pk_d)$

let $K^{enc} = KDF(sharedSecret, epk)$

let $P^{enc} = Sym.Decrypt_{K^{enc}}(C^{enc})$

if $P^{enc} = \perp$, return $\perp$

extract $np = \left(d: \mathbb{B}^{[l_d]}, v: \{0..2^{l_{value}} - 1\}, rcm: \mathbb{B}^{\mathbb{Y}32]}, memo: \mathbb{B}^{\mathbb{Y}32]}\right)$ from P^{enc}

let $rcm = LEOS2IP_{256}(rcm)$ and $g_d = DiversifyHash(d)$

if $rcm \geq r_J$ or $g_d = \perp$, return $\perp$

if $KA.DerivePublic(esk, g_d^{new}) \neq epk$, return $\perp$

let $cm'_u = Extract_{\mathbb{J}^{(r)}}(NoteCommit_{rcm^{new}}(repr_{\mathbb{J}}(g_d), repr_{\mathbb{J}}(pk_d), v))$.

if $LEBS2OSP_{256}(cm'_u \neq cm_u$, return $\perp$, else return np

6.2.9. Broadcast Transaction

After creating transaction, wallet broadcast it to the active peer node.

6.3. Block chain

When receives a shielded transaction, block chain need verify the transaction. After verification, the transaction could be executed.

6.3.1. Verify Transaction

Transaction verification includes the spend authority signature, spend proof, output proof, binding signature, and nullifier.

6.3.1.1. Verify Spend Authority Signature

A verifier assures that the signature of SpendAuthSig is verified according to the verification algorithm defined in §3.7.1.

6.3.1.2. Verify Spend Proof

A verifier assures that given a proof $\pi_{ZKspend}.proof$ and a primary input $spendPrimaryInput$:

$$\begin{gathered}(rt: \mathbb{B}^{l_{Merkle}}, \\ cv^{old}: ValueCommit.output, \\ nf^{old}: \mathbb{B}^{l_{PRFnf}}, \\ rk: SpendAuthSig.Public\ )\end{gathered}$$

$\pi_{ZKspend}.Verify_{VK}\left(spendPrimaryInput, \pi_{ZKspend}.proof\right) = True$ should be checked.

6.3.1.3. Verify Output Proof

A verifier assures that given a proof $\pi_{ZKOutput}.proof$ and a primary input $outPrimaryInput$:

$$\begin{gathered}(cv^{new}: ValueCommit.output, \\ cm_u: \mathbb{B}^{l_{merkle}}, \\ epk: \mathbb{J})\end{gathered}$$

$\pi_{ZKoutput}.Verify_{PK}\left(outPrimaryInput, \pi_{ZKoutput}.proof\right) = True$ should be checked.

6.3.1.4. Verify Binding Signature

The following key is used to verify binding signature.

$$bvk = (\circledplus_{i=1}^{n} cv_i^{old}) \diamond (\circledplus_{j=1}^{m} cv_j^{new}) \diamond valueCommit_0(v^{balance})$$

The verify algorithm is instantiated in §3.7.1.

6.3.1.5. Verify Nullifier

The nullifier $nf: \mathbb{B}^{[l_{PRFnf}]}$ of the spend notes is revealed in spend description. It is enforced to be unique within a valid block chain, in order to prevent double-spends.

A validator checks a nullifier is not in the nullifier set in order to verify the transaction.

6.3.1.6. Verify others

In addition to verify above things, others should also be checked. For instance, transfer contract fee should be equal to what we pre-define.

6.3.2. Execute Transaction

After verification, the transaction could be executed, that is, saved in the block chain.

6.3.2.1. Process Transparent Input

If the transaction input is from transparent address, we will first adjust the balance of the input account, that is, subtract the input value from its account.

6.3.2.2. Save CM, Update Tree

In order to make the output transfer valid and shielded, we need save each note commitment. Get each note commitment from Receive description, and save them into the Merkle tree and update.

6.3.2.3. Save Nullifier

In order to prevent the spend note from double-spent, we need save the nullifier of each spend note into nullifier set. Get each nullifier from spend description, and store them into the database.

6.3.2.4. Process Transparent Output

If one of the transaction output is to transparent address, we will first adjust the balance of the transparent output account, that is, add the output value to its account.

6.4. Contract

6.4.1. User APIs

We support shielded transaction from a single address to multiple addresses.

