# Zcash Protocol Specification

Sean Bowe — Daira Hopwood — Taylor Hornby

February 7, 2016

## 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*).

## 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.

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

### 2.2 Cryptographic Functions

$\mathsf{CRH}$ 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.

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

In **Zcash**, the *SHA-256 compression* function is used to construct all three of these functions. The bits `00`, `01` and `10` 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.

$$a_{pk} := \mathsf{PRF}^{\mathsf{addr}}_{a_{sk}}(0) = \mathsf{CRH}\Big(\boxed{\text{256 bit } a_{sk}}\boxed{0}\boxed{0}\boxed{0^{254}}\Big)$$

$$sn := \mathsf{PRF}^{\mathsf{sn}}_{a_{sk}}(\rho) = \mathsf{CRH}\Big(\boxed{\text{256 bit } a_{sk}}\boxed{0}\boxed{1}\boxed{\mathtt{Leading}_{254}(\rho)}\Big)$$

$$h_i := \mathsf{PRF}^{\mathsf{pk}}_{a_{sk}}(i, h_{Sig}) = \mathsf{CRH}\Big(\boxed{\text{256 bit } a_{sk}}\boxed{1}\boxed{0}\boxed{i}\boxed{\mathtt{Leading}_{253}(h_{Sig})}\Big)$$

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

## 2.3 Confidential Addresses and Private Keys

Nathan: This term, *confidential address*, may be confusing by comparison to a "private key". In the latter case the adjective is reminding a user of their responsibility to protect its privacy, but in the case of *confidential address* we want users to know "transfers to this address are confidential, but the address itself *may* be published or kept confidential depending on your needs. Two different people can compare addresses to know they have the same *confidential address*."

A key pair $(\mathsf{addr}_{pk}, \mathsf{addr}_{sk})$ is generated by users who wish to receive coins under this scheme. The tuple parts embody two distinct keypairs used for different purposes called the *spend authority* and the *key-private encryption* keypair. The *confidential address* $\mathsf{addr}_{pk}$ is a tuple $(a_{pk}, \mathsf{pk}_{enc})$, containing the public components of the *spend authority* and *key-private encryption* respectively. The $\mathsf{addr}_{sk}$ is a tuple $(a_{sk}, \mathsf{sk}_{enc})$, containing the secret components respectively.

Nathan: A diagram could really help here.

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 *confidential address* for each payer.

## 2.4 Coins

A *coin* (denoted $\mathbf{c}$) is a tuple $(a_{pk}, v, \rho, r)$ which represents that a value $v$ is spendable by the recipient who holds the *spend authority* key pair $(a_{pk}, a_{sk})$ such that $a_{pk} = \mathsf{PRF}^{\mathsf{addr}}_{a_{sk}}(0)$. $\rho$ and $r$ are tokens randomly generated by the sender. Only a hash of these values is disclosed publicly, which allows these random tokens to blind the value and recipient *except* to those who possess these tokens.

**In-band secret distribution** In order to transmit the secret $v$, $\rho$, 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 *key-private encryption* 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 $a_{pk}$ and $\mathsf{sk}_{enc}$) is used to reconstruct the original *coin* and *memo field*.

The encryption algorithm is defined in terms of crypto_box (i.e. crypto_box_curve25519xsalsa20poly1305) [**?**] as follows.

Define $\mathsf{nonce}(i, \mathsf{pk}_{eph}, \mathsf{pk}_{enc,i}) = \mathsf{blake2b}\Big(\boxed{\text{1 byte } i-1}\boxed{\text{32 byte } \mathsf{pk}_{eph}}\boxed{\text{32 byte } \mathsf{pk}_{enc,i}}\Big)$.

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

Then to encrypt:

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

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

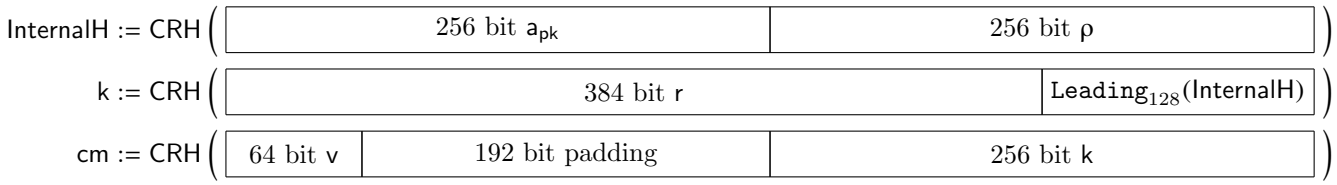
Then for each $i$ in $\{1..N^{new}\}$, the recipient will attempt to decrypt that ciphertext component as follows:

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

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

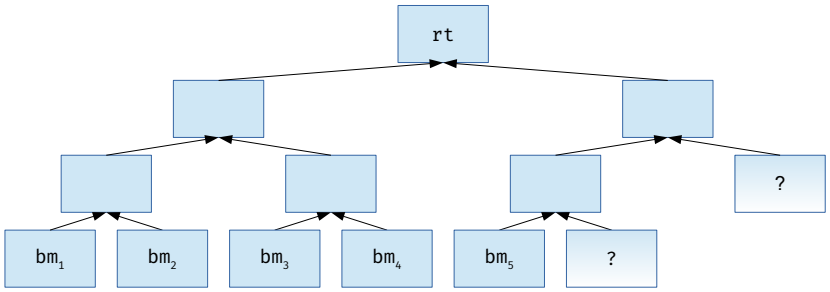
(This is a variation on the crypto_box_seal algorithm defined in libsodium [**?**], but with a single ephemeral key used for all encryptions in a given *Pour description*, and with the nonce for each ciphertext component depending on the index $i$.)

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




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

## 2.5 Coin Commitment Tree




The *coin commitment tree* is an *incremental merkle tree* of depth $\mathsf{d}$ 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 $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$. 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 $N^{old}$ *coins* $\mathbf{c}^{old}_{1..N^{old}}$ and creates $N^{new}$ *coins* $\mathbf{c}^{new}_{1..N^{new}}$. **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 $v^{old}_{pub}$ that the *Pour transfer* removes from the value pool.

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

`anchor` which is a merkle root $\mathsf{rt}$ 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. The `SHA256Compress` hash of this value is $\mathsf{h}_{Sig}$.

Daira: Why `SHA256Compress` and not *SHA-256*? The script is variable-length.

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

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

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

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

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

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

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

**Merkle root validity** A *Pour description* is valid if $\mathsf{rt}$ 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. $v^{old}_{pub}$ takes value from the value pool and $v^{new}_{pub}$ adds value to the value pool. As a result, $v^{old}_{pub}$ is treated like an *output* value, whereas $v^{new}_{pub}$ is treated like an *input* value.

**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**, $N^{old}$ and $N^{new}$ are both 2.

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

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

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

The following conditions hold:

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

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

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

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

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

**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 *confidential address* consists of $\mathsf{a}_{\mathsf{pk}}$ and $\mathsf{pk}_{\mathsf{enc}}$. $\mathsf{a}_{\mathsf{pk}}$ is a SHA-256 compression function output. $\mathsf{pk}_{\mathsf{enc}}$ 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 33-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 [3].

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** | $\mathsf{a_{sk}}$ (32 bytes) | $\mathsf{sk_{enc}}$ (32 bytes) |

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

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* $\mathsf{cm}$.

A *transmitted coins ciphertext* is an encryption of a *coin plaintext* to a *key-private encryption* key $\mathsf{pk_{enc}}$.

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

- $\mathsf{v}$ is a 64-bit unsigned integer representing the value of the *coin* in *zatoshi* (1 **ZEC** $= 10^8$ *zatoshi*).
- $\rho$ is a 32-byte $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$ preimage.
- $\mathsf{r}$ is a 48-byte *COMM trapdoor*.
- $\mathsf{memo}$ 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 [**?**], 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 $\mathsf{s}$ 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.

### 4.6 Raw Encoding

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

| | | | |
|---|---|---|---|
| **0x00** | $\mathsf{v}$ (8 bytes) | $\rho$ (32 bytes) | $\mathsf{r}$ (48 bytes) |

- A byte **0x00** indicating this version of the raw encoding of a *coin plaintext*.
- 8 bytes specifying a big-endian encoding of $\mathsf{v}$.
- 32 bytes specifying $\rho$.
- 48 bytes specifying $\mathsf{r}$.

## 5 Pours (within a transaction on the blockchain)

TBD.

## 6 Transactions

TBD.

## 7 Differences from the Zerocash paper

- Instead of ECIES, we use an encryption scheme based on crypto_box, defined in section "In-band secret distribution".
- Faerie Gold fix (TBD).
- The paper defines a coin as a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines it as $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$. This is just a clarification, because the instantiation of $\mathsf{COMM_s}$ in section 5.1 of the paper does not use $\mathsf{s}$, and $\mathsf{cm}$ 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. Curve25519: new Diffie-Hellman speed records. In *Proceedings of PKC 2006*. Document ID: 4230efdfa673480fc079449d90f322c0. Date: 2006-02-09. `http://cr.yp.to/papers.html#curve25519`.