Zcash Protocol Specification

Version 2016.0-beta-1

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

Daira Hopwood

Sean Bowe — Taylor Hornby — Nathan Wilcox

September 4, 2016

Contents

1 Introduction

Zcash is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash** [BCG+2014], with some security fixes and adjustments to terminology, functionality and performance. It bridges the existing ***transparent*** payment scheme used by **Bitcoin** [Naka2008] with a ***protected*** payment scheme protected by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*).

Changes from the original **Zerocash** are explained in §7 ***'Differences from the Zerocash paper'*** on p. 30, and highlighted in magenta throughout the document.

Technical terms for concepts that play an important role in **Zcash** are written in *slanted text*. ***Italics*** are used for emphasis and for references between sections of the document.

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;
- Consensus Changes from **Bitcoin** — how **Zcash** differs from **Bitcoin** at the consensus layer, including the Proof of Work;
- Differences from the **Zerocash** protocol — a summary of changes from the protocol in [BCG+2014].

1.1 Caution

Zcash security depends on consensus. Should a program interacting with the **Zcash** network diverge from consensus, its security will be 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 that 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** 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.

1.2 High-level Overview

The following overview is intended to give a concise summary of the ideas behind the protocol, for an audience already familiar with *block chain*-based cryptocurrencies such as **Bitcoin**. It is imprecise in some aspects and is not part of the normative protocol specification.

Value in **Zcash** is either *transparent* or *protected*. Transfers of *transparent* value work essentially as in **Bitcoin** and have the same privacy properties. *Protected* value is carried by *notes*[1], which specify an amount and a *paying key*. The *paying key* is part of a *payment address*, which is a destination to which *notes* can be sent. As in **Bitcoin**, this is associated with a private key that can be used to spend *notes* sent to the address; in **Zcash** this is called a *spending key*.

To each *note* there is cryptographically associated a *note commitment*, and a *nullifier*[1] (so that there is a 1:1:1 relation between *notes*, *note commitments*, and *nullifiers*). However, it is infeasible to correlate a commitment with its *nullifier* without knowledge of the *note*. Computing the *nullifier* requires the associated private *spending*

[1] In **Zerocash** [BCG+2014], *notes* were called "coins", and *nullifiers* were called "serial numbers".

key. An unspent valid *note*, at a given point on the *block chain*, is one for which the *note commitment* has been publically revealed on the *block chain* prior to that point, but the *nullifier* has not.

A *transaction* can contain *transparent* inputs, outputs, and scripts, which all work as in **Bitcoin**. They also contain a sequence of zero or more *JoinSplit descriptions*. Each of these describes a *JoinSplit transfer*[2] which takes in a *transparent* value and up to two input *notes*, and produces a *transparent* value and up to two output *notes*. The *nullifiers* of the input *notes* are revealed (preventing them from being spent again) and the commitments of the output *notes* are revealed (allowing them to be spent in future). Each *JoinSplit description* also includes a computationally sound *zk-SNARK* proof, which proves all of the following:

- The inputs and outputs balance (individually for each *JoinSplit transfer*).
- For each input *note* of non-zero value, some revealed *note commitment* exists for that *note*.
- The prover knew the private *spending keys* of the input *notes*.
- The *nullifiers* and *note commitments* are computed correctly.
- The private *spending keys* of the input *notes* are cryptographically linked to a signature over the whole *transaction*, in such a way that the *transaction* cannot be modified by a party who did not know these private keys.
- Each output *note* is generated in such a way that its *nullifier* will not collide with the *nullifier* of any other *note*.

Outside the *zk-SNARK*, it is also checked that the *nullifiers* for the input *notes* had not already been revealed (i.e. they had not already been spent).

A *payment address* includes two public keys: a *paying key* matching that of *notes* sent to the address, and a *transmission key* for a key-private asymmetric encryption scheme. "Key-private" means that ciphertexts do not reveal information about which key they were encrypted to, except to a holder of the corresponding private key, which in this context is called the *viewing key*. This facility is used to communicate encrypted output *notes* on the *block chain* to their intended recipient, who can use the *viewing key* to scan the *block chain* for *notes* addressed to them and then decrypt those *notes*.

The basis of the privacy properties of **Zcash** is that when a *note* is spent, the spender only proves that some commitment for it had been revealed, without revealing which one. This implies that a spent *note* cannot be linked to the *transaction* in which it was created. That is, from an adversary's point of view the set of possibilities for a given *note* input to a *transaction*—its *note traceability set*— includes *all* previous notes that the adversary does not control or know to have been spent. This contrasts with other proposals for private payment systems, such as CoinJoin [Bitcoin-CoinJoin] or CryptoNote [vanS2014], that are based on mixing of a limited number of transactions and that therefore have smaller *note traceability sets*.

The *nullifiers* are necessary to prevent double-spending: each note only has one valid *nullifier*, and so attempting to spend a *note* twice would reveal the *nullifier* twice, which would cause the second *transaction* to be rejected.

2 Notation

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

The notation $\mathbb{B}$ means the set of bit values, i.e. $\{0, 1\}$. $\mathbb{B}^{\ell}$ means the set of sequences of ℓ bits. $\mathbb{B}^{8\star}$ means the set of bit sequences constrained to be of length a multiple of 8 bits.

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 [BCG+2014] and in

[2] *JoinSplit transfers* in **Zcash** generalize "Mint" and "Pour" *transactions* in **Zerocash**; see §7.1 *'Transaction Structure'* on p. 30 for the differences.

[BK2016], this specification uses 1-based indexing and inclusive ranges, notwithstanding the compelling arguments to the contrary made in [EWD-831].)

The notation $\{a \mathinner{..} b\}$ means the set of integers from a through b inclusive.

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 inclusive, 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 inclusive, in descending order.

The notation $a \,||\, b$ means the concatenation of sequences a then b.

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. $\mathbb{N}^+$ means the set of positive integers.

The notation $\mathbb{F}_n$ means the finite field with n elements, and $\mathbb{F}^*_n$ means its group under multiplication. $\mathbb{F}_n[z]$ means the ring of polynomials over z with coefficients in $\mathbb{F}_n$.

The notation $a \bmod q$, for integers $a \geq 0$ and $q > 0$, 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 $\sum\limits_{i=1}^{\mathrm{N}} a_i$ means the sum of $a_{1..\mathrm{N}}$. $\bigoplus\limits_{i=1}^{\mathrm{N}} a_i$ 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 $\bot$ is used to indicate unavailable information or a failed decryption.

The notation $T \subseteq U$ indicates that T is an inclusive subset or subtype of U.

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.

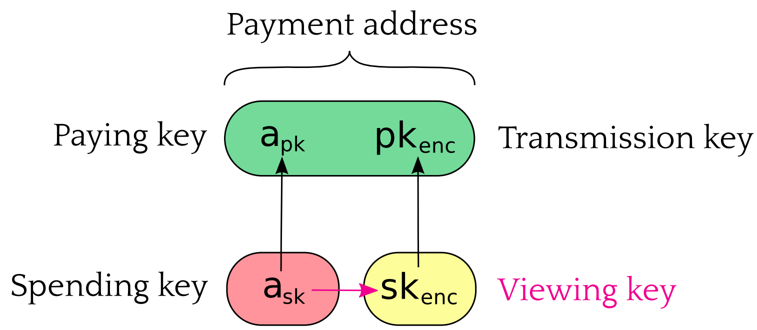
The following integer constants will be instantiated in §5.3 ***'Constants'*** on p. 19: d, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, ℓ_{Merkle}, ℓ_{hSig}, ℓ_{PRF}, ℓ_{r}, ℓ_{Seed}, $\ell_{\mathsf{a_{sk}}}$, ℓ_{φ}, $\mathsf{MAX\_MONEY}$. The bit sequence constant $\mathsf{Uncommitted} \,⦂\, \mathbb{B}^{\ell_{\mathsf{Merkle}}}$ will also be defined in that section.

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* $\mathsf{addr_{pk}}$ 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 ($\mathsf{pk_{enc}}$) 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 §4.10 ***'In-band secret distribution'*** on p. 17), since an adversary would have to know $\mathsf{pk_{enc}}$ in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

A *note* (denoted $\mathbf{n}$) is a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$. It represents that a value v is spendable by the recipient who holds the *spending key* $\mathsf{a_{sk}}$ corresponding to $\mathsf{a_{pk}}$, as described in the previous section.

- $\mathsf{a_{pk}} \mathbin{⦂} \mathbb{B}^{\ell_{\mathsf{PRF}}}$ is the *paying key* of the recipient;
- $\mathsf{v} \mathbin{⦂} \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*);
- $\rho \mathbin{⦂} \mathbb{B}^{\ell_{\mathsf{PRF}}}$ is used as input to $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}$ to derive the *nullifier* of the *note*;
- $\mathsf{r} \mathbin{⦂} \mathbb{B}^{\ell_{\mathsf{r}}}$ is a random bit sequence used as a *commitment trapdoor* as defined in §4.1.7 ***'Commitment'*** on p. 12.

