Zcash Protocol Specification
Version 2.0-alpha-1

Sean Bowe — Daira Hopwood — Taylor Hornby — Nathan Wilcox

March 30, 2016

Contents

1 Introduction .................................................. 3

2 Caution .................................................... 3

3 Conventions ................................................. 3

3.1 Integers, Bit Sequences, and Endianness .................................. 3

3.2 Cryptographic Functions ......................................... 4

4 Concepts .................................................... 4

4.1 Payment Addresses and Spending Keys .................................... 4

4.2 Notes ...................................................... 5

4.2.1 Note Commitments .......................................... 5

4.2.2 Nullifiers ............................................... 6

4.2.3 Note Plaintexts and Memo Fields .................................... 6

4.3 Note Commitment Tree ........................................ 7

4.4 Nullifier Set ............................................... 7

4.5 The Blockchain ............................................ 7

5 JoinSplit Operations and Descriptions .................................. 8

5.1 Computation of hSig ........................................... 9

5.2 Merkle root validity .......................................... 9

5.3 Non-malleability ........................................... 9

5.4 Balance ................................................... 10

5.5 Note Commitments and Nullifiers .................................... 10

5.6 JoinSplit circuit and Proofs .................................... 11

6 In-band secret distribution ........................................ 11
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 in Zcash-specific encodings are unsigned, have a fixed bit length, and are encoded in little-endian byte order. 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.

The notation “...” represents the given string represented as a sequence of bytes in US-ASCII. For example, “abc” represents the byte sequence [0x61, 0x62, 0x63].

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.

Bits within a byte are shown with the most significant bit on the left of a diagram. Where bit fields are used, the text will clarify their position in each case. In cases where a field starts at a position that is not a byte boundary, that field will always be a bit sequence, and its initial bits will occupy the less significant bits of the overlapping first byte.

LeadingBytesk(x), where k is an integer, returns the leading (initial) k bytes of x.

The notation 1..N, used as a subscript, means the sequence of values with indices 1 through N inclusive. For example, a\new_{pk,1..Nnew} means the sequence [a\new_{pk,1}, a\new_{pk,2}, ..., a\new_{pk,Nnew}].

The symbol ⊥ is used to indicate unavailable information or a failed decryption.
3.2 Cryptographic Functions

CRH is a collision-resistant hash function. In Zcash, the \textit{SHA-256 compression} function is used which takes a 512-bit block and produces a 256-bit hash. This is different from the \textit{SHA-256} function, which hashes arbitrary-length sequences.\cite{12}

PRF\textsubscript{x} is a pseudo-random function seeded by \(x\). Four \textit{independent} PRF\textsubscript{x} are needed in our scheme: \(\text{PRF}\textsubscript{addr}{^\text{r}}\), \(\text{PRF}\textsubscript{nf}\), \(\text{PRF}\textsubscript{pk}\), and \(\text{PRF}\textsubscript{ρ}\).

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

In Zcash, the \textit{SHA-256 compression} function is used to construct all of these functions.

\[
\begin{align*}
\text{PRF}\textsuperscript{addr}(t) & := \text{CRH}(1100, 252 \text{ bit } x, 8 \text{ bit } t, [0]^{248}) \\
\text{nf} & := \text{PRF}\textsuperscript{nf}(\rho) := \text{CRH}(1110, 252 \text{ bit } \rho, 256 \text{ bit } \rho) \\
h_i & := \text{PRF}\textsuperscript{pk}(i, h_{\text{Sig}}) := \text{CRH}(0i-100, 252 \text{ bit } \rho, 256 \text{ bit } h_{\text{Sig}}) \\
\rho\text{new}_i & := \text{PRF}\textsubscript{ρ}(i, h_{\text{Sig}}) := \text{CRH}(0i-110, 252 \text{ bit } \rho, 256 \text{ bit } h_{\text{Sig}})
\end{align*}
\]

\textbf{Note:} The first four bits –i.e. the most significant four bits of the first byte– are used to distinguish different uses of CRH, ensuring that the functions are independent. In addition to the inputs shown here, the bits 1011 in this position are used to distinguish uses of the full \textit{SHA-256} hash function — see §4.2.1 ‘Note Commitments ’ on p.5. (The specific bit patterns chosen here are motivated by the possibility of future extensions that either increase \(N\text{old}\) and/or \(N\text{new}\) to 3, or that add an additional bit to \(\rho\) to encode a new key type, or that require an additional PRF.)

BLAKE2b is also used to construct a Key Derivation Function and as a hash function for the computation of \(h_{\text{Sig}}\).

The notation \texttt{BLAKE2b}(\(p, x\)) represents the application of \texttt{BLAKE2b} to a 16-byte personalization string \(p\) and input \(x\), as defined in [1].

4 Concepts

4.1 Payment Addresses and Spending Keys

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

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

\( a_{pk}, sk_{enc} \) and \( pk_{enc} \) are derived as follows:

\[
\begin{align*}
a_{pk} &:= PRF_{a_{sk}}(0) \\
sk_{enc} &:= \text{clamp}_{\text{Curve25519}}(PRF_{a_{sk}}(1)) \\
pk_{enc} &:= \text{Curve25519}(sk_{enc}, 9)
\end{align*}
\]

where

- \( \text{Curve25519}(n, q) \) performs point multiplication of the Curve25519 public key represented by the byte sequence \( q \) by the Curve25519 secret key represented by the byte sequence \( n \), as defined in section 2 of [3];
- \( 9 \) is the public byte sequence representing the Curve25519 base point;
- \( \text{clamp}_{\text{Curve25519}}(x) \) takes a 32-byte sequence \( x \) as input and returns a byte sequence representing a Curve25519 private key, with bits “clamped” as described in section 3 of [3]: “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 \( \text{addr}_{pk} \) 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 \( \text{addr}_{pk} \) 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 \( n \)) is a tuple \( (a_{pk}, v, \rho, r) \) which represents that a value \( v \) is spendable by the recipient who holds the *spending key* \( a_{sk} \) corresponding to \( a_{pk} \), as described in the previous section.

- \( a_{pk} \) is a 32-byte *authorization* public key of the recipient.
- \( v \) is a 64-bit unsigned integer representing the value of the *note* in *zatoshi* (1 ZEC = \( 10^8 \) zatoshi).
- \( \rho \) is a 32-byte \( \text{PRF}_{a_{sk}}^{nf} \) preimage.
- \( r \) is a 32-byte *COMM* *trapdoor*.

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

#### 4.2.1 Note Commitments

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

\[
\begin{array}{cccc}
& 256 \text{ bit } a_{pk} & 64 \text{ bit } v & 256 \text{ bit } \rho & 256 \text{ bit } r \\
\hline
\text{cm} := \text{SHA256} \left( \begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0
\end{array} \right)
\end{array}
\]

**Note:** The leading byte of the \( \text{SHA256} \) input is \( 0xB0 \).
4.2.2 Nullifiers

A nullifier (denoted \(nf\)) is derived from the \(\rho\) component of a note as \(\text{PRF}_{\text{sk}}^{nf}(\rho)\). A note is spent by proving knowledge of \(\rho\) and \(\text{ask}\) in zero knowledge while disclosing its nullifier \(nf\), allowing \(nf\) 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 \(cm\).

The note plaintexts associated with a JoinSplit description are encrypted to the respective transmission keys \(\text{pk}_{\text{new}}, 1..N_{\text{new}}\), 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 \((v, \rho, r, \text{memo})\).

The first three of these fields are as defined earlier. \(\text{memo}\) 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:

<table>
<thead>
<tr>
<th>8 bit 0x00</th>
<th>v (8 bytes)</th>
<th>(\rho) (32 bytes)</th>
<th>(r) (32 bytes)</th>
<th>memo (128 bytes)</th>
</tr>
</thead>
</table>

- A byte, \(0x00\), indicating this version of the encoding of a note plaintext.
- 8 bytes specifying \(v\).
- 32 bytes specifying \(\rho\).
- 32 bytes specifying \(r\).
- 128 bytes specifying \(\text{memo}\).
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.

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 $v^{\text{old}}_{\text{pub}}$ and $v^{\text{new}}_{\text{pub}}$. 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 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 \( N^{\text{old}} \) notes \( n^{\text{old}}_{1...N^{\text{old}}} \) and transparent input \( v^{\text{old}}_{\text{pub}} \), and creates \( N^{\text{new}} \) notes \( n^{\text{new}}_{1...N^{\text{new}}} \) and transparent output \( v^{\text{new}}_{\text{pub}} \).

**Consensus rule:** Either \( v^{\text{old}}_{\text{pub}} \) or \( v^{\text{new}}_{\text{pub}} \) MUST be zero.

Zcash transactions have the following additional fields:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varies</td>
<td>nJoinSplit</td>
<td>compactSize uint</td>
<td>The number of JoinSplit descriptions in vJoinSplit.</td>
</tr>
<tr>
<td>33</td>
<td>joinSplitPubKey *</td>
<td>char[33]</td>
<td>An encoding of a ECDSA public verification key, using the secp256k1 curve and parameters defined in [13] and [5].</td>
</tr>
<tr>
<td>64</td>
<td>joinSplitSig *</td>
<td>char[64]</td>
<td>A signature on a prefix of the transaction encoding, to be verified using joinSplitPubKey.</td>
</tr>
</tbody>
</table>

* The joinSplitPubKey and joinSplitSig fields are present if and only if nJoinSplit > 0.

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:
<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>vpub_old</td>
<td>int64_t</td>
<td>A value $v_{pub}^{old}$ that the JoinSplit operation removes from the value pool.</td>
</tr>
<tr>
<td>8</td>
<td>vpub_new</td>
<td>int64_t</td>
<td>A value $v_{pub}^{new}$ that the JoinSplit operation inserts into the value pool.</td>
</tr>
<tr>
<td>32</td>
<td>anchor</td>
<td>char[32]</td>
<td>A merkle root $rt$ 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. <strong>Sean:</strong> We need to be more specific here.</td>
</tr>
<tr>
<td>64</td>
<td>nullifiers</td>
<td>char[32][N&lt;sup&gt;old&lt;/sup&gt;]</td>
<td>A sequence of nullifiers of the input notes $n_{f}^{old}<em>1...N</em>{f}^{old}$.</td>
</tr>
<tr>
<td>64</td>
<td>commitments</td>
<td>char[32][N&lt;sup&gt;new&lt;/sup&gt;].</td>
<td>A sequence of note commitments for the output notes $cm_{1}^{new}<em>1...N</em>{new}$.</td>
</tr>
<tr>
<td>434</td>
<td>encCiphertexts</td>
<td>char[217][N&lt;sup&gt;new&lt;/sup&gt;]</td>
<td>A sequence of ciphertext components for the encrypted output notes, $C_{1}^{enc}<em>{1...N</em>{new}}$.</td>
</tr>
<tr>
<td>32</td>
<td>randomSeed</td>
<td>char[32]</td>
<td>A 256-bit seed that must be chosen independently at random for each JoinSplit description.</td>
</tr>
<tr>
<td>64</td>
<td>vmacs</td>
<td>char[32][N&lt;sup&gt;old&lt;/sup&gt;]</td>
<td>A sequence of message authentication tags $h_{1...N_{old}}$ that bind $h_{Sig}$ to each $a_{sk}$ of the JoinSplit description.</td>
</tr>
<tr>
<td>288</td>
<td>zkproof</td>
<td>char[288]</td>
<td>An encoding, as determined by the libsnark library [9], of the zero-knowledge proof $\pi_{JoinSplit}$.</td>
</tr>
</tbody>
</table>

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 $h_{Sig}$

Given a JoinSplit description, we define:

$$h_{Sig} := \text{BLAKE2b}(h_{Sigtag}, \text{randomSeed}, 256 \text{ bit } n_{f}^{old}_1...256 \text{ bit } n_{f}^{old}_{N_{old}}, \text{joinSplitPubKey})$$

### 5.2 Merkle root validity

A JoinSplit description is valid if $rt$ 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

Bitcoin defines several SIGHASH types that cover various parts of a transaction. In Zcash, all of these SIGHASH types are extended to cover the Zcash-specific fields $n_{JoinSplit}$, $v_{JoinSplit}$, and $joinSplitPubKey$. They do not cover the field $joinSplitSig$.

**Consensus rule:** If $n_{JoinSplit} > 0$, the transaction MUST NOT use SIGHASH types other than SIGHASH_ALL.
Let dataToBeSigned be the hash of the transaction using the SIGHASH_ALL SIGHASH type. Note that this excludes all of the scriptSig fields in the non-Zcash-specific parts of the transaction.

In order to ensure that a JoinSplit description is cryptographically bound to the transparent inputs and outputs corresponding to v\textsubscript{old}\textsubscript{pub} and v\textsubscript{new}\textsubscript{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.

