Zcash Protocol Specification

Version 2016.0-alpha-3.1-42-gc462a1

as intended for the **Zcash** release of summer 2016

Daira Hopwood
Sean Bowe — Taylor Hornby — Nathan Wilcox

August 12, 2016

Contents

1 Introduction

Zcash is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash** [5] 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.

This specification is structured as follows:

- Notation — definitions of notation used throughout the document;
- Concepts — the principal abstractions needed to understand the protocol;
- Abstract Protocol — a high-level description of the protocol in terms of ideal cryptographic components;
- Concrete Protocol — how the functions and encodings of the abstract protocol are instantiated;
- Differences from the **Zerocash** protocol — a summary of changes from the protocol in [5].

1.1 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.

2 Notation

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

The notation $\mathbb{B}^\ell$ means the set of sequences of ℓ bits. $\mathbb{B}^\star$ means the set of bit sequences of arbitrary length.

The notation **"..."** means the given string represented as a sequence of bytes in US-ASCII. For example, **"abc"** represents the byte sequence $[\mathbf{0x61}, \mathbf{0x62}, \mathbf{0x63}]$.

The notation $a..b$, used as a subscript, means the sequence of values with indices a through b inclusive. For example, $\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1..\mathrm{N}^{\mathsf{new}}}$ means the sequence $[\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1}, \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},2}, \ldots\ \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},\mathrm{N}^{\mathsf{new}}}]$. (For consistency with the notation in [5] and in [9], this specification uses 1-based indexing and inclusive ranges, notwithstanding the compelling arguments to the contrary in [14].)

The notation $\{a\,..\,b\}$ means the set of integers from a through b inclusive. $k\{a\,..\,b\}$ means the set containing integers kn for all $n \in \{a\,..\,b\}$.

The notation $[f(x)$ for x from a up to $b]$ means the sequence formed by evaluating f on each integer from a to b in ascending order. Similarly, $[f(x)$ for x from a down to $b]$ means the sequence formed by evaluating f on each integer from a to b in descending order.

The notation $\mathsf{concat}_{\mathbb{B}}(S)$ means the sequence of bits obtained by concatenating the elements of S viewed as bit sequences. If the elements of S are byte sequences, they are converted to bit sequences with the ***most significant*** bit of each byte first.

The notation $\mathbb{N}$ means the set of nonnegative integers.

The notation $\mathbb{F}_q$ means the finite field with q elements. $\mathbb{F}_q[z]$ means the ring of polynomials over z with coefficients in $\mathbb{F}_q$.

The notation $a \bmod q$, for positive integers a and q, means the remainder on dividing a by q.

The notation $a \oplus b$ means the bitwise exclusive-or of a and b, defined either on integers or bit sequences depending on context.

The notation $\sum_{i=1}^{\mathrm{N}} a_i$ means the sum of $a_{1..\mathrm{N}}$.

The notation $\bigoplus_{i=1}^{\mathrm{N}} a_i$ means the bitwise exclusive-or of $a_{1..\mathrm{N}}$.

The notation $\mathsf{floor}(x)$ means the largest integer $\leq x$. $\mathsf{ceiling}(x)$ means the smallest integer $\geq x$.

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

The notation $x : T$ is used to specify that x has type T. A cartesian product type is denoted by $S \times T$, and a function type by $S \to T$. A subscripted argument of a function is taken to be its first argument, e.g. if $x : X$, $y : Y$, and $\mathsf{PRF}_x(y) : Z$, then $\mathsf{PRF} : X \times Y \to Z$. An argument to a function can determine other argument or result types.

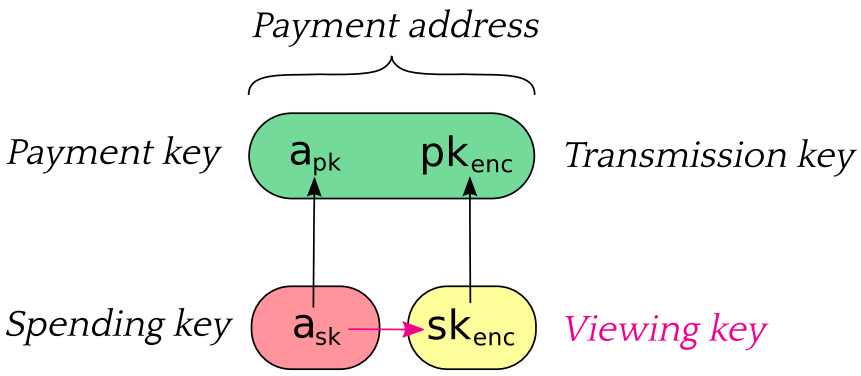
The following integer constants will be instantiated in §5.2 ***'Constants'*** on p. 16: d, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, ℓ_{Merkle}, ℓ_{General}, ℓ_{PRF}, $\ell_{\mathsf{a_{sk}}}$, ℓ_{φ}, MAX_MONEY.

3 Concepts

3.1 Payment Addresses and Keys

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

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*, *viewing keys*, and *spending keys* is a cryptographic protocol detail that should not normally be exposed to users. However, user-visible operations should be provided to obtain a *payment address* or *viewing key* from a *spending key*.

Users can accept payment from multiple parties with a single *payment address* $\mathsf{addr}_{\mathsf{pk}}$ and the fact that these payments are destined to the same payee is not revealed on the *block chain*, even to the paying parties. ***However*** if two parties collude to compare a *payment address* 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.

Note: It is conventional in cryptography to refer to the key used to encrypt a message in an asymmetric encryption scheme as the "public key". However, the public key used as the *transmission key* component of an address ($\mathsf{pk}_{\mathsf{enc}}$) need not be publically distributed; it has the same distribution as the *payment address* itself. As mentioned above, limiting the distribution of the *payment address* is important for some use cases. This also helps to reduce reliance of the overall protocol on the security of the cryptosystem used for *note* encryption (see §4.5 ***'In-band secret distribution'*** on p. 15), since an adversary would have to know $\mathsf{pk}_{\mathsf{enc}}$ in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

A *note* (denoted $\mathbf{n}$) is a tuple (a_{pk}, v, ρ, 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 sequence of ℓ_{PRF} bytes representing the *paying key* of the recipient.
- v is an integer in the range $0 \leq v \leq \mathsf{MAX\_MONEY}$ representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*).
- ρ is a sequence of ℓ_{PRF} bytes, which is used as input to $\mathsf{PRF}^{\mathsf{nf}}_{a_{sk}}$ to obtain the *note*'s *nullifier*.
- r is a *commitment trapdoor*.

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

3.2.1 Note Commitments

The underlying v and a_{pk} are blinded with ρ and r. The resulting hash $\mathsf{cm} = \mathtt{NoteCommitment}(\mathbf{n})$.

$\mathtt{NoteCommitment}$ is required to be a computationally binding and hiding commitment scheme.

3.2.2 Nullifiers

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

3.2.3 Note Plaintexts and Memo Fields

Transmitted *notes* are stored on the *block chain* in encrypted form, together with a *note commitment* cm.

The *note plaintexts* in 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 §4.5 ***'In-band secret distribution'*** on p. 15 for further details).

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

The first three of these fields are as defined earlier.

memo represents a *memo field* associated with this *note*. The usage of the *memo field* is by agreement between the sender and recipient of the *note*.

3.3 Transactions, Blocks, and the Block Chain

At a given point in time, the *block chain 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*. To each *transaction* there is associated an initial *treestate*, which consists of a *note commitment tree* (§ 3.5 *'Note Commitment Tree'* on p. 8), *nullifier set* (§ 3.6 *'Nullifier Set'* on p. 8), and data structures associated with **Bitcoin** such as the UTXO (Unspent Transaction Output) set.

Inputs to a *transaction* insert value into a *value pool*, and outputs remove value from this pool. As in **Bitcoin**, the remaining value in the pool is available to miners as a fee.

An *anchor* is a Merkle tree root of a *note commitment tree*. It uniquely identifies a *note commitment tree* state given the assumed security properties of the Merkle tree's hash function. Since the *nullifier set* is always updated together with the *note commitment tree*, this also identifies a particular state of the *nullifier set*.

In a given node's *block chain view*, *treestates* are chained as follows:

- 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*.

Daira: *JoinSplit descriptions* also have input and output *treestates*.

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*.

3.4 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^{old} *notes* $\mathbf{n}^{old}_{1..N^{old}}$ and transparent input v^{old}_{pub}, and creates N^{new} *notes* $\mathbf{n}^{new}_{1..N^{new}}$ and transparent output v^{new}_{pub}.

Each *transaction* is associated with a sequence of *JoinSplit descriptions*.

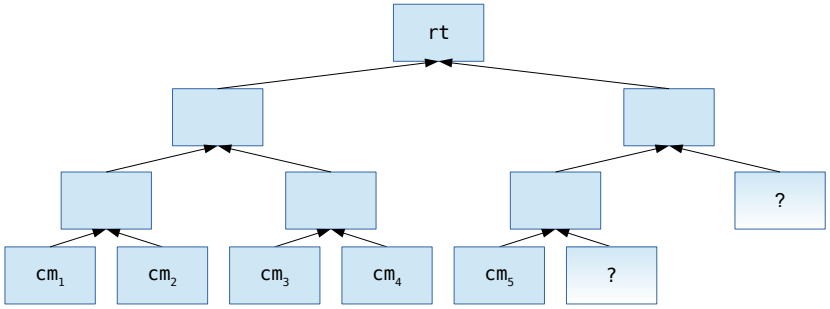
The inputs and outputs of each *JoinSplit operation* **MUST** balance exactly. The total v^{new}_{pub} value adds to, and the total v^{old}_{pub} value subtracts from the value pool of the containing *transaction*.

TODO: Describe the interaction of transparent value flows with the *JoinSplit description*'s v^{old}_{pub} and v^{new}_{pub}.

The *anchor* of each *JoinSplit description* in a *transaction* must refer to either some earlier *block*'s final *treestate*, or to the output *treestate* of any prior *JoinSplit description* in the same *transaction*.

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

3.5 Note Commitment Tree



The *note commitment tree* is an *incremental Merkle tree* of fixed depth 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 *block chain* 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 *note commitment tree* associated with the previous *block*. Daira: Make this more precise.

Each *node* in the *incremental Merkle tree* is associated with a *hash value* of size ℓ_{Merkle} bytes. The *layer* numbered h, counting from *layer* 0 at the *root*, has 2^h *nodes* with *indices* 0 to $2^h - 1$ inclusive. The *hash value* associated with the *node* at *index* i in *layer* h is denoted M_i^h.

3.6 Nullifier Set

Each *full node* maintains a *nullifier set* alongside the *note commitment tree* and UTXO set. As valid *transactions* containing *JoinSplit operations* are processed, the *nullifiers* revealed in *JoinSplit descriptions* are inserted into this *nullifier set*.

If a *JoinSplit description* reveals a *nullifier* that already exists in the *full node*'s *block chain view*, the containing transaction will be rejected, since it would otherwise result in a double-spend.

3.7 Coinbase Transactions

The first *transaction* in a block must be a *coinbase transaction*, which should collect and spend any block reward and transaction fees paid by *transactions* included in this block.

3.7.1 Block Subsidy and Transaction Fees

TODO: Describe money supply curve. TODO: Miner's reward = transaction fees + block subsidy - founder's reward

3.7.2 Coinbase outputs

TODO: Coinbase maturity rule. TODO: Any tx with a coinbase input must have no transparent outputs (vout).

4 Abstract Protocol

4.1 Abstract Cryptographic Functions

4.1.1 Hash Functions

$\mathsf{MerkleCRH} : \mathbb{B}^{\ell_{\mathsf{Merkle}}} \times \mathbb{B}^{\ell_{\mathsf{Merkle}}} \to \mathbb{B}^{\ell_{\mathsf{Merkle}}}$ is a collision-resistant hash function used in §4.4.2 *'Merkle root validity'* on p. 12. It is instantiated in §5.3.1 *'Merkle Tree Hash Function'* on p. 17.

$\mathsf{GeneralCRH} : (\ell : 8\{1\,..\,64\}) \times \mathbb{B}^\star \to \mathbb{B}^\ell$ is another collision-resistant hash function. The first (subscripted) argument indicates the output length in bits. It is used in §4.4.1 *'Computation of* $\mathsf{h}_{\mathsf{Sig}}$*'* on p. 12 and §7.2.1 *'Equihash'* on p. 25, and instantiated in §5.3.2 *'General Hash Function'* on p. 17.

4.1.2 Pseudo Random Functions

PRF_x is a *Pseudo Random Function* seeded by x. Four ***independent*** PRF_x are needed in our protocol:

$$\begin{aligned} \mathsf{PRF}^{\mathsf{addr}} &: \\ \mathsf{PRF}^{\mathsf{nf}} &: \\ \mathsf{PRF}^{\mathsf{pk}} &: \\ \mathsf{PRF}^{\rho} &: . \end{aligned}$$

These are used in §4.4.6 *'JoinSplit Circuit'* on p. 14, and instantiated in §5.3.3 *'Pseudo Random Functions'* on p. 17.

Security requirement: In addition to being *Pseudo Random Functions*, it is required that $\mathsf{PRF}^{\mathsf{nf}}_x$, $\mathsf{PRF}^{\mathsf{addr}}_x$, and PRF^{ρ}_x be collision-resistant across all x – i.e. it should not be feasible to find $(x, y) \neq (x', y')$ such that $\mathsf{PRF}^{\mathsf{nf}}_x(y) = \mathsf{PRF}^{\mathsf{nf}}_{x'}(y')$, and similarly for $\mathsf{PRF}^{\mathsf{addr}}$ and PRF^{ρ}.

4.1.3 Authenticated One-Time Symmetric Encryption

Let Sym be an *authenticated one-time symmetric encryption scheme* with keyspace $\mathsf{Sym}.\mathbf{K}$, encrypting plaintexts in $\mathsf{Sym}.\mathbf{P}$ to produce ciphertexts in $\mathsf{Sym}.\mathbf{C}$.

$\mathsf{Sym}.\mathtt{Encrypt} : \mathsf{Sym}.\mathbf{K} \times \mathsf{Sym}.\mathbf{P} \to \mathsf{Sym}.\mathbf{C}$ is the encryption algorithm.

$\mathsf{Sym}.\mathtt{Decrypt} : \mathsf{Sym}.\mathbf{K} \times \mathsf{Sym}.\mathbf{C} \to \mathsf{Sym}.\mathbf{P} \cup \{\bot\}$ is the corresponding decryption algorithm, such that for any $\mathsf{K} \in \mathsf{Sym}.\mathbf{K}$ and $\mathsf{P} \in \mathsf{Sym}.\mathbf{P}$, $\mathsf{Sym}.\mathtt{Decrypt}_{\mathsf{K}}(\mathsf{Sym}.\mathtt{Encrypt}_{\mathsf{K}}(\mathsf{P})) = \mathsf{P}$. $\bot$ is used to represent the decryption of an invalid ciphertext.

Security requirement: Sym must be one-time (INT-CTXT $\wedge$ IND-CPA)-secure. "One-time" here means that an honest protocol participant will almost surely encrypt only one message with a given key; however, the attacker may make many adaptive chosen ciphertext queries for a given key. The security notions INT-CTXT and IND-CPA are as defined in [4].

4.1.4 Key Agreement

A *key agreement scheme* is a cryptographic protocol in which two parties agree a shared secret, each using their private key and the other party's public key.

A *key agreement scheme* KA defines a type of public keys KA.Public, a type of private keys KA.Private, and a type of shared secrets KA.SharedSecret.

Let KA.`FormatPrivate` $: \mathbb{B}^{\ell_{\mathsf{PRF}}} \to$ KA.Private be a function that converts a bit string of length ℓ_{PRF} to a KA private key.

Let KA.`DerivePublic` $:$ KA.Private $\to$ KA.Public be a function that derives the KA public key corresponding to a given KA public key.

Let KA.`Agree` $:$ KA.Private $\times$ KA.Public $\to$ KA.SharedSecret be the agreement function.

Security requirement: KA.`FormatPrivate` must preserve sufficient entropy from its input to be used as a secure KA private key. TODO: requirements on security of key agreement and KDF

4.1.5 Key Derivation

A *Key Derivation Function* is defined for a particular *key agreement scheme* and *authenticated one-time symmetric encryption scheme*; it takes the shared secret produced by the key agreement and additional arguments, and derives a key suitable for the encryption scheme.

Let KDF $: \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{\ell_{\mathsf{General}}} \times$ KA.SharedSecret $\times$ KA.Public $\times$ KA.Public $\to$ Sym.$\mathbf{K}$ be a *Key Derivation Function* suitable for use with KA, deriving keys for Sym.`Encrypt`.