Let Note be the type of a *note*, i.e. $\mathbb{B}^{\ell_{\mathsf{PRF}}} \times \{0 \,..\, \mathsf{MAX\_MONEY}\} \times \mathbb{B}^{\ell_{\mathsf{PRF}}} \times \mathbb{B}^{\ell_{\mathsf{r}}}$.

Creation of new *notes* is described in §4.4 ***'Sending Notes'*** on p. 14. When *notes* are sent, only a commitment (see §4.1.7 ***'Commitment'*** on p. 12) to the above values is disclosed publically. This allows the value and recipient to be kept private, while the commitment is used by the *zero-knowledge proof* when the *note* is spent, to check that it exists on the *block chain*.

The *note commitment* is computed as $\mathtt{NoteCommitment}(\mathbf{n}) = \mathsf{COMM_r}(\mathsf{a_{pk}}, \mathsf{v}, \rho)$, where COMM is instantiated in §5.4.9 ***'Commitment'*** on p. 22.

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

3.2.1 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* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, and the result forms part of a *transmitted notes ciphertext* (see §4.10 *'In-band secret distribution'* on p. 17 for further details).

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

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. 9), *nullifier set* (§ 3.6 *'Nullifier Set'* on p. 9), and data structures associated with **Bitcoin** such as the UTXO (Unspent Transaction Output) set.

Inputs to a *transaction* insert value into a *transparent 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*.

TODO: *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 Transfers and Descriptions

A *JoinSplit description* is data included in a *transaction* that describes a *JoinSplit transfer*, 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 statement* used for the *zk-SNARK* proof and verification.

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

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

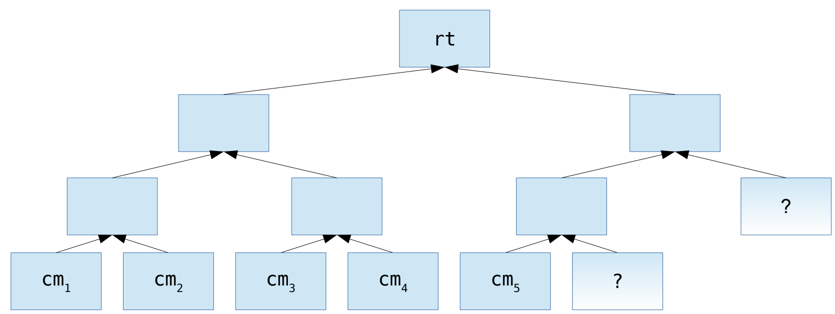
The inputs and outputs of each *JoinSplit transfer* **MUST** balance exactly. The total $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ value adds to, and the total $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ value subtracts from the *transparent value pool* of the containing *transaction*.

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

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 transfers* produce. Just as the *unspent transaction output set* (UTXO set) used in **Bitcoin**, it is used to express the existence of value and the capability to spend it. However, unlike the UTXO set, 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*. TODO: 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 transfers* 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 §4.5 *'Merkle path validity'* on p. 15. It is instantiated in §5.4.1 *'Merkle Tree Hash Function'* on p. 19.

$\mathsf{hSigCRH} : \mathbb{B}^{\ell_{\mathsf{Seed}}} \times (\mathbb{B}^{\ell_{\mathsf{PRF}}})^{\mathrm{N}^{\mathsf{old}}} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{\ell_{\mathsf{hSig}}}$ is a collision-resistant hash function used in § 4.3 *'JoinSplit Descriptions'* on p. 13. It is instantiated in §5.4.2 *'$\mathsf{h_{Sig}}$ Hash Function'* on p. 19.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{B}^{8\star} \times \mathbb{N}^+ \to \mathbb{B}^n$ is another hash function, used in § 6.4.1 *'Equihash'* on p. 29 to generate input to the Equihash solver. The first two arguments, representing the Equihash parameters n and k, are written subscripted. It is instantiated in §5.4.3 *'Equihash Generator'* on p. 20.

4.1.2 Pseudo Random Functions

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

$$
\begin{array}{llll}
\mathsf{PRF}^{\mathsf{addr}} & : \mathbb{B}^{\ell_{\mathsf{a_{sk}}}} & \times \{0 \,..\, 255\} & \to \mathbb{B}^{\ell_{\mathsf{PRF}}} \\
\mathsf{PRF}^{\mathsf{nf}} & : \mathbb{B}^{\ell_{\mathsf{a_{sk}}}} & \times \mathbb{B}^{\ell_{\mathsf{PRF}}} & \to \mathbb{B}^{\ell_{\mathsf{PRF}}} \\
\mathsf{PRF}^{\mathsf{pk}} & : \mathbb{B}^{\ell_{\mathsf{a_{sk}}}} & \times \{1..\mathrm{N}^{\mathsf{old}}\} \times \mathbb{B}^{\ell_{\mathsf{hSig}}} & \to \mathbb{B}^{\ell_{\mathsf{PRF}}} \\
\mathsf{PRF}^{\rho} & : \mathbb{B}^{\ell_{\varphi}} & \times \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{\ell_{\mathsf{hSig}}} & \to \mathbb{B}^{\ell_{\mathsf{PRF}}}
\end{array}
$$

These are used in § 4.9 *'JoinSplit Statement'* on p. 16; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *payment address* from a *spending key* in § 4.2 *'Key Components'* on p. 13. They are instantiated in § 5.4.4 *'Pseudo Random Functions'* on p. 20.

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 $\mathsf{Sym}.\mathbf{C}$.

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

$\mathsf{Sym.Decrypt} : \mathsf{Sym}.\mathbf{K} \times \mathsf{Sym}.\mathbf{C} \to \mathsf{Sym}.\mathbf{P} \cup \{\bot\}$ is the corresponding decryption algorithm, such that for any $\mathsf{K} \in \mathsf{Sym}.\mathbf{K}$ and $\mathsf{P} \in \mathsf{Sym}.\mathbf{P}$, $\mathsf{Sym.Decrypt}_{\mathsf{K}}(\mathsf{Sym.Encrypt}_{\mathsf{K}}(\mathsf{P})) = \mathsf{P}$. $\bot$ 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 [BN2007].

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 private key.

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

Note: The range of KA.`DerivePublic` may be a strict subset of KA.Public.

Security requirements:

- KA.`FormatPrivate` must preserve sufficient entropy from its input to be used as a secure KA private key.
- The key agreement and the KDF defined in the next section must together satisfy a suitable adaptive security assumption along the lines of [Bern2006, section 3] or [ABR1999, Definition 3].

More precise formalization of these requirements is beyond the scope of this specification.

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 $:$ $\{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{\ell_{\mathsf{hSig}}} \times$ 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: In addition to adaptive security of the key agreement and KDF, the following security property is required:

Let $\mathsf{sk}_{\mathsf{enc}}^1$ and $\mathsf{sk}_{\mathsf{enc}}^2$ each be chosen uniformly and independently at random from KA.Private.

Let $\mathsf{pk}_{\mathsf{enc}}^j$:= KA.`DerivePublic`($\mathsf{sk}_{\mathsf{enc}}^j$).

An adversary can adaptively query a function $Q : \{1 \mathinner{..} 2\} \times \mathbb{B}^{\ell_{\mathsf{hSig}}} \to$ KA.Public $\times$ Sym.$\mathbf{K}_{1..\mathrm{N}^{\mathsf{new}}}$ where $Q_j(\mathsf{h_{Sig}})$ is defined as follows:

1. Choose esk uniformly at random from KA.Private.
2. Let epk := KA.`DerivePublic`(esk).
3. For $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$, let K_i := KDF(i, $\mathsf{h_{Sig}}$, KA.`Agree`(esk, $\mathsf{pk}_{\mathsf{enc}}^j$), epk, $\mathsf{pk}_{\mathsf{enc}}^j$)).
4. Return (epk, $\mathsf{K}_{1..\mathrm{N}^{\mathsf{new}}}$).

Then the adversary must make another query to Q_j with random unknown $j \in \{1 \mathinner{..} 2\}$, and guess j with probability greater than chance.

If the adversary's advantage is negligible, then the asymmetric encryption scheme constructed from KA, KDF and Sym in §4.10 *'In-band secret distribution'* on p. 17 will be key-private as defined in [BBDP2001].

Note: The given definition only requires ciphertexts to be indistinguishable between *transmission keys* that are outputs of KA.$\mathtt{DerivePublic}$ (which includes all keys generated as in §4.2 *'Key Components'* on p. 13). If a *transmission key* not in that range is used, it may be distinguishable. This is not considered to be a significant security weakness.

4.1.6 Signatures

TODO: Define JoinSplitSigAlg.

4.1.7 Commitment

A *commitment scheme* is a function that, given a random *commitment trapdoor* and an input, can be used to commit to the input in such a way that:

- no information is revealed about it without the *trapdoor* ("hiding"),
- given the *trapdoor* and input, the commitment can be verified to "open" to that input and no other ("binding").

A *commitment scheme* COMM defines a type of inputs COMM.Input, a type of commitments COMM.Output, and a type of *commitment trapdoors* COMM.Trapdoor.

Let COMM $:$ COMM.Trapdoor $\times$ COMM.Input $\to$ COMM.Output be a function satisfying the security requirements of computational hiding and computational binding, as defined in TODO: need reference.

4.1.8 Zero-Knowledge Proving System

A *zero-knowledge proving system* is a cryptographic protocol that allows proving a particular *statement*, dependent on *primary* and *auxiliary inputs*, in zero knowledge — that is, without revealing information about the *auxiliary inputs* other than that implied by the *statement*. The type of *zero-knowledge proving system* needed by **Zcash** is a *preprocessing zk-SNARK*.

A *preprocessing zk-SNARK* instance ZK defines:

- a type of *zero-knowledge proving keys*, ZK.ProvingKey;
- a type of *zero-knowledge verifying keys*, ZK.VerifyingKey;
- a probability distribution over ZK.ProvingKey $\times$ ZK.VerifyingKey of parameters, ZK.ParameterDistribution;
- a type of *primary inputs* ZK.PrimaryInput;
- a type of *auxiliary inputs* ZK.AuxiliaryInput;
- a type ZK.SatisfyingInputs $\subseteq$ ZK.PrimaryInput $\times$ ZK.AuxiliaryInput of inputs satisfying the *statement*;
- a function ZK.$\mathtt{Prove}$ $:$ ZK.ProvingKey $\times$ ZK.SatisfyingInputs $\to$ ZK.Proof;
- a function ZK.$\mathtt{Verify}$ $:$ ZK.VerifyingKey $\times$ ZK.PrimaryInput $\times$ ZK.Proof $\to \mathbb{B}$;

The security requirements below are supposed to hold with overwhelming probability for $(\mathsf{pk}, \mathsf{vk})$ sampled at random from ZK.ParameterDistribution.

Security requirements:

- **Completeness:** An honestly generated proof will convince a verifier: for any $(x, w) \in$ ZK.SatisfyingInputs, if ZK.$\mathtt{Prove}_{\mathsf{pk}}(x, w)$ outputs π, then ZK.$\mathtt{Verify}_{\mathsf{vk}}(x, \pi) = 1$.
- **Proof of Knowledge:** For any adversary $\mathcal{A}$ able to find an $x :$ ZK.PrimaryInput and proof $\pi :$ ZK.Proof such that ZK.$\mathtt{Verify}_{\mathsf{vk}}(x, \pi) = 1$, there is an efficient extractor $E_{\mathcal{A}}$ such that if $E_{\mathcal{A}}(\mathsf{vk}, \mathsf{pk})$ returns w, then the probability that $(x, w) \notin$ ZK.SatisfyingInputs is negligable.

- **Statistical Zero Knowledge:** An honestly generated proof is statistical zero knowledge. TODO: Full definition.

These definitions are derived from those in [BCTV2014, Appendix C], adapted to state concrete rather than asymptotic security. (ZK.$\mathtt{Prove}$ corresponds to P, ZK.$\mathtt{Verify}$ corresponds to V, and ZK.SatisfyingInputs corresponds to $\mathcal{R}_C$ in the notation of that appendix.)

The Proof of Knowledge definition is a way to formalize the property that it is infeasible to find a new proof π where $\mathsf{ZK.Verify}_{\mathsf{vk}}(x, \pi) = 1$ without ***knowing*** an *auxiliary input* w such that $(x, w) \in$ ZK.SatisfyingInputs. (It is possible to replay proofs, but informally, a proof for a given (x, w) gives no information that helps to find a proof for other (x, w).)

The *proving system* is instantiated in §5.7 ***'Zero-Knowledge Proving System'*** on p. 24. $\mathsf{ZK}_{\mathtt{JoinSplit}}$ refers to this *proving system* specialized to the *JoinSplit statement* given in §4.9 ***'JoinSplit Statement'*** on p. 16. In this case we omit the key subscripts on $\mathsf{ZK}_{\mathtt{JoinSplit}}$.Verify and $\mathsf{ZK}_{\mathtt{JoinSplit}}$.Prove, taking them to be the particular *proving key* and *verifying key* defined by the *JoinSplit parameters* in §5.8 ***'JoinSplit Parameters'*** on p. 25.

4.2 Key Components

Let KA be a *key agreement scheme*, instantiated in §5.4.6 ***'Key Agreement'*** on p. 21.

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

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

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

4.3 JoinSplit Descriptions

A *JoinSplit transfer*, as specified in §3.4 ***'JoinSplit Transfers and Descriptions'*** on p. 8, is encoded in *transactions* as a *JoinSplit description*.

Each *transaction* includes a sequence of zero or more *JoinSplit descriptions*. When this sequence is non-empty, the *transaction* also includes encodings of a JoinSplitSigAlg public verification key and signature.

Each *JoinSplit description* consists of $(\mathsf{v^{old}_{pub}}, \mathsf{v^{new}_{pub}}, \mathsf{rt}, \mathsf{nf^{old}_{1..N^{old}}}, \mathsf{cm^{new}_{1..N^{new}}}, \mathsf{epk}, \mathsf{randomSeed}, \mathsf{h_{1..N^{old}}}, \pi_{\mathtt{JoinSplit}}, \mathsf{C^{enc}_{1..N^{new}}})$ where

- $\mathsf{v^{old}_{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent value pool*;
- $\mathsf{v^{new}_{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent value pool*;
- $\mathsf{rt} ⦂ \mathbb{B}^{\ell_{\mathsf{Merkle}}}$ is an *anchor*, as defined in §3.3 ***'Transactions, Blocks, and the Block Chain'*** on p. 8, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf^{old}_{1..N^{old}}} ⦂ (\mathbb{B}^{\ell_{\mathsf{PRF}}})^{\mathrm{N^{old}}}$ is the sequence of *nullifiers* for the input *notes*;
- $\mathsf{cm^{new}_{1..N^{new}}} ⦂ (\mathsf{COMM.Output})^{\mathrm{N^{new}}}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} ⦂ \mathsf{KA.Public}$ is a key agreement public key, used to derive the key for encryption of the *transmitted notes ciphertext* (§4.10 ***'In-band secret distribution'*** on p. 17);
- $\mathsf{randomSeed} ⦂ \mathbb{B}^{\ell_{\mathsf{Seed}}}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h_{1..N^{old}}} ⦂ (\mathbb{B}^{\ell_{\mathsf{PRF}}})^{\mathrm{N^{old}}}$ is a sequence of tags that bind $\mathsf{h_{Sig}}$ to each $\mathsf{a_{sk}}$ of the input *notes*;
- $\pi_{\mathtt{JoinSplit}} ⦂ \mathsf{ZK.Proof}$ is the *zero-knowledge proof* for the *JoinSplit statement*;

- $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}} \;⦂\; (\mathsf{Sym}.\mathbf{C})^{\mathrm{N}^{\mathsf{new}}}$ is a sequence of ciphertext components for the encrypted output *notes*.

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

The value $\mathsf{h_{Sig}}$ is also computed from $\mathsf{randomSeed}$, $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$, and the `joinSplitPubKey` of the containing *transaction*:

$$\mathsf{h_{Sig}} := \mathsf{hSigCRH}(\mathsf{randomSeed}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \texttt{joinSplitPubKey}).$$

$\mathsf{hSigCRH}$ is instantiated in §5.4.2 '$\mathsf{h_{Sig}}$ ***Hash Function***' on p. 19.

Consensus rules:

- Elements of a *JoinSplit description* **MUST** have the types given above (for example: $0 \leq \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} \leq \mathsf{MAX\_MONEY}$ and $0 \leq \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} \leq \mathsf{MAX\_MONEY}$).
- Either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ **MUST** be zero.
- The proof $\pi_{\texttt{JoinSplit}}$ **MUST** be valid given a *primary input* formed from the other fields and $\mathsf{h_{Sig}}$. I.e. it must be the case that $\mathsf{ZK}_{\texttt{JoinSplit}}.\mathsf{Verify}((\mathsf{rt}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h_{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}), \pi_{\texttt{JoinSplit}}) = 1$.

4.4 Sending Notes

In order to send *protected* value, the sender constructs a *transaction* containing one or more *JoinSplit descriptions*. This involves first generating a new $\mathsf{JoinSplitSigAlg}$ key pair, which includes `joinSplitPubKey`.

For each *JoinSplit description*, the sender chooses $\mathsf{randomSeed}$ uniformly at random on $\mathbb{B}^{\ell_{\mathsf{Seed}}}$, and selects the input *notes*. At this point there is sufficient information to compute $\mathsf{h_{Sig}}$, as described in the previous section. The sender also chooses φ uniformly at random on $\mathbb{B}^{\ell_\varphi}$. Then it creates each output *note* with index $i \;⦂\; \{1..\mathrm{N}^{\mathsf{new}}\}$ as follows:

- Choose $\mathsf{r}^{\mathsf{new}}_i$ uniformly at random on $\mathbb{B}^{\ell_{\mathsf{r}}}$.
- Compute $\rho^{\mathsf{new}}_i := \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}})$.
- Encrypt the *note* to the recipient *transmission key* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}$, as described in §4.10 ***'In-band secret distribution'*** on p. 17, giving the ciphertext component $\mathsf{C}^{\mathsf{enc}}_i$.

In order to minimize information leakage, the sender **SHOULD** randomize the order of the input *notes* and of the output *notes*. Other considerations relating to information leakage from the structure of *transactions* are beyond the scope of this specification.

After generating all of the *JoinSplit descriptions*, the sender constructs the encoded *transaction* as described in §6.2 ***'Encoding of JoinSplit Descriptions'*** on p. 27, signed with the private *JoinSplit signing key*, and submits it to the network.

4.4.1 Dummy Notes

The fields in a *JoinSplit description* allow for $\mathrm{N}^{\mathsf{old}}$ input *notes*, and $\mathrm{N}^{\mathsf{new}}$ output *notes*. In practice, we may wish to encode a *JoinSplit transfer* with fewer input or output *notes*. This is achieved using *dummy notes*.

A *dummy* input *note*, with index i in the *JoinSplit description*, is constructed as follows:

- Generate a new random *spending key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}$ and derive its *paying key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}$.
- Set $\mathsf{v}^{\mathsf{old}}_i := 0$.
- Choose ρ^{old}_i uniformly at random on $\mathbb{B}^{\ell_{\mathsf{PRF}}}$.
- Choose $\mathsf{r}^{\mathsf{old}}_i$ uniformly at random on $\mathbb{B}^{\ell_{\mathsf{r}}}$.

- Compute $\mathsf{nf}_i^{\mathsf{old}} := \mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(\rho_i^{\mathsf{old}})$.
- Construct a *dummy path* path_i for use in the *auxiliary input* to the *JoinSplit statement* (this will not be checked).
- When generating the *JoinSplit proof*, set $\mathsf{enforce}_i$ to 0.

A *dummy* output *note* is constructed as normal but with zero value, and sent to a random *payment address*.

4.5 Merkle path validity

The depth of the *note commitment tree* is d (defined in §5.3 *'Constants'* on p. 19).

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 $0 \le h < \mathsf{d}$ and $0 \le i < 2^h$,

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

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

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

where

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

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* $\mathsf{rt} = \mathsf{M}_0^0$.

4.6 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 $\mathsf{dataToBeSigned}$ 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 $\mathsf{JoinSplitSigAlg}$ key pair is generated for each *transaction*, and the $\mathsf{dataToBeSigned}$ is signed with the private signing key of this key pair. The corresponding public verification key is included in the *transaction* encoding as `joinSplitPubKey`.

$\mathsf{JoinSplitSigAlg}$ is instantiated in §5.4.8 *'Signatures'* on p. 21.

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 $\mathsf{dataToBeSigned}$ as specified above, using `joinSplitPubKey`.

The condition enforced by the *JoinSplit statement* specified in § 4.9 *'Non-malleability'* on p. 17 ensures that a holder of all of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$ for each *JoinSplit description* has authorized the use of the private signing key corresponding to `joinSplitPubKey` to sign this *transaction*.

4.7 Balance

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

Note: Unlike original **Zerocash** [BCG+2014], **Zcash** does not have a distinction between Mint and Pour operations. The addition of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ to a *JoinSplit description* subsumes the functionality of both Mint and Pour. Also, *JoinSplit descriptions* are indistinguishable regardless of the number of real input *notes*.

As stated in §4.3 *'JoinSplit Descriptions'* on p. 13, either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ **MUST** be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

4.8 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.

4.9 JoinSplit Statement

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

$$(\mathsf{rt} \mathrel{⦂} \mathbb{B}^{\ell_{\mathsf{Merkle}}}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} (\mathbb{B}^{\ell_{\mathsf{PRF}}})^{\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} \mathrel{⦂} (\mathsf{COMM.Output})^{\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} \mathrel{⦂} \{0\,..\,2^{64}-1\}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} \mathrel{⦂} \{0\,..\,2^{64}-1\},$$
$$\mathsf{h}_{\mathsf{Sig}} \mathrel{⦂} \mathbb{B}^{\ell_{\mathsf{hSig}}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} (\mathbb{B}^{\ell_{\mathsf{PRF}}})^{\mathrm{N}^{\mathsf{old}}}),$$

the prover knows an *auxiliary input*:

$$(\mathsf{path}_{1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} ((\mathbb{B}^{\ell_{\mathsf{Merkle}}})^{\mathsf{d}})^{\mathrm{N}^{\mathsf{old}}}, \mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} \mathsf{Note}^{\mathrm{N}^{\mathsf{old}}}, \mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} (\mathbb{B}^{\ell_{\mathsf{a_{sk}}}})^{\mathrm{N}^{\mathsf{old}}}, \mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} \mathrel{⦂} \mathsf{Note}^{\mathrm{N}^{\mathsf{old}}},$$
$$\varphi \mathrel{⦂} \mathbb{B}^{\ell_{\varphi}}, \mathsf{enforce}_{1..\mathrm{N}^{\mathsf{old}}} \mathrel{⦂} \mathbb{B}^{\mathrm{N}^{\mathsf{old}}}),$$

where:

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

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforce}_i = 1$: path_i must be a valid *path* of depth d, as defined in §4.5 *'Merkle path validity'* on p. 15, from $\texttt{NoteCommitment}(\mathbf{n}^{\mathsf{old}}_i)$ to *note commitment tree* root rt.

Note: Merkle path validity covers both conditions 1. (a) and 1. (d) of the NP statement given in [BCG+2014, section 4.2].

Commitment Enforcement for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$, if $\mathsf{v}^{\mathsf{old}}_i \neq 0$ then $\mathsf{enforce}_i = 1$.

Balance $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} + \sum_{i=1}^{\mathrm{N}^{\mathsf{old}}} \mathsf{v}^{\mathsf{old}}_i = \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} + \sum_{i=1}^{\mathrm{N}^{\mathsf{new}}} \mathsf{v}^{\mathsf{new}}_i \in \{0 \,..\, 2^{64} - 1\}$.

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

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

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

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

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

For details of the form and encoding of proofs, see §5.7 *'Zero-Knowledge Proving System'* on p. 24.

4.10 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* $\mathsf{pk}_{\mathsf{enc}}$ is used to encrypt these secrets. The recipient's possession of the associated *key tuple* $(\mathsf{a}_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}}, \mathsf{addr}_{\mathsf{pk}})$ is used to reconstruct the original *note* and *memo field*.

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

For both encryption and decryption,

- Let Sym be the *encryption scheme* instantiated in §5.4.5 *'Authenticated One-Time Symmetric Encryption'* on p. 21.
- Let KDF be the *Key Derivation Function* instantiated in §5.4.7 *'Key Derivation'* on p. 21.
- Let KA be the *key agreement scheme* instantiated in §5.4.6 *'Key Agreement'* on p. 21.
- Let $\mathsf{h}_{\mathsf{Sig}}$ be the value computed for this *JoinSplit description* in §4.3 *'JoinSplit Descriptions'* on p. 13.

4.10.1 Encryption

Let $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$ be the *transmission keys* for the intended recipient addresses of each new *note*.

Let $\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *note plaintexts* as defined in §5.5 *'Note Plaintexts and Memo Fields'* on p. 22.

Then to encrypt:

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

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

4.10.2 Decryption by a Recipient

Let $\mathsf{addr_{pk}} = (\mathsf{a_{pk}}, \mathsf{pk_{enc}})$ be the recipient's *payment address*, and let $\mathsf{sk_{enc}}$ be the recipient's *viewing key*.

Let $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *note commitments* of each output coin.

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

- Let $\mathsf{sharedSecret}_i := \mathsf{KA.Agree}(\mathsf{sk_{enc}}, \mathsf{epk})$.
- Let $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
- Return $\mathtt{DecryptNote}(\mathsf{K}^{\mathsf{enc}}_i, \mathsf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i, \mathsf{a_{pk}})$.

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

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

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

Notes:

- The decryption algorithm corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in [BCG+2014, Figure 2].
- 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.

See § 7.7 *'In-band secret distribution'* on p. 33 for further discussion of the security and engineering rationale behind this encryption scheme.

5 Concrete Protocol

5.1 Caution

TODO: Explain the kind of things that can go wrong with linkage between abstract and concrete protocol. E.g. § 7.5 *'Internal hash collision attack and fix'* on p. 32

5.2 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. Diagrams are read from left-to-right, with lines read from top-to-bottom; the breaking of boxes across lines has no significance. 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.

5.3 Constants

Define:

$\mathsf{d} = 29$

$\mathrm{N}^{\mathsf{old}} = 2$

$\mathrm{N}^{\mathsf{new}} = 2$

$\ell_{\mathsf{Merkle}} = 256$

$\ell_{\mathsf{hSig}} = 256$

$\ell_{\mathsf{PRF}} = 256$

$\ell_{\mathsf{r}} = 256$

$\ell_{\mathsf{Seed}} = 256$

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

$\ell_{\varphi} = 252$

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

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

5.4 Concrete Cryptographic Functions

5.4.1 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. [NIST2015]

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

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 $\mathtt{SHA256Compress}(x) = [0]^{256}$.

5.4.2 $\mathsf{h_{Sig}}$ Hash Function

hSigCRH is used to compute the value $\mathsf{h_{Sig}}$ in §4.3 *'JoinSplit Descriptions'* on p. 13.

$$\mathsf{hSigCRH}(\mathsf{randomSeed}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathtt{joinSplitPubKey}) := \mathsf{GeneralCRH}_{256}(\textbf{“}\mathtt{ZcashComputehSig}\textbf{”}, \mathsf{hSigInput})$$

where

$$\mathsf{hSigInput} := \boxed{\text{256-bit randomSeed}}\boxed{\text{256-bit } \mathsf{nf}^{\mathsf{old}}_{1}}\ \ldots\boxed{\text{256-bit } \mathsf{nf}^{\mathsf{old}}_{\mathrm{N}^{\mathsf{old}}}}\boxed{\text{256-bit } \mathtt{joinSplitPubKey}}.$$

$\mathsf{GeneralCRH}_{\ell}(p, x)$ is instantiated by unkeyed BLAKE2b-ℓ [ANWW2013][RFC-7693] in sequential mode, with an output digest length of $\ell/8$ bytes, 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-256(**"ZcashComputehSig"**, x) must be collision-resistant.

5.4.3 Equihash Generator

$\mathsf{EquihashGen}_{n,k}$ is a specialized hash function that maps an input and an index to an output of length n bits. It is used in §6.4.1 ***'Equihash'*** on p. 29.

Let $\mathsf{powtag} :=$ | 64-bit **"ZcashPoW"** | 32-bit n | 32-bit k | .

Let $\mathsf{powcount}(g) :=$ | 32-bit g | .

Let $\mathsf{EquihashGen}_{n,k}(S, i) := T_{h+1\,..\,h+n}$, where

- $m := \mathsf{floor}\left(\frac{512}{n}\right)$;
- $h := (i - 1 \bmod m)\, n$;
- $T := \mathsf{GeneralCRH}_{nm}(\mathsf{powtag},\ S \,||\, \mathsf{powcount}(\mathsf{floor}\left(\frac{i-1}{m}\right)))$.

Indices of bits in T are 1-based. $\mathsf{GeneralCRH}_\ell(p, x)$ is defined as in the previous section.

Security requirement: BLAKE2b-$\ell(\mathsf{powtag}, x)$ must be collision-resistant, for any ℓ and powtag used in the protocol.

5.4.4 Pseudo Random Functions

The four independent PRFs described in §4.1.2 ***'Pseudo Random Functions'*** on p. 10 are all instantiated using the *SHA-256 compression* function:

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

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

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

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

Security requirements:

- The *SHA-256 compression* function must be collision-resistant.
- The *SHA-256 compression* function must be a PRF when keyed by the bits corresponding to x, $\mathsf{a_{sk}}$ or φ in the above diagrams, with input in the remaining bits.

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 §5.4.9 ***'Commitment'*** on p. 22. (The specific bit patterns chosen here are motivated by the possibility of future extensions that either increase $\mathrm{N^{old}}$ and/or $\mathrm{N^{new}}$ to 3, or that add an additional bit to $\mathsf{a_{sk}}$ to encode a new key type, or that require an additional PRF.)

5.4.5 Authenticated One-Time Symmetric Encryption

Let $\mathsf{Sym}.\mathbf{K} := \mathbb{B}^{256}$, $\mathsf{Sym}.\mathbf{P} := \mathbb{B}^{8\star}$, and $\mathsf{Sym}.\mathbf{C} := \mathbb{B}^{8\star}$.

Let $\mathsf{Sym}.\mathtt{Encrypt}_{\mathsf{K}}(\mathsf{P})$ be authenticated encryption using AEAD_CHACHA20_POLY1305 [RFC-7539] encryption of plaintext $\mathsf{P} \in \mathsf{Sym}.\mathbf{P}$, with empty "associated data", all-zero nonce $[0]^{96}$, and 256-bit key $\mathsf{K} \in \mathsf{Sym}.\mathbf{K}$.

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

Note: The "IETF" definition of AEAD_CHACHA20_POLY1305 from [RFC-7539] 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.

5.4.6 Key Agreement

The *key agreement scheme* specified in §4.1.4 *'Key Agreement'* on p. 11 is instantiated using Curve25519 [Bern2006] as follows.

Let KA.Public and KA.SharedSecret be the type of Curve25519 public keys (i.e. a sequence of 32 bytes), and let KA.Private be the type of Curve25519 secret keys.

Let $\mathsf{Curve25519}(\underline{n}, \underline{q})$ be the result of 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 [Bern2006, section 2].

Let $\underline{9}$ be the public byte sequence representing the Curve25519 base point.

Let $\mathsf{clamp}_{\mathsf{Curve25519}}(\underline{x})$ take a 32-byte sequence $\underline{x}$ as input and return a byte sequence representing a Curve25519 private key, with bits "clamped" as described in [Bern2006, section 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.

Define $\mathsf{KA}.\mathtt{FormatPrivate}(x) := \mathsf{clamp}_{\mathsf{Curve25519}}(x)$.

Define $\mathsf{KA}.\mathtt{Agree}(n, q) := \mathsf{Curve25519}(n, q)$.

5.4.7 Key Derivation

The *Key Derivation Function* specified in § 4.1.5 *'Key Derivation'* on p. 11 is instantiated using BLAKE2b-256 as follows:

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

where:

kdftag :=

64-bit **"ZcashKDF"**	8-bit $i-1$	$[0]^{56}$

kdfinput :=

256-bit $\mathsf{h_{Sig}}$	256-bit $\mathsf{sharedSecret}_i$	256-bit epk	256-bit $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}$

.

5.4.8 Signatures

JoinSplitSigAlg is instantiated as Ed25519 [BDL+2012], with the additional requirement that S (the integer represented by $\underline{S}$) must be less than the prime $\ell = 2^{252} + 27742317777372353535851937790883648493$, otherwise the signature is considered invalid. Ed25519 is defined as using SHA-512 internally.

The encoding of a signature is:

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

where $\underline{R}$ and $\underline{S}$ are as defined in [BDL+2012].

The encoding of a public key is as defined in [BDL+2012].

5.4.9 Commitment

The commitment scheme COMM specified in §4.1.7 *'Commitment'* on p. 12 is instantiated using SHA-256 as follows:

$$\mathsf{COMM}_{\mathsf{r}}(\mathsf{v}, \mathsf{a_{pk}}, \rho) := \mathtt{SHA256}\left(\boxed{1|0|1|1|0|0|0|0 \,|\, \text{256-bit } \mathsf{a_{pk}} \,|\, \text{64-bit } \mathsf{v} \,|\, \text{256-bit } \rho \,|\, \text{256-bit } \mathsf{r}} \right).$$

Note: The leading byte of the SHA256 input is **0xB0**.

TODO: Security requirements on SHA-256.

5.5 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* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..N^{\mathsf{new}}}$, and the result forms part of a *transmitted notes ciphertext* (see § 4.10 *'In-band secret distribution'* on p. 17 for further details).

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

The first three of these fields are as defined earlier. memo is a 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 [Unicode], 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.

5.6 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 [Bitcoin-Base58].

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

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

5.6.1 Transparent Payment Addresses

These are encoded in the same way as in **Bitcoin** [Bitcoin-Base58].

5.6.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [Bitcoin-Base58].

5.6.3 Protected Payment Addresses

A *payment address* consists of $\mathsf{a_{pk}}$ and $\mathsf{pk_{enc}}$. $\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{pk_{enc}}$ is a Bern2006 public key, for use with the encryption scheme defined in §4.10 ***'In-band secret distribution'*** on p. 17.

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

8-bit $\mathbf{0x16}$	8-bit $\mathbf{0x9A}$	256-bit $\mathsf{a_{pk}}$	256-bit $\mathsf{pk_{enc}}$

- 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 $\mathsf{a_{pk}}$.
- 256 bits specifying $\mathsf{pk_{enc}}$, using the normal encoding of a Curve25519 public key [Bern2006].

5.6.4 Spending Keys

A *spending key* consists of $\mathsf{a_{sk}}$, which is a sequence of 252 bits.

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

8-bit $\mathbf{0xAB}$	8-bit $\mathbf{0x36}$	$[0]^4$	252-bit $\mathsf{a_{sk}}$

- 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 $\mathsf{a_{sk}}$.

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

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

5.7 Zero-Knowledge Proving System

Zcash uses *zk-SNARKs* generated by its fork of *libsnark* [libsnark-fork] with the *proving system* described in [BCTV2015], which is a refinement of the systems in [PGHR2013] and [BCGTV2013].

