Zcash Protocol Specification

Version 2021.1.17 [Sprout]

Daira Hopwood[†]
Sean Bowe[†] — Taylor Hornby[†] — Nathan Wilcox[†]

March 15, 2021

Abstract. Zcash is an implementation of the *Decentralized Anonymous Payment scheme* **Zerocash**, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by **Bitcoin** with a ***shielded*** payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*). It attempted to address the problem of mining centralization by use of the *Equihash* memory-hard proof-of-work algorithm.
This specification defines the **Zcash** consensus protocol as it was at launch, and explains its differences from **Zerocash** and **Bitcoin**. It is a historical document and no longer specifies the current **Zcash** consensus protocol.

Keywords: anonymity, applications, cryptographic protocols, electronic commerce and payment, financial privacy, proof of work, zero knowledge.

Contents

[†] Electric Coin Company

1 Introduction

Zcash is an implementation of the *Decentralized Anonymous Payment scheme* **Zerocash** [BCGGMTV2014], with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by **Bitcoin** [Nakamoto2008] with a ***shielded*** payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*).

Changes from the original **Zerocash** are explained in § 8 ***'Differences from the Zerocash paper'*** on p. 52, and highlighted in magenta throughout the document.

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

The key words **MUST**, **MUST NOT**, **SHOULD**, and **SHOULD NOT** in this document are to be interpreted as described in [RFC-2119] when they appear in **ALL CAPS**. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

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 [BCGGMTV2014].

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 file an issue at `https://github.com/zcash/zips/issues` or contact `<security@z.cash>`.

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 *shielded*. Transfers of *transparent* value work essentially as in **Bitcoin** and have the same privacy properties. *Shielded* value is carried by *notes*[1], which specify an amount and a *paying key*. The *paying key* is part of a *shielded 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*. Once the *transaction* creating a *note* has been mined, the *note* is associated with a fixed *note position* in a tree of *note commitments*, and with a *nullifier*[1] unique to that *note*. Computing the *nullifier* requires the associated private *spending key*. It is infeasible to correlate the *note commitment* or *note position* with the corresponding *nullifier* without knowledge of at least this *spending 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** [Bitcoin-Protocol]. It also can include 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 that all of the following hold except with insignificant probability:

- The input and output values balance (individually for each *JoinSplit transfer*).
- For each input *note* of nonzero 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 it is infeasible to cause its *nullifier* to 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 *shielded 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 *receiving key*. This facility is used to communicate encrypted output *notes* on the *block chain* to their intended recipient, who can use the *receiving 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.[3] This contrasts with other proposals for private payment systems, such as CoinJoin [Bitcoin-CoinJoin] or **CryptoNote** [vanSaberh2014], that are based on mixing of a limited number of transactions and that therefore have smaller *note traceability sets*.

[1] In **Zerocash** [BCGGMTV2014], *notes* were called "*coins*", and *nullifiers* were called "*serial numbers*".

[2] *JoinSplit transfers* in **Zcash** generalize "*Mint*" and "*Pour*" *transactions* in **Zerocash**; see § 8.1 ***'Transaction Structure'*** on p. 52 for differences.

[3] We make this claim only for ***fully shielded*** *transactions*. It does not exclude the possibility that an adversary may use data present in the cleartext of a *transaction* such as the number of inputs and outputs, or metadata-based heuristics such as timing, to make probabilistic inferences about *transaction* linkage. For consequences of this in the case of partially shielded *transactions*, see [Peterson2017], [Quesnelle2017], and [KYMM2018].

The *nullifiers* are necessary to prevent double-spending: each *note* on the *block chain* 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

$\mathbb{B}$ means the type of bit values, i.e. $\{0, 1\}$. $\mathbb{B}^{\mathbb{Y}}$ means the type of byte values, i.e. $\{0\,..\,255\}$.

$\mathbb{N}$ means the type of nonnegative integers. $\mathbb{N}^+$ means the type of positive integers. $\mathbb{Z}$ means the type of integers. $\mathbb{Q}$ means the type of rationals.

$x : T$ is used to specify that x has type T. A cartesian product type is denoted by $S \times T$, and a function type by $S \to T$. An argument to a function can determine other argument or result types.

The type of a randomized algorithm is denoted by $S \xrightarrow{\mathbb{R}} T$. The domain of a randomized algorithm may be $()$, indicating that it requires no arguments. Given $f : S \xrightarrow{\mathbb{R}} T$ and $s : S$, sampling a variable $x : T$ from the output of f applied to s is denoted by $x \xleftarrow{\mathbb{R}} f(s)$.

Initial arguments to a function or randomized algorithm may be written as subscripts, e.g. if $x : X$, $y : Y$, and $f : X \times Y \to Z$, then an invocation of $f(x, y)$ can also be written $f_x(y)$.

$\{x : T \mid p_x\}$ means the subset of x from T for which p_x (a boolean expression depending on x) holds.

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

$S \cup T$ means the set union of S and T.

$S \cap T$ means the set intersection of S and T, i.e. $\{x : S \mid x \in T\}$.

$T^{[\ell]}$, where T is a type and ℓ is an integer, means the type of sequences of length ℓ with elements in T. For example, $\mathbb{B}^{[\ell]}$ means the set of sequences of ℓ bits, and $\mathbb{B}^{\mathbb{Y}[k]}$ means the set of sequences of k bytes.

$\mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ means the type of byte sequences of arbitrary length.

$\mathsf{length}(S)$ means the length of (number of elements in) S.

`0x` followed by a string of `monospace` hexadecimal digits means the corresponding integer converted from hexadecimal.

`"..."` means the given string represented as a sequence of bytes in US-ASCII. For example, **`"abc"`** represents the byte sequence [`0x61`, `0x62`, `0x63`].

$[0]^{\ell}$ means the sequence of ℓ zero bits.

$a..b$, 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 [BCGGMTV2014] and in [BK2016], this specification uses 1-based indexing and inclusive ranges, notwithstanding the compelling arguments to the contrary made in [EWD-831].)

$\{a\,..\,b\}$ means the set or type of integers from a through b inclusive.

[$f(x)$ 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.

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

$\mathsf{concat}_{\mathbb{B}}(S)$ 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.

$\mathsf{sorted}(S)$ means the sequence formed by sorting the elements of S.

$\mathbb{F}_n$ means the finite field with n elements, and $\mathbb{F}_n^*$ means its group under multiplication (which excludes 0).

Where there is a need to make the distinction, we denote the unique representative of $a : \mathbb{F}_n$ in the range $\{0 \,..\, n-1\}$ (or the unique representative of $a : \mathbb{F}_n^*$ in the range $\{1 \,..\, n-1\}$) as $a \bmod n$. Conversely, we denote the element of $\mathbb{F}_n$ corresponding to an integer $k : \mathbb{Z}$ as $k \pmod n$. We also use the latter notation in the context of an equality $k = k' \pmod n$ as shorthand for $k \bmod n = k' \bmod n$, and similarly $k \neq k' \pmod n$ as shorthand for $k \bmod n \neq k' \bmod n$. (When referring to constants such as 0 and 1 it is usually not necessary to make the distinction between field elements and their representatives, since the meaning is normally clear from context.)

$\mathbb{F}_n[z]$ means the ring of polynomials over z with coefficients in $\mathbb{F}_n$.

$a + b$ means the sum of a and b. This may refer to addition of integers, rationals, finite field elements, or group elements (see § 4.1.8 ***'Represented Group'*** on p. 17) according to context.

$-a$ means the value of the appropriate integer, rational, finite field, or group type such that $(-a) + a = 0$ (or when a is an element of a group $\mathbb{G}$, $(-a) + a = \mathcal{O}_{\mathbb{G}}$), and $a - b$ means $a + (-b)$.

$a \cdot b$ means the product of multiplying a and b. This may refer to multiplication of integers, rationals, or finite field elements according to context (this notation is not used for group elements).

a/b, also written $\frac{a}{b}$, means the value of the appropriate integer, rational, or finite field type such that $(a/b) \cdot b = a$.

$a \bmod q$, for $a : \mathbb{N}$ and $q : \mathbb{N}^+$, means the remainder on dividing a by q. (This usage does not conflict with the notation above for the unique representative of a field element.)

$a \oplus b$ means the bitwise-exclusive-or of a and b, and $a \,\&\, b$ means the bitwise-and of a and b. These are defined on integers (which include bits and bytes), or elementwise on equal-length sequences of integers, according to context.

$\sum_{i=1}^{\mathrm{N}} a_i$ means the sum of $a_{1..\mathrm{N}}$. $\prod_{i=1}^{\mathrm{N}} a_i$ means the product of $a_{1..\mathrm{N}}$. $\bigoplus_{i=1}^{\mathrm{N}} a_i$ means the bitwise exclusive-or of $a_{1..\mathrm{N}}$.

When $N = 0$ these yield the appropriate neutral element, i.e. $\sum_{i=1}^{0} a_i = 0$, $\prod_{i=1}^{0} a_i = 1$, and $\bigoplus_{i=1}^{0} a_i = 0$ or the all-zero bit sequence of length given by the type of a.

a^b, for a an integer or finite field element and $b : \mathbb{Z}$, means the result of raising a to the exponent b, i.e.

$$a^b := \begin{cases} \prod_{i=1}^{b} a, & \text{if } b \geq 0 \\ \prod_{i=1}^{-b} \frac{1}{a}, & \text{otherwise.} \end{cases}$$

The $[k]\,P$ notation for scalar multiplication in a group is defined in § 4.1.8 ***'Represented Group'*** on p. 17.

The convention of affixing $\star$ to a variable name is used for variables that denote bit-sequence representations of group elements.

The binary relations $<$, $\leq$, $=$, $\geq$, and $>$ have their conventional meanings on integers and rationals, and are defined lexicographically on sequences of integers.

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

$\mathsf{bitlength}(x)$, for $x : \mathbb{N}$, means the smallest integer ℓ such that $2^\ell > x$.

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

The following integer constants will be instantiated in § 5.3 ***'Constants'*** on p. 28:

> $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell_{\mathsf{Merkle}}^{\mathsf{Sprout}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, ℓ_{value}, ℓ_{hSig}, $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$, ℓ_{rcm}, ℓ_{Seed}, $\ell_{\mathsf{a_{sk}}}$, $\ell_{\varphi}^{\mathsf{Sprout}}$, MAX_MONEY, SlowStartInterval, HalvingInterval, MaxBlockSubsidy, NumFounderAddresses, PoWLimit, PoWAveragingWindow, PoWMedianBlockSpan, PoWDampingFactor, and PoWTargetSpacing.

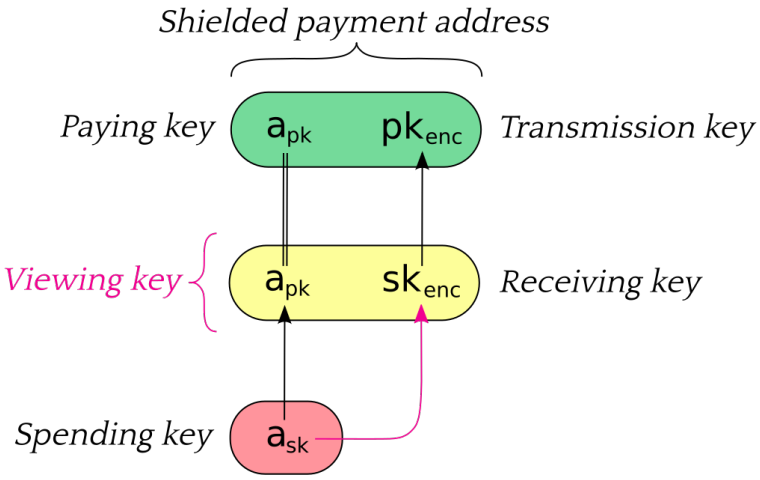
The rational constants FoundersFraction, PoWMaxAdjustDown, and PoWMaxAdjustUp, and the bit sequence constant $\mathsf{Uncommitted}^{\mathsf{Sprout}} : \mathbb{B}^{[\ell_{\mathsf{Merkle}}^{\mathsf{Sprout}}]}$, will also be defined in that section.

3 Concepts

3.1 Payment Addresses and Keys

Users who wish to receive payments in the **Zcash** protocol must have a *shielded payment address*, which is generated from a *spending key*.

The following diagram depicts the relations between key components. Arrows point from a component to any other component(s) that can be derived from it. Double lines indicate that the same component is used in multiple abstractions.



The *receiving key* sk_{enc}, *incoming viewing key* $\mathsf{ivk} = (\mathsf{a}_{pk}, \mathsf{sk}_{enc})$, and *shielded payment address* $\mathsf{addr}_{pk} = (\mathsf{a}_{pk}, \mathsf{pk}_{enc})$ are derived from the *spending key* a_{sk}, as described in § 4.2 ***'Key Components'*** on p. 19.

The composition of *shielded payment addresses*, *incoming 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 *shielded payment address* or *incoming viewing key* from a *spending key*.

Users can accept payment from multiple parties with a single *shielded payment address* 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 *shielded 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 *shielded payment address* for each payer.

Note: It is conventional in cryptography to call the key used to encrypt a message in an asymmetric encryption scheme a "*public key*". However, the *public key* used as the *transmission key* component of an address (pk_{enc}) need not be publically distributed; it has the same distribution as the *shielded payment address* itself. As mentioned above, limiting the distribution of the *shielded 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 § 4.12 ***'In-band secret distribution'*** on p. 26), since an adversary would have to know pk_{enc} in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

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

Let MAX_MONEY and $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$ be as defined in § 5.3 ***'Constants'*** on p. 28.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 5.4.6.1 ***'Sprout Note Commitments'*** on p. 34.

A **Sprout** *note* is a tuple (a_{pk}, v, ρ, rcm), where:

- $a_{pk} : \mathbb{B}^{[\ell^{Sprout}_{PRF}]}$ is the *paying key* of the recipient's *shielded payment address*;
- $v : \{0 .. \text{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*);
- $\rho : \mathbb{B}^{[\ell^{Sprout}_{PRF}]}$ is used as input to $\text{PRF}^{nfSprout}_{a_{sk}}$ to derive the *nullifier* of the *note*;
- $rcm : \text{NoteCommit}^{Sprout}.\text{Trapdoor}$ is a random *commitment trapdoor* as defined in § 4.1.7 *'Commitment'* on p. 17.

Let Note^{Sprout} be the type of a **Sprout** *note*, i.e.

$$\text{Note}^{Sprout} := \mathbb{B}^{[\ell^{Sprout}_{PRF}]} \times \{0 .. \text{MAX\_MONEY}\} \times \mathbb{B}^{[\ell^{Sprout}_{PRF}]} \times \text{NoteCommit}^{Sprout}.\text{Trapdoor}.$$

Creation of new *notes* is described in § 4.4 *'Sending Notes'* on p. 21. When *notes* are sent, only a commitment (see § 4.1.7 *'Commitment'* on p. 17) to the above values is disclosed publically, and added to a data structure called the *note commitment tree*. This allows the value and recipient to be kept private, while the commitment is used by the *zk-SNARK proof* when the *note* is spent, to check that it exists on the *block chain*.

A **Sprout** *note commitment* on a *note* $\mathbf{n} = (a_{pk}, v, \rho, rcm)$ is computed as

$$\text{NoteCommitment}^{Sprout}(\mathbf{n}) = \text{NoteCommit}^{Sprout}_{rcm}(a_{pk}, v, \rho),$$

where $\text{NoteCommit}^{Sprout}$ is instantiated in § 5.4.6.1 *'Sprout Note Commitments'* on p. 34.

The *nullifier* of a *note* is denoted nf.

A *nullifier* for a **Sprout** *note* is derived from the ρ value and the recipient's *spending key* a_{sk}.

The *nullifier* computation uses a *Pseudo Random Function* (see § 4.1.2 *'Pseudo Random Functions'* on p. 14), as described in § 4.10 *'Note Commitments and Nullifiers'* on p. 24.

A *note* is spent by proving knowledge of (ρ, a_{sk}) in zero knowledge while publically disclosing the *note*'s *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 representation of the *note commitment* cm.

The *note plaintexts* in each *JoinSplit description* are encrypted to the respective *transmission keys* $pk^{new}_{enc,1..N^{new}}$.

Each **Sprout** *note plaintext* (denoted $\mathbf{np}$) consists of

$$(\text{leadByte} : \mathbb{B}^{8}, v : \{0 .. 2^{\ell_{value}} - 1\}, \rho : \mathbb{B}^{[\ell^{Sprout}_{PRF}]}, rcm : \text{NoteCommit}^{Sprout}.\text{Trapdoor}, \text{memo} : \mathbb{B}^{8[512]}).$$

memo represents 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*.

Encodings are given in § 5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 37. The result of encryption forms part of a *transmitted note(s) ciphertext*. For further details, see § 4.12 *'In-band secret distribution'* on p. 26.

3.3 The Block Chain

At a given point in time, each *full validator* is aware of a set of candidate *blocks*. These form a tree rooted at the *genesis block*, where each node in the tree refers to its parent via the `hashPrevBlock` *block header* field (see § 7.3 *'Block Header Encoding and Consensus'* on p. 45).

A path from the root toward the leaves of the tree consisting of a sequence of one or more valid *blocks* consistent with consensus rules, is called a *valid block chain*.

Each *block* in a *block chain* has a *block height*. The *block height* of the *genesis block* is 0, and the *block height* of each subsequent *block* in the *block chain* increments by 1.

In order to choose the *best valid block chain* in its view of the overall *block* tree, a node sums the work, as defined in § 7.4.5 ***'Definition of Work'*** on p. 49, of all *blocks* in each *valid block chain*, and considers the *valid block chain* with greatest total work to be best. To break ties between leaf *blocks*, a node will prefer the *block* that it received first.

The consensus protocol is designed to ensure that for any given *block height*, the vast majority of nodes should eventually agree on their *best valid block chain* up to that height.

3.4 Transactions and Treestates

Each *block* contains one or more *transactions*.

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

Consensus rule: The remaining value in the *transparent transaction value pool* **MUST** be nonnegative.

To each *transaction* there is associated an initial *treestate*. A *treestate* consists of:

- a *note commitment tree* (§ 3.6 ***'Note Commitment Trees'*** on p. 12);
- a *nullifier set* (§ 3.7 ***'Nullifier Sets'*** on p. 12).

Validation state associated with *transparent* inputs and outputs, such as the UTXO (Unspent Transaction Output) set, is not described in this document; it is used in essentially the same way as in **Bitcoin**.

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 associated *nullifier set*.

In a given *block chain*, *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*.

JoinSplit descriptions also have interstitial input and output *treestates*, explained in the following section.

3.5 JoinSplit Transfers and Descriptions

A *JoinSplit description* is data included in a *transaction* that describes a *JoinSplit transfer*, i.e. a *shielded* value transfer. This kind of value transfer is the primary **Zcash**-specific operation performed by *transactions*.

A *JoinSplit transfer* spends N^{old} *notes* $\mathbf{n}^{old}_{1..N^{old}}$ and *transparent* input v^{old}_{pub}, and creates N^{new} *notes* $\mathbf{n}^{new}_{1..N^{new}}$ and *transparent* output v^{new}_{pub}. It is associated with a *JoinSplit statement* instance (§ 4.11.1 ***'JoinSplit Statement'*** on p. 25), for which it provides a *zk-SNARK proof*.

Each *transaction* has a sequence of *JoinSplit descriptions*.

The total v_{pub}^{new} value adds to, and the total v_{pub}^{old} value subtracts from the *transparent transaction value pool* of the containing *transaction*.

The *anchor* of each *JoinSplit description* in a *transaction* refers to a **Sprout** *treestate*.

For each of the N^{old} *shielded inputs*, a *nullifier* is revealed. This allows detection of double-spends as described in § 3.7 *'Nullifier Sets'* on p. 12.

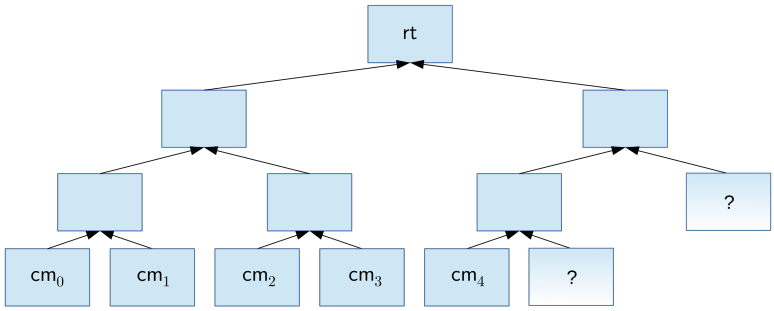
For each *JoinSplit description* in a *transaction*, an interstitial output *treestate* is constructed which adds the *note commitments* and *nullifiers* specified in that *JoinSplit description* to the input *treestate* referred to by its *anchor*. This interstitial output *treestate* is available for use as the *anchor* of subsequent *JoinSplit descriptions* in the same *transaction*. In general, therefore, the set of interstitial *treestates* associated with a *transaction* forms a tree in which the parent of each node is determined by its *anchor*.