The spend address and output address can be either transparent or shielded. However, we do not support that spend address and output address both are transparent, in this case, it is an ordinary transparent transaction.

We provide interfaces to help users build transactions, or users can build by themselves. If user choose to build by themselves, it is him that generate spend proofs and output proofs when needed.

There are 3 conditions if users build transactions by means of APIs we provide.

- transfer from a transparent address to shielded address: users just input the following fields:
  - transparent_from_address: the spend transparent address
  - from_amount: the spend value
  - shieldedReceives: the output note
- transfer from a shielded address to transparent address: users just input the following fields:
  - ask: the spend authorizing key
  - nsk: the proof authorizing key
  - ovk: the outgoing viewing key
  - shieldedSpends: the spend note
  - transparent _to_address: the output address

    - to_mount: the output value
- transfer from a shielded address to shielded address: users just input the following fields:
    - ask: the spend authorizing key
    - nsk: the proof authorizing key
    - ovk: the outgoing viewing key
    - shieldedSpends: the spend note
    - shieldedReceives: the output notes

In addition, we have defined many rpcs in "src/main/protos/api/api.proto" as following.

```
rpc CreateShieldedTransaction (PrivateParameters) returns (TransactionExtention) {
};

rpc GetMerkleTreeVoucherInfo (OutputPointInfo) returns (IncrementalMerkleVoucherInfo) {
}

rpc ScanNoteByIvk (IvkDecryptParameters) returns (DecryptNotes) {
};

rpc ScanAndMarkNoteByIvk (IvkDecryptAndMarkParameters) returns (DecryptNotesMarked) {
};

rpc ScanNoteByOvk (OvkDecryptParameters) returns (DecryptNotes) {
};

rpc GetSpendingKey (EmptyMessage) returns (BytesMessage) {
}

rpc GetExpandedSpendingKey (BytesMessage) returns (ExpandedSpendingKeyMessage) {
}

rpc GetAkFromAsk (BytesMessage) returns (BytesMessage) {
}

rpc GetNkFromNsk (BytesMessage) returns (BytesMessage) {
}

rpc GetIncomingViewingKey (ViewingKeyMessage) returns (IncomingViewingKeyMessage) {
}

rpc GetDiversifier (EmptyMessage) returns (DiversifierMessage) {
}

rpc GetNewShieldedAddress (EmptyMessage) returns (ShieldedAddressInfo) {
}

rpc GetZenPaymentAddress (IncomingViewingKeyDiversifierMessage) returns (PaymentAddressMessage) {
}

rpc GetRcm (EmptyMessage) returns (BytesMessage) {
}

rpc IsSpend (NoteParameters) returns (SpendResult) {
}

rpc CreateShieldedTransactionWithoutSpendAuthSig (PrivateParametersWithoutAsk) returns (TransactionExtention) {
};

rpc GetShieldTransactionHash (Transaction) returns (BytesMessage) {
};

rpc CreateSpendAuthSig (SpendAuthSigParameters) returns (BytesMessage) {
};

rpc CreateShieldNullifier (NfParameters) returns (BytesMessage) {
};
```