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 $y^2 = x^3 + b$. This curve has embedding degree 12 with respect to r.
- $\mathbb{G}_2$ is the subgroup of order r in the 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 μ_r, the subgroup of r^{th} roots of unity in $\mathbb{F}^*_{q^{12}}$.

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

$$\begin{aligned}\text{Let } \mathcal{P}_2 : \mathbb{G}_2 = (&11559732032986387107991004021392285783925812861821192530917403151452391805634\, t \,+ \\ &10857046999023057135944570762232829481370756359578518086990519993285655852781, \\ &4082367875863433681332203403145435568316851327593401208105741076214120093531\, t \,+ \\ &8495653923123431417604973247489272438418190587263600148770280649306958101930).\end{aligned}$$

$\mathcal{P}_1$ 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 using the parameters above as described in [BCTV2015, Appendix B].

Note: Many details of the *proving system* are beyond the scope of this protocol document. For example, the *arithmetic circuit* verifying the *JoinSplit statement*, or its expression as a *Rank 1 Constraint System*, are not specified here. In practice it will be necessary to use the specific proving and verification keys generated for the **Zcash** production *block chain* (see § 5.8 *'JoinSplit Parameters'* on p. 25), and a *proving system* implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

5.7.1 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 most other integers in this protocol. The above encodings are consistent with the definition of EC2OSP for compressed curve points in [IEEE2004, section 5.5.6.2]. 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 [IEEE2000, Appendix A.12.8] for $\mathbb{G}_1$, and [IEEE2004, Appendix A.12.11] 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.

5.7.2 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 [BCTV2015, 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.

5.8 JoinSplit Parameters

For the testnet in release v0.11.2.z9 and later, the SHA-256 hashes of the *proving key* and *verifying key* for the *JoinSplit statement*, encoded in *libsnark* format, are:

```
226913bbdc48b70834f8e044d194ddb61c8e15329f67cdc6014f4e5ac11a82ab  z9-proving.key
4c151c562fce2cdee55ac0a0f8bd9454eb69e6a0db9a8443b58b770ec29b37f5  z9-verifying.key
```

The **Zcash** production *block chain* will use parameters obtained by a multi-party computation, which has yet to be performed.

6 Consensus Changes from Bitcoin

6.1 Encoding of Transactions

The **Zcash** *transaction* format is as follows:

Bytes	Name	Data Type	Description
4	`version`	`uint32_t`	Transaction version number; either 1 or 2.
Varies	`tx_in_count`	`compactSize uint`	Number of *transparent* inputs in this transaction.
Varies	`tx_in`	`tx_in`	Transparent inputs, encoded as in **Bitcoin**.
Varies	`tx_out_count`	`compactSize uint`	Number of *transparent* outputs in this transaction.
Varies	`tx_out`	`tx_out`	Transparent outputs, encoded as in **Bitcoin**.
4	`lock_time`	`uint32_t`	A Unix epoch time or block number, encoded as in **Bitcoin**.
Varies †	`nJoinSplit`	`compactSize uint`	The number of *JoinSplit descriptions* in `vJoinSplit`.
1802 × `nJoinSplit` †	`vJoinSplit`	`JoinSplitDescription [nJoinSplit]`	A sequence of *JoinSplit descriptions*, each encoded as described in § 6.2 *'Encoding of JoinSplit Descriptions'* on p. 27.
32 ‡	`joinSplitPubKey`	`char[32]`	An encoding of a JoinSplitSigAlg public verification key.
64 ‡	`joinSplitSig`	`char[64]`	A signature on a prefix of the *transaction* encoding, to be verified using `joinSplitPubKey`.

† The `nJoinSplit` and `vJoinSplit` fields are present if and only if `version` > 1.

‡ The `joinSplitPubKey` and `joinSplitSig` fields are present if and only if `version` > 1 and `nJoinSplit` > 0.

The encoding of `joinSplitPubKey` and the data to be signed are specified in §4.6 *'Non-malleability'* on p. 15.

The changes relative to **Bitcoin** version 1 transactions as described in [Bitcoin-Format] are:

- The *transaction version number* can be either 1 or 2. A version 1 *transaction* is equivalent to a version 2 *transaction* with `nJoinSplit` = 0. Software that parses *blocks* **MUST NOT** assume, when an encoded *block* starts with an `version` field representing a value other than 1 or 2 (e.g. future versions potentially introduced by hard forks), that it will be parseable according to this format.
- The `nJoinSplit`, `vJoinSplit`, `joinSplitPubKey`, and `joinSplitSig` fields have been added.

Software that creates *transactions* **SHOULD** use version 1 for *transactions* with no *JoinSplit descriptions*.

Note: A *transaction version number* of 2 does not have the same meaning as in **Bitcoin**, where it is associated with support for `OP_CHECKSEQUENCEVERIFY` as specified in [BIP-68]. **Zcash** was forked from **Bitcoin** v0.11.2 and does not support BIP 68, or the related BIPs 9, 112 and 113.

6.2 Encoding of JoinSplit Descriptions

An abstract *JoinSplit description*, as described in §3.4 *'JoinSplit Transfers and Descriptions'* on p. 8, is encoded in a *transaction* as an instance of a `JoinSplitDescription` type as follows:

Bytes	Name	Data Type	Description
8	`vpub_old`	`int64_t`	A value $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ that the *JoinSplit transfer* removes from the *transparent value pool*.
8	`vpub_new`	`int64_t`	A value $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ that the *JoinSplit transfer* inserts into the *transparent 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 transfer* in this *transaction*.
64	`nullifiers`	`char[32][`$\mathrm{N}^{\mathsf{old}}$`]`	A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$.
64	`commitments`	`char[32][`$\mathrm{N}^{\mathsf{new}}$`]`.	A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$.
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][`$\mathrm{N}^{\mathsf{old}}$`]`	A sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the *JoinSplit description*.
296	`zkproof`	`char[296]`	An encoding of the *zero-knowledge proof* $\pi_{\mathtt{JoinSplit}}$ (see §5.7.2 *'Encoding of Zero-Knowledge Proofs'* on p. 25).
1202	`encCiphertexts`	`char[601][`$\mathrm{N}^{\mathsf{new}}$`]`	A sequence of ciphertext components for the encrypted output *notes*, $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$.

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

6.3 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**. [Bitcoin-nBits]
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 §6.4.1 ***'Equihash'*** on p. 29.

The changes relative to **Bitcoin** version 4 blocks as described in [Bitcoin-Block] 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]`.

Note: There is no relation between the values of the `version` field of a *transaction*, and the `nVersion` field of a *block header*.

6.4 Proof of Work

Zcash uses Equihash [BK2016] as its Proof of Work. Motivations for changing the Proof of Work from *SHA-256d* used by **Bitcoin** are described in [WG2016].

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

- The `nSolution` field encodes a *valid Equihash solution* according to §6.4.1 ***'Equihash'*** on p. 29.
- The *block header* satisfies the difficulty check according to §6.4.2 ***'Difficulty filter'*** on p. 30.

6.4.1 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 and test networks are $n = 200, k = 9$.

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 powheader :=

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

For $i \in \{1 .. N\}$, let $X_i = \mathsf{EquihashGen}_{n,k}(\mathsf{powheader}, i)$.

EquihashGen is instantiated in §5.4.3 ***'Equihash Generator'*** on p. 20.

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

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

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

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:

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

Recall from §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 18 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:

<table>
<tr><td colspan="21">$\mathsf{I2BSP}_{21}(68)$</td><td colspan="21">$\mathsf{I2BSP}_{21}(41)$</td><td colspan="21">$\mathsf{I2BSP}_{21}(2^{21}-1)$</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="8">8-bit 0</td><td colspan="8">8-bit 2</td><td colspan="8">8-bit 32</td><td colspan="8">8-bit 0</td><td colspan="8">8-bit 10</td><td colspan="8">8-bit 127</td><td colspan="8">8-bit 255</td><td colspan="7">$\cdots$</td></tr>
</table>

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

Notes:

- I2BSP and BS2IP are big-endian, while the encoding of integer fields in powheader and in the instantiation of EquihashGen 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 [BK2016, section IV A].
- When EquihashGen is used to construct the input list, the index i runs sequentially from 1 to N, allowing the number of calls to BLAKE2b used in the instantiation of EquihashGen to be reduced by a factor of $\mathsf{floor}\left(\frac{512}{n}\right)$ (which is a factor of 2 for $n = 200$).

6.4.2 Difficulty filter

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

6.4.3 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.

7 Differences from the Zerocash paper

7.1 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 transfer* 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.

7.2 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 §5.5 *'Note Plaintexts and Memo Fields'* on p. 22.

7.3 Unification of Mints and Pours

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

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

Only “Pour” transactions included a *zk-SNARK* proof.

In **Zcash**, the sequence of operations added to a *transaction* (described in §7.1 *‘Transaction Structure’* on p. 30) consists only of *JoinSplit transfers*. A *JoinSplit transfer* is a Pour operation generalized to take a *transparent* UTXO as input, allowing *JoinSplit transfers* to subsume the functionality of Mints. An advantage of this is that a **Zcash** *transaction* that takes input from an UTXO can produce up to $\mathrm{N}^{\mathsf{new}}$ output *notes*, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the *JoinSplit transfer*: 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.

7.4 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* ($\mathsf{a_{sk}}$) and ρ. The *note commitment* is derived from the recipient address component $\mathsf{a_{pk}}$, 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 $\mathsf{Receive}$ (as defined in [BCG+2014, Figure 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 [LG2004].

This attack does not violate the security definitions given in [BCG+2014]. 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 [GGM2016]). 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 (see §4.4.1 **'*Dummy Notes*'** on p. 14). The *nullifiers* are used as input to BLAKE2b-256 to derive a public value $\mathsf{h_{Sig}}$ which uniquely identifies the transaction, as described in §4.3 **'*JoinSplit Descriptions*'** on p. 13. ($\mathsf{h_{Sig}}$ was already used in **Zerocash** in a way that requires it to be unique in order to maintain indistinguishability of *JoinSplit descriptions*; adding the *nullifiers* to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the *transaction* creator is an adversary.)

The ρ value for each output *note* is then derived from a random private seed φ and $\mathsf{h_{Sig}}$ using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of ρ for each output *note* is enforced by the *JoinSplit statement* (see §4.9 **'*Uniqueness of* ρ^{new}_i'** on p. 17).

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.

7.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, v, and ρ. However, the instantiation of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of $\mathsf{a_{pk}}$ and ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct values of ρ with colliding outputs of the truncated hash, and therefore the same *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 [HW2016].

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 *zero-knowledge proof* (as described under step 3 in [BCG+2014, section 1.3]). Since **Zcash** combines "Mint" and "Pour" transactions into a generalized *JoinSplit transfer* which always uses a *zero-knowledge 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 statement*.

Note: **Zcash** *note commitments* are not statistically hiding, so **Zcash** does not support the "everlasting anonymity" property described in [BCG+2014, section 8.1], even when used as described in that section. While it is possible to define a statistically hiding, computationally binding commitment scheme for this use at a 128-bit security level, the overhead of doing so within the *JoinSplit statement* was not considered to justify the benefits.

7.6 Changes to PRF inputs and truncation

The format of inputs to the PRFs instantiated in §5.4.4 **'*Pseudo Random Functions*'** on p. 20 has changed relative to **Zerocash**. There is also a requirement for another PRF, PRF^{ρ}, which must be domain-separated from the others.

In the **Zerocash** protocol, ρ^{old}_i is truncated from 256 to 254 bits in the input to $\mathsf{PRF}^{\mathsf{sn}}$ (which corresponds to $\mathsf{PRF}^{\mathsf{nf}}$ in **Zcash**). Also, $\mathsf{h_{Sig}}$ is truncated from 256 to 253 bits in the input to $\mathsf{PRF}^{\mathsf{pk}}$. These truncations are not taken into account in the security proofs.

Both truncations affect the validity of the proof sketch for Lemma D.2 in the proof of Ledger Indistinguishability in [BCG+2014, Appendix D]. In more detail:

- In the argument relating $\mathbf{H}$ and $\eth_2$, it is stated that in $\eth_2$, "for each $i \in \{1, 2\}, \mathsf{sn}_i := \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$ for a random (and not previously used) ρ". It is also argued that "the calls to $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$ are each by definition unique". The latter assertion depends on the fact that ρ is "not previously used". However, the argument is incorrect because the truncated input to $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$, i.e. $[\rho]_{254}$, may repeat even if ρ does not.

- In the same argument, it is stated that "with overwhelming probability, $\mathsf{h_{Sig}}$ is unique". In fact what is required to be unique is the truncated input to $\mathsf{PRF^{pk}}$, i.e. $[\mathsf{h_{Sig}}]_{253} = [\mathsf{CRH}(\mathsf{pk_{sig}})]_{253}$. In practice this value will be unique under a plausible assumption on CRH provided that $\mathsf{pk_{sig}}$ is chosen randomly, but no formal argument for this is presented.

Note that ρ is truncated in the input to $\mathsf{PRF^{sn}}$ but not in the input to $\mathsf{COMM_r}$, which further complicates the analysis.

As further evidence that it is essential for the proofs to explicitly take any such truncations into account, consider a slightly modified protocol in which ρ is truncated in the input to $\mathsf{COMM_r}$ but not in the input to $\mathsf{PRF^{sn}}$. In that case, it would be possible to violate balance by creating two *notes* for which ρ differs only in the truncated bits. These *notes* would have the same *note commitment* but different *nullifiers*, so it would be possible to spend the same value twice.

For resistance to Faerie Gold attacks as described in § 7.4 *'Faerie Gold attack and fix'* on p. 31, **Zcash** depends on collision resistance of both $\mathsf{GeneralCRH}_{256}$ and PRF^{ρ}. Collision resistance of a truncated hash does not follow from collision resistance of the original hash, even if the truncation is only by one bit. This motivated instantiating $\mathsf{GeneralCRH}_{\ell}$ as BLAKE2b-ℓ in **Zcash**, and avoiding truncation along any path from the inputs to the computation of $\mathsf{h_{Sig}}$ to the uses of ρ.

Since the PRFs are instantiated using `SHA256Compress` which has an input block size of 512 bits (of which 256 bits are used for the PRF input and 4 bits are used for domain separation), it was necessary to reduce the size of the PRF key to 252 bits. The key is set to $\mathsf{a_{sk}}$ in the case of $\mathsf{PRF^{addr}}$, $\mathsf{PRF^{nf}}$, and $\mathsf{PRF^{pk}}$, and to φ (which does not exist in **Zerocash**) for PRF^{ρ}, and so those values have been reduced to 252 bits. This is preferable to requiring reasoning about truncation, and 252 bits is quite sufficient for security of these cryptovalues.

7.7 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 [libsodium-Seal].

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 [Bern2006].
- 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 [MAEA2010].
- Although the **Zerocash** paper states that ECIES satisfies key privacy (as defined in [BBDP2001]), 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 [ABR1999], 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 [IEEE2004] 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 AEAD_CHACHA20_POLY1305 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 transfer*, which limits degradation of security with the number of *JoinSplit transfers*. This facilitates security analysis as explained in [DGKM2011] — 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 $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$, 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*.

The security proofs of [ABR1999] can be adapted straightforwardly to the resulting scheme. Although DHAES as defined in that paper does not pass the recipient public key or a public seed to the hash function H, this does not impair the proof because we can consider H to be the specialization of our KDF to a given recipient key and seed. It is necessary to adapt the "HDH independence" assumptions and the proof slightly to take into account that the ephemeral key is reused for two encryptions.

7.8 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 [BCG+2014, Appendix D.3]. 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 §4.9 ***'Spend authority'*** on p. 17), 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.

7.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a_{pk}}, \mathsf{pk_{enc}}), \mathsf{v}, \rho, \mathsf{r}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines it as $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$. The instantiation of $\mathsf{COMM_s}$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of COMM in **Zcash**. $\mathsf{pk_{enc}}$ is also not needed as part of a *note*: it is not an input to COMM nor is it constrained by the **Zerocash** `POUR` *statement* or the **Zcash** *JoinSplit statement*. 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 §5.7.2 ***'Encoding of Zero-Knowledge Proofs'*** on p. 25, 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 [IEEE2004].

- The range of monetary values differs. In **Zcash**, this range is $\{0 \mathinner{..} \mathsf{MAX\_MONEY}\}$; in **Zerocash** it is $\{0 \mathinner{..} 2^{64} - 1\}$. (The *JoinSplit statement* still only directly enforces that the sum of amounts in a given *JoinSplit transfer* is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

8 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, Jake Tarren, and no doubt others.

The Faerie Gold attack was found by Zooko Wilcox. The internal hash collision attack was found by Taylor Hornby. The error in the **Zerocash** proof of Balance relating to collision-resistance of $\mathsf{PRF}^{\mathsf{addr}}$ was found by Daira Hopwood. The errors in the proof of Ledger Indistinguishability mentioned in §7.6 *'Changes to PRF inputs and truncation'* on p. 32 were also found by Daira Hopwood.

9 Change history

2016.0-beta-1

- Major reorganisation to separate the abstract cryptographic protocol from the algorithm instantiations.
- Add type declarations.
- Add a "High-level Overview" section.
- 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[a].
- Add a section on consensus changes from **Bitcoin**, and the specification of Equihash.
- Complete the "Differences from the **Zerocash** paper" section.
- Correct the Merkle tree depth to 29.
- Change the length of *memo fields* to 512 bytes.
- Switch the *JoinSplit signature* scheme to Ed25519, with consequent changes to the computation of $\mathsf{h_{Sig}}$.
- Fix the lead bytes in *payment address* and *spending key* encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $\mathsf{v^{old}_{pub}}$ and $\mathsf{v^{new}_{pub}}$.
- 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, several crypto papers for security definitions, the **Bitcoin** whitepaper, the CryptoNote whitepaper, and several references to **Bitcoin** documentation.
- Reference the extended version of the **Zerocash** paper rather than the Oakland proceedings version.
- Add *JoinSplit transfers* to the Concepts section.
- Add a section on Coinbase Transactions.
- Add acknowledgements for Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, and Jake Tarren.

- Fix a `Makefile` compatibility problem with the escaping behaviour of `echo`.
- Switch to `biber` for the bibliography generation, and add backreferences.
- Make the date format in references more consistent.
- Add visited dates to all URLs in references.
- Terminology changes.

2016.0-alpha-3.1

- 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 (see §7.8 *'Omission in **Zerocash** security proof'* on p. 34).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in §4.9 *'Merkle path validity'* on p. 16 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.

10 References

[ABR1999] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem*. Cryptology ePrint Archive: Report 1999/007. Received March 17, 1999. September 1998. URL: `https://eprint.iacr.org/1999/007` (visited on 2016-08-21) (↑ p11, 33, 34).

[ANWW2013] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. *BLAKE2: simpler, smaller, fast as MD5*. January 29, 2013. URL: `https://blake2.net/#sp` (visited on 2016-08-14) (↑ p19).

[BBDP2001] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. *Key-Privacy in Public-Key Encryption*. September 2001. URL: `https://cseweb.ucsd.edu/~mihir/papers/anonenc.html` (visited on 2016-08-14). Full version. (↑ p11, 33).

[BCG+2014] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version)*. URL: `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf` (visited on 2016-08-06). A condensed version appeared in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland) 2014*, pages 459–474; IEEE, 2014. (↑ p4, 5, 16, 18, 31, 32, 34).

[BCGTV2013] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*. Cryptology ePrint Archive: Report 2013/507. Last revised October 7, 2013. URL: `https://eprint.iacr.org/2013/507` (visited on 2016-08-31). An earlier version appeared in *Proceedings of the 33rd Annual International Cryptology Conference, CRYPTO '13*, pages 90–108; IACR, 2013. (↑ p24).

[BCTV2014] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves (extended version)". In: *Advances in Cryptology - CRYPTO 2014*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pages 276–294. URL: `https://www.cs.tau.ac.il/~tromer/papers/scalablezk-20140803.pdf` (visited on 2016-09-01) (↑ p13).

[BCTV2015] 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. Last revised May 19, 2015. URL: `https://eprint.iacr.org/2013/879` (visited on 2016-08-21) (↑ p24, 25).

[BDL+2012] Daniel Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2 (September 26, 2011), pages 77–89. URL: `http://cr.yp.to/papers.html#ed25519` (visited on 2016-08-14). Document ID: a1a62a2f76d23f65d622484ddd09caf8. (↑ p21, 22).

[Bern2006] 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, February 9, 2006. URL: `http://cr.yp.to/papers.html#curve25519` (visited on 2016-08-14). Document ID: 4230efdfa673480fc079449d90f322c0. (↑ p11, 21, 23, 33).

[BIP-68] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. *Relative lock-time using consensus-enforced sequence numbers*. Bitcoin Improvement Proposal 68. Last revised November 21, 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki` (visited on 2016-09-02) (↑ p26).

[Bitcoin-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: `https://en.bitcoin.it/wiki/Base58Check_encoding` (visited on 2016-01-26) (↑ p23).

[Bitcoin-Block] *Block Headers — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#block-headers` (visited on 2016-08-08) (↑ p28).

[Bitcoin-CoinJoin] *CoinJoin — Bitcoin Wiki*. URL: `https://en.bitcoin.it/wiki/CoinJoin` (visited on 2016-08-17) (↑ p5).

[Bitcoin-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#raw-transaction-format` (visited on 2016-03-15) (↑ p26).

[Bitcoin-nBits] *Target nBits — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#target-nbits` (visited on 2016-08-13) (↑ p28).

[BK2016] 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`. URL: `https://www.internetsociety.org/sites/default/files/blogs-media/equihash-asymmetric-proof-of-work-based-generalized-birthday-problem.pdf` (visited on 2016-08-29) (↑ p6, 28, 30).

[BN2007] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm*. Cryptology ePrint Archive: Report 2000/025. Last revised July 14, 2007. URL: `https://eprint.iacr.org/2000/025` (visited on 2016-09-02) (↑ p10).

[DGKM2011] Dana Dachman-Soled, Rosario Gennaro, Hugo Krawczyk, and Tal Malkin. *Computational Extractors and Pseudorandomness*. Cryptology ePrint Archive: Report 2011/708. December 28, 2011. URL: `https://eprint.iacr.org/2011/708` (visited on 2016-09-02) (↑ p34).

[EWD-831] Edsger W. Dijkstra. *Why numbering should start at zero*. Manuscript. August 11, 1982. URL: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html` (visited on 2016-08-09) (↑ p6).

[GGM2016] Christina Garman, Matthew Green, and Ian Miers. *Accountable Privacy for Decentralized Anonymous Payments*. Cryptology ePrint Archive: Report 2016/061. Last revised January 24, 2016. URL: `https://eprint.iacr.org/2016/061` (visited on 2016-09-02) (↑ p31).

[HW2016] Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the Zcash Protocol*. Zcash blog. April 25, 2016. URL: `https://z.cash/blog/fixing-zcash-vulns.html` (visited on 2016-06-22) (↑ p32).

[IEEE2000] IEEE Computer Society. *IEEE Std 1363-2000: Standard Specifications for Public-Key Cryptography*. IEEE, August 29, 2000. DOI: `10.1109/IEEESTD.2000.92292`. URL: `http://ieeexplore.ieee.org/servlet/opac?punumber=7168` (visited on 2016-08-03) (↑ p25).

[IEEE2004] IEEE Computer Society. *IEEE Std 1363a-2004: Standard Specifications for Public-Key Cryptography – Amendment 1: Additional Techniques*. IEEE, September 2, 2004. DOI: `10.1109/IEEESTD.2004.94612`. URL: `http://ieeexplore.ieee.org/servlet/opac?punumber=9276` (visited on 2016-08-03) (↑ p25, 33, 35).

[LG2004] Eddie Lenihan and Carolyn Eve Green. *Meeting the Other Crowd: The Fairy Stories of Hidden Ireland*. TarcherPerigee, February 2004, pages 109–110. ISBN: 1-58542-206-1 (↑ p31).

[libsnark-fork] *libsnark: C++ library for zkSNARK proofs (Zcash fork)*. URL: `https://github.com/zcash/libsnark` (visited on 2016-08-14) (↑ p24).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: `https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html` (visited on 2016-02-01) (↑ p33).

[MAEA2010] 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*. IEEE, 2010, pages 1–4. DOI: `10.1109/ISIAS.2010.5604194`. URL: `https://digital.csic.es/bitstream/10261/32674/1/Gayoso_A%20Comparison%20of%20the%20Standardized%20Versions%20of%20ECIES.pdf` (visited on 2016-08-14) (↑ p33).

[Naka2008] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. October 31, 2008. URL: `https://bitcoin.org/en/bitcoin-paper` (visited on 2016-08-14) (↑ p4).

[NIST2015] NIST. *FIPS 180-4: Secure Hash Standard (SHS)*. August 2015. DOI: `10.6028/NIST.FIPS.180-4`. URL: `http://csrc.nist.gov/publications/PubsFIPS.html#180-4` (visited on 2016-08-14) (↑ p19).

[PGHR2013] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. *Pinocchio: Nearly Practical Verifiable Computation*. Cryptology ePrint Archive: Report 2013/279. Last revised May 13, 2013. URL: `https://eprint.iacr.org/2013/279` (visited on 2016-08-31) (↑ p24).

[RFC-7539] Yoav Nir and Adam Langley. *Request for Comments 7539: ChaCha20 and Poly1305 for IETF Protocols*. Internet Research Task Force (IRTF). URL: `https://tools.ietf.org/html/rfc7539` (visited on 2016-09-02). As modified by verified errata at `https://www.rfc-editor.org/errata_search.php?rfc=7539` (visited on 2016-09-02). (↑ p21).

[RFC-7693] Markku-Juhani Saarinen (ed.) *Request for Comments 7693: The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. Internet Engineering Task Force (IETF). November 2015. URL: `https://tools.ietf.org/html/rfc7693` (visited on 2016-08-31) (↑ p19).

[Unicode] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2016. URL: `http://www.unicode.org/versions/latest/` (visited on 2016-08-31) (↑ p22).

[vanS2014] Nicolas van Saberhagen. *CryptoNote v 2.0*. Date disputed. URL: `https://cryptonote.org/whitepaper.pdf` (visited on 2016-08-17) (↑ p5).

[WG2016] Zooko Wilcox and Jack Grigg. *Why Equihash?* Zcash blog. April 15, 2016. URL: `https://z.cash/blog/why-equihash.html` (visited on 2016-08-05) (↑ p28).