Security requirement: For any $T = (i \in \{1..\mathrm{N}^{\mathsf{new}}\}, \mathsf{h}_{\mathsf{Sig}} \in \mathbb{B}^{\ell_{\mathsf{General}}}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i} \in$ KA.Public$)$,

> $(\mathsf{epk}, \mathsf{KDF}(i, \mathsf{h}_{\mathsf{Sig}}, \mathsf{KA.Agree}(\mathsf{esk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},}), \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},}))$ must be computationally indistinguishable between different $\mathsf{sk}_{\mathsf{enc}} \in$ KA.Private,
>
> where $\mathsf{epk} =$ KA.`DerivePublic`(esk) and $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},} =$ KA.`DerivePublic`$(\mathsf{sk}_{\mathsf{enc}})$.

This is necessary to ensure that the composition of KA, KDF and Sym as given in §4.5 *'In-band secret distribution'* on p. 15 is a key-private asymmetric encryption scheme. The property of key privacy is defined in [3].

4.1.6 Signatures

TODO:

4.2 Key Components

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

Let KA be a *key agreement scheme*, instantiated in **??** *'??'* on p. **??**.

A new *spending key* a_{sk} is generated by sampling a bit string uniformly at random from $\mathbb{B}^{\ell_{\mathsf{a}_{\mathsf{sk}}}}$.

a_{pk}, $\mathsf{sk}_{\mathsf{enc}}$ and $\mathsf{pk}_{\mathsf{enc}}$ are derived from a_{sk} as follows:

$$
\begin{aligned}
\mathsf{a}_{\mathsf{pk}} &:= \mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a}_{\mathsf{sk}}}(0) \\
\mathsf{sk}_{\mathsf{enc}} &:= \mathsf{KA.FormatPrivate}(\mathsf{PRF}^{\mathsf{addr}}_{\mathsf{a}_{\mathsf{sk}}}(1)) \\
\mathsf{pk}_{\mathsf{enc}} &:= \mathsf{KA.DerivePublic}(\mathsf{sk}_{\mathsf{enc}})
\end{aligned}
$$

where

- 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 [7];
- $\underline{9}$ 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 [7]: "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.

4.3 Note Components

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

4.4 JoinSplit Operations and Descriptions

A *JoinSplit description* is data included in a *transaction* that describes a *JoinSplit operation*, as described in §3.4 ***'JoinSplit Operations and Descriptions'*** on p. 7.

Zcash *transactions* have the following additional fields:

Bytes	Name	Data Type	Description
Varies	`nJoinSplit`	`compactSize uint`	The number of *JoinSplit descriptions* in `vJoinSplit`.
$1802 \times$ `nJoinSplit`	`vJoinSplit`	`JoinSplitDescription [nJoinSplit]`	The sequence of *JoinSplit descriptions* in this *transaction*.
32 †	`joinSplitPubKey`	`char[32]`	An encoding of an JoinSplitSigAlg public verification key
64 †	`joinSplitSig`	`char[64]`	A signature on a prefix of the *transaction* encoding, to be verified using `joinSplitPubKey`.

† 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 §4.4.3 ***'Non-malleability'*** on p. 13.

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 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* $nf^{old}_{1..N^{old}}$.
64	`commitments`	`char[32]` [N^{new}].	A sequence of *note commitments* for the output *notes* $cm^{new}_{1..N^{new}}$.
32	`ephemeralKey`	`char[32]`	A Curve25519 public key epk.
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 $h_{1..N^{old}}$ that bind h_{Sig} to each a_{sk} of the *JoinSplit description*.
296	`zkproof`	`char[296]`	An encoding of the zero-knowledge proof $\pi_{\texttt{JoinSplit}}$ (§6.2 *'Encoding of Zero-Knowledge Proofs'* on p. 23).
1202	`encCiphertexts`	`char[601]` [N^{new}]	A sequence of ciphertext components for the encrypted output *notes*, $C^{enc}_{1..N^{new}}$.

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

Consensus rule: $0 \leq v^{old}_{pub} \leq$ MAX_MONEY, and $0 \leq v^{new}_{pub} \leq$ MAX_MONEY.

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

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

4.4.1 Computation of h_{Sig}

Given a *JoinSplit description* containing the fields `randomSeed` and `nullifiers` = $nf^{old}_{1..N^{old}}$, and embedded in a transaction containing the field `joinSplitPubKey`, we compute h_{Sig} for that *JoinSplit description* as follows:

hSigInput := | 256-bit `randomSeed` | 256-bit nf^{old}_1 ... | 256-bit $nf^{old}_{N^{old}}$ | 256-bit `joinSplitPubKey` |

h_{Sig} := GeneralCRH$_{256}$("ZcashComputehSig", hSigInput)

4.4.2 Merkle root validity

Daira: This paragraph is confusing and only describes one aspect of 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*.

The depth of the *note commitment tree* is d.

Each *node* in the *incremental Merkle tree* is associated with a *hash value*, which is a byte sequence. The *layer* numbered h, counting from *layer* 0 at the *root*, has 2^h *nodes* with *indices* 0 to $2^h - 1$ inclusive.

Let M_i^h be the *hash value* associated with the *node* at *index* i in *layer* h.

The *nodes* at *layer* d are called *leaf nodes*. When a *note commitment* is added to the tree, it occupies the *leaf node hash value* $\mathsf{M}_i^{\mathsf{d}}$ for the next available i. As-yet unused *leaf nodes* are associated with a distinguished *hash value* $\mathsf{Uncommitted}$. It is assumed to be infeasible to find a preimage *note* $\mathbf{n}$ such that $\mathtt{NoteCommitment}(\mathbf{n}) = \mathsf{Uncommitted}$.

The *nodes* at *layers* 0 to $\mathsf{d} - 1$ inclusive are called *internal nodes*, and are associated with $\mathsf{MerkleCRH}$ outputs. *Internal nodes* are computed from their children in the next *layer* as follows: for $0 \leq h < \mathsf{d}$ and $0 \leq i < 2^h$,

$$\mathsf{M}_i^h := \mathsf{MerkleCRH}(\mathsf{M}_{2i}^{h+1}, \mathsf{M}_{2i+1}^{h+1}).$$

A *path* from *leaf node* $\mathsf{M}_i^{\mathsf{d}}$ in the *incremental Merkle tree* is the sequence

$$[\mathsf{M}_{\mathsf{sibling}(h,i)}^h \text{ for } h \text{ from } \mathsf{d} \text{ down to } 1],$$

where

$$\mathsf{sibling}(h, i) = \mathsf{floor}\left(\tfrac{i}{2^{\mathsf{d}-h}}\right) \oplus 1$$

Given such a *path*, it is possible to verify that *leaf node* $\mathsf{M}_i^{\mathsf{d}}$ is in a tree with a given *root* $\mathsf{rt} = \mathsf{M}_0^0$.

4.4.3 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 `nJoinSplit`, `vJoinSplit`, and (if present) `joinSplitPubKey`. They ***do not*** cover the field `joinSplitSig`.

Consensus rule: If `nJoinSplit` > 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*. 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 $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$ and $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$, and to the other *JoinSplit descriptions* in the same *transaction*, an ephemeral JoinSplitSigAlg 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`.

JoinSplitSigAlg is instantiated as Ed25519 [8], with the additional requirement that S (the integer represented by $\underline{S}$) must be less than the prime $\ell = 2^{252} + 27742317777372353535851937790883648493$ defined in [8], otherwise the signature is considered invalid. Ed25519 is defined as using SHA-512 internally.

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 as specified above, using the Ed25519 public key encoded as `joinSplitPubKey`.

The encoding of a signature is:

256-bit $\underline{R}$	256-bit $\underline{S}$

where $\underline{R}$ and $\underline{S}$ are as defined in [8].

The encoding of a public key is as defined in [8].

The condition enforced by the *JoinSplit circuit* specified in §4.4.6 ***'Non-malleability'*** on p. 15 ensures that a holder of all of $\mathsf{a}_{\mathsf{sk},1..\mathsf{N}^{\mathsf{old}}}^{\mathsf{old}}$ for each *JoinSplit description* has authorized the use of the private signing key corresponding to `joinSplitPubKey` to sign this *transaction*.

4.4.4 Balance

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

Note: Unlike original **Zerocash** [5], **Zcash** does not have a distinction between Mint and Pour operations. The addition of v^{old}_{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 §4.4 ***'JoinSplit Operations and Descriptions'*** on p. 11, either v^{old}_{pub} or v^{new}_{pub} **MUST** be zero. No generality is lost because, if a *transaction* in which both v^{old}_{pub} and v^{new}_{pub} were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(v^{old}_{pub}, v^{new}_{pub})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

4.4.5 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 *block chain 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.

4.4.6 JoinSplit Circuit

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

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

there exists a witness of *auxiliary input*:

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

where:

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..N^{old}\} \mid v^{old}_i \neq 0$: path_i must be a valid *path* of depth d, as defined in § 3.5 ***'Note Commitment Tree'*** on p. 8, from $\texttt{NoteCommitment}(\mathbf{n}^{old}_i)$ to *note commitment tree* root rt.

Note: Merkle path validity covers both conditions 1. (a) and 1. (d) of the NP statement given in section 4.2 of [5].

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

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

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

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

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

Commitment integrity for each $i \in \{1..N^{new}\}$: $cm^{new}_i = \texttt{NoteCommitment}(\mathbf{n}^{new}_i)$.

For details of the form and encoding of proofs, see §6 *'Zero-Knowledge Proving System'* on p. 21.

4.5 In-band secret distribution

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

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

4.5.1 Encryption

Let $\text{Sym.}\texttt{Encrypt}_K(P)$ be authenticated encryption using AEAD_CHACHA20_POLY1305 [21] encryption of plaintext $P \in \text{Sym.}\mathbf{P}$, with empty "associated data", all-zero nonce $[0]^{96}$, and 256-bit key $K \in \text{Sym.}\mathbf{K}$.

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

Let $pk^{new}_{enc,1..N^{new}}$ be the Curve25519 public keys for the intended recipient addresses of each new *note*, and let $\mathbf{np}_{1..N^{new}}$ be the *note plaintexts*. Let h_{Sig} be the value computed in §4.4.1 *'Computation of* h_{Sig}*'* on p. 12. Let KDF be the *Key Derivation Function* instantiated in §5.3.6 *'Key Derivation'* on p. 19.

Then to encrypt:

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

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

4.5.2 Decryption by a Recipient

Let $addr_{pk} = (a_{pk}, pk_{enc})$ be the recipient's *payment address*, and let sk_{enc} be the recipient's *viewing key*. Let h_{Sig} be the value computed in §4.4.1 *'Computation of* h_{Sig}*'* on p. 12. Let $cm^{new}_{1..N^{new}}$ be the *note commitments* of each output coin. Then for each $i \in \{1..N^{new}\}$, the recipient will attempt to decrypt that ciphertext component as follows:

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

$\texttt{DecryptNote}(K_i^{enc}, C_i^{enc}, cm_i^{new}, a_{pk})$ is defined as follows:

- Let $P_i^{enc} := \text{Sym.}\texttt{Decrypt}_{K_i^{enc}}(C_i^{enc})$.
- If $P_i^{enc} = \bot$, return $\bot$.
- Extract $\mathbf{np}_i = (v_i^{new}, \rho_i^{new}, r_i^{new}, \text{memo}_i)$ from P_i^{enc}.
- If $\texttt{NoteCommitment}((a_{pk}, v_i^{new}, \rho_i^{new}, r_i^{new})) \neq cm_i^{new}$, return $\bot$, else return $\mathbf{np}_i$.

Note: This corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in Figure 2 of [5].

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

Notes:

- A *note* can change from being unspent to spent on a given *block chain view*, as *transactions* are added to that view. Also, blockchain reorganisations can cause the *transaction* in which a *note* was output to no longer be on the consensus blockchain.
- The nonce parameter to AEAD_CHACHA20_POLY1305 is not used.
- The "IETF" definition of AEAD_CHACHA20_POLY1305 from [21] 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.

See § 8.7 *'In-band secret distribution'* on p. 28 for further discussion of the security and engineering rationale behind this encryption scheme.

5 Concrete Protocol

5.1 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 *unless otherwise specified*.

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.

The entire diagram represents the sequence of ***bytes*** formed by first concatenating these bit sequences, and then treating each subsequence of 8 bits as a byte with the bits ordered from *most significant* to *least significant*. Thus the *most significant* bit in each byte is toward the left of a diagram. Where bit fields are used, the text will clarify their position in each case.

5.2 Constants

Define:

$d = 32$

$N^{old} = 2$

$N^{new} = 2$

$\ell_{Merkle} = 256$

$\ell_{General} = 256$

$\ell_{PRF} = 256$

$\ell_{\mathsf{a_{sk}}} = 252$

$\ell_{\varphi} = 252$

$\mathsf{Uncommitted} = [0]^{\ell_{\mathsf{Merkle}}}$

$\mathsf{MAX\_MONEY} = 2.1 \times 10^{15}$.

5.3 Concrete Cryptographic Functions

5.3.1 Merkle Tree Hash Function

MerkleCRH is used to hash *incremental Merkle tree hash values*. It is instantiated by the *SHA-256 compression* function, which takes a 512-bit block and produces a 256-bit hash. [22]

$\mathsf{MerkleCRH}(\mathsf{left}, \mathsf{right}) := \texttt{SHA256Compress}\left(\boxed{\text{256-bit left}}\boxed{\text{256-bit right}}\right)$.

Note: `SHA256Compress` is not the same as the SHA-256 function, which hashes arbitrary-length sequences.

Security requirement: `SHA256Compress` must be collision-resistant, and it must be infeasible to find a preimage x such that $\texttt{SHA256Compress}(x) = [0]^{256}$.

5.3.2 General Hash Function

$\mathsf{GeneralCRH}_\ell$ is a collision-resistant hash function. It is used in the computation of $\mathsf{h_{Sig}}$ in §4.4.1 *'Computation of* $\mathsf{h_{Sig}}$*'* on p. 12.

It is instantiated by BLAKE2b-ℓ, that is, BLAKE2b with an output digest length of $\ell/8$ bytes. $\mathsf{GeneralCRH}(p, x)$ applies unkeyed BLAKE2b-ℓ, as defined in [2], to a 16-byte personalization string p and input x.

Note: BLAKE2b-ℓ is not the same as BLAKE2b-512 truncated to ℓ bits.

Security requirement: BLAKE2b-$\ell(p, x)$ must be collision-resistant for any given (not adversarially chosen) ℓ and p.

5.3.3 Pseudo Random Functions

PRF_x is a *Pseudo Random Function* seeded by x. Four ***independent*** PRF_x are needed in our scheme: $\mathsf{PRF}^{\mathsf{addr}}_x$, $\mathsf{PRF}^{\mathsf{nf}}_x$, $\mathsf{PRF}^{\mathsf{pk}}_x$, and PRF^{ρ}_x.

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

In **Zcash**, the *SHA-256 compression* function is used to construct all of these functions.

$$\mathsf{PRF}^{\mathsf{addr}}_{x}(t) := \mathtt{SHA256Compress}\Big(\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{252\text{-bit } x}\boxed{8\text{-bit } t}\boxed{[0]^{248}}\Big)$$

$$\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho) := \mathtt{SHA256Compress}\Big(\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{252\text{-bit } \mathsf{a_{sk}}}\boxed{256\text{-bit } \rho}\Big)$$

$$\mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}}) := \mathtt{SHA256Compress}\Big(\boxed{0}\boxed{i\text{-}1}\boxed{0}\boxed{0}\boxed{252\text{-bit } \mathsf{a_{sk}}}\boxed{256\text{-bit } \mathsf{h_{Sig}}}\Big)$$

$$\mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}}) := \mathtt{SHA256Compress}\Big(\boxed{0}\boxed{i\text{-}1}\boxed{1}\boxed{0}\boxed{252\text{-bit } \varphi}\boxed{256\text{-bit } \mathsf{h_{Sig}}}\Big)$$

Note: The first four bits –i.e. the most significant four bits of the first byte– are used to distinguish different uses of `SHA256Compress`, 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 SHA-256 hash function — see § 5.5 *'Note Commitments'* on p. 19. (The specific bit patterns chosen here are motivated by the possibility of future extensions that either increase $\mathsf{N}^{\mathsf{old}}$ and/or $\mathsf{N}^{\mathsf{new}}$ to 3, or that add an additional bit to $\mathsf{a_{sk}}$ to encode a new key type, or that require an additional PRF.)

5.3.4 Authenticated One-Time Symmetric Encryption

Let Sym be an *authenticated one-time symmetric encryption scheme* with keyspace $\mathsf{Sym}.\mathbf{K}$, encrypting plaintexts in $\mathsf{Sym}.\mathbf{P}$ to produce ciphertexts in $\mathsf{Sym}.\mathbf{C}$.

$\mathsf{Sym}.\mathtt{Encrypt} : \mathsf{Sym}.\mathbf{K} \times \mathsf{Sym}.\mathbf{P} \to \mathsf{Sym}.\mathbf{C}$ is the encryption algorithm.

$\mathsf{Sym}.\mathtt{Decrypt} : \mathsf{Sym}.\mathbf{K} \times \mathsf{Sym}.\mathbf{C} \to \mathsf{Sym}.\mathbf{P} \cup \{\bot\}$ is the corresponding decryption algorithm, such that for any $\mathsf{K} \in \mathsf{Sym}.\mathbf{K}$ and $\mathsf{P} \in \mathsf{Sym}.\mathbf{P}$, $\mathsf{Sym}.\mathtt{Decrypt}_{\mathsf{K}}(\mathsf{Sym}.\mathtt{Encrypt}_{\mathsf{K}}(\mathsf{P})) = \mathsf{P}$. $\bot$ is used to represent the decryption of an invalid ciphertext.

Security requirement: Sym must be one-time (INT-CTXT $\wedge$ IND-CPA)-secure. "One-time" here means that an honest protocol participant will almost surely encrypt only one message with a given key; however, the attacker may make many adaptive chosen ciphertext queries for a given key. The security notions INT-CTXT and IND-CPA are as defined in [4].

5.3.5 Key Agreement

A *key agreement scheme* is a cryptographic protocol in which two parties agree a shared secret, each using their private key and the other party's public key.

A *key agreement scheme* KA defines a type of public keys KA.Public, a type of private keys KA.Private, and a type of shared secrets KA.SharedSecret.

Let $\mathsf{KA}.\mathtt{FormatPrivate} : \mathbb{B}^{\ell_{\mathsf{PRF}}} \to \mathsf{KA}.\mathsf{Private}$ be a function that converts a bit string of length ℓ_{PRF} to a KA private key.

Let $\mathsf{KA}.\mathtt{DerivePublic} : \mathsf{KA}.\mathsf{Private} \to \mathsf{KA}.\mathsf{Public}$ be a function that derives the KA public key corresponding to a given KA public key.

Let $\mathsf{KA}.\mathtt{Agree} : \mathsf{KA}.\mathsf{Private} \times \mathsf{KA}.\mathsf{Public} \to \mathsf{KA}.\mathsf{SharedSecret}$ be the agreement function.

Security requirement: KA.FormatPrivate must preserve sufficient entropy from its input to be used as a secure KA private key. TODO: requirements on security of key agreement and KDF

where

- 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 [7];
- $\underline{9}$ is the public byte sequence representing the Curve25519 base point;
- $\text{clamp}_{\text{Curve25519}}(\underline{x})$ 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 [7]: "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.

5.3.6 Key Derivation

The *Key Derivation Function* specified in § 4.1.5 ***'Key Derivation'*** on p. 10 is instantiated using BLAKE2b-256 as follows:

$\mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{dhsecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}) := \mathsf{BLAKE2b\text{-}256}(\mathsf{kdftag}, \mathsf{kdfinput})$

where:

$\mathsf{kdftag} :=$ | 64-bit **"ZcashKDF"** | 8-bit $i-1$ | $[0]^{56}$ |

$\mathsf{kdfinput} :=$ | 256-bit $\mathsf{h_{Sig}}$ | 256-bit $\mathsf{dhsecret}_i$ | 256-bit epk | 256-bit $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}$ |.

5.3.7 Signatures

TODO:

5.4 Note Components

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

5.5 Note Commitments

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

$\mathsf{cm} := \mathsf{SHA256}\Big($ | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 256-bit $\mathsf{a_{pk}}$ | 64-bit v | 256-bit ρ | 256-bit r | $\Big)$

Note: The leading byte of the SHA256 input is $\mathbf{0xB0}$.

5.6 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* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, and the result forms part of a *transmitted notes ciphertext* (see § 4.5 ***'In-band secret distribution'*** on p. 15 for further details).

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

The first three of these fields are as defined earlier. memo is a 512-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 field* **SHOULD** be encoded either as:

- a UTF-8 human-readable string [12], padded by appending zero bytes; or
- an arbitrary sequence of 512 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:

8-bit **0x00**	64-bit v	256-bit ρ	256-bit r	memo (512 bytes)

- A byte, **0x00**, indicating this version of the encoding of a *note plaintext*.
- 8 bytes specifying v.
- 32 bytes specifying ρ.
- 32 bytes specifying r.
- 512 bytes specifying memo.

5.7 Encodings of Addresses and Keys

This section describes how **Zcash** encodes *payment addresses*, *viewing keys*, 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 [10].

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

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

5.7.1 Transparent Payment Addresses

These are encoded in the same way as in **Bitcoin** [10].

5.7.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [10].

5.7.3 Protected Payment Addresses

A *payment address* consists of a_{pk} and pk_{enc}. a_{pk} is a *SHA-256 compression* output. pk_{enc} is a Curve25519 public key, for use with the encryption scheme defined in §4.5 ***'In-band secret distribution'*** on p. 15.

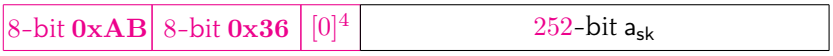
The raw encoding of a *payment address* consists of:

- Two bytes $[\mathbf{0x16}, \mathbf{0x9A}]$, indicating this version of the raw encoding of a **Zcash** *payment address* on the production network. (Addresses on the test network use $[\mathbf{0x14}, \mathbf{0x51}]$ instead.)
- 256 bits specifying a_{pk}.
- 256 bits specifying pk_{enc}, using the normal encoding of a Curve25519 public key [7].

5.7.4 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:



- Two bytes $[\mathbf{0xAB}, \mathbf{0x36}]$, indicating this version of the raw encoding of a **Zcash** *spending key* on the production network. (Addresses on the test network use $[\mathbf{0xB1}, \mathbf{0xEB}]$ instead.)
- 4 zero padding bits.
- 252 bits specifying a_{sk}.

The zero padding occupies the most significant 4 bits of the third 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.

6 Zero-Knowledge Proving System

Zcash uses *zk-SNARKs* generated by its fork of *libsnark* [18] using the proving system described in [6], which is a refinement of the system in [23].

The pairing implementation is `ALT_BN128`.

Let $q = 21888242871839275222246405745257275088696311157297823662689037894645226208583$.

Let $r = 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

Let $b = 3$.

(q and r are prime.)

The pairing is of type $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where:

- $\mathbb{G}_1$ is a Barreto–Naehrig curve over $\mathbb{F}_q$ with equation $y^2 = x^3 + b$.
- $\mathbb{G}_2$ is a twisted Barreto–Naehrig curve over $\mathbb{F}_{q^2}$ with equation $y^2 = x^3 + b/xi$. We represent elements of $\mathbb{F}_{q^2}$ as polynomials $a_1 t + a_0 : \mathbb{F}_q[t]$, modulo the irreducible polynomial $t^2 + 1$.
- $\mathbb{G}_T$ is $\mathbb{F}_{q^{12}}$.

Let $\mathcal{P}_1 : \mathbb{G}_1 = (1, 2)$.

Let $\mathcal{P}_2 : \mathbb{G}_2 = (11559732032986387107991004021392285783925812861821192530917403151452391805634\,t +$
$10857046999023057135944570762232829481370756359578518086990519993285655852781,$
$40823678758634336813322034031454355683168513275934012081057410762141200935 31\,t +$
$84956539231234314176049732474892724384181905872636001487702806493069581019 30)$.

The curves $\mathbb{G}_1$ and $\mathbb{G}_2$ both have prime order r, and so $\mathcal{P}_1$ and $\mathcal{P}_2$ are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

A proof consists of a tuple $(\pi_A : \mathbb{G}_1,\ \pi'_A : \mathbb{G}_1,\ \pi_B : \mathbb{G}_2,\ \pi'_B : \mathbb{G}_1,\ \pi_C : \mathbb{G}_1,\ \pi'_C : \mathbb{G}_1,\ \pi_K : \mathbb{G}_1,\ \pi_H : \mathbb{G}_1)$. It is computed as described in Appendix B of [6].