The above parameters are defined in "src/main/protos/api/api.proto".

```
message IvkDecryptAndMarkParameters {
  int64 start_block_index = 1;
  int64 end_block_index = 2;
  bytes ivk = 5;
  bytes ak = 3;
  bytes nk = 4;
}

message OvkDecryptParameters {
  int64 start_block_index = 1;
  int64 end_block_index = 2;
  bytes ovk = 3;
}

message DecryptNotes {
  message NoteTx {
    Note note = 1;
    bytes txid = 2; //transaction id =  sha256(transaction.rowdata)
    int32 index = 3; //the index of note in receive
  }
  repeated NoteTx noteTxs = 1;
}

message DecryptNotesMarked {
  message NoteTx {
    Note note = 1;
    bytes txid = 2; //transaction id =  sha256(transaction.rowdata)
    int32 index = 3; //the index of note in receive
    bool is_spend = 4;
  }
  repeated NoteTx noteTxs = 1;
}

message Note {
  int64 value = 1;
  string payment_address = 2;
  bytes rcm = 3; // random 32
  bytes memo = 4;
}

message SpendNote {
  Note note = 3;
  bytes alpha = 4; // random number for spend authority signature
  IncrementalMerkleVoucher voucher = 5;
  bytes path = 6; // path for cm from leaf to root in merkle tree
}

message ReceiveNote {
  Note note = 1;
}
```

```
message PrivateParameters {
  bytes transparent_from_address = 1;
  bytes ask = 2;
  bytes nsk = 3;
  bytes ovk = 4;
  int64 from_amount = 5;
  repeated SpendNote shielded_spends = 6;
  repeated ReceiveNote shielded_receives = 7;
  bytes transparent_to_address = 8;
  int64 to_amount = 9;
}

message PrivateParametersWithoutAsk {
  bytes transparent_from_address = 1;
  bytes ak = 2;
  bytes nsk = 3;
  bytes ovk = 4;
  int64 from_amount = 5;
  repeated SpendNote shielded_spends = 6;
  repeated ReceiveNote shielded_receives = 7;
  bytes transparent_to_address = 8;
  int64 to_amount = 9;
}

message SpendAuthSigParameters {
  bytes ask = 1;
  bytes tx_hash = 2;
  bytes alpha = 3;
}

message NfParameters {
  Note note = 1;
  IncrementalMerkleVoucher voucher = 2;
  bytes ak = 3;
  bytes nk = 4;
}

message ExpandedSpendingKeyMessage {
  bytes ask = 1;
  bytes nsk = 2;
  bytes ovk = 3;
}

message ViewingKeyMessage {
  bytes ak = 1;
  bytes nk = 2;
}

message IncomingViewingKeyMessage {
  bytes ivk = 1;
}

message DiversifierMessage {
  bytes d = 1;
}
```

```
message IncomingViewingKeyDiversifierMessage {
  IncomingViewingKeyMessage ivk = 1;
  DiversifierMessage d = 2;
}

message PaymentAddressMessage {
  DiversifierMessage d = 1;
  bytes pkD = 2;
  string payment_address = 3;
}

message ShieldedAddressInfo{
  bytes sk = 1;
  bytes ask = 2;
  bytes nsk = 3;
  bytes ovk = 4;
  bytes ak = 5;
  bytes nk = 6;
  bytes ivk = 7;
  bytes d = 8;
  bytes pkD = 9;
  string payment_address = 10;
}

message NoteParameters {
  bytes ak = 1;
  bytes nk = 2;
  Note note = 3;
  bytes txid = 4;
  int32 index = 5;
}

message SpendResult {
  bool result = 1;
  string message = 2;
}
```

6.4.2. Shielded Transfer Contract

Given spend proof or/and output proof, and other necessary information, we can build shielded transfer contract.

There are 3 contracts we can construct, according to what users provide.

- transfer from a transparent address to shielded address: users need provide the following fields:
  - transparent_from_address: the spend transparent address
  - from_amount: the spend value
  - spend_description: null
  - receive_description: the receive description
  - binding_signature: the binding signature
  - transparent_to_address: the ouput address, null if no transparent output.
  - to_amount: the output amount, 0 if no transparent output.
- transfer from a shielded address to transparent address: users need provide the following fields:
  - transparent_from_address: null
  - from_amount: 0
  - spend_description: the spend description
  - receive_description: null

- binding_signature: the binding signature
- transparent_to_address: the ouput address
- to_amount: the output amount

- transfer from a shielded address to shielded address: users just input the following fields:
  - transparent_from_address: null
  - from_amount: 0
  - spend_description: the spend description
  - receive_description: the receive descriptions
  - binding_signature: the binding signature
  - transparent_to_address: null.
  - to_amount: 0

The shielded contract is as following.

```
message SpendDescription {
  bytes value_commitment = 1;
  bytes anchor = 2; // merkle root
  bytes nullifier = 3; // used for check double spend
  bytes rk = 4; // used for check spend authority signature
  bytes zkproof = 5;
  bytes spend_authority_signature = 6;
}

message ReceiveDescription {
  bytes value_commitment = 1;
  bytes note_commitment = 2;
  bytes epk = 3; // for Encryption
  bytes c_enc = 4; // Encryption for incoming, decrypt it with ivk
  bytes c_out = 5; // Encryption for audit, decrypt it with ovk
  bytes zkproof = 6;
}

message ShieldedTransferContract {
  bytes transparent_from_address = 1; // transparent address
  int64 from_amount = 2;
  repeated SpendDescription spend_description = 3;
  repeated ReceiveDescription receive_description = 4;
  bytes binding_signature = 5;
  bytes transparent_to_address = 6; // transparent address
  int64 to_amount = 7; // the amount to transparent to_address
}
```

7. References

[ANWW2013] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. January 29, 2013. URL:https://blake2.net/ #sp(visited on 2016-08-14) (↑ p52, 136).

