Zcash Protocol Specification
Version 2.0-draft-2

Sean Bowe — Daira Hopwood — Taylor Hornby

March 2, 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 Caution

Zcash security depends on consensus. Should your program diverge from consensus, its security is weakened or destroyed. The cause of the divergence doesn't matter: it could be a bug in your program, it could be an error in this documentation which you implemented as described, or it could be you do everything right but other software on the network behaves unexpectedly. The specific cause will not matter to the users of your software whose wealth is lost.

Having said that, a specification of *intended* behaviour is essential for security analysis, understanding of the protocol, and maintenance of Zcash Core and related software. If you find any mistake in this specification, please contact `<security@z.cash>`. While the production **Zcash** network has yet to be launched, please feel free to do so in public even if you believe the mistake may indicate a security weakness.

3 Conventions

3.1 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 (except in the definition of AEAD_CHACHA20_POLY1305 [7] which internally uses length fields encoded as little-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.

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

The notation 1..N, used as a subscript, means the sequence of values with indices 1 through N inclusive. For example, $\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1..\mathrm{N}^{\mathsf{new}}}$ means the sequence $[\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1}, \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},2}, \ldots\ \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},\mathrm{N}^{\mathsf{new}}}]$.

3.2 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. [8]

PRF_x is a pseudo-random function seeded by x. Five *independent* PRF_x are needed in our scheme: $\mathsf{PRF}^{\mathsf{addr}}_x$, $\mathsf{PRF}^{\mathsf{sn}}_x$, $\mathsf{PRF}^{\mathsf{pk}}_x$, PRF^{ρ}_x, and $\mathsf{PRF}^{\mathsf{dk}}_x$.

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 five of these functions. The bits `0000`, `0001`, `001x`, `010x`, and `011x` are included (respectively) within the blocks that are hashed, ensuring that the functions are independent.

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

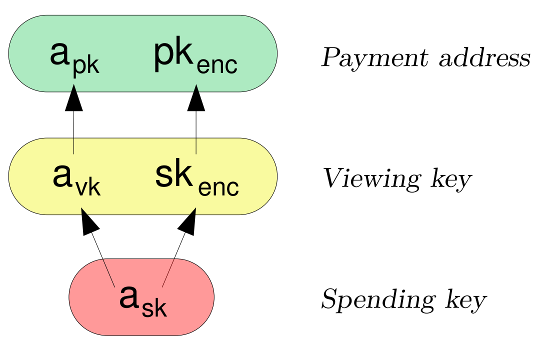
$\mathsf{PRF}^{\mathsf{addr}}_{x}(t)$	$:= \mathsf{CRH}\big($	0	0	0	0	252 bit x	0^{254}	2 bit t	$\big)$
$\mathsf{sn} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$	$:= \mathsf{CRH}\big($	0	0	0	1	252 bit $\mathsf{a_{sk}}$	256 bit ρ		$\big)$
$\mathsf{h}_i = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}})$	$:= \mathsf{CRH}\big($	0	0	1	i-1	252 bit $\mathsf{a_{sk}}$	256 bit $\mathsf{h_{Sig}}$		$\big)$
$\rho^{\mathsf{new}}_i = \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}})$	$:= \mathsf{CRH}\big($	0	1	0	i-1	252 bit φ	256 bit $\mathsf{h_{Sig}}$		$\big)$
$\mathsf{K}^{\mathsf{disclose}}_i = \mathsf{PRF}^{\mathsf{dk}}_{\mathsf{a_{vk}}}(i, \mathsf{h_{Sig}})$	$:= \mathsf{CRH}\big($	0	1	1	i-1	252 bit $\mathsf{a_{vk}}$	256 bit $\mathsf{h_{Sig}}$		$\big)$

4 Concepts

4.1 Payment Addresses, Viewing Keys, and Spending Keys

A *key tuple* $(\mathsf{addr_{sk}}, \mathsf{addr_{vk}}, \mathsf{addr_{pk}})$ is generated by users who wish to receive payments under this scheme. The *viewing key* $\mathsf{addr_{vk}}$ is derived from the *spending key* $\mathsf{addr_{sk}}$, and the *payment address* $\mathsf{addr_{pk}}$ is derived from the *viewing key*.

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.



Note that a *spending key* holder can derive the other components, and a *viewing key* holder can derive $(\mathsf{a_{pk}}, \mathsf{pk_{enc}})$, even though these components are not formally part of the respective keys. Implementations MAY cache these derived components, provided that they are deleted if the corresponding source 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 *payment address* from a *viewing key*; and
- obtain a *payment address* or *viewing key* from a *spending key*.

$\mathsf{a_{sk}}$ and $\mathsf{a_{vk}}$ are each 252 bits. $\mathsf{a_{pk}}$, $\mathsf{sk_{enc}}$, and $\mathsf{pk_{enc}}$, are each 256 bits.

$\mathsf{a_{vk}}$, $\mathsf{a_{pk}}$, $\mathsf{sk_{enc}}$, and $\mathsf{pk_{enc}}$ are derived as follows:

$$\begin{aligned}
\mathsf{a_{vk}} &:= \mathtt{Trailing}_{252}(\mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a_{sk}}}(0)) \\
\mathsf{a_{pk}} &:= \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a_{vk}}}(1) \\
\mathsf{sk_{enc}} &:= \mathsf{clamp}_{\mathsf{Curve25519}}(\mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a_{sk}}}(2)) \\
\mathsf{pk_{enc}} &:= \mathsf{Curve25519}(\mathsf{sk_{enc}}, \underline{9})
\end{aligned}$$

where $\mathsf{clamp}_{\mathsf{Curve25519}}$ performs the clamping of Curve25519 private key bits, $\mathsf{Curve25519}$ performs point multiplication, and $\underline{9}$ is the public string representing a base point, all as defined in [3].

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.

4.2 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 *spending key* $\mathsf{a_{sk}}$ corresponding to $\mathsf{a_{pk}}$, as described in the previous section.

- $\mathsf{a_{pk}}$ is a 32-byte *authorization* public key of the recipient.
- 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 $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$ preimage.
- r is a 32-byte *COMM trapdoor*.

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.

Note that the value s described as being part of a *coin* in the **Zerocash** paper [2] is not encoded because the instantiation of $\mathsf{COMM_s}$ does not use it.

4.2.1 Coin Commitments

The underlying v and $\mathsf{a_{pk}}$ are blinded with ρ and r using the collision-resistant hash function $\mathtt{SHA256}$. The resulting hash $\mathsf{cm} = \mathtt{CoinCommitment}(\mathbf{c})$.

$$\mathsf{cm} := \mathtt{SHA256}\left(\begin{array}{|c|c|c|c|c|}\hline \mathbf{0xF0} & 256\text{ bit } \mathsf{a_{pk}} & 64\text{ bit } \mathsf{v} & 256\text{ bit } \rho & 256\text{ bit } \mathsf{r} \\ \hline\end{array}\right)$$

4.2.2 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.

4.2.3 Coin plaintexts and memo fields

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

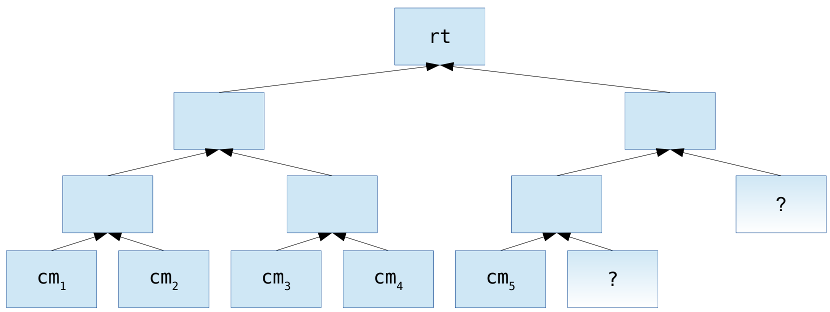
The *coin plaintexts* associated with a *Pour description* are encrypted to the respective *transmission* keys $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, and the result forms part of a *transmitted coins ciphertext* (see section "In-band secret distribution" for further details).

Each *coin plaintext* (denoted **cp**) consists of $(\mathsf{a}_{\mathsf{pk}}, \mathsf{v}, \rho, \mathsf{r}, \mathsf{memo})$.

The first four of these fields are as defined earlier. 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 [4], 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.

4.3 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.

4.4 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.

4.5 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.

5 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* $\mathbf{c}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$ and creates $\mathrm{N}^{\mathsf{new}}$ *coins* $\mathbf{c}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$. **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 $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ that the *Pour transfer* removes from the value pool.

`vpub_new` which is a value $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ 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 $\mathsf{sn}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$.

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

`ephemeralKey` which is a Curve25519 public key epk.

`encCiphertexts` which is a $\mathrm{N}^{\mathsf{new}}$ size sequence of ciphertext components, $\mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$.

`discloseCiphertexts` which is a $\mathrm{N}^{\mathsf{old}}$ size sequence of ciphertext components, $\mathbf{C}^{\mathsf{disclose}}_{1..\mathrm{N}^{\mathsf{old}}}$.

`sharedCiphertext` which is the ciphertext component $\mathbf{C}^{\mathsf{shared}}$.
(The preceding four fields together form the *transmitted coins ciphertext*.)

`randomSeed` which is a random 256-bit seed $\mathsf{randomSeed}$.

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

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

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

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

$$\mathsf{h_{Sig}} := \mathtt{SHA256}\left(\boxed{\mathbf{0xF1}}\,\boxed{256\text{ bit } \mathsf{sn}^{\mathsf{old}}_{0}}\,\boxed{\cdots}\,\boxed{256\text{ bit } \mathsf{sn}^{\mathsf{old}}_{\mathrm{N}^{\mathsf{old}}-1}}\,\boxed{\mathsf{randomSeed}}\,\boxed{\mathtt{scriptPubKey}}\right)$$

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.

5.1 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*:

$$(\mathsf{rt}, \mathsf{sn}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h_{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathbf{C}^{\mathsf{disclose}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{C}^{\mathsf{shared}}),$$

there exists a witness of *auxiliary input*:

$$(\mathsf{path}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{c}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}, \mathsf{a}^{\mathsf{old}}_{\mathsf{vk},1..\mathrm{N}^{\mathsf{old}}}, \mathbf{cp}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \varphi, \mathsf{K}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{K}^{\mathsf{disclose}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{K}^{\mathsf{shared}}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}, \mathsf{esk})$$

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 $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathbf{cp}^{\mathsf{new}}_i = (\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i, \mathsf{memo}_i)$, and $\mathbf{P}^{\mathsf{enc}}_i$ is a raw encoding of $\mathbf{cp}^{\mathsf{new}}_i$;

such that 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 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 $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$: $\mathsf{a}^{\mathsf{old}}_{\mathsf{vk},i} = \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(0)$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i} = \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{vk},i}}(1)$.

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

$\mathbf{C}^{\mathsf{enc}}$ integrity for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathbf{C}^{\mathsf{enc}}_i = \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathbf{P}^{\mathsf{enc}}_i)$.

$\mathbf{C}^{\mathsf{disclose}}$ integrity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$: $\mathbf{C}^{\mathsf{disclose}}_i = \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{disclose}}_i}(\mathbf{P}^{\mathsf{disclose}}_i)$ and $\mathsf{K}^{\mathsf{disclose}}_i = \mathsf{PRF}^{\mathsf{dk}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{vk},i}}(i, \mathsf{h}_{\mathsf{Sig}})$

where $\mathbf{P}^{\mathsf{disclose}}_i =$

256 bit $\mathsf{K}^{\mathsf{shared}}$	64 bit $\mathsf{v}^{\mathsf{old}}_i$

.

$\mathbf{C}^{\mathsf{shared}}$ integrity $\mathbf{C}^{\mathsf{shared}} = \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{shared}}}(\mathbf{P}^{\mathsf{shared}})$

where $\mathbf{P}^{\mathsf{shared}} =$

256 bit $\mathsf{K}^{\mathsf{enc}}_1$	$\cdots$	256 bit $\mathsf{K}^{\mathsf{enc}}_{\mathrm{N}^{\mathsf{new}}}$
256 bit $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1}$	$\cdots$	256 bit $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},\mathrm{N}^{\mathsf{new}}}$
256 bit esk		

Note: $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, esk, and $\mathsf{memo}_{1..\mathrm{N}^{\mathsf{new}}}$ are intentionally not constrained. This implies that for the $\mathbf{C}^{\mathsf{enc}}$ and $\mathbf{C}^{\mathsf{shared}}$ integrity constraints, the circuit need not compute ChaCha20 blocks that are only used to encrypt those fields (although the Poly1305 authenticator must be computed over the whole of each ciphertext).