Note: Many details of the proving system are beyond the scope of this protocol document. For example, the *Rank 1 Constraint System* corresponding to the *JoinSplit circuit* is not specified here. In practice it will be necessary to use the specific proving and verification keys generated for the **Zcash** production *block chain*, and a proving system implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

6.1 Encoding of Points

Define $\mathsf{I2OSP} : (k : \mathbb{N}) \times \{0\,..\,256^k - 1\} \to \{0\,..\,255\}^k$ such that $\mathsf{I2OSP}_\ell(n)$ is the sequence of ℓ bytes representing n in big-endian order.

For a point $P : \mathbb{G}_1 = (x_P, y_P)$:

- The field elements x_P and $y_P : \mathbb{F}_q$ are represented as integers x and $y : \{0\,..\,q-1\}$.
- Let $\tilde{y} = y \bmod 2$.
- P is encoded as

0	0	0	0	0	0	1	1-bit $\tilde{y}$	256-bit $\mathsf{I2OSP}_{32}(x)$

.

For a point $P : \mathbb{G}_2 = (x_P, y_P)$:

- A field element $w : \mathbb{F}_{q^2}$ is represented as a polynomial $a_{w,1}t + a_{w,0} : \mathbb{F}_q[t]$ modulo $t^2 + 1$. Define $\mathsf{FE2IP} : \mathbb{F}_{q^2} \to \{0\,..\,q^2 - 1\}$ such that $\mathsf{FE2IP}(w) = a_{w,1}q + a_{w,0}$.
- Let $x = \mathsf{FE2IP}(x_P)$, $y = \mathsf{FE2IP}(y_P)$, and $y' = \mathsf{FE2IP}(-y_P)$.
- Let $\tilde{y} = \begin{cases} 1, & \text{if } y > y' \\ 0, & \text{otherwise.} \end{cases}$
- P is encoded as

0	0	0	0	1	0	1	1-bit $\tilde{y}$	512-bit $\mathsf{I2OSP}_{64}(x)$

.

Non-normative notes:

- The use of big-endian byte order is different from the encoding of other integers in this protocol. The above encodings are consistent with the definition of EC2OSP for compressed curve points in section 5.5.6.2 of IEEE Std 1363a-2004 [25]. The LSB compressed form (i.e. EC2OSP-XL) is used for points on $\mathbb{G}_1$, and the SORT compressed form (i.e. EC2OSP-XS) for points on $\mathbb{G}_2$.
- Testing $y > y'$ for the compression of $\mathbb{G}_2$ points is equivalent to testing whether $(a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})$ in lexicographic order.
- Algorithms for decompressing points from the above encodings are given in Appendix A.12.8 of [24] for $\mathbb{G}_1$, and Appendix A.12.11 of [25] for $\mathbb{G}_2$.

When computing square roots in $\mathbb{F}_q$ or $\mathbb{F}_{q^2}$ in order to decompress a point encoding, the implementation **MUST NOT** assume that the square root exists, or that the encoding represents a point on the curve.

6.2 Encoding of Zero-Knowledge Proofs

A proof is encoded by concatenating the encodings of its elements:

264-bit π_A	264-bit π'_A	520-bit π_B	264-bit π'_B	264-bit π_C	264-bit π'_C	264-bit π_K	264-bit π_H

The resulting proof size is 296 bytes.

In addition to the steps to verify a proof given in [6] Appendix B, the verifier **MUST** check, for the encoding of each element, that:

- the lead byte is of the required form;
- the remaining bytes encode a big-endian representation of an integer in $\{0 \,..\, q-1\}$ or (in the case of π_B) $\{0 \,..\, q^2-1\}$;
- the encoding represents a point on the relevant curve.

7 Consensus Changes from Bitcoin

7.1 Block Headers

The **Zcash** *block header* format is as follows:

Bytes	Name	Data Type	Description
4	`nVersion`	`int32_t`	The *block version number* indicates which set of *block* validation rules to follow. The current and only defined *block version number* for **Zcash** is 4.
32	`hashPrevBlock`	`char[32]`	A *SHA-256d* hash in internal byte order of the previous *block*'s header. This ensures no previous *block* can be changed without also changing this *block*'s header.
32	`hashMerkleRoot`	`char[32]`	A *SHA-256d* hash in internal byte order. The merkle root is derived from the hashes of all *transactions* included in this *block*, ensuring that none of those *transactions* can be modified without modifying the header.
32	`hashReserved`	`char[32]`	A reserved field which should be ignored.
4	`nTime`	`uint32_t`	The *block time* is a Unix epoch time when the miner started hashing the header (according to the miner). This **MUST** be greater than or equal to the median time of the previous 11 blocks. TODO: has this changed? A *full node* **MUST NOT** accept *blocks* with headers more than two hours in the future according to its clock.
4	`nBits`	`uint32_t`	An encoded version of the target threshold this *block*'s header hash must be less than or equal to, in the same nBits format used by **Bitcoin**.
32	`nNonce`	`char[32]`	An arbitrary field miners change to modify the header hash in order to produce a hash below the target threshold.
1344	`nSolution`	`char[1344]`	The Equihash solution, which **MUST** be valid according to §7.2.1 *'Equihash'* on p. 25.

The changes relative to **Bitcoin** version 4 blocks as described in [11] are:

- The *block version number* **MUST** be 4. Previous versions are not supported. Software that parses blocks **MUST NOT** assume, when an encoded *block* starts with an `nVersion` field representing a value other than 4 (e.g. future versions potentially introduced by hard forks), that it will be parseable according to this format.
- The `hashReserved` and `nSolution` fields have been added.
- The type of the `nNonce` field has changed from `uint32_t` to `char[32]`.

7.2 Proof of Work

Zcash uses Equihash [9] as its Proof of Work. Motivations for changing the Proof of Work from *SHA-256d* used by **Bitcoin** are described in [26].

A *block* satisfies the Proof of Work if and only if:

- The `nSolution` field encodes a *valid Equihash solution* according to §7.2.1 *'Equihash'* on p. 25.
- The *block header* satisfies the difficulty check according to §7.2.2 *'Difficulty filter'* on p. 26.

7.2.1 Equihash

An instance of the Equihash algorithm is parameterized by positive integers n and k, such that n is a multiple of $k + 1$. We assume $k \geq 3$.

The Equihash parameters for the production network are $n = 200, k = 9$. TODO: These may not be final.

The Generalized Birthday Problem is defined as follows: given a sequence $X_{1..\mathrm{N}}$ of n-bit strings, find 2^k distinct X_{i_j} such that $\bigoplus_{j=1}^{2^k} X_{i_j} = 0$.

In Equihash, $\mathrm{N} = 2^{\frac{n}{k+1}+1}$, and the sequence $X_{1..\mathrm{N}}$ is derived from the *block header* and a nonce:

Let powtag := | 64-bit **"ZcashPoW"** | 32-bit n | 32-bit k |.

Let powinput(g) :=

32-bit `nVersion`	256-bit `hashPrevBlock`	256-bit `hashMerkleRoot`
256-bit `hashReserved`	32-bit `nTime`	32-bit `nBits`
256-bit `nNonce`	32-bit g	

Let $\ell := \frac{n}{k+1} + 1$.

Let $m :=$ floor$\left(\frac{512}{n}\right)$.

Let $T :=$ concat$_{\mathbb{B}}$([GeneralCRH$_{nm}$(powtag, powinput(g)) for g from 0 up to $\frac{N}{m} - 1$]).

For $h \in \{1 .. N\}$, let $X_h = T_{n(h-1)+1..nh}$.

(In other words, the bit sequence T is split into N subsequences of n bits. Indices of bits in T are 1-based.)

Define I2BSP $: (u : \mathbb{N}) \times \{0 .. 2^u - 1\} \to \mathbb{B}^u$ such that I2BSP$_u(x)$ is the sequence of u bits representing x in big-endian order.

Define BS2IP $: (u : \mathbb{N}) \times \mathbb{B}^u \to \{0 .. 2^u - 1\}$ such that BS2IP$_u$ is the inverse of I2BSP$_u$.

Define $\Xi_r(a, b) :=$ BS2IP$_{2^{r-1}\ell}$(concat$_{\mathbb{B}}(X_{i_{a..b}})$).