[Bowe2017] Sean Bowe. ebfull/pairing source code, BLS12-381 – README.md as of commit e726600. URL: https://github.com/ebfull/pairing/tree/e72660056e00c93d6b054dfb08ff34a1c67cb799/src/bls12_381 (visited on 2017-07-16) (↑ p67).

[Groth2016] Jens Groth. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive: Report 2016/260. Last revised May 31, 2016. URL: https://eprint.iacr.org/2016/260 (visited on 2017-08-03) (↑ p71, 72, 98, 143).

[Zcash] Sean Bowe, Taylor Hornby, Nathan Wilcox, Zcash Protocol Specification, Sep. 2018, [online]Available:https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

Appendices

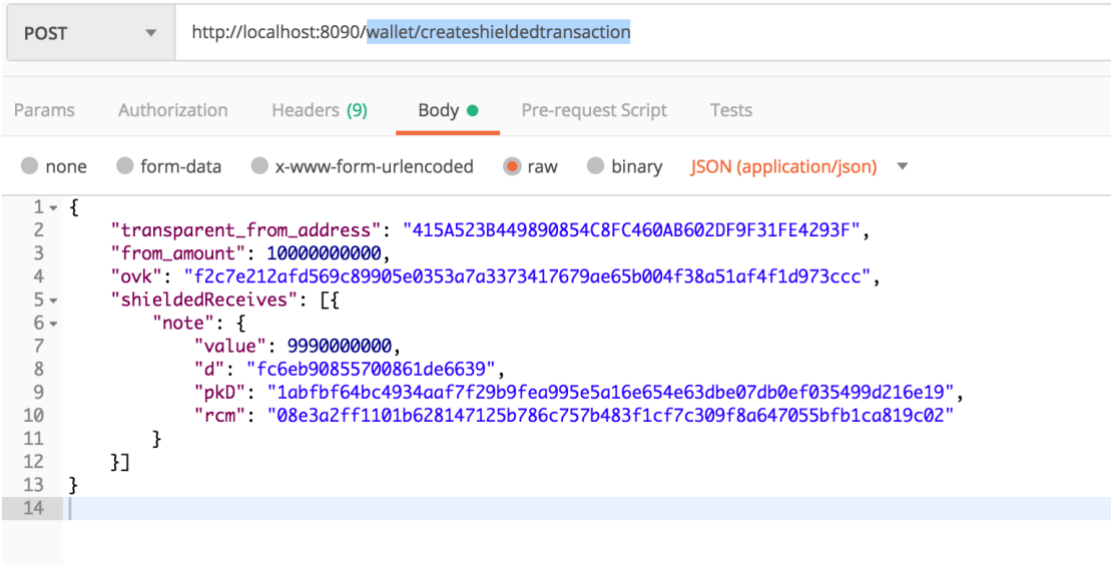
A. Demo

In this section, we give 2 demos. One is transaction from transparent address to shielded address, the other one is from shielded address to shielded address.

A.1 Transaction from public address to shielded address

A.1.1 Create shielded transaction

call api: wallet/createshieldedtransaction



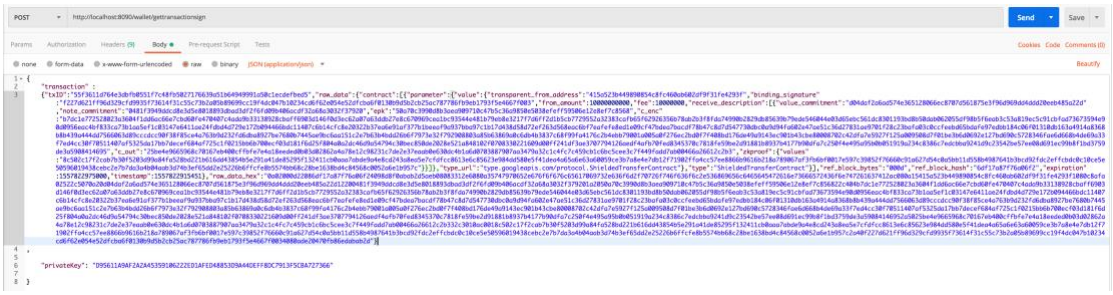
Result:

A.1.2 Signature

call api to sign (use the private key of the public address)

api: wallet/gettransactionsign

param:



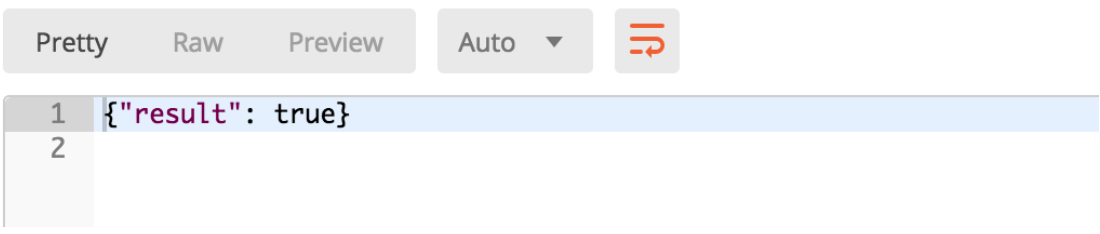
Result:

A.1.3 Broadcast Transaction

api: wallet/broadcasttransaction

param:

Result:

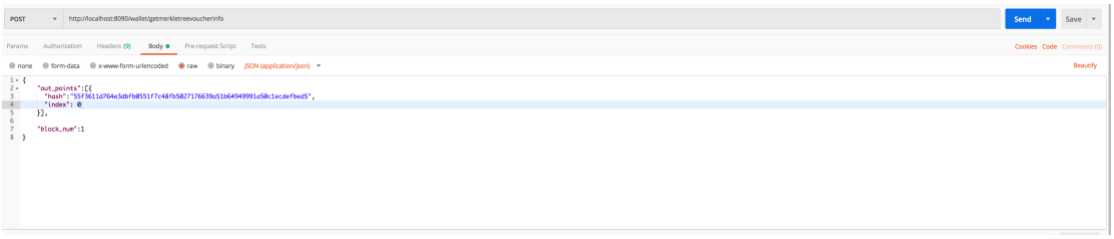


A.2 Transaction from shielded address to shielded address

A. 2. 1 Get voucher

Call api getmerkletreevoucherinfo to get the voucher of the shield address, this info will be used when create shielded transaction

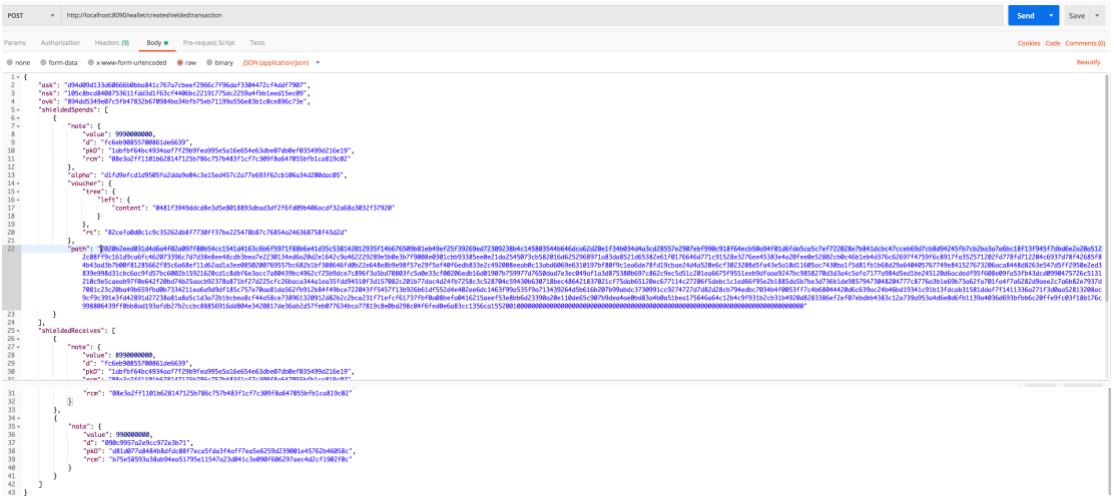
param:



result:

A. 2. 2 Create transaction

call api: wallet/createshieldedtransaction



result:

A.2.3 Broadcast Transaction

There is no need to sign this transaction.

call api: wallet/broadcasttransaction

param: