Zcash Protocol Specification
Version 2.0-draft-2

Sean Bowe — Daira Hopwood — Taylor Hornby

February 12, 2016

Contents

1 Introduction

Zcash is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash** [2] with some adjustments to terminology, functionality and performance. It bridges the existing *transparent* payment scheme used by **Bitcoin** with a *confidential* payment scheme protected by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*).

Changes from the original **Zerocash** are highlighted in magenta.

2 Concepts

2.1 Integers, Bit Sequences, and Endianness

All integers visible in **Zcash**-specific encodings are unsigned, have a fixed bit length, and are encoded as big-endian.

In bit layout diagrams, each box of the diagram represents a sequence of bits. If the content of the box is a byte sequence, it is implicitly converted to a sequence of bits using big endian order. The bit sequences are then concatenated in the order shown from left to right, and the result is converted to a sequence of bytes, again using big-endian order.

Nathan: An example would help here. It would be illustrative if it had a few differently-sized fields.

$\mathtt{Leading}_k(x)$, where k is an integer and x is a bit sequence, returns the leading (initial) k bits of its input.

2.2 Cryptographic Functions

CRH is a collision-resistant hash function. In **Zcash**, the *SHA-256 compression* function is used which takes a 512-bit block and produces a 256-bit hash. This is different from the *SHA-256* function, which hashes arbitrary-length strings. [7]

PRF_x is a pseudo-random function seeded by x. Four *independent* PRF_x are needed in our scheme: $\mathsf{PRF}^{\mathsf{addr}}_x$, $\mathsf{PRF}^{\mathsf{sn}}_x$, $\mathsf{PRF}^{\mathsf{pk}}_x$, and PRF^{ρ}_x. It is required that $\mathsf{PRF}^{\mathsf{sn}}_x$ and PRF^{ρ}_x be collision-resistant across all x — i.e. it should not be feasible to find $(x, y) \neq (x', y')$ such that $\mathsf{PRF}^{\mathsf{sn}}_x(y) = \mathsf{PRF}^{\mathsf{sn}}_{x'}(y')$, and similarly for PRF^{ρ}.

In **Zcash**, the *SHA-256 compression* function is used to construct all four of these functions. The bits `00`, `01`, `10`, and `11` are included (respectively) within the blocks that are hashed, ensuring that the functions are independent.

Nathan: Note: If we change input arity (i.e. $\mathrm{N}^{\mathsf{old}}$), we need to be aware of how it is associated with this bit-packing.

$$
\begin{array}{lll}
\mathsf{a_{pk}} := \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a_{sk}}}(0) & = \mathsf{CRH}\Big(& \boxed{256 \text{ bit } \mathsf{a_{sk}}}\boxed{0}\boxed{0}\boxed{0^{254}} \Big) \\
\mathsf{sn} := \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho) & = \mathsf{CRH}\Big(& \boxed{256 \text{ bit } \mathsf{a_{sk}}}\boxed{0}\boxed{1}\boxed{\mathtt{Leading}_{254}(\rho)} \Big) \\
\mathsf{h}_i := \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}}) & = \mathsf{CRH}\Big(& \boxed{256 \text{ bit } \mathsf{a_{sk}}}\boxed{1}\boxed{0}\boxed{i}\boxed{\mathtt{Leading}_{253}(\mathsf{h_{Sig}})} \Big) \\
\rho^{\mathsf{new}}_i := \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}}) & = \mathsf{CRH}\Big(& \boxed{256 \text{ bit } \varphi}\boxed{1}\boxed{1}\boxed{i}\boxed{\mathtt{Leading}_{253}(\mathsf{h_{Sig}})} \Big)
\end{array}
$$

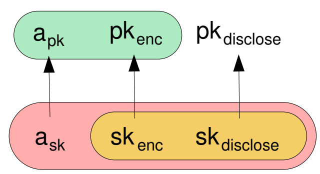
Daira: Should we instead define ρ to be 254 bits and $\mathsf{h_{Sig}}$ to be 253 bits?

2.3 Payment Addresses, Viewing Keys, and Spending Keys

A *key tuple* $(\mathsf{addr_{pk}}, \mathsf{addr_{viewkey}}, \mathsf{addr_{sk}})$ is generated by users who wish to receive payments under this scheme. The parts of the *key tuple* are composed from three distinct keypairs, called the *authorization*, *transmission*, and *disclosure* keypairs.

- The *payment address* $\mathsf{addr_{pk}}$ is a pair $(\mathsf{a_{pk}}, \mathsf{pk_{enc}})$, containing the *public* components of the *authorization* and *transmission* keypairs respectively.
- The *viewing key* $\mathsf{addr_{viewkey}}$ is a pair $(\mathsf{sk_{enc}}, \mathsf{sk_{disclose}})$, containing the *private* components of the *transmission* and *disclosure* keypairs respectively.
- The *spending key* $\mathsf{addr_{sk}}$ is a triple $(\mathsf{a_{sk}}, \mathsf{sk_{enc}}, \mathsf{sk_{disclose}})$, containing the *private* components of the *authorization*, *transmission*, and *disclosure* keypairs respectively.

The following diagram depicts the relations between key components. Arrows point from a private component to the corresponding public component derived from it.



Note that a *spending key* holder can derive $(\mathsf{a_{pk}}, \mathsf{pk_{enc}}, \mathsf{pk_{disclose}})$, and a *viewing key* holder can derive $(\mathsf{pk_{enc}}, \mathsf{pk_{disclose}})$, even though these components are not formally part of the respective keys. Implementations MAY cache these derived public components, provided that they are deleted if the corresponding private component is deleted.

The composition of *payment addresses*, *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 *viewing key* from a *spending key*; and
- obtain a *payment address* from a *spending key*.

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

2.4 Coins

A *coin* (denoted $\mathbf{c}$) is a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$ which represents that a value v is spendable by the recipient who holds the *authorization* key pair $(\mathsf{a_{pk}}, \mathsf{a_{sk}})$ such that $\mathsf{a_{pk}} = \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a_{sk}}}(0)$.

r is randomly generated by the sender. ρ is generated from a random seed φ using $\mathsf{PRF}^{\rho}_{\varphi}$. Only a commitment to these values is disclosed publicly, which allows the tokens r and ρ to blind the value and recipient *except* to those who possess these tokens.

2.4.1 In-band secret distribution

In order to transmit the secret v, ρ, and r (necessary for the recipient to later spend) and also a *memo field* to the recipient *without* requiring an out-of-band communication channel, the *transmission* public key $\mathsf{pk_{enc}}$ is used to encrypt these secrets to form a *transmitted coins ciphertext*. The recipient's possession of the associated $(\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ (which contains both $\mathsf{a_{pk}}$ and $\mathsf{sk_{enc}}$) is used to reconstruct the original *coin* and *memo field*.

The encryption algorithm is defined in terms of $\mathsf{crypto\_box}$ (i.e. $\mathsf{crypto\_box\_curve25519xsalsa20poly1305}$) [3] as follows.

Let $\mathsf{pk}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$ be the Curve25519 public keys for the intended recipient addresses of each new *coin*, and let $\mathbf{P}_{1..\mathrm{N}^{\mathsf{new}}}$ be their *coin plaintexts*.

Define:

$$\mathsf{prenonce}(i, \mathsf{pk}_{\mathsf{eph}}, \mathsf{pk}_{\mathsf{enc},i}) := \mathtt{SHA256}\left(\begin{array}{|c|c|c|} \hline 64 \text{ bit } i-1 & 256 \text{ bit } \mathsf{pk}_{\mathsf{eph}} & 256 \text{ bit } \mathsf{pk}_{\mathsf{enc},i} \\ \hline \end{array} \right)$$

$$\mathsf{nonce}(i, \mathsf{pk}_{\mathsf{eph}}, \mathsf{pk}_{\mathsf{enc},i}) := \begin{array}{|c|c|} \hline \mathtt{Leading}_{128}(\mathsf{prenonce}) & 64 \text{ bit } i-1 \\ \hline \end{array}$$

Then to encrypt:

- Generate a new Curve25519 (public, private) key pair $(\mathsf{pk}_{\mathsf{eph}}, \mathsf{sk}_{\mathsf{eph}})$.
- For i in $\{1..\mathrm{N}^{\mathsf{new}}\}$, let $\mathbf{C}_i^{\mathsf{enc}} = \mathsf{crypto\_box}(\mathbf{P}_i, \mathsf{pk}_{\mathsf{enc},i}, \mathsf{sk}_{\mathsf{eph}}, \mathsf{nonce}(i, \mathsf{pk}_{\mathsf{eph}}, \mathsf{pk}_{\mathsf{enc},i}))$.
- Let $\mathsf{Encrypt}_{\mathsf{pk}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}}(\mathbf{P}_{1..\mathrm{N}^{\mathsf{new}}}) = (\mathsf{pk}_{\mathsf{eph}}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$.

Let $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}})$ be the recipient's Curve25519 (public, private) key pair, and let $(\mathsf{pk}_{\mathsf{eph}}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathbf{C}^{\mathsf{view}}_{1..\mathrm{N}^{\mathsf{new}}})$ be the *transmitted coins ciphertext*.

Then for each i in $\{1..\mathrm{N}^{\mathsf{new}}\}$, the recipient will attempt to decrypt that ciphertext component as follows:

- $\mathsf{Decrypt}_{\mathsf{sk}_{\mathsf{enc}}}(i, \mathsf{pk}_{\mathsf{eph}}, \mathbf{C}_i^{\mathsf{enc}}) = \mathsf{crypto\_box\_open}(\mathbf{C}_i^{\mathsf{enc}}, \mathsf{pk}_{\mathsf{eph}}, \mathsf{sk}_{\mathsf{enc}}, \mathsf{nonce}(i, \mathsf{pk}_{\mathsf{eph}}, \mathsf{pk}_{\mathsf{enc}}))$

Any ciphertext components that fail to decrypt with a given recipient's private key will be ignored.

This is a variation on the $\mathsf{crypto\_box\_seal}$ algorithm defined in libsodium [6], but with a single ephemeral key used for all encryptions in a given *Pour description*, and with the nonce for each ciphertext component depending on the index i. Also, $\mathtt{SHA256}$ (the full hash, not the compression function) is used instead of $\mathsf{blake2b}$. The particular nonce construction is chosen so that a known-nonce distinguisher for $\mathsf{Salsa20}$ would not directly lead to a break of the IK-CCA (key privacy) property.

2.4.2 Coin Commitments

The underlying v and $\mathsf{a_{pk}}$ are blinded with ρ and r using the collision-resistant hash function CRH in a multi-layered process. The resulting hash $\mathsf{cm} = \mathtt{CoinCommitment}(\mathbf{c})$.

$$\mathsf{InternalH} := \mathsf{CRH}\left(\begin{array}{|c|c|} \hline 256 \text{ bit } \mathsf{a_{pk}} & 256 \text{ bit } \rho \\ \hline \end{array} \right)$$

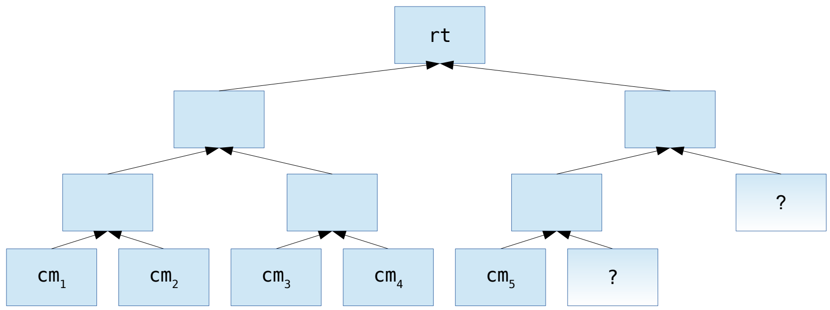
$$\mathsf{k} := \mathsf{CRH}\left(\begin{array}{|c|c|} \hline 384 \text{ bit } \mathsf{r} & \mathtt{Leading}_{128}(\mathsf{InternalH}) \\ \hline \end{array} \right)$$

$$\mathsf{cm} := \mathsf{CRH}\left(\begin{array}{|c|c|c|} \hline 64 \text{ bit } \mathsf{v} & 192 \text{ bit padding} & 256 \text{ bit } \mathsf{k} \\ \hline \end{array} \right)$$

2.4.3 Serial numbers

A *serial number* (denoted sn) equals $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$. A *coin* is spent by proving knowledge of ρ and $\mathsf{a_{sk}}$ in zero knowledge while disclosing sn, allowing sn to be used to prevent double-spending.

2.5 Coin Commitment Tree



The *coin commitment tree* is an *incremental merkle tree* of depth d used to store *coin commitments* that *Pour transfers* produce. Just as the *unspent transaction output set* (UTXO) used in Bitcoin, it is used to express the existence of value and the capability to spend it. However, unlike the UTXO, it is *not* the job of this tree to protect against double-spending, as it is append-only.

Blocks in the blockchain are associated (by all nodes) with the root of this tree after all of its constituent *Pour descriptions*' *coin commitments* have been entered into the tree associated with the previous block.

2.6 Spent Serials Map

Transactions insert *serial numbers* into a *spent serial numbers map* which is maintained alongside the UTXO by all nodes.

Eli: a tx is just a string, so it doesn't insert anything. Rather, nodes process tx's and the "good" ones lead to the addition of serials to the spent serials map.

Transactions that attempt to insert a *serial number* into this map that already exists within it are invalid as they are attempting to double-spend.

Eli: After defining *transaction*, one should define what a *legal tx* is (this definition depends on a particular blockchain [view]) and only then can one talk about "attempts" of transactions, and insertions of serial numbers into the spent serials map.

2.7 The Blockchain

At a given point in time, the *blockchain view* of each *full node* consists of a sequence of one or more valid *blocks*. Each *block* consists of a sequence of one or more *transactions*. In a given node's *blockchain view*, *treestates* are chained in an obvious way:

- 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*.

An *anchor* is a Merkle tree root of a *treestate*, and uniquely identifies that *treestate* given the assumed security properties of the Merkle tree's hash function.

Each *transaction* is associated with a sequence of *Pour descriptions*. TODO: They also have a transparent value flow that interacts with the Pour $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ and $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$. Inputs and outputs are associated with a value.

The total value of the outputs must not exceed the total value of the inputs.

The *anchor* of the first *Pour description* in a *transaction* must refer to some earlier *block*'s final *treestate*.

The *anchor* of each subsequent *Pour description* may refer either to some earlier *block*'s final *treestate*, or to the output *treestate* of the immediately preceding *Pour description*.

These conditions act as constraints on the blocks that a *full node* will accept into its *blockchain view*.

We rely on Bitcoin-style consensus for *full nodes* to eventually converge on their views of valid *blocks*, and therefore of the sequence of *treestates* in those *blocks*.

Value pool Transaction inputs insert value into a *value pool*, and transaction outputs remove value from this pool. The remaining value in the pool is available to miners as a fee.

3 Pour Transfers and Descriptions

A *Pour description* is data included in a *block* that describes a *Pour transfer*, i.e. a confidential value transfer. This kind of value transfer is the primary **Zerocash**-specific operation performed by transactions; it uses, but should not be confused with, the `POUR` *circuit* used for the *zk-SNARK* proof and verification.