A *valid Equihash solution* is then a sequence $i : \{1 .. N\}^{2^k}$ that satisfies the following conditions:

Generalized Birthday condition $\bigoplus_{j=1}^{2^k} X_{i_j} = 0$.

Algorithm Binding conditions For all $r \in \{1 .. k-1\}$, for all $w \in \{0 .. 2^{k-r} - 1\}$:

- $\bigoplus_{j=1}^{2^r} X_{i_{w2^r+j}}$ has $\frac{nr}{k+1}$ leading zeroes; and
- $\Xi_r(w2^r + 1, w2^r + 2^{r-1}) < \Xi_r(w2^r + 2^{r-1} + 1, w2^r + 2^r)$.

Note: This does not include a difficulty condition, because here we are defining validity of an Equihash solution independent of difficulty.

An Equihash solution with $n = 200$ and $k = 9$ is encoded in the `nSolution` field of a *block header* as follows:

I2BSP$_{21}(i_1 - 1)$	I2BSP$_{21}(i_2 - 1)$	$\cdots$	I2BSP$_{21}(i_{512} - 1)$

Recall from § 5.1 ***'Integers, Bit Sequences, and Endianness'*** on p. 16 that bits in the above diagram are ordered

from most to least significant in each byte. For example, if the first 3 elements of i are $[69, 42, 2^{21}]$, then the corresponding bit array is:

$\mathsf{I2BSP}_{21}(68)$			$\mathsf{I2BSP}_{21}(41)$		$\mathsf{I2BSP}_{21}(2^{21}-1)$		
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0			0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1		1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1		
8-bit 0	8-bit 2	8-bit 32	8-bit 0	8-bit 10	8-bit 127	8-bit 255	$\cdots$

and so the first 7 bytes of `nSolution` would be $[0, 2, 32, 0, 10, 127, 255]$.

Note: I2BSP and BS2IP are big-endian, while the encoding of integer fields in powtag and powinput is little-endian. The rationale for this is that little-endian serialization of *block headers* is consistent with **Bitcoin**, but using little-endian ordering of bits in the solution encoding would require bit-reversal (as opposed to only shifting). The comparison of Ξ_r values obtained by a big-endian conversion is equivalent to lexicographic comparison as specified in section IV A. of [9].

7.2.2 Difficulty filter

The difficulty filter is unchanged from **Bitcoin**, and is calculated using *SHA-256d* on the whole *block header* (including `nSolution`).

7.2.3 Difficulty adjustment

Zcash uses a difficulty adjustment algorithm based on DigiShield v3/v4, with simplifications and altered parameters, to adjust difficulty to target the desired 2.5-minute block time. Unlike **Bitcoin**, the difficulty adjustment occurs after every block.

TODO: Describe the algorithm.

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^{old} protected *notes* as input, and produces up to N^{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. 26) 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^{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 §5.6 *'Note Plaintexts and Memo Fields'* on p. 19.

8.4 Faerie Gold attack and fix

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

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 [5]), 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 [17].

This attack does not violate the security definitions given in [5]. 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 ρ values for all *notes* they have ever received, and reject duplicates (as proposed in [15]). 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 ρ, by making use of the fact that all of

the *nullifiers* in *JoinSplit descriptions* that appear in a valid *block chain view* must be distinct. This is true regardless of whether the *nullifiers* corresponded to real or dummy notes. The *nullifiers* are used as input to BLAKE2b-256 to derive a public value h_{Sig} which uniquely identifies the transaction, as described in §4.4.1 ***'Computation of*** h_{Sig}***'*** on p. 12. (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 ρ value for each output *note* is then derived from a random private seed φ and h_{Sig} using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of ρ for each output *note* is enforced by the circuit (see §4.4.6 ***'Uniqueness of*** ρ_i^{new}***'*** on p. 15).

Now even if the creator of a *JoinSplit description* does not choose φ randomly, uniqueness of *nullifiers* and collision resistance of both BLAKE2b-256 and PRF^{ρ} will ensure that the derived ρ values are unique, at least for any two *JoinSplit descriptions* that get into a valid *block chain 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 COMM_r and COMM_s is a computationally binding commitment to its inputs a_{pk}, v, and ρ. However, the instantiation of COMM_r and 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 ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct values of ρ with colliding outputs of the truncated hash, and therefore the same *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. [16]

Zcash uses a simpler construction with a single SHA-256 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 [5]). 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: **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 [5], 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

...

The need for collision resistance of CRH truncated to 253 bits was not explicitly stated in the **Zerocash** paper; this does not follow from collision resistance of CRH.

8.7 In-band secret distribution

Zerocash specified ECIES (referencing Certicom's SEC 1 standard) as the encryption scheme used for the in-band secret distribution. This has been changed to a scheme based on Curve25519 key agreement, and the authenticated encryption algorithm AEAD_CHACHA20_POLY1305. This scheme is still loosely based on ECIES, and on the crypto_box_seal scheme defined in libsodium [19].

The motivations for this change were as follows:

- The **Zerocash** paper did not specify the curve to be used. We believe that Curve25519 has significant side-channel resistance, performance, implementation complexity, and robustness advantages over most other available curve choices, as explained in [7].

- ECIES permits many options, which were not specified. There are at least –counting conservatively– 576 possible combinations of options and algorithms over the four standards (ANSI X9.63, IEEE Std 1363a-2004, ISO/IEC 18033-2, and SEC 1) that define ECIES variants [20].
- Although the **Zerocash** paper states that ECIES satisfies key privacy (as defined in [3]), it is not clear that this holds for all curve parameters and key distributions. For example, if a group of non-prime order is used, the distribution of ciphertexts could be distinguishable depending on the order of the points representing the ephemeral and recipient public keys. Public key validity is also a concern. Curve25519 key agreement is defined in a way that avoids these concerns due to the curve structure and the "clamping" of private keys.
- Unlike the DHAES/DHIES proposal on which it is based [1], ECIES does not require a representation of the sender's ephemeral public key to be included in the input to the KDF, which may impair the security properties of the scheme. (The Std 1363a-2004 version of ECIES [25] has a "DHAES mode" that allows this, but the representation of the key input is underspecified, leading to incompatible implementations.) The scheme we use has both the ephemeral and recipient public key encodings –which are unambiguous for Curve25519– and also $\mathsf{h_{Sig}}$ and a nonce as described below, as input to the KDF. Note that because $\mathsf{pk_{enc}}$ is included in the KDF input, being able to break the Elliptic Curve Diffie-Hellman Problem on Curve25519 (without breaking ChaCha20 as an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the *transmitted notes ciphertext* unless $\mathsf{pk_{enc}}$ is known or guessed.
- The KDF also takes a public seed $\mathsf{h_{Sig}}$ as input. This can be modeled as using a different "randomness extractor" for each *JoinSplit operation*, which limits degradation of security with the number of *JoinSplit operations*. This facilitates security analysis as explained in [13] — see section 7 of that paper for a security proof that can be applied to this construction under the assumption that single-block BLAKE2b-256 is a "weak PRF". Note that $\mathsf{h_{Sig}}$ is authenticated, by the ZK proof, as having been chosen with knowledge of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..N^{\mathsf{old}}}$, so an adversary cannot modify it in a ciphertext from someone else's transaction for use in a chosen-ciphertext attack without detection.
- The scheme used by **Zcash** includes an optimization that uses the same ephemeral key (with different nonces) for the two ciphertexts encrypted in each *JoinSplit description*.

8.8 Omission in Zerocash security proof

The abstract **Zerocash** protocol requires $\mathsf{PRF}^{\mathsf{addr}}$ only to be a PRF; it is not specified to be collision-resistant. This reveals a flaw in the proof of the Balance property.

Suppose that an adversary finds a collision on $\mathsf{PRF}^{\mathsf{addr}}$ such that $\mathsf{a}^1_{\mathsf{sk}}$ and $\mathsf{a}^2_{\mathsf{sk}}$ are distinct *spending keys* for the same $\mathsf{a_{pk}}$. Because the *note commitment* is to $\mathsf{a_{pk}}$, but the *nullifier* is computed from $\mathsf{a_{sk}}$ (and ρ), the adversary is able to double-spend the note, once with each $\mathsf{a_{sk}}$. This is not detected because each spend reveals a different *nullifier*. The *JoinSplit statements* are still valid because they can only check that the $\mathsf{a_{sk}}$ in the witness is ***some*** preimage of the $\mathsf{a_{pk}}$ used in the *note commitment*.

