# Zcash Protocol Specification
Version 2.0-draft-4

Sean Bowe — Daira Hopwood — Taylor Hornby — Nathan Wilcox

March 30, 2016

## Contents

# 1 Introduction

**Zcash** is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash** [1] 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

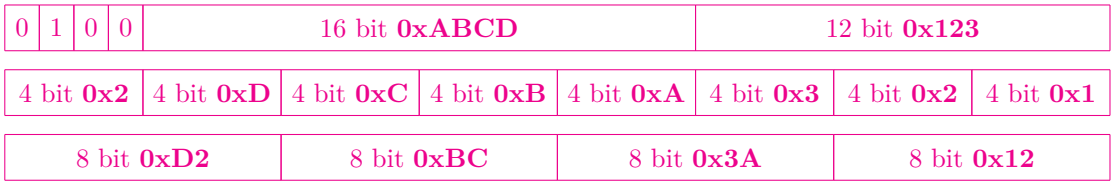
## 3.1 Integers, Bit Sequences, and Endianness

All integers in ***Zcash**-specific* encodings are unsigned, have a fixed bit length, and are encoded as little-endian. The definition of the encryption scheme based on AEAD_CHACHA20_POLY1305 [11] in § 6 *'In-band secret distribution'* on p. 11 uses length fields encoded as little-endian. Also, Curve25519 public and private keys are defined as byte sequences, which are converted from integers using little-endian encoding.

The notation **0x** followed by a string of **boldface** hexadecimal digits represents the corresponding integer converted from hexadecimal.

In bit layout diagrams, each box of the diagram represents a sequence of bits. The bit length is given explicitly in each box, except for the case of a single bit, or for the notation $[0]^n$ which represents the sequence of $n$ zero bits. If the content of the box is a byte sequence, it is implicitly converted to a sequence of bits using little-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 little-endian order. Note that the *least significant* bit of each byte is first, i.e. shown toward the left of the diagram.

For example, the following diagrams are all equivalent:




and represent the byte sequence $[\mathbf{0xD2}, \mathbf{0xBC}, \mathbf{0x3A}, \mathbf{0x12}]$.

$\texttt{Trailing}_k(x)$, where $k$ is an integer, converts the input $x$ to a bit sequence using little-endian order, and returns the trailing (final) $k$ bits of that bit sequence.

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 $[\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1}, \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},2}, \ldots \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},\mathrm{N}^{\mathsf{new}}}]$.

The symbol $\perp$ is used to indicate unavailable information or a failed decryption.

### 3.2 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 sequences. [12]

$\mathsf{PRF}_x$ is a pseudo-random function seeded by $x$. Four *independent* $\mathsf{PRF}_x$ are needed in our scheme: $\mathsf{PRF}^{\mathsf{addr}}_x$, $\mathsf{PRF}^{\mathsf{nf}}_x$, $\mathsf{PRF}^{\mathsf{pk}}_x$, and $\mathsf{PRF}^{\rho}_x$.

It is required that $\mathsf{PRF}^{\mathsf{nf}}_x$ and $\mathsf{PRF}^{\rho}_x$ be collision-resistant across all $x$ — i.e. it should not be feasible to find $(x, y) \neq (x', y')$ such that $\mathsf{PRF}^{\mathsf{nf}}_x(y) = \mathsf{PRF}^{\mathsf{nf}}_{x'}(y')$, and similarly for $\mathsf{PRF}^{\rho}$.