A *Pour transfer* spends $\mathrm{N}^{\mathsf{old}}$ *coins* $\mathbf{c}_{1..\mathrm{N}^{\mathsf{old}}}^{\mathsf{old}}$ and creates $\mathrm{N}^{\mathsf{new}}$ *coins* $\mathbf{c}_{1..\mathrm{N}^{\mathsf{new}}}^{\mathsf{new}}$. **Zcash** transactions have an additional field `vpour`, which is a sequence of *Pour descriptions*.

Each *Pour description* consists of:

`vpub_old` which is a value $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ that the *Pour transfer* removes from the value pool.

`vpub_new` which is a value $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$ that the *Pour transfer* inserts into the value pool.

`anchor` which is a merkle root rt of the *coin commitment tree* at some block height in the past, or the merkle root produced by a previous pour in this transaction. Sean: We need to be more specific here.

`scriptSig` which is a *script* that creates conditions for acceptance of a *Pour description* in a transaction.

`scriptPubKey` which is a *script* used to satisfy the conditions of the `scriptSig`.

`serials` which is an $\mathrm{N}^{\mathsf{old}}$ size sequence of serials $\mathsf{sn}_{1..\mathrm{N}^{\mathsf{old}}}^{\mathsf{old}}$.

`commitments` which is a $\mathrm{N}^{\mathsf{new}}$ size sequence of *coin commitments* $\mathsf{cm}_{1..\mathrm{N}^{\mathsf{new}}}^{\mathsf{new}}$.

`ephemeralKey` which is a Curve25519 public key $\mathsf{pk}_{\mathsf{eph}}$.

`ciphertexts` which is a $\mathrm{N}^{\mathsf{new}}$ size sequence of ciphertext components. (`ephemeralKey` and `ciphertexts` together form the *transmitted coins ciphertext*.)

`vmacs` which is a $\mathrm{N}^{\mathsf{old}}$ size sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the *Pour description*.

`zkproof` which is the zero-knowledge proof $\pi_{\texttt{POUR}}$.

Computation of $\mathsf{h}_{\mathsf{Sig}}$ Given a *Pour description*, we define:

$$\mathsf{h}_{\mathsf{Sig}} := \mathtt{SHA256}\left(\boxed{\mathbf{0x00}}\boxed{256 \text{ bit } \mathsf{sn}_{0}^{\mathsf{old}}}\boxed{\cdots}\boxed{256 \text{ bit } \mathsf{sn}_{\mathrm{N}^{\mathsf{old}}-1}^{\mathsf{old}}}\boxed{\mathtt{scriptPubKey}}\right)$$

Merkle root validity A *Pour description* is valid if rt is a *coin commitment tree* root found in either the blockchain or a merkle root produced by inserting the *coin commitments* of a previous *Pour description* in the transaction to the *coin commitment tree* identified by that previous *Pour description*'s *anchor*.

Non-malleability A *Pour description* is valid if the script formed by appending `scriptPubKey` to `scriptSig` returns *true*. The `scriptSig` is cryptographically bound to $\pi_{\mathtt{POUR}}$.

Balance A *Pour transfer* can be seen, from the perspective of the transaction, as an input and an output simultaneously. $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ takes value from the value pool and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ adds value to the value pool. As a result, $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ is treated like an *output* value, whereas $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ is treated like an *input* value.

Note that unlike original **Zerocash** [2], **Zcash** does not have a distinction between Mint and Pour transfers. The addition of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ to a *Pour description* subsumes the functionality of Mint. Also, *Pour descriptions* are indistinguishable regardless of the number of real input *coins*.

Commitments and Serials A *transaction* that contains one or more *Pour descriptions*, when entered into the blockchain, appends to the *coin commitment tree* with all constituent *coin commitments*. All of the constituent *serial numbers* are also entered into the *spent serial numbers map* of the *blockchain view and mempool*. A *transaction* is not valid if it attempts to add a *serial number* to the *spent serial numbers map* that already exists in the map.