Interstitial *treestates* are necessary because when a *transaction* is constructed, it is not known where it will eventually appear in a mined *block*. Therefore the *anchors* that it uses must be independent of its eventual position.

Consensus rules:

- The input and output values of each *JoinSplit transfer* **MUST** balance exactly.
- For the first *JoinSplit description* of a *transaction*, the *anchor* **MUST** be the output **Sprout** *treestate* of a previous *block*.
- The *anchor* of each *JoinSplit description* in a *transaction* **MUST** refer to either some earlier *block*'s final **Sprout** *treestate*, or to the interstitial output *treestate* of any prior *JoinSplit description* in the same *transaction*.

3.6 Note Commitment Trees



A *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.

A *root* of a *note commitment tree* is associated with each *treestate* (§ 3.4 *'Transactions and Treestates'* on p. 11).

Each *node* in the *incremental Merkle tree* is associated with a *hash value* of size $\ell_{\mathsf{Merkle}}^{\mathsf{Sprout}}$ bits. 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.

The *index* of a *note's commitment* at the leafmost layer ($\mathsf{MerkleDepth}^{\mathsf{Sprout}}$) is called its *note position*.

3.7 Nullifier Sets

Each *full validator* maintains a *nullifier set* logically associated with each *treestate*. As valid *transactions* containing *JoinSplit transfers* are processed, the *nullifiers* revealed in *JoinSplit descriptions* are inserted into the *nullifier set* associated with the new *treestate*. *Nullifiers* are enforced to be unique within a *valid block chain*, in order to prevent double-spends.

Consensus rule: A *nullifier* **MUST NOT** repeat either within a *transaction*, or across *transactions* in a *valid block chain*.

3.8 Block Subsidy and Founders' Reward

Like **Bitcoin**, **Zcash** creates currency when *blocks* are mined. The value created on mining a *block* is called the *block subsidy*.

The *block subsidy* is composed of a *miner subsidy* and a *Founders' Reward*.

As in **Bitcoin**, the miner of a *block* also receives *transaction fees*.

The calculations of the *block subsidy*, *miner subsidy*, and *Founders' Reward* depend on the *block height*, as defined in § 3.3 ***'The Block Chain'*** on p. 10.

The calculations are described in § 7.5 ***'Calculation of Block Subsidy and Founders' Reward'*** on p. 50.

3.9 Coinbase Transactions

The first (and only the first) *transaction* in a block is a *coinbase transaction*, which collects and spends any *miner subsidy* and *transaction fees* paid by *transactions* included in this *block*.

As described in § 7.6 ***'Payment of Founders' Reward'*** on p. 50, the *coinbase transaction* **MUST** also pay the *Founders' Reward*.

3.10 Mainnet and Testnet

The production **Zcash** *network*, which supports the **ZEC** token, is called *Mainnet*. Governance of its protocol is by agreement between the Electric Coin Company and the Zcash Foundation [ECCZF2019]. Subject to errors and omissions, each version of this document intends to describe some version (or planned version) of that agreed protocol.

All *block hashes* given in this section are in *RPC byte order* (that is, byte-reversed relative to the normal order for a SHA-256 hash).

Mainnet genesis block: `00040fe8ec8471911baa1db1266ea15dd06b4a8a5c453883c000b031973dce08`

Mainnet **Canopy** *activation block*: `000000000002038016f976744c369dce7419fca30e7171dfac703af5e5f7ad1d4`

There is also a public test *network* called *Testnet*. It supports a **TAZ** token which is intended to have no monetary value. By convention, *Testnet* activates *network upgrades* (as described in § 6 ***'Network Upgrades'*** on p. 41) before *Mainnet*, in order to allow for errors or ambiguities in their specification and implementation to be discovered. The *Testnet block chain* is subject to being rolled back to a prior *block* at any time.

Testnet genesis block: `05a60a92d99d85997cce3b87616c089f6124d7342af37106edc76126334a2c38`

Testnet **Canopy** *activation block*: `01a4d7c6aada30c87762c1bf33fff5df7266b1fd7616bfdb5227fa59bd79e7a2`

We call the smallest units of currency (on either *network*) *zatoshi*.

On *Mainnet*, 1 **ZEC** = 10^8 *zatoshi*. On *Testnet*, 1 **TAZ** = 10^8 *zatoshi*.

Other *networks* using variants of the **Zcash** protocol may exist, but are not described by this specification.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

Let $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, ℓ_{Seed}, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, ℓ_{hSig}, and $\mathrm{N}^{\mathsf{old}}$ be as defined in § 5.3 ***'Constants'*** on p. 28.

$\mathsf{MerkleCRH} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$ is a *collision-resistant hash function* used in § 4.6 ***'Merkle Path Validity'*** on p. 22. It is instantiated in § 5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 30.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a *collision-resistant hash function* used in § 4.3 ***'JoinSplit Descriptions'*** on p. 19. It is instantiated in § 5.4.1.4 ***'$\mathsf{h}_{\mathsf{Sig}}$ Hash Function'*** on p. 31.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \times \mathbb{N}^+ \to \mathbb{B}^{[n]}$ is another *hash function*, used in § 7.4.1 ***'Equihash'*** on p. 47 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 § 5.4.1.5 ***'Equihash Generator'*** on p. 31.

4.1.2 Pseudo Random Functions

PRF_x denotes a *Pseudo Random Function* keyed by x.

Let $\ell_{\mathsf{a_{sk}}}$, ℓ_{hSig}, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell^{\mathsf{Sprout}}_{\varphi}$, $\mathrm{N}^{\mathsf{old}}$, and $\mathrm{N}^{\mathsf{new}}$ be as defined in § 5.3 ***'Constants'*** on p. 28.

Four ***independent*** PRF_x are needed in our protocol:

$$
\begin{aligned}
\mathsf{PRF}^{\mathsf{addr}} &: \mathbb{B}^{[\ell_{\mathsf{a_{sk}}}]} \times \mathbb{B}^{\mathbb{Y}} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \\
\mathsf{PRF}^{\mathsf{pk}} &: \mathbb{B}^{[\ell_{\mathsf{a_{sk}}}]} \times \{1..\mathrm{N}^{\mathsf{old}}\} \times \mathbb{B}^{[\ell_{\mathsf{hSig}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \\
\mathsf{PRF}^{\rho} &: \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\varphi}]} \times \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{[\ell_{\mathsf{hSig}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \\
\mathsf{PRF}^{\mathsf{nfSprout}} &: \mathbb{B}^{[\ell_{\mathsf{a_{sk}}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}
\end{aligned}
$$

These are used in § 4.11.1 ***'JoinSplit Statement'*** on p. 25; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in § 4.2 ***'Key Components'*** on p. 19.

They are instantiated in § 5.4.2 ***'Pseudo Random Functions'*** on p. 31.

Security requirements:

- Security definitions for *Pseudo Random Functions* are given in [BDJR2000, section 4].
- In addition to being *Pseudo Random Functions*, it is required that $\mathsf{PRF}^{\mathsf{addr}}_x$, PRF^{ρ}_x, and $\mathsf{PRF}^{\mathsf{nfSprout}}_x$ be *collision-resistant* across all x — i.e. finding $(x, y) \neq (x', y')$ such that $\mathsf{PRF}^{\mathsf{addr}}_x(y) = \mathsf{PRF}^{\mathsf{addr}}_{x'}(y')$ should not be feasible, and similarly for PRF^{ρ} and $\mathsf{PRF}^{\mathsf{nfSprout}}$.

Non-normative note: $\mathsf{PRF}^{\mathsf{nfSprout}}$ was called $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash** [BCGGMTV2014], and just $\mathsf{PRF}^{\mathsf{nf}}$ in previous versions of this specification.

4.1.3 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 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 ∧ IND-CPA)-secure [BN2007]. *"One-time"* here means that an honest protocol participant will almost surely encrypt only one message with a given key; however, the adversary may make many adaptive chosen ciphertext queries for a given key.

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 $: \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]} \to$ KA.Private be a function to convert a bit string of length $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$ to a KA *private key*.

Let KA.DerivePublic $:$ KA.Private $\times$ KA.Public $\to$ KA.Public be a function that derives the KA *public key* corresponding to a given KA *private key* and base point.

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

Let KA.Base $:$ KA.Public be a public base point.

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 [Bernstein2006, 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 $\mathsf{KDF}^{\mathsf{Sprout}} : \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{[\ell_{\mathsf{hSig}}]} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{SharedSecret} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public} \to \mathsf{Sym}.\mathbf{K}$ be a *Key Derivation Function* suitable for use with $\mathsf{KA}^{\mathsf{Sprout}}$, 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 $\mathsf{g}Sprout := \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Base}$.

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

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

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

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

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

Note: The given definition only requires ciphertexts to be indistinguishable between *transmission keys* that are outputs of $\mathsf{KA}^{\mathsf{Sprout}}$.DerivePublic (which includes all keys generated as in § 4.2 ***'Key Components'*** on p. 19). 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 Signature

A *signature scheme* Sig defines:

- a type of *signing keys* Sig.Private;
- a type of *validating keys* Sig.Public;
- a type of messages Sig.Message;
- a type of signatures Sig.Signature;
- a randomized *signing key* generation algorithm $\mathsf{Sig.GenPrivate} : () \xrightarrow{\mathrm{R}} \mathsf{Sig.Private}$;
- an injective *validating key* derivation algorithm $\mathsf{Sig.DerivePublic} : \mathsf{Sig.Private} \to \mathsf{Sig.Public}$;
- a randomized signing algorithm $\mathsf{Sig.Sign} : \mathsf{Sig.Private} \times \mathsf{Sig.Message} \xrightarrow{\mathrm{R}} \mathsf{Sig.Signature}$;
- a validating algorithm $\mathsf{Sig.Validate} : \mathsf{Sig.Public} \times \mathsf{Sig.Message} \times \mathsf{Sig.Signature} \to \mathbb{B}$;

such that for any *signing key* $\mathsf{sk} \xleftarrow{\mathrm{R}} \mathsf{Sig.GenPrivate}()$ and corresponding *validating key* $\mathsf{vk} = \mathsf{Sig.DerivePublic}(\mathsf{sk})$, and any $m : \mathsf{Sig.Message}$ and $s : \mathsf{Sig.Signature} \xleftarrow{\mathrm{R}} \mathsf{Sig.Sign}_{\mathsf{sk}}(m)$, $\mathsf{Sig.Validate}_{\mathsf{vk}}(m, s) = 1$.

Zcash uses two *signature schemes*:

- one used for signatures that can be validated by script operations such as `OP_CHECKSIG` and `OP_CHECKMULTISIG` as in **Bitcoin**;
- one called JoinSplitSig which is used to sign *transactions* that contain at least one *JoinSplit description* (instantiated in § 5.4.5 **'Ed25519'** on p. 33).

The signature scheme used in script operations is instantiated by ECDSA on the secp256k1 curve. JoinSplitSig is instantiated by Ed25519.

Security requirement: JoinSplitSig must be Strongly Unforgeable under (non-adaptive) Chosen Message Attack (SU-CMA), as defined for example in [BDEHR2011, Definition 6].[4] This allows an adversary to obtain signatures on chosen messages, and then requires it to be infeasible for the adversary to forge a previously unseen valid (message, signature) pair without access to the *signing key*.

Non-normative notes:

- A fresh signature key pair is generated for each *transaction* containing a *JoinSplit description*. Since each key pair is only used for one signature (see § 4.8 ***'Non-malleability'*** on p. 23), a *one-time signature scheme* would suffice for JoinSplitSig. This is also the reason why only security against ***non-adaptive*** chosen message attack is needed. In fact the instantiation of JoinSplitSig uses a scheme designed for security under adaptive attack even when multiple signatures are signed under the same key.
- SU-CMA security requires it to be infeasible for the adversary, not knowing the *private key*, to forge a distinct signature on a previously seen message. That is, *JoinSplit signatures* are intended to be *nonmalleable* in the sense of [BIP-62].
- The terminology used in this specification is that we "validate" signatures, and "verify" *zk-SNARK proofs*.

[4] The scheme defined in that paper was attacked in [LM2017], but this has no impact on the applicability of the definition.

4.1.7 Commitment

A *commitment scheme* is a function that, given a *commitment trapdoor* generated at random 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 $\mathsf{COMM.Input}$, a type of commitments $\mathsf{COMM.Output}$, a type of *commitment trapdoors* $\mathsf{COMM.Trapdoor}$, and a *trapdoor* generator $\mathsf{COMM.GenTrapdoor} : () \xrightarrow{\mathrm{R}} \mathsf{COMM.Trapdoor}$.

Let $\mathsf{COMM} : \mathsf{COMM.Trapdoor} \times \mathsf{COMM.Input} \to \mathsf{COMM.Output}$ be a function satisfying the following security requirements.

Security requirements:

- **Computational hiding:** For all $x, x' : \mathsf{COMM.Input}$, the distributions $\{ \mathsf{COMM}_r(x) \mid r \xleftarrow{\mathrm{R}} \mathsf{COMM.GenTrapdoor}() \}$ and $\{ \mathsf{COMM}_r(x') \mid r \xleftarrow{\mathrm{R}} \mathsf{COMM.GenTrapdoor}() \}$ are computationally indistinguishable.
- **Computational binding:** It is infeasible to find $x, x' : \mathsf{COMM.Input}$ and $r, r' : \mathsf{COMM.Trapdoor}$ such that $x \neq x'$ and $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(x')$.

Note: If it were only feasible to find $x : \mathsf{COMM.Input}$ and $r, r' : \mathsf{COMM.Trapdoor}$ such that $r \neq r'$ and $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(x)$, this would not contradict the computational binding security requirement.

Let ℓ_{rcm}, $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, and ℓ_{value} be as defined in § 5.3 ***'Constants'*** on p. 28.

Define $\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor} := \mathbb{B}^{[\ell_{\mathsf{rcm}}]}$ and $\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output} := \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$.

Sprout uses a *note commitment scheme*

$$\mathsf{NoteCommit}^{\mathsf{Sprout}} : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \times \{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]} \to \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output},$$

instantiated in § 5.4.6.1 ***'Sprout Note Commitments'*** on p. 34.

4.1.8 Represented Group

A *represented group* $\mathbb{G}$ consists of:

- a subgroup order parameter $r_{\mathbb{G}} : \mathbb{N}^+$, which must be prime;
- a cofactor parameter $h_{\mathbb{G}} : \mathbb{N}^+$;
- a group $\mathbb{G}$ of order $h_{\mathbb{G}} \cdot r_{\mathbb{G}}$, written additively with operation $+ : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, and additive identity $\mathcal{O}_{\mathbb{G}}$;
- a bit-length parameter $\ell_{\mathbb{G}} : \mathbb{N}$;
- a representation function $\mathsf{repr}_{\mathbb{G}} : \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]}$ and an abstraction function $\mathsf{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\}$, such that $\mathsf{abst}_{\mathbb{G}}$ is the left inverse of $\mathsf{repr}_{\mathbb{G}}$, i.e. for all $P \in \mathbb{G}$, $\mathsf{abst}_{\mathbb{G}}\big(\mathsf{repr}_{\mathbb{G}}(P)\big) = P$.

Note: Ideally, we would also have that for all S not in the image of $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{abst}_{\mathbb{G}}(S) = \bot$. This may not be true in all cases, i.e. there can be *non-canonical* encodings $P\star$ such that $\mathsf{repr}_{\mathbb{G}}\big(\mathsf{abst}_{\mathbb{G}}(P\star)\big) \neq P\star$.

Define $\mathbb{G}^{(r)}$ as the order-$r_{\mathbb{G}}$ subgroup of $\mathbb{G}$, which is called a *represented subgroup*. Note that this includes $\mathcal{O}_{\mathbb{G}}$. For the set of points of order $r_{\mathbb{G}}$ (which excludes $\mathcal{O}_{\mathbb{G}}$), we write $\mathbb{G}^{(r)*}$.

Define $\mathbb{G}^{(r)}_{\star} := \{\mathsf{repr}_{\mathbb{G}}(P) : \mathbb{B}^{[\ell_{\mathbb{G}}]} \mid P \in \mathbb{G}^{(r)}\}$. (This intentionally excludes *non-canonical* encodings if there are any.)

For $G : \mathbb{G}$ we write $-G$ for the negation of G, such that $(-G) + G = \mathcal{O}_{\mathbb{G}}$. We write $G - H$ for $G + (-H)$.

We also extend the $\sum$ notation to addition on group elements.

For $G : \mathbb{G}$ and $k : \mathbb{Z}$ we write $[k]\,G$ for scalar multiplication on the group, i.e.

$$[k]\,G := \begin{cases} \sum_{i=1}^{k} G, & \text{if } k \geq 0 \\ \sum_{i=1}^{-k} (-G), & \text{otherwise.} \end{cases}$$

For $G : \mathbb{G}$ and $a : \mathbb{F}_{r_{\mathbb{G}}}$, we may also write $[a]\,G$ meaning $[a \bmod r_{\mathbb{G}}]\,G$ as defined above. (This variant is not defined for fields other than $\mathbb{F}_{r_{\mathbb{G}}}$.)

4.1.9 Represented Pairing

A *represented pairing* $\mathbb{PAIR}$ consists of:

- a group order parameter $r_{\mathbb{PAIR}} : \mathbb{N}^+$ which must be prime;
- two *represented subgroups* $\mathbb{PAIR}^{(r)}_{1,2}$, both of order $r_{\mathbb{PAIR}}$;
- a group $\mathbb{PAIR}^{(r)}_T$ of order $r_{\mathbb{PAIR}}$, written multiplicatively with operation $\cdot : \mathbb{PAIR}^{(r)}_T \times \mathbb{PAIR}^{(r)}_T \to \mathbb{PAIR}^{(r)}_T$ and group identity $\mathbf{1}_{\mathbb{PAIR}}$;
- three generators $\mathcal{P}_{\mathbb{PAIR}_{1,2,T}}$ of $\mathbb{PAIR}^{(r)}_{1,2,T}$ respectively;
- a pairing function $\hat{e}_{\mathbb{PAIR}} : \mathbb{PAIR}^{(r)}_1 \times \mathbb{PAIR}^{(r)}_2 \to \mathbb{PAIR}^{(r)}_T$ satisfying:
  - (Bilinearity) for all $a, b : \mathbb{F}^*_r$, $P : \mathbb{PAIR}^{(r)}_1$, and $Q : \mathbb{PAIR}^{(r)}_2$, $\hat{e}_{\mathbb{PAIR}}([a]\,P, [b]\,Q) = \hat{e}_{\mathbb{PAIR}}(P, Q)^{a \cdot b}$; and
  - (Nondegeneracy) there does not exist $P : \mathbb{PAIR}^{(r)*}_1$ such that for all $Q : \mathbb{PAIR}^{(r)}_2$, $\hat{e}_{\mathbb{PAIR}}(P, Q) = \mathbf{1}_{\mathbb{PAIR}}$.

4.1.10 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* [BCCGLRT2014].

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 type of *primary inputs* ZK.PrimaryInput;
- a type of *auxiliary inputs* ZK.AuxiliaryInput;
- a type of *zk-SNARK proofs* ZK.Proof;
- a type ZK.SatisfyingInputs $\subseteq$ ZK.PrimaryInput $\times$ ZK.AuxiliaryInput of inputs satisfying the *statement*;
- a randomized key pair generation algorithm ZK.Gen : () $\xrightarrow{R}$ ZK.ProvingKey $\times$ ZK.VerifyingKey;
- a proving algorithm ZK.Prove : ZK.ProvingKey $\times$ ZK.SatisfyingInputs $\to$ ZK.Proof;
- a verifying algorithm ZK.Verify : ZK.VerifyingKey $\times$ ZK.PrimaryInput $\times$ ZK.Proof $\to \mathbb{B}$;

The security requirements below are supposed to hold with overwhelming probability for $(\mathsf{pk}, \mathsf{vk}) \xleftarrow{R}$ ZK.Gen().