In **Zcash**, the *SHA-256 compression* function is used to construct all of these functions. *SHA-512* is also used to construct a Key Derivation Function.

$$\mathsf{PRF}^{\mathsf{addr}}_x(t) := \mathsf{CRH}\Big(\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{252\text{ bit }x}\boxed{8\text{ bit }t}\boxed{[0]^{248}}\Big)$$

$$\mathsf{nf} = \mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho) := \mathsf{CRH}\Big(\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{252\text{ bit }\mathsf{a_{sk}}}\boxed{256\text{ bit }\rho}\Big)$$

$$\mathsf{h}_i = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}}) := \mathsf{CRH}\Big(\boxed{i\text{-}1}\boxed{0}\boxed{0}\boxed{0}\boxed{252\text{ bit }\mathsf{a_{sk}}}\boxed{256\text{ bit }\mathsf{h_{Sig}}}\Big)$$

$$\rho^{\mathsf{new}}_i = \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}}) := \mathsf{CRH}\Big(\boxed{i\text{-}1}\boxed{0}\boxed{1}\boxed{0}\boxed{252\text{ bit }\varphi}\boxed{256\text{ bit }\mathsf{h_{Sig}}}\Big)$$

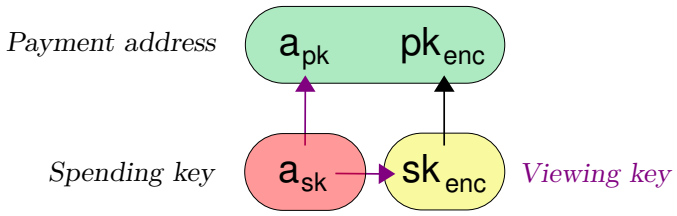
**Note:** The first four bits –i.e. the least significant four bits of the first byte– are used to distinguish different uses of $\mathsf{CRH}$, ensuring that the functions are independent. In addition to the inputs shown here, the bits `0111` in this position are used to distinguish uses of the full *SHA-256* hash function — see § 4.2.1 *'Note Commitments '* on p. 6, § 5.1 *'Computation of* $\mathsf{h_{Sig}}$*'* on p. 9, and § 6 *'In-band secret distribution'* on p. 11. (The specific bit patterns chosen here are motivated by the possibility of future extensions that either increase $\mathrm{N}^{\mathsf{old}}$ and/or $\mathrm{N}^{\mathsf{new}}$ to 3, or that add an additional bit to $\mathsf{a_{sk}}$ to encode a new key type, or that require an additional PRF.)

## 4 Concepts

### 4.1 Payment Addresses and Spending Keys

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

The following diagram depicts the relations between key components. Arrows point from a component to any other component(s) that can be derived from it.




The composition of *payment addresses* 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* or *viewing key* from a *spending key*.

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

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

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

where

- $\mathsf{Curve25519}(\underline{n}, \underline{q})$ performs point multiplication of the Curve25519 public key represented by the byte sequence $\underline{q}$ by the Curve25519 secret key represented by the byte sequence $\underline{n}$, as defined in section 2 of [2];
- $\underline{9}$ is the public byte sequence representing the Curve25519 base point;
- $\mathsf{clamp}_{\mathsf{Curve25519}}(\underline{x})$ takes a 32-byte sequence $\underline{x}$ as input and returns a byte sequence representing a Curve25519 private key, with bits "clamped" as described in section 3 of [2]: "clear bits 0, 1, 2 of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte." Here the bits of a byte are numbered such that bit $b$ has numeric weight $2^b$.

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 Notes

A *note* (denoted $\mathbf{n}$) is a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$ which represents that a value $\mathsf{v}$ 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.
- $\mathsf{v}$ is a 64-bit unsigned integer representing the value of the *note* in *zatoshi* (1 **ZEC** = $10^8$ *zatoshi*).
- $\rho$ is a 32-byte $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}$ preimage.
- $\mathsf{r}$ is a 32-byte *COMM trapdoor*.

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

### 4.2.1 Note Commitments

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


### 4.2.2 Nullifiers

A *nullifier* (denoted $\mathsf{nf}$) is derived from the $\rho$ component of a *note* as $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho)$. A *note* is spent by proving knowledge of $\rho$ and $\mathsf{a_{sk}}$ in zero knowledge while disclosing its *nullifier* $\mathsf{nf}$, allowing $\mathsf{nf}$ to be used to prevent double-spending.

### 4.2.3 Note Plaintexts and Memo Fields

Transmitted *notes* are stored on the blockchain in encrypted form, together with a *note commitment* $\mathsf{cm}$.