If nJoinSplit is zero, the joinSplitPubKey and joinSplitSig fields are omitted. Otherwise, a transaction has a correct JoinSplit signature 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 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, 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 xp and compressed y-coordinate y\textsubscript{p}:

| 0 0 0 0 | 1 bit y\textsubscript{p} | 256 bit x\textsubscript{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 a\textsubscript{old,1}..N\textsubscript{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. v\textsubscript{old}\textsubscript{pub} takes value from the value pool and v\textsubscript{new}\textsubscript{pub} adds value to the value pool. As a result, v\textsubscript{old}\textsubscript{pub} is treated like an output value, whereas v\textsubscript{new}\textsubscript{pub} is treated like an input value.

Note that unlike original Zerocash [2], Zcash does not have a distinction between Mint and Pour operations. The addition of v\textsubscript{old}\textsubscript{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 v\textsubscript{old}\textsubscript{pub} or v\textsubscript{new}\textsubscript{pub} MUST be zero. No generality is lost because, if a transaction in which both v\textsubscript{old}\textsubscript{pub} and v\textsubscript{new}\textsubscript{pub} were nonzero were allowed, it could be replaced by an equivalent one in which min(v\textsubscript{old}\textsubscript{pub}, v\textsubscript{new}\textsubscript{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, N\text{old} and N\text{new} are both 2.

A valid instance of \( \pi_{\text{JoinSplit}} \) assures that given a primary input:

\[
(rt, \rho_{1..N_{\text{old}}}^{\text{old}}, \text{cm}_{1..N_{\text{new}}}^{\text{new}}, v_{1..N_{\text{old}}}^{\text{old}}, v_{1..N_{\text{new}}}^{\text{new}}, H_{1..N_{\text{old}}}, h_{1..N_{\text{new}}}, h_{\text{Sig}})
\]

there exists a witness of auxiliary input:

\[
(\text{path}_{1..N_{\text{old}}}, \rho_{1..N_{\text{old}}}^{\text{old}}, \text{a}_{\text{pk},1..N_{\text{old}}}, \text{cm}_{1..N_{\text{new}}}, \text{v})
\]

where:

for each \( i \in \{1..N_{\text{old}}\} \):
\[
\rho_{i}^{\text{old}} = (\text{a}_{\text{pk},i}, \text{v}_{i}^{\text{old}}, \rho_{i}^{\text{old}}, r_{i}^{\text{old}});
\]

for each \( i \in \{1..N_{\text{new}}\} \):
\[
\rho_{i}^{\text{new}} = (\text{a}_{\text{pk},i}, \text{v}_{i}^{\text{new}}, \rho_{i}^{\text{new}}, r_{i}^{\text{new}})
\]

such that the following conditions hold:

**Merkle path validity** for each \( i \in \{1..N_{\text{old}}\} \mid v_{i}^{\text{old}} \neq 0 \): \( \text{path}_{i} \) must be a valid path of depth \( d \) from \( \text{NoteCommitment}(\rho_{i}^{\text{old}}) \) to note commitment tree root \( rt \).

**Balance** \( v_{1..N_{\text{old}}}^{\text{old}} + \sum_{i=1}^{N_{\text{old}}} v_{i}^{\text{old}} = v_{1..N_{\text{new}}}^{\text{new}} + \sum_{i=1}^{N_{\text{new}}} v_{i}^{\text{new}}. \)

**Nullifier integrity** for each \( i \in \{1..N_{\text{new}}\} \):
\[
\rho_{i}^{\text{old}} = \text{PRF}_{\text{sk}, i}^{nf}(\rho_{i}^{\text{old}}).
\]

**Spend authority** for each \( i \in \{1..N_{\text{old}}\} \):
\[
\text{a}_{\text{pk},i}^{\text{old}} = \text{PRF}_{\text{sk}, i}^{addr}(0).
\]

**Non-malleability** for each \( i \in \{1..N_{\text{old}}\} \):
\[
\text{h}_{i} = \text{PRF}_{\text{sk}, i}^{pk}(i, h_{\text{Sig}}).
\]

**Uniqueness of \( \rho_{i}^{\text{new}} \)** for each \( i \in \{1..N_{\text{new}}\} \):
\[
\rho_{i}^{\text{new}} = \text{PRF}_{\text{sk}, i}^{\phi}(i, h_{\text{Sig}}).
\]