6 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 $\mathsf{pk}_{\mathsf{enc}}$ is used to encrypt these secrets. The recipient's possession of the associated $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ (which contains both a_{pk} and $\mathsf{sk}_{\mathsf{enc}}$) is used to reconstruct the original *coin* and *memo field*.

Several more encryptions are used to also reveal these values to a holder of a *viewing key* for any of the input *coins*, and also to permit them to check whether the other encryptions are valid.

All of the resulting ciphertexts are combined to form a *transmitted coins ciphertext*.

6.1 Encryption

Let $\mathsf{SymEncrypt}_{\mathsf{K}}(\mathbf{P})$ be the $\mathsf{AEAD\_CHACHA20\_POLY1305}$ [7] encryption of plaintext $\mathbf{P}$ with empty "associated data", all-zero nonce, and key K.

Similarly, let $\mathsf{SymDecrypt}_{\mathsf{K}}(\mathbf{C})$ be the $\mathsf{AEAD\_CHACHA20\_POLY1305}$ decryption of ciphertext $\mathbf{C}$ with empty "associated data", all-zero nonce, and key K. The result is either the plaintext byte sequence, or $\perp$ indicating failure to decrypt.

Define:

$$\mathsf{KDF}(\mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, i) := \mathtt{SHA256}\left(\begin{array}{|c|c|c|c|}\hline \text{256 bit } \mathsf{dhsecret}_i & \text{256 bit } \mathsf{epk} & \text{256 bit } \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i} & \text{8 bit } i-1 \\ \hline \end{array}\right).$$

Let $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$ be the Curve25519 public keys for the intended recipient addresses of each new *coin*, let $\mathsf{a}^{\mathsf{old}}_{\mathsf{vk},1..\mathrm{N}^{\mathsf{old}}}$ be the *disclosure key* for each of the addresses from which the old *coins* are sent, and let $\mathbf{cp}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *coin plaintexts*.

Then to encrypt:

- Generate a new Curve25519 (public, private) key pair $(\mathsf{epk}, \mathsf{esk})$, and a new $\mathsf{AEAD\_CHACHA20\_POLY1305}$ key $\mathsf{K}^{\mathsf{shared}}$.
- For i in $\{1..\mathrm{N}^{\mathsf{new}}\}$,
  - Let $\mathbf{P}^{\mathsf{enc}}_i$ be the raw encoding of $\mathbf{cp}_i$.
  - Let $\mathsf{dhsecret}_i := \mathsf{Curve25519}(\mathsf{esk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
  - Let $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}(\mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, i)$.
  - Let $\mathbf{C}^{\mathsf{enc}}_i := \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathbf{P}^{\mathsf{enc}}_i)$.
- For i in $\{1..\mathrm{N}^{\mathsf{old}}\}$,
  - Let $\mathbf{P}^{\mathsf{disclose}}_i := \begin{array}{|c|c|}\hline \text{256 bit } \mathsf{K}^{\mathsf{shared}} & \text{64 bit } \mathsf{v}^{\mathsf{old}}_i \\ \hline \end{array}$.
  - Let $\mathsf{K}^{\mathsf{disclose}}_i := \mathsf{PRF}^{\mathsf{dk}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{vk},i}}(i, \mathsf{h}_{\mathsf{Sig}})$.
  - Let $\mathbf{C}^{\mathsf{disclose}}_i := \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{disclose}}_i}(\mathbf{P}^{\mathsf{disclose}}_i)$.
- Let $\mathbf{P}^{\mathsf{shared}} := \begin{array}{|c|c|c|}\hline \text{256 bit } \mathsf{K}^{\mathsf{enc}}_1 & \cdots & \text{256 bit } \mathsf{K}^{\mathsf{enc}}_{\mathrm{N}^{\mathsf{new}}} \\ \hline \text{256 bit } \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1} & \cdots & \text{256 bit } \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},\mathrm{N}^{\mathsf{new}}} \\ \hline \text{256 bit } \mathsf{esk} \\ \cline{1-1} \end{array}$
- Let $\mathbf{C}^{\mathsf{shared}} := \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{shared}}}(\mathbf{P}^{\mathsf{shared}})$.

The resulting *transmitted coins ciphertext* is $(\mathsf{epk}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathbf{C}^{\mathsf{disclose}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{C}^{\mathsf{shared}})$.

6.2 Decryption by a Recipient

Let $(\mathsf{pk_{enc}}, \mathsf{sk_{enc}})$ be the recipient's Curve25519 (public, private) key pair, and let $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ be the coin commitments of each output coin. Then for each i in $\{1..\mathrm{N}^{\mathsf{new}}\}$, the recipient will attempt to decrypt that ciphertext component as follows:

- Let $\mathsf{dhsecret}_i := \mathsf{Curve25519}(\mathsf{sk_{enc}}, \mathsf{epk})$.
- Let $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}(\mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, i)$.
- Return $\mathtt{DecryptCoin}(\mathsf{K}^{\mathsf{enc}}_i, \mathbf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i)$.

$\mathtt{DecryptCoin}(\mathsf{K}^{\mathsf{enc}}_i, \mathbf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i)$ is defined as follows:

- Let $\mathbf{P}^{\mathsf{enc}}_i := \mathsf{SymDecrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathbf{C}^{\mathsf{enc}}_i)$.
- If $\mathbf{P}^{\mathsf{enc}}_i = \bot$, return $\bot$.
- Extract $\mathbf{cp}_i = (\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i, \mathsf{memo}_i)$ from $\mathbf{P}^{\mathsf{enc}}_i$.
- If $\mathtt{CoinCommitment}((\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i)) \neq \mathsf{cm}^{\mathsf{new}}_i$, return $\bot$, else return $\mathbf{cp}_i$.

Note that this corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in Figure 2 of [2].

To test whether a *coin* is unspent in a particular *blockchain view* also requires the *authorization* private key $\mathsf{a_{sk}}$; the coin is unspent if and only if $\mathsf{sn} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$ is not in the *spent serial number set* for that *blockchain view*.

Note that a coin may change from being unspent to spent on a given *blockchain view*, as transactions are added to that view. Also, blockchain reorganisations may cause the transaction in which a coin was output to no longer be on the consensus blockchain.

6.3 Decryption by a Viewing Key Holder

A *viewing key* holder also acts as a recipient using its $\mathsf{sk_{enc}}$ key component. How to decrypt transactions using this key component is described in the preceding section. The following applies to decryption using the $\mathsf{a_{vk}}$ component of the *viewing key*.

Let $\mathsf{a_{vk}}$ be a *viewing key* holder's *disclosure key*. Then for each *Pour description* in its *blockchain view*, the *viewing key* holder will attempt to decrypt the corresponding *transmitted coins ciphertext* as follows:

1. For i in $\{1..\mathrm{N}^{\mathsf{old}}\}$,
   - Let $\mathsf{K}^{\mathsf{disclose}}_i := \mathsf{PRF}^{\mathsf{dk}}_{\mathsf{a_{vk}}}(i, \mathsf{h_{Sig}})$.
   - Let $\mathbf{P}^{\mathsf{disclose}}_i := \mathsf{SymDecrypt}_{\mathsf{K}^{\mathsf{disclose}}_i}(\mathbf{C}^{\mathsf{disclose}}_i)$.
   - If $\mathbf{P}^{\mathsf{disclose}}_i = \bot$ then set $\mathbf{P}^{\mathsf{shared}}_i := \bot$ and $\mathsf{v}^{\mathsf{old}}_i := \bot$, and continue with the next i.
   - Extract $\mathsf{K}^{\mathsf{shared}}_i$ and $\mathsf{v}^{\mathsf{old}}_i$ from $\mathbf{P}^{\mathsf{disclose}}_i$.
   - Let $\mathbf{P}^{\mathsf{shared}}_i := \mathsf{SymDecrypt}_{\mathsf{K}^{\mathsf{shared}}_i}(\mathbf{C}^{\mathsf{shared}})$.
2. If $\mathbf{P}^{\mathsf{shared}}_i = \bot$ for all i in $\{1..\mathrm{N}^{\mathsf{old}}\}$, then set $\mathbf{cp}_i = \bot$ for i in $\{1..\mathrm{N}^{\mathsf{new}}\}$ and return $(\mathsf{v}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{cp}_{1..\mathrm{N}^{\mathsf{new}}})$.
3. Otherwise, let $\mathbf{P}^{\mathsf{shared}}$ be the first non-$\bot$ value in $\mathbf{P}^{\mathsf{shared}}_{1..\mathrm{N}^{\mathsf{old}}}$.
4. Extract $\mathsf{K}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$, $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, and esk from $\mathbf{P}^{\mathsf{shared}}$.

5. For i in $\{1..\mathrm{N}^{\mathsf{new}}\}$,

   - Let $\mathbf{cp}_i := \mathtt{DecryptCoin}(\mathsf{K}_i^{\mathsf{enc}}, \mathbf{C}_i^{\mathsf{enc}}, \mathsf{cm}_i^{\mathsf{new}})$.
   - Let $\mathsf{epk}^* := \mathsf{Curve25519}(\mathsf{esk}, \underline{9})$.
   - Let $\mathsf{dhsecret}_i := \mathsf{Curve25519}(\mathsf{esk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
   - Let $\mathsf{K}_i^* := \mathsf{KDF}(\mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}, i)$.
   - If $\mathbf{cp}_i \neq \bot$ and either $(\mathsf{K}_i^* \neq \mathsf{K}_i^{\mathsf{enc}}$ or $\mathsf{epk}^* \neq \mathsf{epk})$, then set the *memo field* of $\mathbf{cp}_i$ to be $\bot$ (indicating that, although this is a valid coin, the recipient would not have been able to decrypt it, and that the *memo field* cannot be verified).

6. Return $(\mathsf{v}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{cp}_{1..\mathrm{N}^{\mathsf{new}}})$.

Note: The above algorithm is not constant-time. An equivalent but constant-time algorithm should be used whenever it is desirable to avoid leakage of which ciphertext components were decryptable.

If a party holds more than one *viewing key*, it may optimize the above procedure by performing the loop in step 1 for the $\mathsf{a_{vk}}$ of each *viewing key*. It may be assumed that the first $\mathbf{P}_i^{\mathsf{shared}}$ that decrypts correctly is the one that should be used in step 3 onward. (However, additional information is provided by which *viewing key* was able to decrypt each $\mathbf{C}_i^{\mathsf{disclose}}$.)

The public key encryption used in this part of the protocol is based loosely on other encryption schemes based on Diffie-Hellman over an elliptic curve, such as ECIES or the $\mathsf{crypto\_box\_seal}$ algorithm defined in libsodium [6]. Note that:

- The same ephemeral key is used for all encryptions to the recipient keys in a given *Pour description*.
- In addition to the Diffie-Hellman secret, the KDF takes as input the public keys of both parties, and the index i.
- The nonce parameter to $\mathsf{AEAD\_CHACHA20\_POLY1305}$ is not used.
- The ephemeral secret esk is included together with the *transmission* public keys of the recipients, symmetrically encrypted to the *disclosure key*. This allows a *viewing key* holder to check whether the indicated recipients would be able to decrypt a given component, and if so to decrypt the memo field. (We do not rely on this to ensure that a *viewing key* holder can decrypt the other components of the output coins; instead, those are symmetrically encrypted to the *viewing key* and the correctness of this encryption is checked by the *POUR circuit*.)

7 Encoding Addresses, Keys, and Coin plaintexts

This section describes how **Zcash** encodes *payment addresses*, *spending keys*, *viewing keys*, *coin plaintexts*, 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.

7.1 Transparent Payment Addresses

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

7.2 Transparent Private Keys

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

7.3 Private Payment 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".

The raw encoding of a *payment address* consists of:

0x??	256 bit $\mathsf{a_{pk}}$	256 bit $\mathsf{pk_{enc}}$

- A byte, **0x92**, indicating this version of the raw encoding of a **Zcash** public address.
- 256 bits specifying $\mathsf{a_{pk}}$.
- 256 bits 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?

7.4 Spending Keys

A *spending key* consists of $\mathsf{a_{sk}}$.

The raw encoding of a *spending key* consists of, in order:

0x??	0^4	252 bit $\mathsf{a_{sk}}$

- A byte **0x**?? indicating this version of the raw encoding of a **Zcash** *spending key*.
- 4 zero padding bits.
- 252 bits specifying $\mathsf{a_{sk}}$.

Note that, consistent with big-endian encoding, the zero padding occupies the high-order 4 bits of the second byte.

Daira: check that this lead byte is distinct from other Bitcoin stuff, and produces a suitable Base58Check leading character.