The *note plaintexts* associated with a *JoinSplit description* are encrypted to the respective *transmission* keys $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..N^{\mathsf{new}}}$, and the result forms part of a *transmitted notes ciphertext* (see § 6 *'In-band secret distribution'* on p. 11 for further details).

Each *note plaintext* (denoted **np**) consists of $(\mathsf{v}, \rho, \mathsf{r}, \mathsf{memo})$.

The first three of these fields are as defined earlier. memo is a 128-byte *memo field* associated with this *note*.

The usage of the *memo field* is by agreement between the sender and recipient of the *note*. The memo SHOULD be encoded either as:

- a UTF-8 human-readable string [6], padded with zero bytes; or
- an arbitrary sequence of 128 bytes starting with a byte value of **0xF5** or greater, which is therefore not a valid UTF-8 string.

In the former case, 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).

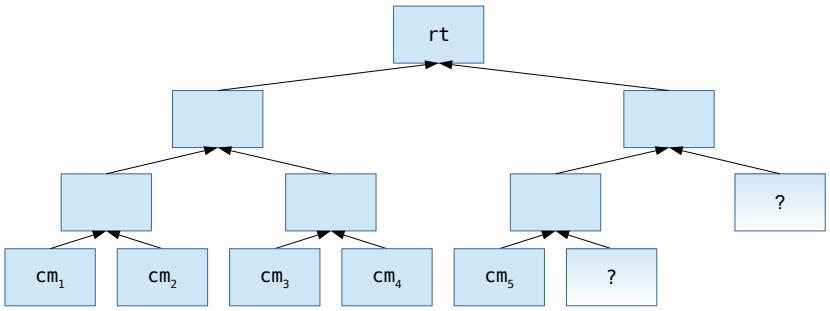
In the latter case, the contents of the *memo field* SHOULD NOT be displayed. A start byte of **0xF5** is reserved for use by automated software by private agreement. A start byte of **0xF6** or greater is reserved for use in future **Zcash** protocol extensions.

The encoding of a *note plaintext* consists of, in order:

| $\mathsf{v}$ (8 bytes) | $\rho$ (32 bytes) | $\mathsf{r}$ (32 bytes) | memo (128 bytes) |
|---|---|---|---|

- 8 bytes specifying $\mathsf{v}$.
- 32 bytes specifying $\rho$.
- 32 bytes specifying $\mathsf{r}$.
- 128 bytes specifying memo.

### 4.3 Note Commitment Tree




The *note commitment tree* is an *incremental merkle tree* of depth d used to store *note commitments* that *JoinSplit operations* 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 *JoinSplit descriptions*' *note commitments* have been entered into the tree associated with the previous block.

### 4.4 Nullifier Set

Transactions insert *nullifiers* into a *nullifier set* 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 *nullifiers* to the *nullifier set*.

Transactions that attempt to insert a *nullifier* into this set 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 *nullifiers* into the *nullifier set*.

### 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 *JoinSplit descriptions*. TODO: They also have a transparent value flow that interacts with the *JoinSplit description*'s $v_{pub}^{old}$ and $v_{pub}^{new}$. 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 *JoinSplit description* in a *transaction* must refer to some earlier *block*'s final *treestate*.

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

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

A *JoinSplit operation* spends $\mathrm{N}^{\mathsf{old}}$ *notes* $\mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$ and transparent input $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$, and creates $\mathrm{N}^{\mathsf{new}}$ *notes* $\mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ and transparent output $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$.

Either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ MUST be zero.

**Zcash** *transactions* have the following additional fields:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| *Varies* | `nJoinSplit` | `compactSize uint` | The number of *JoinSplit descriptions* in `vJoinSplit`. |
| 1024 × `nJoinSplit` | `vJoinSplit` | `JoinSplitDescription [nJoinSplit]` | The sequence of *JoinSplit descriptions* in this *transaction*. |
| 33 | `joinSplitPubKey` | `char[33]` | An encoding of a ECDSA public verification key, using the secp256k1 curve and parameters defined in [13] and [4]. |
| 64 | `joinSplitSig` | `char[64]` | A signature on a prefix of the *transaction* encoding, to be verified using `joinSplitPubKey`. |

The encoding of `joinSplitPubKey` and the data to be signed are specified in more detail in § 5.3 *'Non-malleability'* on p. 9.

Each `JoinSplitDescription` consists of:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 8 | vpub_old | int64_t | A value $v^{old}_{pub}$ that the *JoinSplit operation* removes from the value pool. |
| 8 | vpub_new | int64_t | A value $v^{new}_{pub}$ that the *JoinSplit operation* inserts into the value pool. |
| 32 | anchor | char[32] | A merkle root $\mathsf{rt}$ of the *note commitment tree* at some block height in the past, or the merkle root produced by a previous *JoinSplit operation* in this *transaction*. Sean: We need to be more specific here. |
| 64 | nullifiers | char[32][$N^{old}$] | A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{old}_{1..N^{old}}$. |
| 64 | commitments | char[32][$N^{new}$]. | A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{new}_{1..N^{new}}$. |
| 32 | ephemeralKey | char[32] | A Curve25519 public key $\mathsf{epk}$. |
| 432 | encCiphertexts | char[216][$N^{new}$] | A sequence of ciphertext components for the encrypted output *notes*, $\mathbf{C}^{enc}_{1..N^{new}}$. |
| 32 | randomSeed | char[32] | A 256-bit seed that must be chosen independently at random for each *JoinSplit description*. |
| 64 | vmacs | char[32][$N^{old}$] | A sequence of message authentication tags $\mathsf{h}_{1..N^{old}}$ that bind $\mathsf{h}_{Sig}$ to each $\mathsf{a}_{sk}$ of the *JoinSplit description*. |
| 288 | zkproof | char[288] | An encoding, as determined by the libsnark library [9], of the zero-knowledge proof $\pi_{\texttt{JoinSplit}}$. |

The ephemeralKey and encCiphertexts fields together form the *transmitted notes ciphertext*.

TODO: Describe case where there are fewer than $N^{old}$ real input *notes*.

## 5.1 Computation of $\mathsf{h}_{Sig}$

Given a *JoinSplit description*, we define:

$$\mathsf{h}_{Sig} := \texttt{SHA256}\Big(\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{4\text{ bit }\mathbf{1}}\boxed{256\text{ bit }\mathsf{nf}^{old}_{1}\quad ...}\boxed{256\text{ bit }\mathsf{nf}^{old}_{N^{old}}}\boxed{\texttt{randomSeed}}\boxed{\texttt{joinSplitPubKey}}\Big)$$

## 5.2 Merkle root validity

A *JoinSplit description* is valid if $\mathsf{rt}$ is a *note commitment tree* root found in either the blockchain or a merkle root produced by inserting the *note commitments* of a previous *JoinSplit description* in the *transaction* to the *note commitment tree* identified by that previous *JoinSplit description*'s *anchor*.

## 5.3 Non-malleability

Let dataToBeSigned be the raw-format [5] encoding of the *transaction* up to and not including the joinSplitPubKey and joinSplitSig fields.

In order to ensure that a *JoinSplit description* is cryptographically bound to the transparent inputs and outputs corresponding to $v^{new}_{pub}$ and $v^{old}_{pub}$, and to the other *JoinSplit descriptions* in the same *transaction*, an ephemeral ECDSA key pair is generated for each *transaction*, and the dataToBeSigned is signed with the private signing key of this key pair. The corresponding public verification key is included in the *transaction* encoding as joinSplitPubKey.

A *transaction* is correctly signed if:

- `joinSplitSig` can be verified as an encoding of a signature on `dataToBeSigned`, using the ECDSA public key encoded as `joinSplitPubKey`; and
- `joinSplitSig` has an `s` value in the lower half of the possible range (i.e. `s` must be in the range from 0x1 to 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, inclusive).

If `s` is not in the given range, the signature is treated as invalid.

The encoding of a signature is:

| 256 bit `r` | 256 bit `s` |
|---|---|

where `r` and `s` are as defined in [13].

The encoding of a public key is as defined in section E.2.3.2 of [14] for a compressed elliptic curve point with $x$-coordinate $x_P$ and compressed $y$-coordinate $\tilde{y}_P$:

| 1 bit $\tilde{y}_P$ | 1 | $[0]^6$ | 256 bit $x_P$ |
|---|---|---|---|

Note that only compressed public keys are valid.

The condition enforced by the *`JoinSplit` circuit* specified in § 5.6 *'Non-malleability'* on p. 11 ensures that a holder of all of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$ for each *JoinSplit description* has authorized the use of the private signing key corresponding to `joinSplitPubKey` to sign this *transaction*.

## 5.4 Balance

A *JoinSplit operation* 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** [1], **Zcash** does not have a distinction between Mint and Pour operations. The addition of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ to a *JoinSplit description* subsumes the functionality of both Mint and Pour. Also, *JoinSplit descriptions* are indistinguishable regardless of the number of real input *notes*.

As stated in § 5 *'JoinSplit Operations and Descriptions'* on p. 8, either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ MUST be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

## 5.5 Note Commitments and Nullifiers

A *transaction* that contains one or more *JoinSplit descriptions*, when entered into the blockchain, appends to the *note commitment tree* with all constituent *note commitments*. All of the constituent *nullifiers* are also entered into the *nullifier set* of the *blockchain view and mempool*. A *transaction* is not valid if it attempts to add a *nullifier* to the *nullifier set* that already exists in the set.

## 5.6 *`JoinSplit` circuit* and Proofs

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

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

$$(\mathsf{rt}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}),$$

there exists a witness of *auxiliary input*:

$$(\mathsf{path}_{1..\mathrm{N}^{\mathsf{old}}}, \mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}, \mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \varphi)$$

where:

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

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

such that the following conditions hold:

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

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

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

**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 $\rho^{\mathsf{new}}_i$** 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 $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$: $\mathsf{cm}^{\mathsf{new}}_i = \mathtt{NoteCommitment}(\mathbf{n}^{\mathsf{new}}_i)$.

# 6 In-band secret distribution

In order to transmit the secret $\mathsf{v}$, $\rho$, and $\mathsf{r}$ (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 $\mathsf{a}_{\mathsf{pk}}$ and $\mathsf{sk}_{\mathsf{enc}}$) is used to reconstruct the original *note* and *memo field*.

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

## 6.1 Encryption

Let $\mathsf{SymEncrypt}_{\mathsf{K}}(\mathbf{P})$ be authenticated encryption using AEAD_CHACHA20_POLY1305 [11] encryption of plaintext $\mathbf{P}$, with empty "associated data", all-zero nonce $[0]^{96}$, and 256-bit key $\mathsf{K}$.

Similarly, let $\mathsf{SymDecrypt}_{\mathsf{K}}(\mathbf{C})$ be AEAD_CHACHA20_POLY1305 decryption of ciphertext $\mathbf{C}$, with empty "associated data", all-zero nonce $[0]^{96}$, and 256-bit key $\mathsf{K}$. The result is either the plaintext byte sequence, or $\perp$ indicating failure to decrypt.

Define $\mathsf{KDF}(i, \mathsf{h}_{\mathsf{Sig}}, \mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}) := \mathtt{Trailing}_{256}($

$\mathtt{SHA512}\Big($

| $i$-1 | 0 | $\mathtt{Trailing}_{246}(\mathsf{h_{Sig}})$ | 256 bit $\mathsf{dhsecret}_i$ | 256 bit epk | 256 bit $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}$ |
|---|---|---|---|---|---|

$\Big)$.

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 *note*, and let $\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *note plaintexts*. Let $\mathsf{h_{Sig}}$ be the value computed in § 5.1 *'Computation of* $\mathsf{h_{Sig}}$*'* on p. 9.

Then to encrypt:

- Generate a new Curve25519 (public, private) key pair $(\mathsf{epk}, \mathsf{esk})$.
- For $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$,
  - Let $\mathbf{P}^{\mathsf{enc}}_i$ be the raw encoding of $\mathbf{np}_i$.
  - Let $\mathsf{dhsecret}_i := \mathsf{Curve25519}(\mathsf{esk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
  - Let $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
  - Let $\mathbf{C}^{\mathsf{enc}}_i := \mathsf{SymEncrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathbf{P}^{\mathsf{enc}}_i)$.

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

## 6.2 Decryption by a Recipient

Let $\mathsf{addr_{pk}} = (\mathsf{a_{pk}}, \mathsf{pk_{enc}})$ be the recipient's *payment address*, and let $\mathsf{sk_{enc}}$ be the recipient's Curve25519 private key. Let $\mathsf{h_{Sig}}$ be the value computed in § 5.1 *'Computation of* $\mathsf{h_{Sig}}$*'* on p. 9. Let $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *note 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 $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
- Return $\mathtt{DecryptNote}(\mathsf{K}^{\mathsf{enc}}_i, \mathbf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i, \mathsf{a_{pk}})$.

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

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

Note that this corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in Figure 2 of [1].

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

Note that a *note* 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 *note* was output to no longer be on the consensus blockchain.

## 6.3 Commentary

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 crypto_box_seal algorithm defined in libsodium [10]. Note that:

- The same ephemeral key is used for all encryptions to the recipient keys in a given *JoinSplit 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 AEAD_CHACHA20_POLY1305 is not used.

# 7 Encoding Addresses and Keys

This section describes how **Zcash** encodes *payment addresses* and *spending keys*.

Addresses and keys can be encoded as a byte sequence; this is called the *raw encoding*. This byte sequence can then be further encoded using Base58Check. The Base58Check layer is the same as for upstream **Bitcoin** addresses [3].

SHA-256 compression function outputs are always represented as sequences 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** [3].

## 7.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [3].

## 7.3 Protected Payment Addresses

A *payment 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 § 6 *'In-band secret distribution'* on p. 11.

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

| 8 bit **0x92** | 256 bit $\mathsf{a}_{\mathsf{pk}}$ | 256 bit $\mathsf{pk}_{\mathsf{enc}}$ |
|---|---|---|

- A byte, **0x92**, indicating this version of the raw encoding of a **Zcash** public address.
- 256 bits specifying $\mathsf{a}_{\mathsf{pk}}$.
- 256 bits specifying $\mathsf{pk}_{\mathsf{enc}}$, using the normal encoding of a Curve25519 public key [2].

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 $\mathsf{a}_{\mathsf{sk}}$.

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

| 8 bit **0x**?? | $[0]^4$ | 252 bit $\mathsf{a}_{\mathsf{sk}}$ |
|---|---|---|

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

Consistent with little-endian encoding, the zero padding occupies the low-order 4 bits of the second byte.

**Note:** If an implementation represents $\mathsf{a_{sk}}$ internally as a sequence of 32 bytes with the 4 bits of zero padding intact, it will be in the correct form for use as an input to $\mathsf{PRF^{addr}}$, $\mathsf{PRF^{nf}}$, and $\mathsf{PRF^{pk}}$ without need for bit-shifting. Future key representations may make use of these padding bits.

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?

# 8 Differences from the Zerocash paper

## 8.1 Transaction Structure

**Zerocash** introduces two new operations, which are described in the paper as new transaction types, in addition to the original transaction type of the cryptocurrency on which it is based (e.g. **Bitcoin**).

In **Zcash**, there is only the original **Bitcoin** transaction type, which is extended to contain a sequence of zero or more **Zcash**-specific operations.

This allows for the possibility of chaining transfers of protected value in a single **Zcash** *transaction*, e.g. to spend a protected *note* that has just been created. (In **Zcash**, we refer to value stored in UTXOs as "transparent", and value stored in *JoinSplit operation* output *notes* as "protected".) This was not possible in the **Zerocash** design without using multiple transactions. It also allows transparent and protected transfers to happen atomically — possibly under the control of nontrivial script conditions, at some cost in distinguishability.

TODO: Describe changes to signing.

## 8.2 Unification of Mints and Pours

In the original **Zerocash** protocol, there were two kinds of transaction relating to protected *notes*:

- a "Mint" transaction takes value from transparent UTXOs as input and produces a new protected *note* as output.
- a "Pour" transaction takes up to $\mathrm{N}^{\mathsf{old}}$ protected *notes* as input, and produces up to $\mathrm{N}^{\mathsf{new}}$ protected *notes* and a transparent UTXO as output.

Only "Pour" transactions included a *zk-SNARK* proof.

In **Zcash**, the sequence of operations added to a *transaction* (described in § 8.1 *'Transaction Structure'* on p. 14) consists only of *JoinSplit operations*. A *JoinSplit operation* is a Pour operation generalized to take a transparent UTXO as input, allowing *JoinSplit operations* to subsume the functionality of Mints. An advantage of this is that a **Zcash** *transaction* that takes input from an UTXO can produce up to $\mathrm{N}^{\mathsf{new}}$ output *notes*, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the *JoinSplit operation*: an unused (zero-value) input is indistinguishable from an input that takes value from a *note*.

This unification also simplifies the fix to the Faerie Gold attack described below, since no special case is needed for Mints.

## 8.3 Memo Fields

**Zcash** adds a *memo field* sent from the creator of a *JoinSplit description* to the recipient of each output *note*. This feature is described in more detail in § 4.2.3 *'Note Plaintexts and Memo Fields '* on p. 6.

## 8.4 Faerie Gold attack and fix

When a protected *note* is created in **Zerocash**, the creator is supposed to choose a new $\rho$ value at random. The *nullifier* of the *note* is derived from its *spending key* ($\mathsf{a_{sk}}$) and $\rho$. The *note commitment* is derived from the recipient address component $\mathsf{a_{pk}}$, the value $\mathsf{v}$, and the commitment trapdoor $\mathsf{r}$, as well as $\rho$. However nothing prevents creating multiple *notes* with different $\mathsf{v}$ and $\mathsf{r}$ (hence different *note commitments*) but the same $\rho$.

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by $\mathsf{Receive}$ (as defined in Figure 2 of [1]), but only one of which can be spent.

We call this a "Faerie Gold" attack — referring 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 [8].

This attack does not violate the security definitions given in [1]. The issue could be framed as a problem either with the definition of Completeness, or the definition of Balance:

- The Completeness property asserts that a validly received *note* can be spent provided that its *nullifier* does not appear on the ledger. This does not take into account the possibility that distinct *notes*, which are validly received, could have the same *nullifier*. That is, the security definition depends on a protocol detail –*nullifiers*– that is not part of the intended abstract security property, and that could be implemented incorrectly.
- The Balance property only asserts that an adversary cannot obtain *more* funds than they have minted or received via payments. It does not prevent an adversary from causing others' funds to decrease. In a Faerie Gold attack, an adversary can cause spending of a *note* to reduce (to zero) the effective value of another *note* for which the attacker does not know the *spending key*, which violates an intuitive conception of global balance.

These problems with the security definitions need to be repaired, but doing so is outside the scope of this specification. Here we only describe how **Zcash** addresses the immediate attack.

It would be possible to address the attack by requiring that a recipient remember all of the $\rho$ values for all *notes* they have ever received, and reject duplicates (as proposed in [7]). However, this requirement would interfere with the intended **Zcash** feature that a holder of a *spending key* can recover access to (and be sure that they are able to spend) all of their funds, even if they have forgotten everything but the *spending key*.