3.1 Pour Circuit and Proofs

In **Zcash**, $\mathrm{N}^{\mathsf{old}}$ and $\mathrm{N}^{\mathsf{new}}$ are both 2.

A valid instance of $\pi_{\mathtt{POUR}}$ assures that given a *primary input* $(\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}})$, a witness of *auxiliary input* $(\mathsf{path}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{c}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}, \mathbf{c}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \varphi)$ exists, where:

for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$: $\mathbf{c}^{\mathsf{old}}_i = (\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}, \mathsf{v}^{\mathsf{old}}_i, \rho^{\mathsf{old}}_i, \mathsf{r}^{\mathsf{old}}_i)$

for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathbf{c}^{\mathsf{new}}_i = (\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i)$

The following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{v}^{\mathsf{old}}_i \neq 0$: path_i must be a valid path of depth d from $\mathtt{CoinCommitment}(\mathbf{c}^{\mathsf{old}}_i)$ to Coin commitment merkle tree root rt.

Balance $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} + \sum_{i=1}^{\mathrm{N}^{\mathsf{old}}} \mathsf{v}^{\mathsf{old}}_i = \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} + \sum_{i=1}^{\mathrm{N}^{\mathsf{new}}} \mathsf{v}^{\mathsf{new}}_i$.

Serial integrity for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathsf{sn}^{\mathsf{old}}_i = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(\rho^{\mathsf{old}}_i)$.

Spend authority for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$: $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i} = \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(0)$.

Non-malleability for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$: $\mathsf{h}_i = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(i, \mathsf{h}_{\mathsf{Sig}})$

Uniqueness of ρ^{new}_i for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\rho^{\mathsf{new}}_i = \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h}_{\mathsf{Sig}})$

Commitment integrity for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathsf{cm}_i^{\mathsf{new}} = \mathtt{CoinCommitment}(\mathbf{c}_i^{\mathsf{new}})$

4 Encoding Addresses, Private keys, Coins, and Pour descriptions

This section describes how **Zcash** encodes public addresses, private keys, coins, and *Pour descriptions*.

Addresses, keys, and coins, can be encoded as a byte string; this is called the *raw encoding*. This byte string can then be further encoded using Base58Check. The Base58Check layer is the same as for upstream **Bitcoin** addresses [1].

SHA-256 compression function outputs are always represented as strings of 32 bytes.

The language consisting of the following encoding possibilities is prefix-free.

4.1 Transparent Public Addresses

These are encoded in the same way as in **Bitcoin** [1].

4.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [1].

4.3 Confidential Public Addresses

A *payment address* consists of $\mathsf{a_{pk}}$ and $\mathsf{pk_{enc}}$. $\mathsf{a_{pk}}$ is a SHA-256 compression function output. $\mathsf{pk_{enc}}$ is a Curve25519 public key, for use with the encryption scheme defined in section "In-band secret distribution".

4.3.1 Raw Encoding

The raw encoding of a confidential address consists of:

0x92	$\mathsf{a_{pk}}$ (32 bytes)	A 32-byte encoding of $\mathsf{pk_{enc}}$

- A byte, **0x92**, indicating this version of the raw encoding of a **Zcash** public address.
- 32 bytes specifying $\mathsf{a_{pk}}$.
- 32 bytes specifying $\mathsf{pk_{enc}}$, using the normal encoding of a Curve25519 public key [4].

Daira: check that this lead byte is distinct from other Bitcoin stuff, and produces 'z' as the Base58Check leading character.

Nathan: what about the network version byte?

4.4 Confidential Address Secrets

A confidential address secret consists of $\mathsf{a_{sk}}$ and $\mathsf{sk_{enc}}$. $\mathsf{a_{sk}}$ is a SHA-256 compression function output. $\mathsf{sk_{enc}}$ is a Curve25519 private key, for use with the encryption scheme defined in section "In-band secret distribution".