**Commitment integrity** for each \( i \in \{1..N_{\text{new}}\} \):
\[
\text{cm}_{i}^{\text{new}} = \text{NoteCommitment}(\rho_{i}^{\text{new}}).
\]

6 In-band secret distribution

In order to transmit the secret \( v, \rho, \text{ and } r \) (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 \( \text{pk}_{\text{enc}} \) is used to encrypt these secrets. The recipient’s possession of the associated \( (\text{addr}_{\text{pk}}, \text{addr}_{\text{sk}}) \) (which contains both \( \text{a}_{\text{pk}} \) and \( \text{sk}_{\text{enc}} \)) 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


Similarly, let SymDecrypt_K(C) be AEAD_CHACHA20_POLY1305 decryption of ciphertext C, with empty “associated data”, all-zero nonce [0]96, and 256-bit key K. The result is either the plaintext byte sequence, or ⊥ indicating failure to decrypt.

Define:

\[
\begin{align*}
\text{KDF}(i, h_{\text{Sig}}, \text{dhsecret}_i, \text{epk}, \text{pk}^\text{new}_{\text{enc}, i}) & := \text{LeadingBytes}_{32}(\text{BLAKE2b}(\text{kdftag}, \text{kdfinput})) \\
\text{kdftag} & := \begin{tabular}{|c|c|}
\hline
\text{64 bit} & \text{ZcashKDF} \& \text{8 bit} & i - 1 \& \text{[0]}^56 \\
\hline
\end{tabular} \\
\text{kdfinput} & := \begin{tabular}{|c|c|c|c|}
\hline
\text{256-bit} & \text{h}_{\text{Sig}} & \text{256 bit} & \text{dhsecret}_i & \text{256 bit} & \text{epk} & \text{256 bit} & \text{pk}^\text{new}_{\text{enc}, i} \\
\hline
\end{tabular} .
\end{align*}
\]

Let \( \text{pk}^\text{new}_{\text{enc}, 1..N_{\text{new}}} \) be the Curve25519 public keys for the intended recipient addresses of each new note, and let \( \text{np}_{1..N_{\text{new}}} \) be the note plaintexts. Let \( h_{\text{Sig}} \) be the value computed in § 5.1 ‘Computation of \( h_{\text{Sig}} \)’ on p. 9.

Then to encrypt:

- For \( i \in \{1..N_{\text{new}}\} \),
  - Let \( P^\text{enc}_i \) be the raw encoding of \( \text{np}_i \),
  - Let \( \text{dhsecret}_i := \text{Curve25519}(\text{esk}, \text{pk}^\text{new}_{\text{enc}, i}) \).
  - Let \( K^\text{enc}_i := \text{KDF}(i, h_{\text{Sig}}, \text{dhsecret}_i, \text{epk}, \text{pk}^\text{new}_{\text{enc}, i}) \).
  - Let \( C^\text{enc}_i := \text{SymEncrypt}_{K^\text{enc}_i}(P^\text{enc}_i) \).

The resulting transmitted notes ciphertext is \( (\text{epk}, C^\text{enc}_{1..N_{\text{new}}}) \).

6.2 Decryption by a Recipient

Let \( \text{addr}_{\text{pk}} = (a_{\text{pk}}, \text{pk}_{\text{enc}}) \) be the recipient’s payment address, and let \( \text{sk}_{\text{enc}} \) be the recipient’s Curve25519 private key. Let \( h_{\text{Sig}} \) be the value computed in § 5.1 ‘Computation of \( h_{\text{Sig}} \)’ on p. 9. Let \( \text{cm}^\text{new}_{1..N_{\text{new}}} \) be the note commitments of each output coin. Then for each \( i \in \{1..N_{\text{new}}\} \), the recipient will attempt to decrypt that ciphertext component as follows:

- Let \( \text{dhsecret}_i := \text{Curve25519}(\text{sk}_{\text{enc}}, \text{epk}) \).
- Let \( K^\text{enc}_i := \text{KDF}(i, h_{\text{Sig}}, \text{dhsecret}_i, \text{epk}, \text{pk}^\text{new}_{\text{enc}, i}) \).
- Return DecryptNote\( (K^\text{enc}_i, C^\text{enc}_i, \text{cm}^\text{new}_i, a_{\text{pk}}) \).