Security requirements:

- **Completeness:** An honestly generated proof will convince a verifier: for any $(x, w) \in \mathsf{ZK.SatisfyingInputs}$, if $\mathsf{ZK.Prove}_{\mathsf{pk}}(x, w)$ outputs π, then $\mathsf{ZK.Verify}_{\mathsf{vk}}(x, \pi) = 1$.
- **Knowledge Soundness:** For any adversary $\mathcal{A}$ able to find an $x : \mathsf{ZK.PrimaryInput}$ and proof $\pi : \mathsf{ZK.Proof}$ such that $\mathsf{ZK.Verify}_{\mathsf{vk}}(x, \pi) = 1$, there is an efficient extractor $\mathcal{E}_{\mathcal{A}}$ such that if $\mathcal{E}_{\mathcal{A}}(\mathsf{vk}, \mathsf{pk})$ returns w, then the probability that $(x, w) \notin \mathsf{ZK.SatisfyingInputs}$ is insignificant.
- **Statistical Zero Knowledge:** An honestly generated proof is statistical zero knowledge. That is, there is a feasible stateful simulator $\mathcal{S}$ such that, for all stateful distinguishers $\mathcal{D}$, the following two probabilities are not significantly different:

$$\Pr\left[\begin{array}{c|l} (x,w) \in \mathsf{ZK.SatisfyingInputs} & (\mathsf{pk},\mathsf{vk}) \xleftarrow{R} \mathsf{ZK.Gen}() \\ \mathcal{D}(\pi) = 1 & (x,w) \xleftarrow{R} \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ & \pi \xleftarrow{R} \mathsf{ZK.Prove}_{\mathsf{pk}}(x,w) \end{array}\right] \text{ and } \Pr\left[\begin{array}{c|l} (x,w) \in \mathsf{ZK.SatisfyingInputs} & (\mathsf{pk},\mathsf{vk}) \xleftarrow{R} \mathcal{S}() \\ \mathcal{D}(\pi) = 1 & (x,w) \xleftarrow{R} \mathcal{D}(\mathsf{pk},\mathsf{vk}) \\ & \pi \xleftarrow{R} \mathcal{S}(x) \end{array}\right]$$

These definitions are derived from those in [BCTV2014b, Appendix C], adapted to state concrete security for a fixed circuit, rather than asymptotic security for arbitrary circuits. ($\mathsf{ZK.Prove}$ corresponds to P, $\mathsf{ZK.Verify}$ corresponds to V, and $\mathsf{ZK.SatisfyingInputs}$ corresponds to $\mathcal{R}_C$ in the notation of that appendix.)

The Knowledge Soundness 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 $(x, w) \in \mathsf{ZK.SatisfyingInputs}$. Note that Knowledge Soundness implies Soundness — i.e. the property that it is infeasible to find a new proof π where $\mathsf{ZK.Verify}_{\mathsf{vk}}(x, \pi) = 1$ without ***there existing*** an *auxiliary input* w such that $(x, w) \in \mathsf{ZK.SatisfyingInputs}$.

Non-normative notes:

- The above properties do not include *nonmalleability* [DSDCOPS2001], and the design of the protocol using the *zero-knowledge proving system* must take this into account.
- The terminology used in this specification is that we "validate" signatures, and "verify" *zk-SNARK proofs*.

The *proving system* is instantiated in § 5.4.8.1 'BCTV14' on p. 36. ZKJoinSplit refers to this *proving system* with the BN-254 pairing, specialized to the *JoinSplit statement* given in § 4.11.1 ***'JoinSplit Statement'*** on p. 25. In this case we omit the key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be the particular *proving key* and *verifying key* defined by the *JoinSplit parameters* in § 5.7 'BCTV14 ***zk-SNARK Parameters***' on p. 40.

4.2 Key Components

Let $\ell_{\mathsf{a_{sk}}}$ be as defined in § 5.3 ***'Constants'*** on p. 28.

Let $\mathsf{PRF}^{\mathsf{addr}}$ be a *Pseudo Random Function*, instantiated in § 5.4.2 ***'Pseudo Random Functions'*** on p. 31.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be a *key agreement scheme*, instantiated in § 5.4.4.1 ***'Sprout Key Agreement'*** on p. 32.

A new **Sprout** *spending key* $\mathsf{a_{sk}}$ is generated by choosing a bit sequence 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:

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

4.3 JoinSplit Descriptions

A *JoinSplit transfer*, as specified in § 3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 11, 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 JoinSplitSig public *validating key* and signature.

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, ℓ_{Seed}, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, and MAX_MONEY be as defined in § 5.3 *'Constants'* on p. 28.

Let hSigCRH be as defined in § 4.1.1 *'Hash Functions'* on p. 14.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.7 *'Commitment'* on p. 17.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 4.1.4 *'Key Agreement'* on p. 15.

Let Sym be as defined in § 4.1.3 *'Symmetric Encryption'* on p. 14.

Let ZKJoinSplit be as defined in § 4.1.10 *'Zero-Knowledge Proving System'* on p. 18.

A *JoinSplit description* comprises $(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{rt}^{\mathsf{Sprout}}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{epk}, \mathsf{randomSeed}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}, \pi_{\mathsf{ZKJoinSplit}}, \mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$ where

- $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} : \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent transaction value pool*;
- $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} : \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent transaction value pool*;
- $\mathsf{rt}^{\mathsf{Sprout}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in § 3.3 *'The Block Chain'* on p. 10, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is the sequence of *nullifiers* for the input *notes*;
- $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} : \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ is a key agreement *public key*, used to derive the key for encryption of the *transmitted notes ciphertext* (§ 4.12 *'In-band secret distribution'* on p. 26);
- $\mathsf{randomSeed} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is a sequence of tags that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the input *notes*;
- $\pi_{\mathsf{ZKJoinSplit}} : \mathsf{ZKJoinSplit.Proof}$ is a *zk proof* with *primary input* $(\mathsf{rt}^{\mathsf{Sprout}}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}})$ for the *JoinSplit statement* defined in § 4.11.1 *'JoinSplit Statement'* on p. 25;
- $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}} : \mathsf{Sym}.\mathbf{C}^{[\mathrm{N}^{\mathsf{new}}]}$ is a sequence of ciphertext components for the encrypted output *notes*.

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

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

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

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 $0 \leq \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} \leq \mathsf{MAX\_MONEY}$).
- The proof $\pi_{\mathsf{ZKJoinSplit}}$ **MUST** be valid given a *primary input* formed from the relevant other fields and $\mathsf{h}_{\mathsf{Sig}}$ — i.e. $\mathsf{ZKJoinSplit.Verify}\big((\mathsf{rt}^{\mathsf{Sprout}}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}), \pi_{\mathsf{ZKJoinSplit}}\big) = 1$.
- Either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ **MUST** be zero.

4.4 Sending Notes

In order to send **Sprout** *shielded* value, the sender constructs a *transaction* containing one or more *JoinSplit descriptions*.

Let JoinSplitSig be as specified in § 4.1.6 *'Signature'* on p. 16.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as specified in § 4.1.7 *'Commitment'* on p. 17.

Let ℓ_{Seed} and $\ell_{\varphi}^{\mathsf{Sprout}}$ be as specified in § 5.3 *'Constants'* on p. 28.

Sending a *transaction* containing *JoinSplit descriptions* involves first generating a new JoinSplitSig key pair:

$\texttt{joinSplitPrivKey} \xleftarrow{R} \mathsf{JoinSplitSig.GenPrivate}()$

$\texttt{joinSplitPubKey} := \mathsf{JoinSplitSig.DerivePublic}(\texttt{joinSplitPrivKey})$.

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

- Choose uniformly random $\mathsf{rcm}_i \xleftarrow{R} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{GenTrapdoor}()$.
- Compute $\rho_i = \mathsf{PRF}_{\varphi}^{\rho}(i, \mathsf{h}_{\mathsf{Sig}})$.
- Compute $\mathsf{cm}_i = \mathsf{NoteCommit}_{\mathsf{rcm}_i}^{\mathsf{Sprout}}(\mathsf{a}_{\mathsf{pk},i}, \mathsf{v}_i, \rho_i)$.
- Let $\mathbf{np}_i = (\mathtt{0x00}, \mathsf{v}_i, \rho_i, \mathsf{rcm}_i, \mathsf{memo}_i)$.

$\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ are then encrypted to the recipient *transmission keys* $\mathsf{pk}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, giving the *transmitted notes ciphertext* $(\mathsf{epk}, \mathsf{C}_{1..\mathrm{N}^{\mathsf{new}}}^{\mathsf{enc}})$, as described in § 4.12 *'In-band secret distribution'* on p. 26.

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 obtains $\mathsf{dataToBeSigned} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}}$ as described in § 4.8 *'Non-malleability'* on p. 23, and signs it with the private *JoinSplit signing key*:

$\texttt{joinSplitSig} \xleftarrow{R} \mathsf{JoinSplitSig.Sign}_{\texttt{joinSplitPrivKey}}(\mathsf{dataToBeSigned})$

Then the encoded *transaction* including joinSplitSig is submitted to the peer-to-peer network.

The facility to send to **Sprout** addresses is **OPTIONAL** for a particular node or wallet implementation.

4.5 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*.

Let $\ell_{\mathsf{a_{sk}}}$ and $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$ be as defined in § 5.3 *'Constants'* on p. 28.

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 14.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.7 *'Commitment'* on p. 17.

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

- Generate a new uniformly random *spending key* $\mathsf{a}_{\mathsf{sk},i}^{\mathsf{old}} \xleftarrow{R} \mathbb{B}^{[\ell_{\mathsf{a}_{\mathsf{sk}}}]}$ and derive its *paying key* $\mathsf{a}_{\mathsf{pk},i}^{\mathsf{old}}$.
- Set $\mathsf{v}_i^{\mathsf{old}} = 0$.
- Choose uniformly random $\rho_i^{\mathsf{old}} \xleftarrow{R} \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]}$ and $\mathsf{rcm}_i^{\mathsf{old}} \xleftarrow{R} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{GenTrapdoor}()$.
- Compute $\mathsf{nf}_i^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSprout}}_{\mathsf{a}_{\mathsf{sk},i}^{\mathsf{old}}}(\rho_i^{\mathsf{old}})$.
- Let path_i be a *dummy Merkle path* for the *auxiliary input* to the *JoinSplit statement* (this will not be checked).
- When generating the *JoinSplit proof*, set $\mathsf{enforceMerklePath}_i$ to 0.

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

4.6 Merkle Path Validity

The depth of the *note commitment tree* is MerkleDepth (defined in § 5.3 ***'Constants'*** on p. 28).

Each *node* in the *incremental Merkle tree* is associated with a *hash value*, which is a bit 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 $\mathsf{M}_i^{\mathsf{h}}$ be the *hash value* associated with the *node* at *index* i in *layer* h.

The *nodes* at *layer* MerkleDepth are called *leaf nodes*. When a *note commitment* is added to the tree, it occupies the *leaf node hash value* $\mathsf{M}_i^{\mathsf{MerkleDepth}}$ for the next available i.

As-yet unused *leaf nodes* are associated with a distinguished *hash value* $\mathsf{Uncommitted}^{\mathsf{Sprout}}$. It is assumed to be infeasible to find a preimage *note* $\mathbf{n}$ such that $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}) = \mathsf{Uncommitted}^{\mathsf{Sprout}}$.

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

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

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

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

where

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

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

4.7 SIGHASH Transaction Hashing

Bitcoin and **Zcash** use signatures and/or non-interactive proofs associated with *transaction* inputs to authorize spending. Because these signatures or proofs could otherwise be replayed in a different *transaction*, it is necessary to "bind" them to the *transaction* for which they are intended. This is done by hashing information about the *transaction* and (where applicable) the specific input, to give a *SIGHASH transaction hash* which is then used for the Spend authorization. The means of authorization differs between *transparent inputs* and inputs to **Sprout** *JoinSplit transfers*, but for a given *transaction version* the same *SIGHASH transaction hash* algorithm is used.

In the case of **Zcash**, the BCTV14 proving system used is ***malleable***, meaning that there is the potential for an adversary who does not know all of the *auxiliary inputs* to a proof, to malleate it in order to create a new proof involving related *auxiliary inputs* [DSDCOPS2001]. This can be understood as similar to a malleability attack on an encryption scheme, in which an adversary can malleate a ciphertext in order to create an encryption of a related plaintext, without knowing the original plaintext. **Zcash** has been designed to mitigate malleability attacks, as described in § 4.8 *'Non-malleability'* on p. 23.

To provide additional flexibility when combining spend authorizations from different sources, **Bitcoin** defines several *SIGHASH types* that cover various parts of a transaction [Bitcoin-SigHash]. One of these types is SIGHASH_ALL, which is used for **Zcash**-specific signatures, i.e. *JoinSplit signatures*. In this case the *SIGHASH transaction hash* is not associated with a *transparent input*, and so the input to hashing excludes ***all*** of the `scriptSig` fields in the non-**Zcash**-specific parts of the *transaction*.

In **Zcash**, all *SIGHASH types* are extended to cover the **Zcash**-specific fields `nJoinSplit`, `vJoinSplit`, and if present `joinSplitPubKey`. These fields are described in § 7.1 *'Transaction Encoding and Consensus'* on p. 42. The hash ***does not*** cover the field `joinSplitSig`.

The *SIGHASH algorithm* used prior to **Overwinter** activation, i.e. for version 1 and 2 *transactions*, will be defined in [ZIP-76] (to be written).

4.8 Non-malleability

Let dataToBeSigned be the hash of the *transaction*, not associated with an input, using the SIGHASH_ALL *SIGHASH type*.

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

JoinSplitSig is instantiated in § 5.4.5 'Ed25519' on p. 33.

If `nJoinSplit` is zero, the `joinSplitPubKey` and `joinSplitSig` fields are omitted. Otherwise, a *transaction* has a correct *JoinSplit signature* if and only if $\mathsf{JoinSplitSig.Validate}_{\texttt{joinSplitPubKey}}(\mathsf{dataToBeSigned}, \texttt{joinSplitSig}) = 1$.

Let $\mathsf{h}_{\mathsf{Sig}}$ be computed as specified in § 4.3 *'JoinSplit Descriptions'* on p. 19.

Let $\mathsf{PRF}^{\mathsf{pk}}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 14.

For each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$, the creator of a *JoinSplit description* calculates $\mathsf{h}_i = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(i, \mathsf{h}_{\mathsf{Sig}})$.

The correctness of $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ is enforced by the *JoinSplit statement* given in § 4.11.1 *'JoinSplit Statement'* on p. 25. This ensures that a holder of all of the $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$ for every *JoinSplit description* in the *transaction* has authorized the use of the private *signing key* corresponding to `joinSplitPubKey` to sign this *transaction*.

4.9 Balance

In **Bitcoin**, all inputs to and outputs from a *transaction* are transparent. The total value of *transparent outputs* must not exceed the total value of *transparent inputs*. The net value of *transparent inputs* minus *transparent outputs* is transferred to the miner of the *block* containing the *transaction*; it is added to the *miner subsidy* in the *coinbase transaction* of the *block*.

Zcash Sprout extends this by adding *JoinSplit transfers*. Each *JoinSplit transfer* can be seen, from the perspective of the *transparent transaction value pool*, as an input and an output simultaneously.

$\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ takes value from the *transparent transaction value pool* and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ adds value to the *transparent transaction 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.

As defined in [ZIP-209], the ***Sprout*** *chain value pool balance* for a given *block chain* is the sum of all $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ field values for *transactions* in the *block chain*, minus the sum of all $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ fields values for transactions in the *block chain*.

Consensus rule: If the ***Sprout*** *chain value pool balance* would become negative in the *block chain* created as a result of accepting a *block*, then all nodes **MUST** reject the block as invalid.

Unlike original **Zerocash** [BCGGMTV2014], **Zcash** does not have a distinction between Mint and Pour operations. The addition of v^{old}_{pub} to a *JoinSplit description* subsumes the functionality of both Mint and Pour.

Also, a difference in the number of real input *notes* does not by itself cause two *JoinSplit descriptions* to be distinguishable.

As stated in § 4.3 ***'JoinSplit Descriptions'*** on p. 19, either v^{old}_{pub} or v^{new}_{pub} **MUST** be zero. No generality is lost because, if a *transaction* in which both v^{old}_{pub} and v^{new}_{pub} were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(v^{old}_{pub}, v^{new}_{pub})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

4.10 Note Commitments and Nullifiers

A *transaction* that contains one or more *JoinSplit descriptions*, when entered into the *block chain*, 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 associated *treestate*. A *transaction* is not valid if it would have added a *nullifier* to the *nullifier set* that already exists in the set (see § 3.7 ***'Nullifier Sets'*** on p. 12).

Each *note* has a ρ component.

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ be as instantiated in § 5.4.2 ***'Pseudo Random Functions'*** on p. 31.

For a **Sprout** *note*, the *nullifier* is derived as $\mathsf{PRF}^{\mathsf{nfSprout}}_{a_{sk}}(\rho)$, where a_{sk} is the *spending key* associated with the *note*.

Security requirement: The requirements on *nullifier* derivation are as follows:

- The derived *nullifier* must be determined completely by the fields of the *note*, in a way that can be checked in the corresponding statement that controls spends (i.e. the *JoinSplit statement*).
- Under the assumption that ρ values are unique, it must not be possible to generate two *notes* with distinct *note commitments* but the same *nullifier*. (See § 8.4 ***'Faerie Gold attack and fix'*** on p. 53 for further discussion.)
- Given a set of *nullifiers* of ***a priori*** unknown *notes*, they must not be linkable to those *notes* with probability greater than expected by chance, even to an adversary with the corresponding *incoming viewing keys* (but not *full viewing keys*), and even if the adversary may have created the *notes*.

4.11 Zk-SNARK Statements

4.11.1 JoinSplit Statement

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, ℓ_{value}, $\ell_{\mathsf{a_{sk}}}$, $\ell^{\mathsf{Sprout}}_{\varphi}$, ℓ_{hSig}, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$ be as defined in § 5.3 *'Constants'* on p. 28.
Let $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ} be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 14.
Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.7 *'Commitment'* on p. 17, and let $\mathsf{Note}^{\mathsf{Sprout}}$ and $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$ be as defined in § 3.2 *'Notes'* on p. 9.

A valid instance of a *JoinSplit statement*, $\pi_{\mathsf{ZKJoinSplit}}$, assures that given a *primary input*:

$$
\begin{aligned}
&(\mathsf{rt}^{\mathsf{Sprout}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]},\\
&\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]},\\
&\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]},\\
&\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} : \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} : \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\mathsf{h}_{\mathsf{Sig}} : \mathbb{B}^{[\ell_{\mathsf{hSig}}]},\\
&\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}),
\end{aligned}
$$

the prover knows an *auxiliary input*:

$$
\begin{aligned}
&(\mathsf{path}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sprout}}][\mathrm{N}^{\mathsf{old}}]},\\
&\mathsf{pos}_{1..\mathrm{N}^{\mathsf{old}}} : \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sprout}}}-1\}^{[\mathrm{N}^{\mathsf{old}}]},\\
&\mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} : \mathsf{Note}^{\mathsf{Sprout}[\mathrm{N}^{\mathsf{old}}]},\\
&\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\ell_{\mathsf{a_{sk}}}][\mathrm{N}^{\mathsf{old}}]},\\
&\mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} : \mathsf{Note}^{\mathsf{Sprout}[\mathrm{N}^{\mathsf{new}}]},\\
&\varphi : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\varphi}]},\\
&\mathsf{enforceMerklePath}_{1..\mathrm{N}^{\mathsf{old}}} : \mathbb{B}^{[\mathrm{N}^{\mathsf{old}}]}),
\end{aligned}
$$

where:

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforceMerklePath}_i = 1$: $(\mathsf{path}_i, \mathsf{pos}_i)$ is a valid *Merkle path* (see § 4.6 *'Merkle Path Validity'* on p. 22) of depth $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ from $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}^{\mathsf{old}}_i)$ to the *anchor* $\mathsf{rt}^{\mathsf{Sprout}}$.

Note: Merkle path validity covers conditions 1. (a) and 1. (d) of the NP *statement* in [BCGGMTV2014, section 4.2].

Merkle path enforcement for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\}$, if $\mathsf{v}^{\mathsf{old}}_i \neq 0$ then $\mathsf{enforceMerklePath}_i = 1$.

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

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

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

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

For details of the form and encoding of proofs, see § 5.4.8.1 'BCTV14' on p. 36.

4.12 In-band secret distribution

The secrets that need to be transmitted to a recipient of funds in order for them to later spend, are v, ρ, and rcm. A *memo field* (§ 3.2.1 *'Note Plaintexts and Memo Fields'* on p. 10) is also transmitted.

To transmit these secrets securely to a recipient ***without*** requiring an out-of-band communication channel, the *transmission key* pk_{enc} is used to encrypt them. The recipient's possession of the associated *incoming viewing key* ivk is used to reconstruct the original *note* and *memo field*.

A single *ephemeral public key* is shared between encryptions of the N^{new} *shielded outputs* in a *JoinSplit description*. All of the resulting ciphertexts are combined to form a *transmitted notes ciphertext*.

For both encryption and decryption,

- let Sym be the scheme instantiated in § 5.4.3 *'Symmetric Encryption'* on p. 32;
- let KDF^{Sprout} be the *Key Derivation Function* instantiated in § 5.4.4.2 *'Sprout Key Derivation'* on p. 33;
- let KA^{Sprout} be the *key agreement scheme* instantiated in § 5.4.4.1 *'Sprout Key Agreement'* on p. 32;
- let h_{Sig} be the value computed for this *JoinSplit description* in § 4.3 *'JoinSplit Descriptions'* on p. 19.