Nathan: what about the network version byte?

7.5 Viewing Keys

A *viewing key* consists of a *disclosure key* $\mathsf{a_{vk}}$, and a *transmission* private key $\mathsf{sk_{enc}}$.

The raw encoding of a *viewing key* consists of, in order:

0x??	0^4	252 bit $\mathsf{a_{vk}}$	256 bit $\mathsf{sk_{enc}}$

- A byte **0x**?? indicating this version of the raw encoding of a **Zcash** *viewing key*.
- 4 zero padding bits.
- 252 bits specifying $\mathsf{a_{vk}}$.
- 256 bits specifying $\mathsf{sk_{enc}}$.

Note that, consistent with big-endian encoding, the zero padding occupies the high-order 4 bits of the second byte.

Daira: check that this lead byte is distinct from other Bitcoin stuff, and produces a suitable Base58Check leading character.

Nathan: what about the network version byte?

7.6 Coin Plaintexts

The raw encoding of a *coin plaintext* $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r}, \mathsf{memo})$ consists of, in order:

0x00	$\mathsf{a_{pk}}$ (32 bytes)	v (8 bytes)	ρ (32 bytes)	r (32 bytes)	memo (64 bytes)

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

8 Differences from the Zerocash paper

8.1 Unification of Mints and Pours

TODO:

8.2 Faerie Gold attack and fix

TODO:

(The name "Faerie Gold" refers to various Celtic legends in which faeries pay mortals in what appears to be gold, but which soon after reveals itself to be leaves, gorse blossoms, gingerbread cakes, or other less valuable things [5].)

8.3 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, v, and ρ. However, the instantiation of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of $\mathsf{a_{pk}}$ and ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct values of ρ with colliding outputs of the truncated hash, and therefore the same *coin commitment*. This would have allowed such an attacker to break the balance property by double-spending coins, potentially creating arbitrary amounts of currency for themself.

Zcash uses a simpler construction with a single `SHA256` evaluation for the commitment. The motivation for the nested construction in **Zerocash** was to allow Mint transactions to be publically verified without requiring a ZK proof (as described under step 3 in section 1.3 of [2]). Since **Zcash** combines "Mint" and "Pour" transactions into a generalized Pour which always uses a ZK proof, it does not require the nesting. A side benefit is that this reduces the number of `SHA256Compress` evaluations needed to compute each *coin commitment* from three to two, saving a total of four `SHA256Compress` evaluations in the *POUR circuit*.

Note that **Zcash** coin commitments are not statistically hiding, and so **Zcash** does not support the "everlasting anonymity" property described in section 8.1 of the **Zerocash** paper [2], even when used as described in that section. While it is possible to define a statistically hiding, computationally binding commitment scheme for this use at a 128-bit security level, the overhead of doing so within the circuit was not considered to justify the benefits.

8.4 Viewing keys

TODO:

8.5 Changes to PRF inputs and truncation

TODO:

8.6 In-band secret distribution

TODO:

8.7 Miscellaneous

- 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 did not use s (and neither does the new instantiation of `CoinCommitment`). cm can be computed from the other fields.

9 Acknowledgements

The inventors of **Zerocash** are Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza.

The authors would like to thank everyone with whom they have discussed the **Zerocash** protocol design; in addition to the inventors, this includes Mike Perry, Isis Lovecruft, Leif Ryge, Andrew Miller, Zooko Wilcox, Nathan Wilcox, Samantha Hulsey, and no doubt others.

Mike Perry, Zooko Wilcox, and Nathan Wilcox contributed to the design of selective transparency features, now called viewing keys.

The Faerie Gold attack was found by Zooko Wilcox. The internal hash collision attack was found by Taylor Hornby.

10 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 *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`.

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

[5] Eddie Lenihan and Carolyn Eve Green. *Meeting the Other Crowd: The Fairy Stories of Hidden Ireland*. 2004. Pages 109–110. ISBN: 1-58542-206-1.

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

[7] Yoav Nir and Adam Langley. Request for Comments 7539: ChaCha20 and Poly1305 for IETF Protocols. Internet Research Task Force (IRTF). `https://tools.ietf.org/html/rfc7539`. As modified by verified errata at `https://www.rfc-editor.org/errata_search.php?rfc=7539`.

[8] 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.