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

- Let \( P^\text{enc}_i := \text{SymDecrypt}_{K^\text{enc}_i}(C^\text{enc}_i) \).
- If \( P^\text{enc}_i \) = ⊥, return ⊥.
- Extract \( \text{np}_i = (v^\text{new}_i, \rho^\text{new}_i, r^\text{new}_i, \text{memo}_i) \) from \( P^\text{enc}_i \).
- If \( \text{NoteCommitment}((a_{\text{pk}}, v^\text{new}_i, \rho^\text{new}_i, r^\text{new}_i)) \neq \text{cm}^\text{new}_i \), return ⊥, else return \( \text{np}_i \).
Note that this corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in Figure 2 of [2].

To test whether a note is unspent in a particular blockchain view also requires the authorization private key \( a_{sk} \); the coin is unspent if and only if \( nf = \text{PRF}_{a_{sk}}(\rho) \) 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 seed \( h_{\text{Sig}} \), the public keys of both parties, and the index \( i \).
- The nonce parameter to AEAD\_CHACHA20\_POLY1305 is not used.
- The “IETF” definition of AEAD\_CHACHA20\_POLY1305 from [11] is used; this uses a 32-bit block count and a 96-bit nonce, rather than a 64-bit block count and 64-bit nonce as in the original definition of ChaCha20.

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 [4]. 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 [4].

7.2 Transparent Private Keys

These are encoded in the same way as in Bitcoin [4].

7.3 Protected Payment Addresses

A payment address consists of \( a_{pk} \) and \( pk_{\text{enc}} \). \( a_{pk} \) is a SHA-256 compression function output. \( pk_{\text{enc}} \) 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bit</td>
<td>0x92</td>
</tr>
<tr>
<td>256 bit</td>
<td>( a_{pk} )</td>
</tr>
<tr>
<td>256 bit</td>
<td>( pk_{\text{enc}} )</td>
</tr>
</tbody>
</table>
• A byte, 0x92, indicating this version of the raw encoding of a Zcash public address.
• 256 bits specifying $a_{pk}$.
• 256 bits specifying $p_{k_{enc}}$, 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 $a_{sk}$, which is a sequence of 252 bits.

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

| 8 bit $0x?? | 0|^4 | 252 bit $a_{sk}$ |

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

The zero padding occupies the most significant 4 bits of the second byte.

Note: If an implementation represents $a_{sk}$ 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 $PRF_{addr}$, $PRF_{nf}$, and $PRF_{pk}$ 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 $N^{\text{old}}$ protected notes as input, and produces up to $N^{\text{new}}$ 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 $N^{\text{new}}$ 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 ($a_{sk}$) and $\rho$. The note commitment is derived from the recipient address component $a_{pk}$, the value $v$, and the commitment trapdoor $r$, as well as $\rho$. However nothing prevents creating multiple notes with different $v$ and $r$ (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 Receive (as defined in Figure 2 of [2]), 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 [2]. 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 $\text{BLAKE2b}$ to derive a public value $h_{\text{Sig}}$ which uniquely identifies the transaction, as described in §5.1 ‘Computation of $h_{\text{Sig}}$’ on p. 9. ($h_{\text{Sig}}$ 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 $\phi$ and $h_{\text{Sig}}$ using $\text{PRF}^\rho_{\phi}$. The correct construction of $\rho$ for each output note is enforced by the circuit (see §5.6 ‘Uniqueness of $\rho_i^{\text{new}}$’ on p. 11).

Now even if the creator of a JoinSplit description does not choose $\phi$ randomly, uniqueness of nullifiers and collision resistance of both $\text{BLAKE2b}$ and $\text{PRF}^\rho$ 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 $\text{COMM}_r$ and $\text{COMM}_s$ is a computationally binding commitment to its inputs $a_{pk}$, $v$, and $\rho$. However, the instantiation of $\text{COMM}_r$ and $\text{COMM}_s$ 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 $a_{pk}$ 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 $\text{SHA256}$ 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 JoinSplit operation which always uses a ZK proof, it does not require the nesting. A side benefit is that this reduces the number of $\text{SHA256Compress}$ evaluations needed to compute each note commitment from three to two, saving a total of four $\text{SHA256Compress}$ 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 [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.

### 8.6 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 $(a_{pk}, v, \rho, r, s, \text{cm})$, whereas this specification defines it as $(a_{pk}, v, \rho, r)$. This is just a clarification, because the instantiation of $\text{COMM}_s$ in section 5.1 of the paper did not use $s$ (and neither does the new instantiation of NoteCommitment). $\text{cm}$ 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