4.12.1 Encryption

Let KA^{Sprout} be the *key agreement scheme* instantiated in § 5.4.4.1 *'Sprout Key Agreement'* on p. 32.

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

Let $\mathbf{np}_{1..N^{new}}$ be **Sprout** *note plaintexts* defined in § 5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 37.

Then to encrypt:

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

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

Note: It is technically possible to replace C_i^{enc} for a given *note* with a random (and undecryptable) dummy ciphertext, relying instead on out-of-band transmission of the *note* to the recipient. In this case the ephemeral key **MUST** still be generated as a random *public key* (rather than a random bit sequence) to ensure indistinguishability from other *JoinSplit descriptions*. This mode of operation raises further security considerations, for example of how to validate a **Sprout** *note* received out-of-band, which are not addressed in this document.

4.12.2 Decryption

Let $ivk = (a_{pk}, sk_{enc})$ be the recipient's *incoming viewing key*, and let pk_{enc} be the corresponding *transmission key* derived from sk_{enc} as specified in § 4.2 *'Key Components'* on p. 19.

Let $\mathsf{cm}_{1..N^{new}}$ 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 $(\mathsf{epk}, \mathsf{C}_i^{\mathsf{enc}})$ as follows:

$\mathsf{let\ sharedSecret}_i = \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Agree}(\mathsf{sk}_{\mathsf{enc}}, \mathsf{epk})$

$\mathsf{let}\ \mathsf{K}_i^{\mathsf{enc}} = \mathsf{KDF}^{\mathsf{Sprout}}(i, \mathsf{h}_{\mathsf{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk}_{\mathsf{enc}})$

$\mathsf{return}\ \mathtt{DecryptNoteSprout}(\mathsf{K}_i^{\mathsf{enc}}, \mathsf{C}_i^{\mathsf{enc}}, \mathsf{cm}_i, \mathsf{a}_{\mathsf{pk}})$.

$\mathtt{DecryptNoteSprout}(\mathsf{K}_i^{\mathsf{enc}}, \mathsf{C}_i^{\mathsf{enc}}, \mathsf{cm}_i, \mathsf{a}_{\mathsf{pk}})$ is defined as follows:

$\mathsf{let}\ \mathsf{P}_i^{\mathsf{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K}_i^{\mathsf{enc}}}(\mathsf{C}_i^{\mathsf{enc}})$

$\mathsf{if}\ \mathsf{P}_i^{\mathsf{enc}} = \bot, \mathsf{return}\ \bot$

$\mathsf{extract}\ \mathbf{np}_i = (\mathsf{leadByte}_i : \mathbb{B}^{\mathbb{Y}}, \mathsf{v}_i : \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}, \rho_i : \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]}, \mathsf{rcm}_i : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}, \mathsf{memo}_i : \mathbb{B}^{\mathbb{Y}^{[512]}})$ $\mathsf{from}\ \mathsf{P}_i^{\mathsf{enc}}$

$\mathsf{if\ leadByte}_i \neq \mathsf{0x00\ or\ NoteCommitment}^{\mathsf{Sprout}}((\mathsf{a}_{\mathsf{pk}}, \mathsf{v}_i, \rho_i, \mathsf{rcm}_i)) \neq \mathsf{cm}_i, \mathsf{return}\ \bot, \mathsf{else\ return}\ \mathbf{np}_i.$

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

Notes:

- The decryption algorithm corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in [BCGGMTV2014, Figure 2].
- A *note* can change from being unspent to spent as a node's view of the *best valid block chain* is extended by new *transactions*. Also, *block chain reorganizations* can cause a node to switch to a different *best valid block chain* that does not contain the *transaction* in which a *note* was output.

See § 8.7 *'In-band secret distribution'* on p. 55 for further discussion of the security and engineering rationale behind this encryption scheme.

4.13 Block Chain Scanning

Let $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$ be as defined in § 5.3 *'Constants'* on p. 28.

Let $\mathsf{Note}^{\mathsf{Sprout}}$ be as defined in § 3.2 *'Notes'* on p. 9.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.4.1 *'Sprout Key Agreement'* on p. 32.

The following algorithm can be used, given the *block chain* and a **Sprout** *spending key* a_{sk}, to obtain each *note* sent to the corresponding *shielded payment address*, its *memo field* field, and its final status (spent or unspent).

Let $\mathsf{ivk} = (\mathsf{a}_{\mathsf{pk}} : \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]}, \mathsf{sk}_{\mathsf{enc}} : \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private})$ be the *incoming viewing key* corresponding to a_{sk}, and let $\mathsf{pk}_{\mathsf{enc}}$ be the associated *transmission key*, as specified in § 4.2 *'Key Components'* on p. 19.

$\mathsf{let\ mutable\ ReceivedSet} : \mathscr{P}(\mathsf{Note}^{\mathsf{Sprout}} \times \mathbb{B}^{\mathbb{Y}^{[512]}}) \leftarrow \{\}$

$\mathsf{let\ mutable\ SpentSet} : \mathscr{P}(\mathsf{Note}^{\mathsf{Sprout}}) \leftarrow \{\}$

$\mathsf{let\ mutable\ NullifierMap} : \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]} \to \mathsf{Note}^{\mathsf{Sprout}} \leftarrow \text{the empty mapping}$

for each *transaction* tx:

  for each *JoinSplit description* in tx:

    let $(\mathsf{epk}, \mathsf{C}_{1..\mathrm{N}^{\mathsf{new}}}^{\mathsf{enc}})$ be the *transmitted notes ciphertext* of the *JoinSplit description*

    for i in $1..\mathrm{N}^{\mathsf{new}}$:

      Attempt to decrypt the *transmitted notes ciphertext* component $(\mathsf{epk}, \mathsf{C}_i^{\mathsf{enc}})$ using ivk with the

algorithm in § 4.12.2 *'Decryption'* on p. 26. If this succeeds giving $\mathbf{np}$:

Extract $\mathbf{n}$ and memo $: \mathbb{B}^{[512]}$ from $\mathbf{np}$ (taking the a_{pk} field of the *note* to be a_{pk} from ivk).

Add $(\mathbf{n}, \mathsf{memo})$ to ReceivedSet.

Calculate the nullifier nf of $\mathbf{n}$ using a_{sk} as described in § 3.2 *'Notes'* on p. 9.

Add the mapping $\mathsf{nf} \to \mathbf{n}$ to NullifierMap.

let $\mathsf{nf}_{1..\mathrm{N}^{\mathsf{old}}}$ be the *nullifiers* of the *JoinSplit description*

for i in $1..\mathrm{N}^{\mathsf{old}}$:

if nf_i is present in NullifierMap, add NullifierMap(nf_i) to SpentSet

return (ReceivedSet, SpentSet).

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. § 8.5 *'Internal hash collision attack and fix'* on p. 54

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***.

Define $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0 .. 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ such that $\mathsf{I2BEBSP}_\ell(x)$ is the sequence of ℓ bits representing x in ***big-endian*** order.

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 ℓ is given explicitly in each box, except when it is obvious (e.g. for a single bit, or for the notation $[0]^\ell$ representing the sequence of ℓ 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:

$\mathsf{MerkleDepth}^{\mathsf{Sprout}} : \mathbb{N} := 29$

$\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}} : \mathbb{N} := 256$

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

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

$\ell_{\mathsf{value}} : \mathbb{N} := 64$

$\ell_{\mathsf{hSig}} : \mathbb{N} := 256$

$\ell_{\mathsf{PRF}}^{\mathsf{Sprout}} : \mathbb{N} := 256$

$\ell_{\mathsf{rcm}} : \mathbb{N} := 256$

$\ell_{\mathsf{Seed}} : \mathbb{N} := 256$

$\ell_{\mathsf{a_{sk}}} : \mathbb{N} := 252$

$\ell_{\varphi}^{\mathsf{Sprout}} : \mathbb{N} := 252$

$\mathsf{Uncommitted}^{\mathsf{Sprout}} : \mathbb{B}^{[\ell_{\mathsf{Merkle}}^{\mathsf{Sprout}}]} := [0]^{\ell_{\mathsf{Merkle}}^{\mathsf{Sprout}}}$

$\mathsf{MAX\_MONEY} : \mathbb{N} := 2.1 \cdot 10^{15}$ (*zatoshi*)

$\mathsf{SlowStartInterval} : \mathbb{N} := 20000$

$\mathsf{HalvingInterval} : \mathbb{N} := 840000$

$\mathsf{MaxBlockSubsidy} : \mathbb{N} := 1.25 \cdot 10^{9}$ (*zatoshi*)

$\mathsf{NumFounderAddresses} : \mathbb{N} := 48$

$\mathsf{FoundersFraction} : \mathbb{Q} := \frac{1}{5}$

$$\mathsf{PoWLimit} : \mathbb{N} := \begin{cases} 2^{243} - 1, & \text{for } \textit{Mainnet} \\ 2^{251} - 1, & \text{for } \textit{Testnet} \end{cases}$$

$\mathsf{PoWAveragingWindow} : \mathbb{N} := 17$

$\mathsf{PoWMedianBlockSpan} : \mathbb{N} := 11$

$\mathsf{PoWMaxAdjustDown} : \mathbb{Q} := \frac{32}{100}$

$\mathsf{PoWMaxAdjustUp} : \mathbb{Q} := \frac{16}{100}$

$\mathsf{PoWDampingFactor} : \mathbb{N} := 4$

$\mathsf{PoWTargetSpacing} : \mathbb{N} := 150$ (seconds).

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

5.4.1.1 SHA-256, SHA-256d, SHA256Compress, and SHA-512 Hash Functions

SHA-256 and SHA-512 are defined by [NIST2015].

Zcash uses the full SHA-256 *hash function* to instantiate $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$.

$\mathsf{SHA\text{-}256} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{B}^{\mathbb{Y}^{[32]}}$

[NIST2015] strictly speaking only specifies the application of SHA-256 to messages that are bit sequences, producing outputs ("message digests") that are also bit sequences. In practice, SHA-256 is universally implemented with a byte-sequence interface for messages and outputs, such that the ***most significant*** bit of each byte corresponds to the first bit of the associated bit sequence. (In the NIST specification "first" is conflated with "leftmost".)

SHA-256d, defined as a double application of SHA-256, is used to hash *block headers*:

$$\mathsf{SHA\text{-}256d} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{B}^{\mathbb{Y}^{[32]}}$$

Zcash also uses the SHA-256 compression function, SHA256Compress. This operates on a single 512-bit block and ***excludes*** the padding step specified in [NIST2015, section 5.1].

That is, the input to SHA256Compress is what [NIST2015, section 5.2] refers to as "the message and its padding". The Initial Hash Value is the same as for full SHA-256.

SHA256Compress is used to instantiate several *Pseudo Random Functions* and $\mathsf{MerkleCRH}^{\mathsf{Sprout}}$.

$$\mathsf{SHA256Compress} : \mathbb{B}^{[512]} \to \mathbb{B}^{[256]}$$

The ordering of bits within words in the interface to SHA256Compress is consistent with [NIST2015, section 3.1], i.e. big-endian.

Ed25519 uses SHA-512:

$$\mathsf{SHA\text{-}512} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{B}^{\mathbb{Y}^{[64]}}$$

The comment above concerning bit vs byte-sequence interfaces also applies to SHA-512.

5.4.1.2 BLAKE2b Hash Function

BLAKE2 is defined by [ANWW2013].

BLAKE2b-$\ell(p, x)$ refers to unkeyed BLAKE2b-ℓ in sequential mode, with an output digest length of $\ell/8$ bytes, 16-byte personalization string p, and input x.

BLAKE2b is used to instantiate hSigCRH, EquihashGen, and $\mathsf{KDF}^{\mathsf{Sprout}}$.

$$\mathsf{BLAKE2b\text{-}}\ell : \mathbb{B}^{\mathbb{Y}^{[16]}} \times \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{B}^{\mathbb{Y}^{[\ell/8]}}$$

Note: BLAKE2b-ℓ is not the same as BLAKE2b-512 truncated to ℓ bits, because the digest length is encoded in the parameter block.

5.4.1.3 Merkle Tree Hash Function

$\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ is used to hash *incremental Merkle tree hash values*.

$\mathsf{MerkleCRH}^{\mathsf{Sprout}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$ is defined as follows:

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

SHA256Compress is defined in § 5.4.1.1 'SHA-256, SHA-256d, SHA256Compress, *and* SHA-512 ***Hash Functions***' on p. 29.

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

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

5.4.1.4 $\mathsf{h_{Sig}}$ Hash Function

hSigCRH is used to compute the value $\mathsf{h_{Sig}}$ in § 4.3 *'JoinSplit Descriptions'* on p. 19.

$$\mathsf{hSigCRH}(\mathsf{randomSeed}, \mathsf{nf}^{\mathsf{old}}_{1..N^{\mathsf{old}}}, \mathtt{joinSplitPubKey}) := \mathsf{BLAKE2b\text{-}256}(\texttt{"ZcashComputehSig"}, \mathsf{hSigInput})$$

where

$$\mathsf{hSigInput} := \boxed{\text{256-bit randomSeed} \mid \text{256-bit } \mathsf{nf}^{\mathsf{old}}_1 \mid \ldots \mid \text{256-bit } \mathsf{nf}^{\mathsf{old}}_{N^{\mathsf{old}}} \mid \text{256-bit } \mathtt{joinSplitPubKey}}.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 *'BLAKE2b Hash Function'* on p. 30.

Security requirement: $\mathsf{BLAKE2b\text{-}256}(\texttt{"ZcashComputehSig"}, x)$ must be *collision-resistant* on x.

5.4.1.5 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 § 7.4.1 *'Equihash'* on p. 47.

$$\mathsf{Let\ powtag} := \boxed{\text{64-bit } \texttt{"ZcashPoW"} \mid \text{32-bit } n \mid \text{32-bit } k}.$$

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

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

$m = \mathsf{floor}\left(\frac{512}{n}\right)$;

$h = (i - 1 \bmod m) \cdot n$;

$T = \mathsf{BLAKE2b\text{-}}(n \cdot m)(\mathsf{powtag},\ S \mid\mid \mathsf{powcount}(\mathsf{floor}(\frac{i-1}{m})))$.

Indices of bits in T are 1-based.

$\mathsf{BLAKE2b\text{-}}\ell(p, x)$ is defined in § 5.4.1.2 *'BLAKE2b Hash Function'* on p. 30.

Security requirement: $\mathsf{BLAKE2b\text{-}}\ell(\mathsf{powtag}, x)$ must generate output that is sufficiently unpredictable to avoid short-cuts to the *Equihash* solution process. It would suffice to model it as a *random oracle*.

Note: When EquihashGen is evaluated for sequential indices, as in the *Equihash* solving process (§ 7.4.1 *'Equihash'* on p. 47), the number of calls to BLAKE2b can be reduced by a factor of $\mathsf{floor}\left(\frac{512}{n}\right)$ in the best case (which is a factor of 2 for $n = 200$).

5.4.2 Pseudo Random Functions

Let SHA256Compress be as given in § 5.4.1.1 *'SHA-256, SHA-256d, SHA256Compress, and SHA-512 Hash Functions'* on p. 29.

The *Pseudo Random Functions* $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ} from § 4.1.2 *'Pseudo Random Functions'* on p. 14, are all instantiated using SHA256Compress:

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

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

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

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

Security requirements:

- SHA256Compress must be *collision-resistant*.
- SHA256Compress 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 separate distinct uses of SHA256Compress, ensuring that the functions are independent. As well as the inputs shown here, bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function; see § 5.4.6.1 ***'Sprout Note Commitments'*** on p. 34.

(The specific bit patterns chosen here were motivated by the possibility of future extensions that might have increased $\mathrm{N}^{\mathsf{old}}$ and/or $\mathrm{N}^{\mathsf{new}}$ to 3, or added an additional bit to $\mathsf{a_{sk}}$ to encode a new key type, or that would have required an additional *PRF*.)

5.4.3 Symmetric Encryption

Let $\mathsf{Sym}.\mathbf{K} := \mathbb{B}^{[256]}$, $\mathsf{Sym}.\mathbf{P} := \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}}$, and $\mathsf{Sym}.\mathbf{C} := \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}}$.

Let the *authenticated one-time symmetric encryption scheme* $\mathsf{Sym.Encrypt_K(P)}$ 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 $\mathsf{Sym.Decrypt_K(C)}$ 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 has 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.

5.4.4 Key Agreement And Derivation

5.4.4.1 Sprout Key Agreement

$\mathsf{KA}^{\mathsf{Sprout}}$ is a *key agreement scheme* as specified in § 4.1.4 ***'Key Agreement'*** on p. 15.

It is instantiated as Curve25519 key agreement, described in [Bernstein2006], as follows.

Let $\mathsf{KA}^{\mathsf{Sprout}}$.Public and $\mathsf{KA}^{\mathsf{Sprout}}$.SharedSecret be the type of Curve25519 *public keys* (i.e. $\mathbb{B}^{\mathbb{Y}^{[32]}}$), and let $\mathsf{KA}^{\mathsf{Sprout}}$.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 [Bernstein2006, section 2].

Let $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Base} := \underline{9}$ 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 [Bernstein2006, 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 $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{FormatPrivate}(x) := \mathsf{clamp}_{\mathsf{Curve25519}}(x)$.

Define $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{DerivePublic}(n, q) := \mathsf{Curve25519}(n, q)$.

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

5.4.4.2 Sprout Key Derivation

$\mathsf{KDF}^{\mathsf{Sprout}}$ is a *Key Derivation Function* as specified in § 4.1.5 *'Key Derivation'* on p. 15.

It is instantiated using BLAKE2b-256 as follows:

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

where:

$\mathsf{kdftag} :=$

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

$\mathsf{kdfinput} :=$

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

.

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 *'BLAKE2b Hash Function'* on p. 30.

5.4.5 Ed25519

Ed25519 is a *signature scheme* as specified in § 4.1.6 *'Signature'* on p. 16. It is used to instantiate JoinSplitSig as described in § 4.8 *'Non-malleability'* on p. 23.

Let $\mathsf{ExcludedPointEncodings} : \mathscr{P}(\mathbb{Y}^{[32]}) = \{$

[0x00, 0x00],
[0x01, 0x00],
[0x26, 0xe8, 0x95, 0x8f, 0xc2, 0xb2, 0x27, 0xb0, 0x45, 0xc3, 0xf4, 0x89, 0xf2, 0xef, 0x98, 0xf0, 0xd5, 0xdf, 0xac, 0x05, 0xd3, 0xc6, 0x33, 0x39, 0xb1, 0x38, 0x02, 0x88, 0x6d, 0x53, 0xfc, 0x05],
[0xc7, 0x17, 0x6a, 0x70, 0x3d, 0x4d, 0xd8, 0x4f, 0xba, 0x3c, 0x0b, 0x76, 0x0d, 0x10, 0x67, 0x0f, 0x2a, 0x20, 0x53, 0xfa, 0x2c, 0x39, 0xcc, 0xc6, 0x4e, 0xc7, 0xfd, 0x77, 0x92, 0xac, 0x03, 0x7a],
[0x13, 0xe8, 0x95, 0x8f, 0xc2, 0xb2, 0x27, 0xb0, 0x45, 0xc3, 0xf4, 0x89, 0xf2, 0xef, 0x98, 0xf0, 0xd5, 0xdf, 0xac, 0x05, 0xd3, 0xc6, 0x33, 0x39, 0xb1, 0x38, 0x02, 0x88, 0x6d, 0x53, 0xfc, 0x85],
[0xb4, 0x17, 0x6a, 0x70, 0x3d, 0x4d, 0xd8, 0x4f, 0xba, 0x3c, 0x0b, 0x76, 0x0d, 0x10, 0x67, 0x0f, 0x2a, 0x20, 0x53, 0xfa, 0x2c, 0x39, 0xcc, 0xc6, 0x4e, 0xc7, 0xfd, 0x77, 0x92, 0xac, 0x03, 0xfa],
[0xec, 0xff, 0x7f],
[0xed, 0xff, 0x7f],
[0xee, 0xff, 0x7f],
[0xd9, 0xff],
[0xda, 0xff]

$\}$.

Let $p = 2^{255} - 19$.

Let $a = -1$.

Let $d = -121665/121666 \pmod p$.

Let $\ell = 2^{252} + 27742317777372353535851937790883648493$ (the order of the Ed25519 curve's prime-order subgroup).

Let B be the base point given in [BDLSY2012].

Define I2LEOSP, LEOS2BSP, and LEBS2IP as in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 28.

Define $\mathsf{reprBytes}_{\mathsf{Ed25519}} : \mathsf{Ed25519} \to \mathbb{Y}^{[32]}$ such that $\mathsf{reprBytes}_{\mathsf{Ed25519}}(x, y) = \mathsf{I2LEOSP}_{256}\big(y + 2^{255} \cdot \tilde{x}\big)$, where $\tilde{x} = x \bmod 2$.

Define $\mathsf{abstBytes}_{\mathsf{Ed25519}} : \mathbb{Y}^{[32]} \to \mathsf{Ed25519} \cup \{\bot\}$ such that $\mathsf{abstBytes}_{\mathsf{Ed25519}}(\underline{P})$ is computed as follows:

- let $y\star : \mathbb{B}^{[255]}$ be the first 255 bits of $\mathsf{LEOS2BSP}_{256}(\underline{P})$ and let $\tilde{x} : \mathbb{B}$ be the last bit.
- let $y : \mathbb{F}_p = \mathsf{LEBS2IP}_{255}(y\star) \pmod p$.
- let $x = \sqrt[?]{\dfrac{1 - y^2}{a - d \cdot y^2}}$. (The denominator $a - d \cdot y^2$ cannot be zero, since $\frac{a}{d}$ is not square in $\mathbb{F}_p$.)
- if $x = \bot$, return $\bot$.
- if $x \bmod 2 = \tilde{x}$ then return (x, y) else return $(p - x, y)$.

Note: This definition of point decoding differs from that of [RFC-8032, section 5.1.3, as corrected by the errata]. In the latter there is an additional step "`If x = 0, and x_0 = 1, decoding fails.`", which rejects the encodings {
[0x01, 0x00, 0x80],
[0xee, 0xff],
[0xec, 0xff]
}.
In this specification, the first two of these are accepted as encodings of $(0, 1)$, and the third is accepted as an encoding of $(0, -1)$.

Ed25519 is defined as in [BDLSY2012], using SHA-512 as the internal *hash function*, with the additional requirements below. A valid Ed25519 *validating key* is defined as a sequence of 32 bytes encoding a point on the Ed25519 curve. All conversions between Ed25519 points, byte sequences, and integers used in this section are as specified in [BDLSY2012].

The requirements on a signature $(\underline{R}, \underline{S})$ with *validating key* $\underline{A}$ on a message M are:

- $\underline{S}$ **MUST** represent an integer less than ℓ.
- $\underline{R}$ and $\underline{A}$ **MUST** be encodings of points R and A respectively on the Ed25519 curve;
- $\underline{R}$ **MUST NOT** be in ExcludedPointEncodings;
- The validation equation **MUST** be equivalent to $[S]\,B = R + [c]\,A$.

where c is computed as the integer corresponding to SHA-512$(\underline{R}\,||\,\underline{A}\,||\,M)$ as specified in [BDLSY2012].

If these requirements are not met or the validation equation does not hold, then the signature is considered invalid.

The encoding of an Ed25519 signature is:

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

where $\underline{R}$ and $\underline{S}$ are as defined in [BDLSY2012].

Notes:

- It is *not* required that the integer encoding of the y-coordinate of the points represented by $\underline{R}$ or $\underline{A}$ are less than $2^{255} - 19$.
- It is *not* required that $\underline{A} \notin$ ExcludedPointEncodings.

Non-normative note: The exclusion of ExcludedPointEncodings from $\underline{R}$ is due to a quirk of version 1.0.15 of the libsodium library [libsodium] which was initially used to implement Ed25519 signature validation in zcashd. (The `ED25519_COMPAT` compile-time option was not set.) The intent was to exclude points of order less than ℓ; however, not all such points were covered. It is possible, with due attention to detail, to reproduce this quirk without using libsodium v1.0.15.

5.4.6 Commitment schemes

5.4.6.1 Sprout Note Commitments

The commitment scheme NoteCommit$^{\mathsf{Sprout}}$ specified in § 4.1.7 *'Commitment'* on p. 17 is instantiated using SHA-256 as follows:

$\mathsf{NoteCommit}^{\mathsf{Sprout}}_{\mathsf{rcm}}(\mathsf{a_{pk}}, \mathsf{v}, \rho) := \mathsf{SHA\text{-}256}\Big($

1	0	1	1	0	0	0	0	256-bit $\mathsf{a_{pk}}$	64-bit v	256-bit ρ	256-bit rcm

$\Big)$

$\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$.

Note: The leading byte of the SHA-256 input is `0xB0`.

Security requirements:

- SHA256Compress must be *collision-resistant*.
- SHA256Compress must be a *PRF* when keyed by the bits corresponding to the position of rcm in the second block of SHA-256 input, with input to the *PRF* in the remaining bits of the block and the chaining variable.

5.4.7 Represented Groups and Pairings

5.4.7.1 BN-254

The *represented pairing* BN-254 is defined in this section.

Let $q_{\mathbb{G}} := 21888242871839275222246405745257275088696311157297823662689037894645226208583$.

Let $r_{\mathbb{G}} := 21888242871839275222246405745257275088548364400416034343698204186575808495617$.

Let $b_{\mathbb{G}} := 3$.

($q_{\mathbb{G}}$ and $r_{\mathbb{G}}$ are prime.)

Let $\mathbb{G}_1^{(r)}$ be the group (of order $r_{\mathbb{G}}$) of rational points on a Barreto–Naehrig ([BN2005]) curve $E_{\mathbb{G}_1}$ over $\mathbb{F}_{q_{\mathbb{G}}}$ with equation $y^2 = x^3 + b_{\mathbb{G}}$. This curve has embedding degree 12 with respect to $r_{\mathbb{G}}$.

Let $\mathbb{G}_2^{(r)}$ be the subgroup of order $r_{\mathbb{G}}$ in the sextic twist $E_{\mathbb{G}_2}$ of $E_{\mathbb{G}_1}$ over $\mathbb{F}_{q_{\mathbb{G}}{}^2}$ with equation $y^2 = x^3 + \frac{b_{\mathbb{G}}}{\xi}$, where $\xi : \mathbb{F}_{q_{\mathbb{G}}{}^2}$.

We represent elements of $\mathbb{F}_{q_{\mathbb{G}}{}^2}$ as polynomials $a_1 \cdot t + a_0 : \mathbb{F}_{q_{\mathbb{G}}}[t]$, modulo the irreducible polynomial $t^2 + 1$; in this representation, ξ is given by $t + 9$.

Let $\mathbb{G}_T^{(r)}$ be the subgroup of $r_{\mathbb{G}}{}^{\text{th}}$ roots of unity in $\mathbb{F}^*_{q_{\mathbb{G}}{}^{12}}$, with multiplicative identity $\mathbf{1}_{\mathbb{G}}$.

Let $\hat{e}_{\mathbb{G}}$ be the optimal ate pairing (see [Vercauter2009] and [AKLGL2010, section 2]) of type $\mathbb{G}_1^{(r)} \times \mathbb{G}_2^{(r)} \to \mathbb{G}_T^{(r)}$.

For $i : \{1 .. 2\}$, let $\mathcal{O}_{\mathbb{G}_i}$ be the point at infinity (which is the additive identity) in $\mathbb{G}_i^{(r)}$, and let $\mathbb{G}_i^{(r)*} := \mathbb{G}_i^{(r)} \setminus \{\mathcal{O}_{\mathbb{G}_i}\}$.

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

Let $\mathcal{P}_{\mathbb{G}_2} : \mathbb{G}_2^{(r)*} := (11559732032986387107991004021392285783925812861821192530917403151452391805634 \cdot t +$
$10857046999023057135944570762232829481370756359578518086990519993285655852781,$
$4082367875863433681332203403145435568316851327593401208105741076214120093531 \cdot t +$
$8495653923123431417604973247489272438418190587263600148770280649306958101930)$.

$\mathcal{P}_{\mathbb{G}_1}$ and $\mathcal{P}_{\mathbb{G}_2}$ are generators of $\mathbb{G}_1^{(r)}$ and $\mathbb{G}_2^{(r)}$ respectively.

Define I2BEBSP $: (\ell : \mathbb{N}) \times \{0 .. 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 28.

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

- 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 I2BEBSP$_{256}(x)$

.

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

- Define $\mathsf{FE2IP} : \mathbb{F}_{q_\mathbb{G}}[t]/(t^2 + 1) \to \{0 \,..\, {q_\mathbb{G}}^2 - 1\}$ such that $\mathsf{FE2IP}(a_{w,1} \cdot t + a_{w,0}) = a_{w,1} \cdot q + a_{w,0}$.
- 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 $\mathsf{I2BEBSP}_{512}(x)$

.

Non-normative notes:

- Only the $r_\mathbb{G}$-order subgroups $\mathbb{G}_{2,T}^{(r)}$ are used in the protocol, not their containing groups $\mathbb{G}_{2,T}$. Points in $\mathbb{G}_2^{(r)*}$ are ***always*** checked to be of order $r_\mathbb{G}$ when decoding from external representation. (The group of rational points $\mathbb{G}_1$ on $E_{\mathbb{G}_1}/\mathbb{F}_{q_\mathbb{G}}$ is of order $r_\mathbb{G}$ so no subgroup checks are needed in that case, and elements of $\mathbb{G}_T^{(r)}$ are never represented externally.) The (r) superscripts on $\mathbb{G}_{1,2,T}^{(r)}$ are used for consistency with notation elsewhere in this specification.
- The points at infinity $\mathcal{O}_{\mathbb{G}_{1,2}}$ never occur in proofs and have no defined encodings in this protocol.
- A rational point $P \neq \mathcal{O}_{\mathbb{G}_2}$ on the curve $E_{\mathbb{G}_2}$ can be verified to be of order $r_\mathbb{G}$, and therefore in $\mathbb{G}_2^{(r)*}$, by checking that $r_\mathbb{G} \cdot P = \mathcal{O}_{\mathbb{G}_2}$.
- The use of big-endian order by I2BEBSP is different from the encoding of most other integers in this protocol. The encodings for $\mathbb{G}_{1,2}^{(r)*}$ 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 in $\mathbb{G}_1^{(r)*}$, and the SORT compressed form (i.e. EC2OSP-XS) for points in $\mathbb{G}_2^{(r)*}$.
- Testing $y > y'$ for the compression of $\mathbb{G}_2^{(r)*}$ 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 $\mathbb{G}_1^{(r)*}$, and [IEEE2004, Appendix A.12.11] for $\mathbb{G}_2^{(r)*}$.

When computing square roots in $\mathbb{F}_{q_\mathbb{G}}$ or $\mathbb{F}_{{q_\mathbb{G}}^2}$ 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.

5.4.8 Zero-Knowledge Proving Systems

5.4.8.1 BCTV14

Zcash uses *zk-SNARKs* generated by a fork of *libsnark* [Zcash-libsnark] with the BCTV14 *proving system* described in [BCTV2014a], which is a modification of the systems in [PHGR2013] and [BCGTV2013].

A BCTV14 proof comprises $(\pi_A : \mathbb{G}_1^{(r)*}, \pi'_A : \mathbb{G}_1^{(r)*}, \pi_B : \mathbb{G}_2^{(r)*}, \pi'_B : \mathbb{G}_1^{(r)*}, \pi_C : \mathbb{G}_1^{(r)*}, \pi'_C : \mathbb{G}_1^{(r)*}, \pi_K : \mathbb{G}_1^{(r)*}, \pi_H : \mathbb{G}_1^{(r)*})$. It is computed as described in [BCTV2014a, Appendix B], using the pairing parameters specified in § 5.4.7.1 'BN-254' on p. 35.

Note: Many details of the *proving system* are beyond the scope of this protocol document. For example, the *quadratic constraint program* verifying the *JoinSplit statement*, or its translation to a *Quadratic Arithmetic Program* [BCTV2014a, section 2.3], are not specified in this document. In 2015, Bryan Parno found a bug in this translation, which is corrected by the *libsnark* implementation[5] [WCBTV2015] [Parno2015] [BCTV2014a, Remark 2.5]. In practice

[5] Confusingly, the bug found by Bryan Parno was fixed in *libsnark* in 2015, but that fix was incompletely described in the May 2015 update [BCTV2014a-old, Theorem 2.4]. It is described completely in [BCTV2014a, Theorem 2.4] and in [Gabizon2019].

it will be necessary to use the specific proving and verifying keys that were generated for the **Zcash** production *block chain*, given in § 5.7 'BCTV14 *zk-SNARK Parameters*' on p. 40, together with a *proving system* implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

Vulnerability disclosure: BCTV14 is subject to a security vulnerability, separate from [Parno2015], that could allow violation of Knowledge Soundness (and Soundness) [CVE-2019-7167] [SWB2019] [Gabizon2019]. The consequence for **Zcash** is that balance violation could have occurred before activation of the **Sapling** *network upgrade*, although there is no evidence of this having happened. Use of the vulnerability to produce false proofs is believed to have been fully mitigated by activation of **Sapling**. The use of BCTV14 in **Zcash** is now limited to verifying proofs that were made prior to the **Sapling** *network upgrade*.

Due to this issue, new forks of **Zcash MUST NOT** use BCTV14, and any other users of the **Zcash** protocol **SHOULD** discontinue use of BCTV14 as soon as possible.

The vulnerability does not affect the Zero Knowledge property of the scheme (as described in any version of [BCTV2014a] or as implemented in any version of *libsnark* that has been used in **Zcash**), even under subversion of the parameter generation [BGG2017, Theorem 4.10].

Encoding of BCTV14 Proofs

A BCTV14 proof is encoded by concatenating the encodings of its elements; for the BN-254 pairing this is:

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 [BCTV2014a, 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 $\{0 \,..\, q_{\mathbb{S}} - 1\}$ or (in the case of π_B) $\{0 \,..\, q_{\mathbb{S}}^2 - 1\}$;
- the encoding represents a point in $\mathbb{G}_1^{(r)*}$ or (in the case of π_B) $\mathbb{G}_2^{(r)*}$, including checking that it is of order $r_{\mathbb{G}}$ in the latter case.

5.5 Encodings of Note Plaintexts and Memo Fields

As explained in § 3.2.1 *'Note Plaintexts and Memo Fields'* on p. 10, transmitted *notes* are stored on the *block chain* in encrypted form.

The *note plaintexts* in a *JoinSplit description* are encrypted to the respective *transmission keys* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$. Each *note plaintext* (denoted $\mathbf{np}$) consists of:

$$(\mathsf{leadByte} : \mathbb{B}^{\mathbb{Y}},\ \mathsf{v} : \{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\},\ \rho : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]},\ \mathsf{rcm} : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output},\ \mathsf{memo} : \mathbb{B}^{\mathbb{Y}^{[512]}})$$

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 as one of:

- a UTF-8 human-readable string [Unicode], padded by appending zero bytes; or
- the byte 0xF6 followed by 511 0x00 bytes, indicating "no memo"; or
- any other sequence of 512 bytes starting with a byte value 0xF5 or greater (which is therefore not a valid UTF-8 string), as specified in [ZIP-302].

When the first byte value is less than 0xF5, 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 other cases, the contents of the *memo field* **SHOULD NOT** be displayed unless otherwise specified by [ZIP-302].

Other fields are as defined in § 3.2 *'Notes'* on p. 9.

The encoding of a **Sprout** *note plaintext* consists of:

8-bit leadByte	64-bit v	256-bit ρ	256-bit rcm	memo (512 bytes)

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

5.6 Encodings of Addresses and Keys

This section describes how **Zcash** encodes *shielded payment addresses*, *incoming 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].

SHA256Compress outputs are always represented as sequences of 32 bytes.

5.6.1 Transparent Encodings

5.6.1.1 Transparent Addresses

Transparent addresses are either P2SH (Pay to Script Hash) addresses [BIP-13] or P2PKH (Pay to Public Key Hash) addresses [Bitcoin-P2PKH].

The *raw encoding* of a P2SH address consists of:

8-bit 0x1C	8-bit 0xBD	160-bit script hash

- Two bytes [0x1C, 0xBD], indicating this version of the *raw encoding* of a P2SH address on *Mainnet*. (Addresses on *Testnet* use [0x1C, 0xBA] instead.)
- 20 bytes specifying a script hash [Bitcoin-P2SH].

The *raw encoding* of a P2PKH address consists of:

8-bit 0x1C	8-bit 0xB8	160-bit *validating key* hash

- Two bytes [0x1C, 0xB8], indicating this version of the *raw encoding* of a P2PKH address on *Mainnet*. (Addresses on *Testnet* use [0x1D, 0x25] instead.)
- 20 bytes specifying a *validating key* hash, which is a RIPEMD-160 hash [RIPEMD160] of a SHA-256 hash [NIST2015] of a compressed ECDSA key encoding.

Notes:

- In **Bitcoin** a single byte is used for the version field identifying the address type. In **Zcash** two bytes are used. For addresses on *Mainnet*, this and the encoded length cause the first two characters of the Base58Check encoding to be fixed as **"t3"** for P2SH addresses, and as **"t1"** for P2PKH addresses. (This does ***not*** imply that a *transparent* **Zcash** address can be parsed identically to a **Bitcoin** address just by removing the **"t"**.)
- **Zcash** does not yet support Hierarchical Deterministic Wallet addresses [BIP-32].

5.6.1.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [Bitcoin-Base58], for both *Mainnet* and *Testnet*.

5.6.2 Sprout Encodings

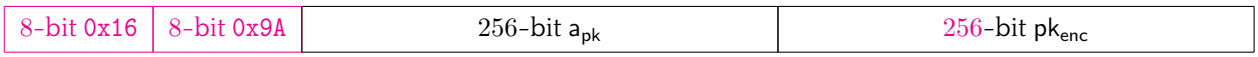
5.6.2.1 Sprout Payment Addresses

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.4.1 ***'Sprout Key Agreement'*** on p. 32.

A **Sprout** *shielded payment address* consists of $\mathsf{a_{pk}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}$ and $\mathsf{pk_{enc}} : \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$.

$\mathsf{a_{pk}}$ is a SHA256Compress output. $\mathsf{pk_{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ key, for use with the encryption scheme defined in § 4.12 ***'In-band secret distribution'*** on p. 26. These components are derived from a *spending key* as described in § 4.2 ***'Key Components'*** on p. 19.

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



- Two bytes [0x16, 0x9A], indicating this version of the *raw encoding* of a **Sprout** *shielded payment address* on *Mainnet*. (Addresses on *Testnet* use [0x16, 0xB6] instead.)
- 32 bytes specifying $\mathsf{a_{pk}}$.
- 32 bytes specifying $\mathsf{pk_{enc}}$, using the normal encoding of a Curve25519 *public key* [Bernstein2006].

Note: For addresses on *Mainnet*, the lead bytes and encoded length cause the first two characters of the Base58Check encoding to be fixed as **"zc"**. For *Testnet*, the first two characters are fixed as **"zt"**.

5.6.2.2 Sprout Incoming Viewing Keys

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.4.1 ***'Sprout Key Agreement'*** on p. 32.

An *incoming viewing key* consists of $\mathsf{a_{pk}} : \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}$ and $\mathsf{sk_{enc}} : \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$.

$\mathsf{a_{pk}}$ is a SHA256Compress output. $\mathsf{sk_{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$ key, for use with the encryption scheme defined in § 4.12 ***'In-band secret distribution'*** on p. 26. These components are derived from a *spending key* as described in § 4.2 ***'Key Components'*** on p. 19.

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

8-bit 0xA8	8-bit 0xAB	8-bit 0xD3	256-bit a_{pk}	256-bit sk_{enc}

- Three bytes [0xA8, 0xAB, 0xD3], indicating this version of the *raw encoding* of a **Zcash** *incoming viewing key* on *Mainnet*. (Addresses on *Testnet* use [0xA8, 0xAC, 0x0C] instead.)
- 32 bytes specifying a_{pk}.
- 32 bytes specifying sk_{enc}, using the normal encoding of a Curve25519 *private key* [Bernstein2006].

sk_{enc} **MUST** be "clamped" using KA^{Sprout}.FormatPrivate as specified in § 4.2 *'Key Components'* on p. 19. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $sk_{enc} \neq KA^{Sprout}.FormatPrivate(sk_{enc})$.

KA^{Sprout}.FormatPrivate is defined in § 5.4.4.1 *'Sprout Key Agreement'* on p. 32.

Note: For addresses on *Mainnet*, the lead bytes and encoded length cause the first four characters of the Base58Check encoding to be fixed as **"ZiVK"**. For *Testnet*, the first four characters are fixed as **"ZiVt"**.

5.6.2.3 Sprout Spending Keys

A **Sprout** *spending key* consists of a_{sk}, which is a sequence of 252 bits (see § 4.2 *'Key Components'* on p. 19).

The *raw encoding* of a **Sprout** *spending key* consists of:

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

- Two bytes [0xAB, 0x36], indicating this version of the *raw encoding* of a **Zcash** *spending key* on *Mainnet*. (Addresses on *Testnet* use [0xAC, 0x08] instead.)
- 32 bytes: 4 zero padding bits and 252 bits specifying a_{sk}.

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

Notes:

- If an implementation represents a_{sk} internally as a sequence of 32 bytes with the 4 bits of zero padding intact, it will be in the correct form for use as an input to PRF^{addr}, $PRF^{nfSprout}$, and PRF^{pk} without need for bit-shifting. Future key representations may make use of these padding bits.
- For addresses on *Mainnet*, the lead bytes and encoded length cause the first two characters of the Base58Check encoding to be fixed as **"SK"**. For *Testnet*, the first two characters are fixed as **"ST"**.

5.7 BCTV14 zk-SNARK Parameters

The SHA-256 hashes of the *proving key* and *verifying key* for the **Sprout** *JoinSplit circuit*, encoded in *libsnark* format, are:

```
8bc20a7f013b2b58970cddd2e7ea028975c88ae7ceb9259a5344a16bc2c0eef7 sprout-proving.key
4bd498dae0aacfd8e98dc306338d017d9c08dd0918ead18172bd0aec2fc5df82 sprout-verifying.key
```

These parameters were obtained by a multi-party computation described in [BGG-mpc] and [BGG2017]. Due to the security vulnerability described in § 5.4.8.1 'BCTV14' on p. 36, it is not recommended to use these parameters in new protocols, and it is recommended to stop using them in protocols other than **Zcash** where they are currently used.

6 Network Upgrades

Zcash launched with a protocol revision that we call **Sprout**. A first *network upgrade*, called **Overwinter**, activated on *Mainnet* on 26 June, 2018 at *block height* 347500 [Swihart2018] [ZIP-201]. A second upgrade, called **Sapling**, activated on *Mainnet* on 28 October, 2018 at *block height* 419200 [Hamdon2018] [ZIP-205]. A third upgrade, called **Blossom**, activated on *Mainnet* on 11 December, 2019 at *block height* 653600 [Zcash-Blossom] [ZIP-206]. A fourth upgrade, called **Heartwood**, activated on *Mainnet* on 16 July, 2020 at *block height* 903000 [Zcash-Heartwd] [ZIP-250]. A fifth upgrade, called **Canopy**, activated on *Mainnet* on 18 November, 2020 at *block height* 1046400 (coinciding with the first *block subsidy halving*) [Zcash-Canopy] [ZIP-251].

This draft specification describes a set of changes codenamed **NU5**, which are proposed to activate in a future *network upgrade*.

This section summarizes the strategy for upgrading from **Sprout** to subsequent versions of the protocol (**Overwinter**, **Sapling**, **Blossom**, **Heartwood**, and **Canopy**), and for future upgrades.

The *network upgrade* mechanism is described in [ZIP-200].

Each *network upgrade* is introduced as a "*bilateral consensus rule change*". In this kind of upgrade,

- there is an *activation block height* at which the *consensus rule change* takes effect;
- *blocks* and *transactions* that are valid according to the post-upgrade rules are not valid before the upgrade *block height*;
- *blocks* and *transactions* that are valid according to the pre-upgrade rules are no longer valid at or after the *activation block height*.

Full support for each *network upgrade* is indicated by a minimum version of the peer-to-peer protocol. At the planned *activation block height*, nodes that support a given upgrade will disconnect from (and will not reconnect to) nodes with a protocol version lower than this minimum.

This ensures that upgrade-supporting nodes transition cleanly from the old protocol to the new protocol. Nodes that do not support the upgrade will find themselves on a network that uses the old protocol and is fully partitioned from the upgrade-supporting network. This allows us to specify arbitrary protocol changes that take effect at a given *block height*.

Note, however, that a *block chain reorganization* across the upgrade *activation block height* is possible. In the case of such a reorganization, *blocks* at a height before the *activation block height* will still be created and validated according to the pre-upgrade rules, and upgrade-supporting nodes **MUST** allow for this.

7 Consensus Changes from Bitcoin

7.1 Transaction Encoding and Consensus

The **Zcash** *transaction* format up to and including *transaction version* 4 is as follows (this should be read in the context of consensus rules later in the section):

Version*	Bytes	Name	Data Type	Description
1 .. 4	4	`header`	`uint32`	Contains: · `fOverwintered` flag (bit 31) · `version` (bits 30 .. 0) – *transaction version*.
1 .. 4	*Varies*	`tx_in_count`	`compactSize`	Number of *transparent* inputs.
1 .. 4	*Varies*	`tx_in`	`tx_in`	*Transparent* inputs, encoded as in **Bitcoin**.
1 .. 4	*Varies*	`tx_out_count`	`compactSize`	Number of *transparent* outputs.
1 .. 4	*Varies*	`tx_out`	`tx_out`	*Transparent* outputs, encoded as in **Bitcoin**.
1 .. 4	4	`lock_time`	`uint32`	Unix-epoch UTC time or *block height*, encoded as in **Bitcoin**.
2 .. 4	*Varies*	`nJoinSplit`	`compactSize`	The number of *JoinSplit descriptions* in `vJoinSplit`.
2 .. 3	1802· `nJoinSplit`	`vJoinSplit`	`JSDescriptionBCTV14 [nJoinSplit]`	A sequence of *JoinSplit descriptions* using BCTV14 proofs, encoded per § 7.2 ***'JoinSplit Description Encoding and Consensus'*** on p. 44.
2 .. 4 †	32	`joinSplitPubKey`	`byte[32]`	An encoding of a JoinSplitSig public *validating key*.
2 .. 4 †	64	`joinSplitSig`	`byte[64]`	A signature on a prefix of the *transaction* encoding, validated using `joinSplitPubKey` as specified in § 4.8 ***'Non-malleability'*** on p. 23.

* Version constraints apply to the effectiveVersion, which is equal to min(2, `version`) when `fOverwintered` = 0 and to `version` otherwise.

† The `joinSplitPubKey` and `joinSplitSig` fields are present if and only if effectiveVersion ≥ 2 and `nJoinSplit` > 0.

Consensus rules:

- The *transaction version number* **MUST** be greater than or equal to 1.
- The `fOverwintered` flag **MUST NOT** be set in the protocol version described by this document.
- The encoded size of the *transaction* **MUST** be less than or equal to 100000 bytes.
- If effectiveVersion = 1 or `nJoinSplit` = 0, then both `tx_in_count` and `tx_out_count` **MUST** be nonzero.
- A *transaction* with one or more *transparent* inputs from *coinbase transactions* **MUST** have no *transparent* outputs (i.e. `tx_out_count` **MUST** be 0). Inputs from *coinbase transactions* include *Founders' Reward* outputs.
- If effectiveVersion ≥ 2 and `nJoinSplit` > 0, then:
  - `joinSplitPubKey` **MUST** be a valid encoding (see § 5.4.5 **'Ed25519'** on p. 33) of an Ed25519 *validating key*.
  - `joinSplitSig` **MUST** represent a valid signature under `joinSplitPubKey` of dataToBeSigned, as defined in § 4.8 ***'Non-malleability'*** on p. 23.
- The total value in *zatoshi* of *transparent outputs* from a *coinbase transaction* **MUST NOT** be greater than the value in *zatoshi* of *miner subsidy* plus the *transaction fees* paid by *transactions* in this *block*.
- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions*.

- A *coinbase transaction* for a *block* at *block height* greater than 0 **MUST** have a script that, as its first item, encodes the *block height* as follows. Let heightBytes be the signed little-endian representation of the number, using the minimum number of bytes such that the most significant byte is $< 0x80$. Then the encoding is the length of heightBytes encoded as one byte, followed by heightBytes itself. This matches the encoding used by **Bitcoin** in the implementation of [BIP-34] (but the description here is to be considered normative).
- A *transaction* **MUST NOT** spend a *transparent* output of a *coinbase transaction* from a *block* less than 100 *blocks* prior to the spend. Note that *transparent* outputs of *coinbase transactions* include *Founders' Reward* outputs .
- A *transaction* **MUST NOT** spend an output of the *genesis block coinbase transaction*. (There is one such zero-valued output, on each of *Testnet* and *Mainnet*.)
- TODO: Other rules inherited from **Bitcoin**.

Consensus rules associated with each *JoinSplit description* (§ 7.2 *'JoinSplit Description Encoding and Consensus'* on p. 44) **MUST** also be followed.

Notes:

- Previous versions of this specification defined what is now the `header` field as a signed `int32` field which was required to be positive. The consensus rule that the `fOverwintered` flag **MUST NOT** be set before **Overwinter** has activated, has the same effect. (**Overwinter** is an upgrade of the **Zcash** protocol, not specified in this document.)
- The semantics of *transactions* with *transaction version number* not equal to 1, 2, is not currently defined.
- The exclusion of *transactions* with *transaction version number* ***greater than*** 2 is not a consensus rule. Such *transactions* may exist in the *block chain* and **MUST** be treated identically to version 2 *transactions*.
- The *transaction version number* `0x7FFFFFFF`, and the *version group ID* `0xFFFFFFFF`, are reserved for use in experimental extensions to *transaction* format or semantics on private testnets. They **MUST NOT** be used on the **Zcash** *Mainnet* or *Testnet*.
- Note that a future upgrade might use ***any*** *transaction version number*. It is likely that an upgrade that changes the *transaction version number* will also change the *transaction* format, and software that parses *transactions* **SHOULD** take this into account.
- 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 currently support BIP 68.

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

- *Transaction version* 0 is not supported.
- A version 1 *transaction* is equivalent to a version 2 *transaction* with `nJoinSplit` $= 0$.
- The `nJoinSplit`, `vJoinSplit`, `joinSplitPubKey`, and `joinSplitSig` fields have been added.
- In **Zcash** it is permitted for a *transaction* to have no *transparent* inputs provided that `nJoinSplit` > 0.
- A consensus rule limiting *transaction* size has been added. In **Bitcoin** there is a corresponding standard rule but no consensus rule.

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

7.2 JoinSplit Description Encoding and Consensus

An abstract *JoinSplit description*, as described in § 3.5 *'JoinSplit Transfers and Descriptions'* on p. 11, is encoded in a *transaction* as an instance of a `JoinSplitDescription` type as follows:

Bytes	Name	Data Type	Description
8	`vpub_old`	`uint64`	A value v_{pub}^{old} that the *JoinSplit transfer* removes from the *transparent transaction value pool*.
8	`vpub_new`	`uint64`	A value v_{pub}^{new} that the *JoinSplit transfer* inserts into the *transparent transaction value pool*.
32	`anchor`	`byte[32]`	A *root* rt^{Sprout} of the **Sprout** *note commitment tree* at some *block height* in the past, or the *root* produced by a previous *JoinSplit transfer* in this *transaction*.
64	`nullifiers`	`byte[32]` [N^{old}]	A sequence of *nullifiers* of the input *notes* $nf_{1..N^{old}}^{old}$.
64	`commitments`	`byte[32]` [N^{new}]	A sequence of *note commitments* for the output *notes* $cm_{1..N^{new}}^{new}$.
32	`ephemeralKey`	`byte[32]`	A Curve25519 *public key* epk.
32	`randomSeed`	`byte[32]`	A 256-bit seed that must be chosen independently at random for each *JoinSplit description*.
64	`vmacs`	`byte[32]` [N^{old}]	A sequence of message authentication tags $h_{1..N^{old}}$ binding h_{Sig} to each a_{sk} of the *JoinSplit description*, computed as described in § 4.8 *'Non-malleability'* on p. 23.
296	`zkproof`	`byte[296]`	An encoding of the *zk-SNARK proof* $\pi_{ZKJoinSplit}$ (see § 5.4.8.1 'BCTV14' on p. 36).
1202	`encCiphertexts`	`byte[601]` [N^{new}]	A sequence of ciphertext components for the encrypted output *notes*, $C_{1..N^{new}}^{enc}$.

The `ephemeralKey` and `encCiphertexts` fields together form the *transmitted notes ciphertext*, which is computed as described in § 4.12 *'In-band secret distribution'* on p. 26.

Consensus rules applying to a *JoinSplit description* are given in § 4.3 *'JoinSplit Descriptions'* on p. 19.

7.3 Block Header Encoding and Consensus

The **Zcash** *block header* format is as follows (this should be read in the context of consensus rules later in the section):

Bytes	Name	Data Type	Description
4	`nVersion`	`int32`	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`	`byte[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`	`byte[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`	`byte[32]`	A reserved field which should be ignored. The *root* $\mathsf{LEBS2OSP}_{256}\left(\mathsf{rt}^{\mathsf{Sapling}}\right)$ of the **Sapling** *note commitment tree* corresponding to the final **Sapling** *treestate* of this *block*.
4	`nTime`	`uint32`	The *block timestamp* is a Unix epoch time (UTC) when the miner started hashing the *header* (according to the miner).
4	`nBits`	`uint32`	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`	`byte[32]`	An arbitrary field that miners can change to modify the *header* hash in order to produce a hash less than or equal to the *target threshold*.
3	`solutionSize`	`compactSize`	The size of an *Equihash* solution in bytes (always 1344).
1344	`solution`	`byte[1344]`	The *Equihash* solution.

A *block* consists of a *block header* and a sequence of *transactions*. How transactions are encoded in a *block* is part of the Zcash peer-to-peer protocol but not part of the consensus protocol.

Let ThresholdBits be as defined in §7.4.3 *'Difficulty adjustment'* on p. 48, and let PoWMedianBlockSpan be the constant defined in §5.3 *'Constants'* on p. 28.

Define the *median-time-past* of a *block* to be the median (as defined in §7.4.3 *'Difficulty adjustment'* on p. 48) of the `nTime` fields of the *preceding* PoWMedianBlockSpan *blocks* (or all preceding *blocks* if there are fewer than PoWMedianBlockSpan). The *median-time-past* of a *genesis block* is not defined.

Consensus rules:

- The *block version number* **MUST** be greater than or equal to 4.
- For a *block* at *block height* height, `nBits` **MUST** be equal to ThresholdBits(height).
- The *block* **MUST** pass the difficulty filter defined in § 7.4.2 ***'Difficulty filter'*** on p. 48.
- `solution` **MUST** represent a *valid Equihash solution* as defined in § 7.4.1 ***'Equihash'*** on p. 47.
- For each *block* other than the *genesis block*, `nTime` **MUST** be strictly greater than the *median-time-past* of that *block*.
- For each *block* at *block height* 2 or greater on *Mainnet*, or *block height* 653606 or greater on *Testnet*, `nTime` **MUST** be less than or equal to the *median-time-past* of that *block* plus $90 \cdot 60$ seconds.
- The size of a *block* **MUST** be less than or equal to 2000000 bytes.
- TODO: Other rules inherited from **Bitcoin**.

In addition, a *full validator* **MUST NOT** accept *blocks* with `nTime` more than two hours in the future according to its clock. This is not strictly a consensus rule because it is nondeterministic, and clock time varies between nodes. Also note that a *block* that is rejected by this rule at a given point in time may later be accepted.

Notes:

- The semantics of blocks with *block version number* not equal to 4 is not currently defined. Miners **MUST NOT** create such *blocks*.
- The exclusion of *blocks* with *block version number* ***greater than*** 4 is not a consensus rule; such *blocks* may exist in the *block chain* and **MUST** be treated identically to version 4 *blocks* by *full validators*. Note that a future upgrade might use *block version number* either greater than or less than 4. It is likely that such an upgrade will change the *block* header and/or *transaction* format, and software that parses *blocks* **SHOULD** take this into account.
- The `nVersion` field is a signed integer. (It was specified as unsigned in a previous version of this specification.) A future upgrade might use negative values for this field, or otherwise change its interpretation.
- There is no relation between the values of the `version` field of a *transaction*, and the `nVersion` field of a *block header*.
- Like other serialized fields of type `compactSize`, the `solutionSize` field **MUST** be encoded with the minimum number of bytes (3 in this case), and other encodings **MUST** be rejected. This is necessary to avoid a potential attack in which a miner could test several distinct encodings of each *Equihash* solution against the difficulty filter, rather than only the single intended encoding.
- As in **Bitcoin**, the `nTime` field **MUST** represent a time ***strictly greater than*** the median of the timestamps of the past PoWMedianBlockSpan *blocks*. The Bitcoin Developer Reference [Bitcoin-Block] was previously in error on this point, but has now been corrected.
- The rule limiting `nTime` to be no later than $90 \cdot 60$ seconds after the *median-time-past* is a retrospective consensus change, applied as a soft fork in `zcashd` v2.1.1-1. It had not been violated by any *block* from the given *block heights* in the consensus *block chains* of either *Mainnet* or *Testnet*.

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

- *Block versions* less than 4 are not supported.
- The `hashReserved`, `solutionSize`, and `solution` fields have been added.
- The type of the `nNonce` field has changed from `uint32` to `byte[32]`.
- The maximum *block* size has been doubled to 2000000 bytes.

7.4 Proof of Work

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

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

- The `solution` field encodes a *valid Equihash solution* according to § 7.4.1 ***'Equihash'*** on p. 47.
- The *block header* satisfies the difficulty check according to § 7.4.2 ***'Difficulty filter'*** on p. 48.

7.4.1 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 *Mainnet* and *Testnet* are $n = 200, k = 9$.

Equihash is based on a variation of the Generalized Birthday Problem [AR2017]: 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 § 5.4.1.5 ***'Equihash Generator'*** on p. 31.

Define I2BEBSP $: (\ell : \mathbb{N}) \times \{0\,..\,2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 28.

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 $w \in \{0\,..\,2^{k-r}-1\}$: $\bigoplus_{j=1}^{2^r} X_{i_{w \cdot 2^r + j}}$ has $\frac{n \cdot r}{k+1}$ leading zeros; and
- For all $r \in \{1\,..\,k\}$, for all $w \in \{0\,..\,2^{k-r}-1\}$: $i_{w \cdot 2^r + 1\,..\,w \cdot 2^r + 2^{r-1}} < i_{w \cdot 2^r + 2^{r-1} + 1\,..\,w \cdot 2^r + 2^r}$ lexicographically.

Notes:

- This does not include a difficulty condition, because here we are defining validity of an *Equihash* solution independent of difficulty.
- Previous versions of this specification incorrectly specified the range of r to be $\{1\,..\,k-1\}$ for both parts of the algorithm binding condition. The implementation in zcashd was as intended.

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

$\mathsf{I2BEBSP}_{21}(i_1 - 1)$	$\mathsf{I2BEBSP}_{21}(i_2 - 1)$	$\cdots$	$\mathsf{I2BEBSP}_{21}(i_{512} - 1)$

Recall from § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 28 that bits in the above diagram are ordered from most to least significant in each byte. For example, if the first 3 elements of i are $[69, 42, 2^{21}]$, then the corresponding bit array is:

$\mathsf{I2BEBSP}_{21}(68)$			$\mathsf{I2BEBSP}_{21}(41)$		$\mathsf{I2BEBSP}_{21}(2^{21}-1)$		
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0			0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1		1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1		
8-bit 0	8-bit 2	8-bit 32	8-bit 0	8-bit 10	8-bit 127	8-bit 255	⋯

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

Note: I2BEBSP is big-endian, while integer field encodings in powheader and in the instantiation of EquihashGen are little-endian. The rationale for this is that little-endian serialization of *block headers* is consistent with **Bitcoin**, but little-endian ordering of bits in the solution encoding would require bit-reversal (as opposed to only shifting).

7.4.2 Difficulty filter

Let ToTarget be as defined in § 7.4.4 *'nBits conversion'* on p. 49.

Difficulty is defined in terms of a *target threshold*, which is adjusted for each *block* according to the algorithm defined in § 7.4.3 *'Difficulty adjustment'* on p. 48.

The difficulty filter is unchanged from **Bitcoin**, and is calculated using SHA-256d on the whole *block header* (including `solutionSize` and `solution`). The result is interpreted as a 256-bit integer represented in little-endian byte order, which **MUST** be less than or equal to the *target threshold* given by ToTarget(`nBits`).

7.4.3 Difficulty adjustment

The desired time between *blocks* is called the *block target spacing*. **Zcash** uses a difficulty adjustment algorithm based on DigiShield v3/v4 [DigiByte-PoW], with simplifications and altered parameters, to adjust difficulty to target the desired *block target spacing*. Unlike **Bitcoin**, the difficulty adjustment occurs after every *block*.

PoWLimit, HalvingInterval, PoWAveragingWindow, PoWMaxAdjustDown, PoWMaxAdjustUp, PoWDampingFactor, and PoWTargetSpacing are specified in section § 5.3 *'Constants'* on p. 28.

Let ToCompact and ToTarget be as defined in § 7.4.4 *'nBits conversion'* on p. 49.

Let nTime(height) be the value of the `nTime` field in the *header* of the *block* at *block height* height.

Let nBits(height) be the value of the `nBits` field in the *header* of the *block* at *block height* height.

Block header fields are specified in § 7.3 *'Block Header Encoding and Consensus'* on p. 45.

Define:

$$\mathsf{mean}(S) := \frac{\sum_{i=1}^{\mathsf{length}(S)} S_i}{\mathsf{length}(S)}$$

$$\mathsf{median}(S) := \mathsf{sorted}(S)_{\mathsf{ceiling}((\mathsf{length}(S)+1)/2)}$$

$$\mathsf{bound}_{\mathsf{lower}}^{\mathsf{upper}}(x) := \mathsf{max}(\mathsf{lower}, \mathsf{min}(\mathsf{upper}, x)))$$

$$\mathsf{trunc}(x) := \begin{cases} \mathsf{floor}(x)\,, & \text{if } x \geq 0 \\ -\mathsf{floor}(-x)\,, & \text{otherwise} \end{cases}$$

$$\mathsf{AveragingWindowTimespan} := \mathsf{PoWAveragingWindow} \cdot \mathsf{PoWTargetSpacing}$$

$$\mathsf{MinActualTimespan} := \mathsf{floor}(\mathsf{AveragingWindowTimespan} \cdot (1 - \mathsf{PoWMaxAdjustUp}))$$

$$\mathsf{MaxActualTimespan} := \mathsf{floor}(\mathsf{AveragingWindowTimespan} \cdot (1 + \mathsf{PoWMaxAdjustDown}))$$

$$\mathsf{MedianTime}(\mathsf{height} : \mathbb{N}) := \mathsf{median}([\![\, \mathsf{nTime}(i) \text{ for } i \text{ from } \mathsf{max}(0, \mathsf{height} - \mathsf{PoWMedianBlockSpan}) \text{ up to } \mathsf{height} - 1 \,]\!])$$

$$\mathsf{ActualTimespan}(\mathsf{height} : \mathbb{N}) := \mathsf{MedianTime}(\mathsf{height}) - \mathsf{MedianTime}(\mathsf{height} - \mathsf{PoWAveragingWindow})$$

$$\mathsf{ActualTimespanDamped}(\mathsf{height} : \mathbb{N}) :=$$
$$\mathsf{AveragingWindowTimespan} + \mathsf{trunc}\left(\frac{\mathsf{ActualTimespan}(\mathsf{height}) - \mathsf{AveragingWindowTimespan}}{\mathsf{PoWDampingFactor}}\right)$$

$$\mathsf{ActualTimespanBounded}(\mathsf{height} : \mathbb{N}) := \mathsf{bound}^{\mathsf{MaxActualTimespan}}_{\mathsf{MinActualTimespan}}(\mathsf{ActualTimespanDamped}(\mathsf{height}))$$

$$\mathsf{MeanTarget}(\mathsf{height} : \mathbb{N}) := \begin{cases} \mathsf{PoWLimit}, & \text{if } \mathsf{height} \leq \mathsf{PoWAveragingWindow} \\ \mathsf{mean}([\mathsf{ToTarget}(\mathsf{nBits}(i)) \text{ for } i \text{ from } \mathsf{height} - \mathsf{PoWAveragingWindow} \text{ up to } \mathsf{height} - 1]), & \\ & \text{otherwise.} \end{cases}$$

The *target threshold* for a given *block height* height is then calculated as:

$$\mathsf{Threshold}(\mathsf{height} : \mathbb{N}) \;:=\; \begin{cases} \mathsf{PoWLimit}, & \text{if } \mathsf{height} = 0 \\ \mathsf{min}(\mathsf{PoWLimit}, \mathsf{floor}\left(\frac{\mathsf{MeanTarget}(\mathsf{height})}{\mathsf{AveragingWindowTimespan}}\right) \cdot \mathsf{ActualTimespanBounded}(\mathsf{height})), & \\ & \text{otherwise} \end{cases}$$

$$\mathsf{ThresholdBits}(\mathsf{height} : \mathbb{N}) := \mathsf{ToCompact}(\mathsf{Threshold}(\mathsf{height})).$$

Notes:

- The convention used for the height parameters to the functions MedianTime, MeanTarget, ActualTimespan, ActualTimespanDamped, ActualTimespanBounded, Threshold, and ThresholdBits is that these functions use only information from *blocks **preceding*** the given *block height*.
- When the median function is applied to a sequence of even length (which only happens in the definition of MedianTime during the first PoWAveragingWindow $- 1$ *blocks* of the *block chain*), the element that begins the second half of the sequence is taken. This corresponds to the zcashd implementation, but was not specified correctly in versions of this specification prior to 2019.0-beta-40.

On *Testnet* from *block height* 299188 onward, the difficulty adjustment algorithm is changed to allow minimum-difficulty *blocks*, as described in [ZIP-205]. This change does not apply to *Mainnet*.

7.4.4 nBits conversion

Deterministic conversions between a *target threshold* and a "compact" nBits value are not fully defined in the Bitcoin documentation [Bitcoin-nBits], and so we define them here:

$$\mathsf{size}(x) := \mathsf{ceiling}\left(\frac{\mathsf{bitlength}(x)}{8}\right)$$

$$\mathsf{mantissa}(x) := \mathsf{floor}\left(x \cdot 256^{3-\mathsf{size}(x)}\right)$$

$$\mathsf{ToCompact}(x) := \begin{cases} \mathsf{mantissa}(x) + 2^{24} \cdot \mathsf{size}(x), & \text{if } \mathsf{mantissa}(x) < 2^{23} \\ \mathsf{floor}\left(\frac{\mathsf{mantissa}(x)}{256}\right) + 2^{24} \cdot (\mathsf{size}(x) + 1), & \text{otherwise} \end{cases}$$

$$\mathsf{ToTarget}(x) := \begin{cases} 0, & \text{if } x \;\&\; 2^{23} = 2^{23} \\ (x \;\&\; (2^{23} - 1)) \cdot 256^{\mathsf{floor}\left(x/2^{24}\right)-3}, & \text{otherwise.} \end{cases}$$

7.4.5 Definition of Work

As explained in § 3.3 *'The Block Chain'* on p. 10, a node chooses the "best" *block chain* visible to it by finding the chain of valid *blocks* with the greatest total work.

Let ToTarget be as defined in § 7.4.4 *'nBits conversion'* on p. 49.

The work of a *block* with value nBits for the `nBits` field in its *block header* is defined as $\mathsf{floor}\left(\frac{2^{256}}{\mathsf{ToTarget}(\mathsf{nBits}) + 1}\right)$.

7.5 Calculation of Block Subsidy and Founders' Reward

§ 3.8 *'Block Subsidy and Founders' Reward'* on p. 13 defines the *block subsidy*, *miner subsidy*, and *Founders' Reward*. Their amounts in *zatoshi* are calculated from the *block height* using the formulae below.

Let SlowStartInterval, HalvingInterval, MaxBlockSubsidy, and FoundersFraction be as defined in § 5.3 *'Constants'* on p. 28.

$$\mathsf{SlowStartShift} : \mathbb{N} := \frac{\mathsf{SlowStartInterval}}{2}$$

$$\mathsf{SlowStartRate} : \mathbb{N} := \frac{\mathsf{MaxBlockSubsidy}}{\mathsf{SlowStartInterval}}$$

$$\mathsf{Halving}(\mathsf{height} : \mathbb{N}) := \begin{cases} 0, & \text{if } \mathsf{height} < \mathsf{SlowStartShift} \\ \mathsf{floor}\left(\frac{\mathsf{height} - \mathsf{SlowStartShift}}{\mathsf{HalvingInterval}}\right), & \text{otherwise} \end{cases}$$

$$\mathsf{BlockSubsidy}(\mathsf{height} : \mathbb{N}) := \begin{cases} \mathsf{SlowStartRate} \cdot \mathsf{height}, & \text{if } \mathsf{height} < \mathsf{SlowStartShift} \\ \mathsf{SlowStartRate} \cdot (\mathsf{height} + 1), & \text{if } \mathsf{SlowStartShift} \leq \mathsf{height} \\ & \quad \text{and } \mathsf{height} < \mathsf{SlowStartInterval} \\ \mathsf{floor}\left(\frac{\mathsf{MaxBlockSubsidy}}{2^{\mathsf{Halving}(\mathsf{height})}}\right), & \text{otherwise} \end{cases}$$

$$\mathsf{FoundersReward}(\mathsf{height} : \mathbb{N}) := \begin{cases} \mathsf{BlockSubsidy}(\mathsf{height}) \cdot \mathsf{FoundersFraction}, & \text{if } \mathsf{Halving}(\mathsf{height}) < 1 \\ 0, & \text{otherwise} \end{cases}$$

$$\mathsf{MinerSubsidy}(\mathsf{height}) := \mathsf{BlockSubsidy}(\mathsf{height}) - \mathsf{FoundersReward}(\mathsf{height}).$$

7.6 Payment of Founders' Reward

The *Founders' Reward* is paid by a *transparent* output in the *coinbase transaction*, to one of NumFounderAddresses *transparent* addresses, depending on the *block height*.

For *Mainnet*, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ is:

["t3Vz22vK5z2LcKEdg16Yv4FFneEL1zg9ojd", "t3cL9AucCajm3HXDhb5jBnJK2vapVoXsop3",
"t3fqvkzrrNaMcamkQMwAyHRjfDdM2xQvDTR", "t3TgZ9ZT2CTSK44AnUPi6qeNaHa2eC7pUyF",
"t3SpkcPQPfuRYHsP5vz3Pv86PgKo5m9KVmx", "t3Xt4oQMRPagwbpQqkgAViQgtST4VoSWR6S",
"t3ayBkZ4w6kKXynwoHZFUSSgXRKtogTXNgb", "t3adJBQuaa21u7NxbR8YMzp3km3TbSZ4MGB",
"t3K4aLYagSSBySdrfAGGeUd5H9z5Qvz88t2", "t3RYnsc5nhEvKiva3ZPhfRSk7eyh1CrA6Rk",
"t3Ut4KUq2ZSMTPNE67pBU5LqYCi2q36KpXQ", "t3ZnCNAvgu6CSyHm1vWtrx3aiN98dSAGpnD",
"t3fB9cB3eSYim64BS9xfwAHQUKLgQQroBDG", "t3cwZfKNNj2vXMAHBQeewm6pXhKFdhk18kD",
"t3YcoujXfspWy7rbNUsGKxFEWZqNstGpeG4", "t3bLvCLigc6rbNrUTS5NwkgyVrZcZumTRa4",
"t3VvHWa7r3oy67YtU4LZKGCWa2J6eGHvShi", "t3eF9X6X2dSo7MCvTjfZEzwWrVzquxRLNeY",
"t3esCNwwmcyc8i9qQfyTbYhTqmYXZ9AwK3X", "t3M4jN7hYE2e27yLsuQPPjuVek81WV3VbBj",
"t3gGWxdC67CYNoBbPjNvrrWLAWxPqZLxrVY", "t3LTWeoxeWPbmdkUD3NWBquk4WkazhFBmvU",
"t3P5KKX97gXYFSaSjJPiruQEX84yF5z3Tjq", "t3f3T3nCWsEpzmD35VK62JgQfFig74dV8C9",
"t3Rqonuzz7afkF7156ZA4vi4iimRSEn41hj", "t3fJZ5jYsyxDtvNrWBeoMbvJaQCj4JJgbgX",
"t3Pnbg7XjP7FGPBUuz75H65aczphHgkpoJW", "t3WeKQDxCijL5X7rwFem1MTL9ZwVJkUFhpF",
"t3Y9FNi26J7UtAUC4moaETLbMo8KS1Be6ME", "t3aNRLLsL2y8xcjPheZZwFy3Pcv7CsTwBec",
"t3gQDEavk5VzAAHK8TrQu2BWDLxEiF1unBm", "t3Rbykhx1TUFrgXrmBYrAJe2STxRKFL7G9r",
"t3aaW4aTdP7a8d1VTE1Bod2yhbeggHgMajR", "t3YEiAa6uEjXwFL2v5ztU1fn3yKgzMQqNyo",
"t3g1yUUwt2PbmDvMDevTCPWUcbDatL2iQGP", "t3dPWnep6YqGPuY1CecgbeZrY9iUwH8Yd4z",
"t3QRZXHDPh2hwU46iQs2776kRuuWfwFp4dV", "t3enhACRxi1ZD7e8ePomVGKn7wp7N9fFJ3r",
"t3PkLgT71TnF112nSwBToXsD77yNbx2gJJY", "t3LQtHUDoe7ZhhvddRv4vnaoNAhCr2f4oFN",
"t3fNcdBUbycvbCtsD2n9q3LuxG7jVPvFB8L", "t3dKojUU2EMjs28nHV84TvkVEUDu1M1FaEx",
"t3aKH6NiWN1ofGd8c19rZiqgYpkJ3n679ME", "t3MEXDF9Wsi63KwpPuQdD6by32Mw2bNTbEa",
"t3WDhPfik343yNmPTqtkZAoQZeqA83K7Y3f", "t3PSn5TbMMAEw7Eu36DYctFezRzpX1hzf3M",
"t3R3Y5vnBLrEn8L6wFjPjBLnxSUQsKnmFpv", "t3Pcm737EsVkGTbhsu2NekKtJeG92mvYyoN"]

For *Testnet*, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ is:

["t2UNzUUx8mWBCRYPRezvA363EYXyEpHokyi", "t2N9PH9Wk9xjqYg9iin1Ua3aekJqfAtE543",
"t2NGQjYMQhFndDHguvUw4wZdNdsssA6K7x2", "t2ENg7hHVqqs9JwU5cgjvSbxnT2a9USNfhy",
"t2BkYdVCHzvTJJUTx4yZB8qeegD8QsPx8bo", "t2J8q1xH1EuigJ52MfExyyjYtN3VgvshKDf",
"t2Crq9mydTm37kZokC68HzT6yez3t2FBnFj", "t2EaMPUiQ1kthqcP5UEkF42CAFKJqXCkXC9",
"t2F9dtQc63JDDyrhnfpzvVYTJcr57MkqA12", "t2LPirmnfYSZc481GgZBa6xUGcoovfytBnC",
"t26xfxoSw2UV9Pe5o3C8V4YybQD4SESfxtp", "t2D3k4fNdErd66YxtvXEdft9xuLoKD7CcVo",
"t2DWYBkxKNivdmsMiivNJzutaQGqmoRjRnL", "t2C3kFF9iQRxfc4B9zgbWo4dQLLqzqjpuGQ",
"t2MnT5tzu9HSKcppRyUNwoTp8MUueuSGNaB", "t2AREsWdoW1F8EQYsScsjkgqobmgrkKeUkK",
"t2Vf4wKcJ3ZFtLj4jezUUKkwYR92BLHn5UT", "t2K3fdViH6R5tRuXLphKyoYXyZhyWGghDNY",
"t2VEn3KiKyHSGyzd3nDw6ESWtaCQHwuv9WC", "t2F8XouqdNMq6zzEvxQXHV1TjwZRHwRg8gC",
"t2BS7Mrbaef3fA4xrmkvDisFVXVrRBnZ6Qj", "t2FuSwoLCdBVPwdZuYoHrEzxAb9qy4qjbnL",
"t2SX3U8NtrT6gz5Db1AtQCSGjrpptr8JC6h", "t2V51gZNSoJ5kRL74bf9YTtbZuv8Fcqx2FH",
"t2FyTsLjjdm4jeVwir4xzj7FAkUidbr1b4R", "t2EYbGLekmpqHyn8UBF6kqpahrYm7D6N1Le",
"t2NQTrStZHtJECNFT3dUBLYA9AErxPCmkka", "t2GSWZZJzoesYxfPTWXkFn5UaxjiYxGBU2a",
"t2RpffkzyLRevGM3w9aWdqMX6bd8uuAK3vn", "t2JzjoQqnuXtTGSN7k7yk5keURBGvYofh1d",
"t2AEefc72ieTnsXKmgK2bZNckiwvZe3oPNL", "t2NNs3ZGZFsNj2wvmVd8BSwSfvETgiLrD8J",
"t2ECCQPVcxUCSSQopdNquguEPE14HsVfcUn", "t2JabDUkG8TaqVKYfqDJ3rqkVdHKp6hwXvG",
"t2FGzW5Zdc8Cy98ZKmRygsVGi6oKcmYir9n", "t2DUD8a21FtEFn42oVLp5NGbogY13uyjy9t",
"t2UjVSd3zheHPgAkuX8WQW2CiC9xHQ8EvWp", "t2TBUAhELyHUn8i6SXYsXz5Lmy7kDzA1uT5",
"t2Tz3uCyhP6eizUWDc3bGH7XUC9GQsEyQNc", "t2NysJSZtLwMLWEJ6MH3BsxRh6h27mNcsSy",
"t2KXJVVyyrjVxxSeazbY9ksGyft4qsXUNm9", "t2J9YYtH31cveiLZzjaE4AcuwVho6qjTNzp",
"t2QgvW4sP9zaGpPMH1GRzy7cpydmuRfB4AZ", "t2NDTJP9MosKpyFPHJmfjc5pGCvAU58XGa4",
"t29pHDBWq7qN4EjwSEHg8wEqYe9pkmVrtRP", "t2Ez9KM8VJLuArcxuEkNRAkhNvidKkzXcjJ",
"t2D5y7J5fpXajLbGrMBQkFg2mFN8fo3n8cX", "t2UV2wr1PTaUiybpkV3FdSdGxUJeZdZztyt"]

Note: For *Testnet* only, the addresses from index 4 onward have been changed from what was implemented at launch. This reflects an upgrade on *Testnet*, starting from *block height* 53127. [Zcash-Issue2113]

Each address representation in $\mathsf{FounderAddressList}$ denotes a *transparent* P2SH multisig address.

Let $\mathsf{SlowStartShift}$ and $\mathsf{Halving}$ be defined as in the previous section.

Define:

$$\mathsf{FounderAddressChangeInterval} := \mathsf{ceiling}\left(\frac{\mathsf{SlowStartShift} + \mathsf{HalvingInterval}}{\mathsf{NumFounderAddresses}}\right)$$

$$\mathsf{FounderAddressIndex}(\mathsf{height} : \mathbb{N}) := 1 + \mathsf{floor}\left(\frac{\mathsf{height}}{\mathsf{FounderAddressChangeInterval}}\right)$$

$$\mathsf{FoundersRewardLastBlockHeight} := \mathsf{SlowStartShift} + \mathsf{HalvingInterval} - 1\,.$$

Let $\mathsf{FounderRedeemScriptHash}(\mathsf{height} : \mathbb{N})$ be the standard redeem script hash, as specified in [Bitcoin-Multisig], for the P2SH multisig address with Base58Check form given by $\mathsf{FounderAddressList}_{\mathsf{FounderAddressIndex}(\mathsf{height})}$.

Consensus rule: A *coinbase transaction* at $\mathsf{height} \in \{1\,..\,\mathsf{FoundersRewardLastBlockHeight}\}$ **MUST** include at least one output that pays exactly $\mathsf{FoundersReward}(\mathsf{height})$ *zatoshi* with a standard P2SH script of the form `OP_HASH160` $\mathsf{FounderRedeemScriptHash}(\mathsf{height})$ `OP_EQUAL` as its `scriptPubKey`.

Notes:

- No *Founders' Reward* is required to be paid for $\mathsf{height} > \mathsf{FoundersRewardLastBlockHeight}$ (i.e. after the first *halving*), or for $\mathsf{height} = 0$ (i.e. the *genesis block*).
- The *Founders' Reward* addresses are not treated specially in any other way, and there can be other outputs to them, in *coinbase transactions* or otherwise. In particular, it is valid for a *coinbase transaction* with $\mathsf{height} \in \{1\,..\,\mathsf{FoundersRewardLastBlockHeight}\}$ to have other outputs, possibly to the same address, that do not meet the criterion in the above consensus rule, as long as at least one output meets it.
- The assertion $\mathsf{FounderAddressIndex}(\mathsf{FoundersRewardLastBlockHeight}) \leq \mathsf{NumFounderAddresses}$ holds, ensuring that the *Founders' Reward* address index remains in range for the whole period in which the *Founders' Reward* is paid.

7.7 Changes to the Script System

The `OP_CODESEPARATOR` opcode has been disabled. This opcode also no longer affects the calculation of *SIGHASH transaction hashes*.

7.8 Bitcoin Improvement Proposals

In general, Bitcoin Improvement Proposals (BIPs) do not apply to **Zcash** unless otherwise specified in this section.

All of the BIPs referenced below should be interpreted by replacing "BTC", or "bitcoin" used as a currency unit, with "ZEC"; and "satoshi" with "zatoshi".

The following BIPs apply, otherwise unchanged, to **Zcash**: [BIP-11], [BIP-14], [BIP-31], [BIP-35], [BIP-37], [BIP-61].

The following BIPs apply starting from the **Zcash** *genesis block*, i.e. any activation rules or exceptions for particular *blocks* in the **Bitcoin** *block chain* are to be ignored: [BIP-16], [BIP-30], [BIP-65], [BIP-66].

The effect of [BIP-34] has been incorporated into the consensus rules (§ 7.1 *'Transaction Encoding and Consensus'* on p. 42). This excludes the *Mainnet* and *Testnet genesis blocks*, for which the "height in coinbase" was inadvertently omitted.