The error is in the proof of Balance in section D.3 of [5]. For the "$\mathcal{A}$ violates Condition I" case, the proof says:

> "(i) If $\mathsf{cm}^{\mathsf{old}}_1 = \mathsf{cm}^{\mathsf{old}}_2$, then the fact that $\mathsf{sn}^{\mathsf{old}}_1 \neq \mathsf{sn}^{\mathsf{old}}_2$ implies that the witness a contains two distinct openings of $\mathsf{cm}^{\mathsf{old}}_1$ (the first opening contains $(\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}, \rho^{\mathsf{old}}_1)$, while the second opening contains $(\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}, \rho^{\mathsf{old}}_2)$). This violates the binding property of the commitment scheme COMM."

In fact the openings do not contain $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}$; they contain $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}$.

A similar error occurs in the argument for the "$\mathcal{A}$ violates Condition II" case.

The flaw is not exploitable for the actual instantiations of $\mathsf{PRF}^{\mathsf{addr}}$ in **Zerocash** and **Zcash**, which ***are*** collision-resistant assuming that `SHA256Compress` is.

The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of $\mathsf{PRF}^{\mathsf{addr}}$ (on both its arguments) to argue that distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}$, together with constraint 1(b) of the *JoinSplit statement* (see §4.4.6 ***'Spend authority'*** on p. 14), implies distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},2}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

8.9 Miscellaneous

- The paper defines a *note* as a tuple $(a_{pk}, v, \rho, r, s, cm)$, whereas this specification defines it as (a_{pk}, v, ρ, r). The instantiation of $COMM_s$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of `NoteCommitment` in **Zcash**. cm can be computed from the other fields.
- The length of proof encodings given in the paper is 288 bytes. This differs from the 296 bytes specified in §6.2 *'Encoding of Zero-Knowledge Proofs'* on p. 23, because the paper did not take into account the need to encode compressed y-coordinates. The fork of *libsnark* used by **Zcash** uses a different format to upstream *libsnark*, in order to follow [25].

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, Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, and no doubt others.

The Faerie Gold attack was found by Zooko Wilcox. The internal hash collision attack was found by Taylor Hornby. The omission in the **Zerocash** security proof relating to collision-resistance of PRF^{addr} was found by Daira Hopwood.

10 Change history

2016.0-beta-1

- Major reorganisation to separate the abstract cryptographic protocol from the algorithm instantiations.
- Add a section specifying the zero-knowledge proving system and the encoding of proofs. Change the encoding of points in proofs to follow IEEE Std 1363.
- Add a section on consensus changes from **Bitcoin**, and the specification of Equihash.
- Complete the "Differences from the **Zerocash** paper" section.
- Change the length of *memo fields* to 512 bytes.
- Switch the *JoinSplit signature* scheme to Ed25519, with consequent changes to the computation of h_{Sig}.
- Fix the lead bytes in *payment address* and *spending key* encodings to match the implemented protocol.
- Clarify cryptographic security requirements and added definitions relating to the in-band secret distribution.
- Add various citations: the "Fixing Vulnerabilities in the Zcash Protocol" and "Why Equihash?" blog posts, and several crypto papers for security definitions.
- Reference the extended version of the **Zerocash** paper rather than the Oakland proceedings version.
- Add *JoinSplit operations* to the Concepts section.
- Add a section on Coinbase Transactions.
- Add type declarations for functions.
- Add acknowledgements for Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, and Alex Balducci.
- Fix a `Makefile` compatibility problem with the escaping behaviour of `echo`.
- Make the date format in references more consistent.
- Change main font to Quattrocento.

2016.0-alpha-3

- Change version numbering convention (no other changes).

2.0-alpha-3

- Allow anchoring to any previous output *treestate* in the same *transaction*, rather than just the immediately preceding output *treestate*.
- Add change history.

2.0-alpha-2

- Change from truncated BLAKE2b-512 to BLAKE2b-256.
- Clarify endianness, and that uses of BLAKE2b are unkeyed.
- Minor correction to what *SIGHASH types* cover.
- Add "as intended for the **Zcash** release of summer 2016" to title page.
- Require $\mathsf{PRF}^{\mathsf{addr}}$ to be collision-resistant. [**?**]
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in § 4.4.6 *'**Merkle path validity**'* on p. 14 about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

2.0-alpha-1

- First version intended for public review.

11 References

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem. Cryptology ePrint Archive: Report 1999/007. `https://eprint.iacr.org/1999/007`. March 17, 1999.

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. `https://blake2.net/#sp`, January 29, 2013.

[3] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-Privacy in Public-Key Encryption. `https://cseweb.ucsd.edu/~mihir/papers/anonenc.html`. Full version, September 2001.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. Cryptology ePrint Archive: Report 2000/025. `https://eprint.iacr.org/2000/025`. Last revised July 14, 2007.

[5] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version). `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf`. Accessed: 2016-08-06. A condensed version appeared in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland) 2014*, pages 459–474; IEEE, 2014.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. Cryptology ePrint Archive: Report 2013/879. `https://eprint.iacr.org/2013/879`. Last revised May 19, 2015.

[7] 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. http://cr.yp.to/papers.html#curve25519. Date: 2006-02-09. Document ID: 4230efdfa673480fc079449d90f322c0.

[8] Daniel Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77–89, 2012. http://cr.yp.to/papers.html#ed25519. Date: 2011-09-26. Document ID: a1a62a2f76d23f65d622484ddd09caf8.

[9] Alex Biryukov and Dmitry Khovratovich. Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem. In *Proceedings of NDSS '16, 21–24 February 2016, San Diego, CA, USA. ISBN 1-891562-41-X*. Internet Society, 2016. DOI: 10.14722/ndss.2016.23108. https://www.internetsociety.org/sites/default/files/blogs-media/equihash-asymmetric-proof-of-work-based-generalized-birthday-problem.pdf.

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

[11] Block Headers – Bitcoin Developer Reference. https://bitcoin.org/en/developer-reference#block-headers. Accessed: 2016-08-08.

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

[13] Dana Dachman-Soled, Rosario Gennaro, Hugo Krawczyk, and Tal Malkin. Computational Extractors and Pseudorandomness. Cryptology ePrint Archive: Report 2011/708. https://eprint.iacr.org/2011/708. December 28, 2011.

[14] Edsger W. Dijkstra. Why numbering should start at zero. Manuscript, August 11, 1982. Transcribed at https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html. Accessed 2016-08-09.

[15] 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 January 24, 2016.

[16] Taylor Hornby and Zooko Wilcox. Fixing Vulnerabilities in the Zcash Protocol. Zcash blog. https://z.cash/blog/fixing-zcash-vulns.html, April 25, 2016. Accessed 2016-06-22.

[17] 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.

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

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

[20] V. Gayoso Martínez, F. Hernández Alvarez, L. Hernández Encinas, and C. Sánchez Ávila. A Comparison of the Standardized Versions of ECIES. In *Proceedings of Sixth International Conference on Information Assurance and Security, 23–25 August 2010, Atlanta, GA, USA. ISBN: 978-1-4244-7407-3*, pages 1–4. IEEE, 2010. DOI: 10.1109/ISIAS.2010.5604194. https://digital.csic.es/bitstream/10261/32674/1/Gayoso_A%20Comparison%20of%20the%20Standardized%20Versions%20of%20ECIES.pdf.

[21] 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.

[22] 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.

[23] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. Cryptology ePrint Archive: Report 2013/279. https://eprint.iacr.org/2013/279. Last revised May 13, 2013.

[24] IEEE Computer Society. *IEEE Std 1363-2000: Standard Specifications for Public-Key Cryptography*. IEEE, August 29, 2000. http://ieeexplore.ieee.org/servlet/opac?punumber=7168. Accessed: 2016-08-03. DOI: 10.1109/IEEESTD.2000.92292.

[25] IEEE Computer Society. *IEEE Std 1363a-2004: Standard Specifications for Public-Key Cryptography – Amendment 1: Additional Techniques*. IEEE, September 2, 2004. http://ieeexplore.ieee.org/servlet/opac?punumber=9276. Accessed: 2016-08-03. DOI: 10.1109/IEEESTD.2004.94612.

[26] Zooko Wilcox and Jack Grigg. Why Equihash? Zcash blog. https://z.cash/blog/why-equihash.html, April 15, 2016. Accessed 2016-08-05.