Instead, **Zcash** enforces that an adversary must choose distinct values for each $\rho$, by making use of the fact that all of the *nullifiers* in *JoinSplit descriptions* that appear in a valid *blockchain view* must be distinct. The *nullifiers* are used as input to $\mathtt{SHA256}$ to derive a public value $\mathsf{h_{Sig}}$ which uniquely identifies the transaction, as described in § 5.1 *'Computation of* $\mathsf{h_{Sig}}$*'* on p. 9. ($\mathsf{h_{Sig}}$ was already used in **Zerocash** in a way that requires it to be unique in order to maintain indistinguishability of *JoinSplit descriptions*; adding the *nullifiers* to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the *transaction* creator is an adversary.)

The $\rho$ value for each output *note* is then derived from a random private seed $\varphi$ and $\mathsf{h_{Sig}}$ using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of $\rho$ for each output *note* is enforced by the circuit (see § 5.6 *'Uniqueness of* $\rho_i^{\mathsf{new}}$*'* on p. 11).

Now even if the creator of a *JoinSplit description* does not choose $\varphi$ randomly, uniqueness of *nullifiers* and collision resistance of both $\mathtt{SHA256}$ and $\mathsf{PRF}^{\rho}$ will ensure that the derived $\rho$ values are unique, at least for any two *JoinSplit descriptions* that get into a valid *blockchain view*. This is sufficient to prevent the Faerie Gold attack.

### 8.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM}_\mathsf{r}$ and $\mathsf{COMM}_\mathsf{s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, $\mathsf{v}$, and $\rho$. However, the instantiation of $\mathsf{COMM}_\mathsf{r}$ and $\mathsf{COMM}_\mathsf{s}$ 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 $\rho$ 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 $\rho$ with colliding outputs of the truncated hash, and therefore the same *note commitment*. This would have allowed such an attacker to break the balance property by double-spending *notes*, 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 [1]). Since **Zcash** combines "Mint" and "Pour" transactions into a generalized *JoinSplit operation* 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 *note commitment* from three to two, saving a total of four `SHA256Compress` evaluations in the *`JoinSplit` circuit*.

Note that **Zcash** *note commitments* are not statistically hiding, and so **Zcash** does not support the "everlasting anonymity" property described in section 8.1 of the **Zerocash** paper [1], 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.

### 8.6 Changes to PRF inputs and truncation

TODO:

### 8.7 In-band secret distribution

TODO:

### 8.8 Miscellaneous

- The paper defines a *note* 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 $\mathsf{COMM}_\mathsf{s}$ in section 5.1 of the paper did not use $\mathsf{s}$ (and neither does the new instantiation of `NoteCommitment`). $\mathsf{cm}$ 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, Samantha Hulsey, and no doubt others.

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

# 10 References


[1] 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.

[2] 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`.

[3] Base58Check encoding – Bitcoin Wiki. `https://en.bitcoin.it/wiki/Base58Check_encoding`. Accessed: 2016-01-26.

[4] Secp256k1 – Bitcoin Wiki. `https://en.bitcoin.it/wiki/Secp256k1`. Accessed: 2016-03-14.

[5] Raw Transaction Format – Bitcoin Developer Reference. `https://bitcoin.org/en/developer-reference#raw-transaction-format`. Accessed: 2016-03-15.

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

[7] Christina Garman, Matthew Green, and Ian Miers. Accountable Privacy for Decentralized Anonymous Payments. Cryptology ePrint Archive: Report 2016/061. `https://eprint.iacr.org/2016/061`. Last revised 24 Jan 2016.

[8] 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.

[9] libsnark: a C++ library for zkSNARK proofs. `https://github.com/scipr-lab/libsnark`. Accessed: 2016-03-15.

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

[11] 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`.

[12] 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.

[13] Certicom Research. Standards for Efficient Cryptography 2 (SEC 2). `http://www.secg.org/sec2-v2.pdf`, January 27 2010. Version 2.0.

[14] IEEE Computer Society. *IEEE Std 1363-2000: Standard Specifications for Public-Key Cryptography*. IEEE, August 29 2000. `http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=891000&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F7168%2F19282%2F00891000`. Accessed 2016-03-15.