4.4.1 Raw Encoding

The raw encoding of a confidential address secret consists of, in order:

0x93	a_{sk} (32 bytes)	sk_{enc} (32 bytes)

- A byte **0x93** indicating this version of the raw encoding of a **Zcash** private key.
- 32 bytes specifying a_{sk}.
- 32 bytes specifying sk_{enc}.

Daira: check that this lead byte is distinct from other Bitcoin stuff, and produces 'z' as the Base58Check leading character.

Nathan: what about the network version byte?

4.5 Coins

Transmitted coins are stored on the blockchain in encrypted form, together with a *coin commitment* cm.

The *coin plaintexts* associated with a *Pour description* are encrypted to the respective *transmission* keys $\mathsf{pk}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, and the result forms a *transmitted coins ciphertext*.

Each *coin plaintext* consists of $(\mathsf{v}, \rho, \mathsf{r}, \mathsf{memo})$, where:

- v is a 64-bit unsigned integer representing the value of the *coin* in *zatoshi* (1 **ZEC** $= 10^8$ *zatoshi*).
- ρ is a 32-byte $\mathsf{PRF}^{\mathsf{sn}}_{a_{sk}}$ preimage.
- r is a 48-byte *COMM trapdoor*.
- memo is a 64-byte *memo field* associated with this *coin*.

The usage of the *memo field* is by agreement between the sender and recipient of the *coin*. It should be encoded as a UTF-8 human-readable string [5], padded with zero bytes. Wallet software is expected to strip any trailing zero bytes and then display the resulting UTF-8 string to the recipient user, where applicable. Incorrect UTF-8-encoded byte sequences should be displayed as replacement characters (U+FFFD). This does not preclude uses of the *memo field* by automated software, but specification of such usage is not in the scope of this document.

Note that the value s described as being part of a *coin* in the **Zerocash** paper is not encoded because the instantiation of $\mathsf{COMM_s}$ does not use it.

4.5.1 Raw Encoding

The raw encoding of a *coin plaintext* consists of, in order:

0x00	v (8 bytes)	ρ (32 bytes)	r (48 bytes)	memo (64 bytes)

- A byte **0x00** indicating this version of the raw encoding of a *coin plaintext*.
- 8 bytes specifying a big-endian encoding of v.
- 32 bytes specifying ρ.
- 48 bytes specifying r.
- 64 bytes specifying memo.

5 Pours (within a transaction on the blockchain)

TBD.

Describe case where there are fewer than $\mathrm{N}^{\mathsf{old}}$ real input coins.

6 Transactions

TBD.

7 Differences from the Zerocash paper

- Instead of ECIES, we use an encryption scheme based on $\mathsf{crypto\_box}$, defined in section "In-band secret distribution".
- Faerie Gold fix (TBD).
- The paper defines a coin as a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \mathsf{\rho}, \mathsf{r}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines it as $(\mathsf{a_{pk}}, \mathsf{v}, \mathsf{\rho}, \mathsf{r})$. This is just a clarification, because the instantiation of $\mathsf{COMM_s}$ in section 5.1 of the paper does not use s, and cm can be computed from the other fields.

8 References

[1] Base58Check encoding. `https://en.bitcoin.it/wiki/Base58Check_encoding`. Accessed: 2016-01-26.

[2] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland) 2014*, pages 459–474. IEEE, 2014.

[3] Daniel Bernstein. Cryptography in NaCl. `https://nacl.cr.yp.to/valid.html`. Accessed: 2016-02-01.

[4] Daniel Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography - PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26*. Springer-Verlag, 2006. Document ID: 4230efdfa673480fc079449d90f322c0. Date: 2006-02-09. `http://cr.yp.to/papers.html#curve25519`.

[5] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2015. `http://www.unicode.org/versions/latest/`.

[6] libsodium documentation: Sealed boxes. `https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html`. Accessed: 2016-02-01.

[7] NIST. FIPS 180-4: Secure Hash Standard (SHS). `http://csrc.nist.gov/publications/PubsFIPS.html#180-4`, August 2015. DOI: 10.6028/NIST.FIPS.180-4.