[BIP-13] applies with the changes to address version bytes described in § 5.6.1.1 *'Transparent Addresses'* on p. 38.

[BIP-111] applies from peer-to-peer network protocol version 170004 onward; that is:

- references to protocol version 70002 are to be replaced by 170003;
- references to protocol version 70011 are to be replaced by 170004;
- the reference to protocol version 70000 is to be ignored (**Zcash** nodes have supported Bloom-filtered connections since launch).

8 Differences from the Zerocash paper

8.1 Transaction Structure

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

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

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

Computation of *SIGHASH transaction hashes*, as described in § 4.7 *'SIGHASH Transaction Hashing'* on p. 22, was changed to clean up handling of an error case for SIGHASH_SINGLE, to remove the special treatment of `OP_CODESEPARATOR`, and to include **Zcash**-specific fields in the hash [ZIP-76].

8.2 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 *'Encodings of Note Plaintexts and Memo Fields'* on p. 37.

8.3 Unification of Mints and Pours

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

- a "Mint" transaction takes value from *transparent* UTXOs as input and produces a new *shielded note* as output.
- a "Pour" transaction takes up to N^{old} *shielded notes* as input, and produces up to N^{new} *shielded 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* (see § 8.1 ***'Transaction Structure'*** on p. 52) 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 N^{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.

8.4 Faerie Gold attack and fix

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

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by Receive (as defined in [BCGGMTV2014, 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 [BCGGMTV2014]. 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 adversary 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* must be distinct. This is true regardless

of whether the *nullifiers* corresponded to real or *dummy notes* (see § 4.5 *'Dummy Notes'* on p. 21). The *nullifiers* are used as input to hSigCRH to derive a public value h_{Sig} which uniquely identifies the transaction, as described in § 4.3 *'JoinSplit Descriptions'* on p. 19. (h_{Sig} was already used in **Zerocash** in a way that requires it to be unique in order to maintain indistinguishability of *JoinSplit descriptions*; adding the *nullifiers* to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the *transaction* creator is an adversary.)

The ρ value for each output *note* is then derived from a random private seed φ and h_{Sig} using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of ρ for each output *note* is enforced by § 4.11.1 *'JoinSplit Statement'* on p. 25 in the *JoinSplit statement*.

Now even if the creator of a *JoinSplit description* does not choose φ randomly, uniqueness of *nullifiers* and *collision resistance* of both hSigCRH 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*. This is sufficient to prevent the Faerie Gold attack.

A variation on the attack attempts to cause the *nullifier* of a sent *note* to be repeated, without repeating ρ. However, since the *nullifier* is computed as $\mathsf{PRF}^{\mathsf{nfSprout}}_{a_{sk}}(\rho)$; this is only possible if the adversary finds a collision across both inputs on $\mathsf{PRF}^{\mathsf{nfSprout}}$, which is assumed to be infeasible — see § 4.1.2 *'Pseudo Random Functions'* on p. 14.

Crucially, "*nullifier* integrity" is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note* (§ 4.11.1 *'JoinSplit Statement'* on p. 25). If this were not the case then an adversary could perform the attack by creating a zero-valued *note* with a repeated *nullifier*, since the *nullifier* would not depend on the value.

Nullifier integrity also prevents a "roadblock attack" in which the adversary sees a victim's *transaction*, and is able to publish another *transaction* that is mined first and blocks the victim's *transaction*. This attack would be possible if the public value(s) used to enforce uniqueness of ρ could be chosen arbitrarily by the *transaction* creator: the victim's *transaction*, rather than the adversary's, would be considered to be repeating these values. In the chosen solution that uses *nullifiers* for these public values, they are enforced to be dependent on *spending keys* controlled by the original *transaction* creator (whether or not each input *note* is a *dummy*), and so a roadblock attack cannot be performed by another party who does not know these keys.

8.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of COMM_{rcm} and COMM_{s} is a computationally binding commitment to its inputs a_{pk}, v, and ρ. However, the instantiation of COMM_{rcm} and COMM_{s} in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of a_{pk} and ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct pairs (a_{pk}, ρ) and (a_{pk}', 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 *zk-SNARK proof* ([BCGGMTV2014, section 1.3, under step 3]). Since **Zcash** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers*, and each transfer always uses a *zk-SNARK proof*, it does not require the nesting. A side benefit is that this reduces the cost of computing the *note commitments*: it 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 [BCGGMTV2014, 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.

8.6 Changes to PRF inputs and truncation

The format of inputs to the *PRFs* instantiated in § 5.4.2 *'Pseudo Random Functions'* on p. 31 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_i^{old} is truncated from 256 to 254 bits in the input to $\mathsf{PRF}^{\mathsf{sn}}$ (which corresponds to $\mathsf{PRF}^{\mathsf{nfSprout}}$ in **Zcash**). Also, $\mathsf{h}_{\mathsf{Sig}}$ 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 [BCGGMTV2014, Appendix D].

In more detail:

- In the argument relating $\mathbf{H}$ and $\mathfrak{D}_2$, it is stated that in $\mathfrak{D}_2$, "for each $i \in \{1, 2\}$, $\mathsf{sn}_i := \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a}_{\mathsf{sk}}}(\rho)$ for a random (and not previously used) ρ". It is also argued that "the calls to $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a}_{\mathsf{sk}}}$ 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 $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a}_{\mathsf{sk}}}$, i.e. $[\rho]_{254}$, may repeat even if ρ does not.
- In the same argument, it is stated that "with overwhelming probability, $\mathsf{h}_{\mathsf{Sig}}$ is unique". In fact what is required to be unique is the truncated input to $\mathsf{PRF}^{\mathsf{pk}}$, i.e. $[\mathsf{h}_{\mathsf{Sig}}]_{253} = [\mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})]_{253}$. In practice this value will be unique under a plausible assumption on CRH provided that $\mathsf{pk}_{\mathsf{sig}}$ is chosen randomly, but no formal argument for this is presented.

Note that ρ is truncated in the input to $\mathsf{PRF}^{\mathsf{sn}}$ but not in the input to $\mathsf{COMM}_{\mathsf{rcm}}$, 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 $\mathsf{COMM}_{\mathsf{rcm}}$ but not in the input to $\mathsf{PRF}^{\mathsf{sn}}$. 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 § 8.4 *'Faerie Gold attack and fix'* on p. 53, **Zcash** depends on *collision resistance* of hSigCRH and PRF^{ρ} (instantiated using BLAKE2b-256 and SHA256Compress respectively). *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 avoiding truncation along any path from the inputs to the computation of $\mathsf{h}_{\mathsf{Sig}}$ 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 a_{sk} in the case of $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, and $\mathsf{PRF}^{\mathsf{pk}}$, 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.

8.7 In-band secret distribution

Zerocash specified ECIES (referencing Certicom's SEC 1 standard) as the encryption scheme used for the in-band secret distribution. This has been changed to a key agreement scheme based on Curve25519, 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 [Bernstein2006].
- 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 [MAEÁ2010].
- 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 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 note(s) 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-SNARK 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 **Sprout** includes an optimization that reuses 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. (Passing the recipient *public key* to the KDF could in principle compromise *key privacy*, but not confidentiality of encryption.) 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.

Note that the 256-bit key for AEAD_CHACHA20_POLY1305 maintains a high concrete security level even under attacks using parallel hardware [Bernstein2005] in the multi-user setting [Zaverucha2012]. This is especially necessary because the privacy of **Zcash** transactions may need to be maintained far into the future, and upgrading the encryption algorithm would not prevent a future adversary from attempting to decrypt ciphertexts encrypted before the upgrade. Other cryptovalues that could be attacked to break the privacy of transactions are also sufficiently long to resist parallel brute force in the multi-user setting: $\mathsf{a_{sk}}$ is 252 bits, and $\mathsf{sk_{enc}}$ is no shorter than $\mathsf{a_{sk}}$.

8.8 Omission in Zerocash security proof

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

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

The error is in the proof of Balance in [BCGGMTV2014, 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 $\mathsf{a}_{\mathsf{sk},i}^{\mathsf{old}}$; they contain $\mathsf{a}_{\mathsf{pk},i}^{\mathsf{old}}$. (In **Sprout** $\mathsf{cm}_i^{\mathsf{old}}$ opens directly to $(\mathsf{a}_{\mathsf{pk},i}^{\mathsf{old}}, \mathsf{v}_i^{\mathsf{old}}, \rho_i^{\mathsf{old}})$, and in **Zerocash** it opens to $(\mathsf{v}_i^{\mathsf{old}}, \mathsf{COMM}_{\mathsf{s}}(\mathsf{a}_{\mathsf{pk},i}^{\mathsf{old}}, \rho_i^{\mathsf{old}}))$.)

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 **Sprout**, 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 $\mathsf{a}_{\mathsf{sk},1}^{\mathsf{old}}$ and $\mathsf{a}_{\mathsf{sk},2}^{\mathsf{old}}$, together with constraint 1(b) of the *JoinSplit statement* (see § 4.11.1 *'JoinSplit Statement'* on p. 25), implies distinctness of $\mathsf{a}_{\mathsf{pk},1}^{\mathsf{old}}$ and $\mathsf{a}_{\mathsf{pk},2}^{\mathsf{old}}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

8.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a}_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}}), \mathsf{v}, \rho, \mathsf{rcm}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines it as $(\mathsf{a}_{\mathsf{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$. The instantiation of $\mathsf{COMM}_{\mathsf{s}}$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ in **Sprout**. $\mathsf{pk}_{\mathsf{enc}}$ is also not needed as part of a *note*: it is not an input to $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ 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 § 5.4.8.1 'BCTV14' on p. 36, because both the x-coordinate and compressed y-coordinate of each point need to be represented. Although it is possible to encode a proof in 288 bytes by making use of the fact that elements of $\mathbb{F}_q$ can be represented in 254 bits, we prefer to use the standard formats for points defined in [IEEE2004]. The fork of *libsnark* used by **Zcash** uses this standard encoding rather than the less efficient (uncompressed) one used by upstream *libsnark*.
- The range of monetary values differs. In **Zcash** this range is $\{0 \,..\, \mathsf{MAX\_MONEY}\}$, while in **Zerocash** it is $\{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\}$. (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.)

9 Acknowledgements

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

The designers of the **Zcash** protocol are the **Zerocash** inventors and also Daira Hopwood, Sean Bowe, Jack Grigg, Simon Liu, Taylor Hornby, Nathan Wilcox, Zooko Wilcox, Jay Graber, Ariel Gabizon, George Tankersley, Ying Tong Lai, Kris Nuttycombe, Jack Gavigan, and Steven Smith. The *Equihash* proof-of-work algorithm was designed by Alex Biryukov and Dmitry Khovratovich.

The authors would like to thank everyone with whom they have discussed the **Zerocash** and **Zcash** protocol designs; in addition to the preceding, this includes Mike Perry, isis agora lovecruft, Leif Ryge, Andrew Miller, Ben Blaxill, Samantha Hulsey, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, John Tromp, Paige Peterson, jl777, Alison Stevenson, Maureen Walsh, Filippo Valsorda, Zaki Manian, Tracy Hu, Brian Warner, Mary Maller, Michael Dixon, Andrew Poelstra, Eirik Ogilvie-Wigley, Benjamin Winston, Kobi Gurkan, Weikeng Chen, Henry de Valence, Deirdre Connolly, Chelsea Komlo, Zancas Wilcox, Jane Lusby, Teor, Izaak Meckler, Zac Williamson, Vitalik Buterin, Jakub Zalewski, Oana Ciobotaru, and no doubt others. We would also like to thank the designers and developers of **Bitcoin**.

Zcash has benefited from security audits performed by NCC Group, Coinspect, Least Authority, Mary Maller, Kudelski Security, QEDIT, and Trail of Bits.

The Faerie Gold attack was found by Zooko Wilcox; subsequent analysis of variations on the attack was performed by Daira Hopwood and Sean Bowe. 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 § 8.6 *'Changes to PRF inputs and truncation'* on p. 54 were also found by Daira Hopwood.

The 2015 Soundness vulnerability in BCTV14 [Parno2015] was found by Bryan Parno. An additional condition needed to resist this attack was documented by Ariel Gabizon [Gabizon2019, section 3]. The 2019 Soundness vulnerability in BCTV14 [Gabizon2019] was found by Ariel Gabizon.

The design of **Sapling** is primarily due to Matthew Green, Ian Miers, Daira Hopwood, Sean Bowe, Jack Grigg, and Jack Gavigan. A potential attack linking *diversified payment addresses*, avoided in the adopted design, was found by Brian Warner.

The design of **Orchard** is primarily due to Daira Hopwood, Sean Bowe, Jack Grigg, Kris Nuttycombe, Ying Tong Lai, and Steven Smith.

We thank Ariel Gabizon for teaching us the techniques of [BFIJSV2010] , by applying them to BCTV14.

The arithmetization used by Halo 2 is based on that used by PLONK [GWC2019], which was designed by Ariel Gabizon, Zachary Williamson, and Oana Ciobotaru.

Numerous people have contributed to the science of zero-knowledge proving systems, but we would particularly like to acknowledge the work of Shafi Goldwasser, Silvio Micali, Oded Goldreich, Charles Rackoff, Rosario Gennaro, Bryan Parno, Jon Howell, Craig Gentry, Mariana Raykova, Jens Groth, Rafail Ostrovsky, and Amit Sahai.

We thank the organizers of the ZKProof standardization effort and workshops; and also Anna Rose and Fredrik Harrysson for their work on the Zero Knowledge Podcast, ZK Summits, and ZK Study Club. These efforts have enriched the zero knowledge community immeasurably.

Many of the ideas used in **Zcash** —including the use of zero-knowledge proofs to resolve the tension between privacy and auditability, Merkle trees over note commitments, and the use of "serial numbers" or *nullifiers* to detect or prevent double-spends— were first applied to privacy-preserving digital currencies by Tomas Sander and Amnon Ta-Shma. To a large extent **Zcash** is a refinement of their "Auditable, Anonymous Electronic Cash" proposal in [ST1999].

We thank Alexandra Elbakyan for her tireless work in dismantling barriers to scientific research.

10 Change History

2021.1.17 2021-03-15

- The definition of an abstraction function in § 4.1.8 *'Represented Group'* on p. 17 incorrectly required canonicity, i.e. that $\mathsf{abst}_{\mathbb{G}}$ does not accept inputs outside the range of $\mathsf{repr}_{\mathbb{G}}$.
- Rename `char` to `byte` in field type declarations.

2021.1.16 2021-01-11

- Add macros and `Makefile` support for building the **NU5** draft specification.
- Clarify the encoding of *block heights* for the "height in coinbase" rule. The description of this rule has also moved from § 7.3 on p. 45 to § 7.1 *'Transaction Encoding and Consensus'* on p. 42.
- Include the activation dates of **Heartwood** and **Canopy** in § 6 *'Network Upgrades'* on p. 41.
- Section links in the **Heartwood** and **Canopy** versions of the specification now go to the correct document URL.
- Attempt to improve search and cut-and-paste behaviour for ligatures in some PDF readers.

2020.1.15 2020-11-06

- Add a missing consensus rule that has always been implemented in zcashd: there must be at least one *transparent output*, *Output description*, or *JoinSplit description* in a *transaction*.
- Add a consensus rule that the (zero-valued) *coinbase transaction* output of the *genesis block* cannot be spent.
- Define ***Sprout*** *chain value pool balance* and include consensus rules from [ZIP-209].
- Reserve *transaction version number* `0x7FFFFFFF` and *version group ID* `0xFFFFFFFF` for experimental use.
- Remove a statement that the language consisting of key and address encoding possibilities is prefix-free. (The human-readable forms are prefix-free but the raw encodings are not; for example, the *raw encoding* of a **Sapling** *spending key* can be a prefix of several of the other encodings.)
- Use "let mutable" to introduce mutable variables in algorithms.
- Acknowledge Jack Gavigan as a co-designer of **Sapling** and of the **Zcash** protocol.
- Acknowledge Izaak Meckler, Zac Williamson, Vitalik Buterin, and Jakub Zalewski.
- Acknowledge Alexandra Elbakyan.

2020.1.14 2020-08-19

- The consensus rule that a *coinbase transaction* must not spend more than is available from the *block subsidy* and *transaction fees*, was not explicitly stated. (This rule was correctly implemented in zcashd.)

2020.1.13 2020-08-11

- Make Halving(height) return 0 (rather than -1) for height $<$ SlowStartShift. This has no effect on consensus since the Halving function is not used in that case, but it makes the definition match the intuitive meaning of the function.
- Rename sections under § 7 *'Consensus Changes from Bitcoin'* on p. 42 to clarify that these sections do not only concern encoding, but also consensus rules.
- Make the **Canopy** specification the default.

2020.1.12 2020-08-03

- Include SHA-512 in § 5.4.1.1 *'SHA-256, SHA-256d, SHA256Compress, and SHA-512 Hash Functions'* on p. 29.
- Add a reference to [BCCGLRT2014] in § 4.1.10 *'Zero-Knowledge Proving System'* on p. 18.

2020.1.11 2020-07-13

- Change instances of "the production network" to "*Mainnet*", and "the test network" to *Testnet*. This follows the terminology used in ZIPs.
- Update stale references to **Bitcoin** documentation.

2020.1.10 2020-07-05

- Corrections to a note in § 5.4.5 *'Ed25519'* on p. 33.

2020.1.9 2020-07-05

- Add § 3.10 *'Mainnet and Testnet'* on p. 13.
- Acknowledge Jane Lusby and Teor.
- Precisely specify the encoding and decoding of Ed25519 points.

2020.1.8 2020-07-04

- Add Ying Tong Lai and Kris Nuttycombe as **Zcash** protocol designers.

2020.1.7 2020-06-26

- Delete some 'new' superscripts that only added notational clutter.
- Add an explicit lead byte field to **Sprout** *note plaintexts*, and clearly specify the error handling when it is invalid.

2020.1.6 2020-06-17

- Correct an error in the specification of Ed25519 *validating keys*: they should not have been specified to be checked against ExcludedPointEncodings, since libsodium v1.0.15 does not do so.
- Consistently use "validating" for signatures and "verifying" for proofs.

2020.1.5 2020-06-02

- Mark more index entries as definitions.

2020.1.4 2020-05-27

- Reference [BIP-32] and [ZIP-32] when describing keys and their encodings.
- Network Upgrade 4 has been given the name **Canopy**.
- Improve LaTeX portability of this specification.

2020.1.3 2020-04-22

- Correct a wording error transposing *transparent inputs* and *transparent outputs* in § 4.9 ***'Balance'*** on p. 23.

2020.1.2 2020-03-20

- The implementation of **Sprout** Ed25519 signature validation in zcashd differed from what was specified in § 5.4.5 **'Ed25519'** on p. 33. The specification has been changed to match the implementation.
- Remove "pvc" `Makefile` targets.
- Make the **Heartwood** specification the default.
- Add macros and `Makefile` support for building the **Canopy** specification.

2020.1.1 2020-02-13

- Resolve conflicts in the specification of *memo fields* by deferring to [ZIP-302].

2020.1.0 2020-02-06

- Specify a retrospective soft fork implemented in zcashd v2.1.1-1 that limits the `nTime` field of a *block* relative to its *median-time-past*.
- Correct the definition of *median-time-past* for the first PoWMedianBlockSpan *blocks* in a *block chain*.
- Add acknowledgements to Henry de Valence, Deirdre Connolly, Chelsea Komlo, and Zancas Wilcox.
- Add an acknowledgement to Trail of Bits for their security audit.
- Change indices in the *incremental Merkle tree* diagram to be zero-based.

2019.0.9 2019-12-27

- No changes to **Sprout**.
- `Makefile` updates for **Heartwood**.

2019.0.8 2019-09-24

- No changes to **Sprout**.

2019.0.7 2019-09-24

- Update references to ZIPs and to the Electric Coin Company blog.
- `Makefile` improvements to suppress unneeded output.

2019.0.6 2019-08-23

- No changes to **Sprout**.

2019.0.5 2019-08-23

- Remove "optimized" `Makefile` targets (which actually produced a larger PDF, with TeXLive 2019).
- Remove "html" `Makefile` targets.
- Make the **Blossom** spec the default.

2019.0.4 2019-07-23

- Clicking on a section heading now shows section labels.
- Changes needed to support TeXLive 2019.

2019.0.3 2019-07-08

- Experimental support for building using LuaTeX and XeTeX.
- Add an **Index**.

2019.0.2 2019-06-18

- Ensure that this document builds correctly and without missing characters on recent versions of TeXLive.
- Update the `Makefile` to use Ghostscript for PDF optimization.
- Ensure that hyperlinks are preserved, and available as "Destination names" in URL fragments and links from other PDF documents.

2019.0.1 2019-05-20

- No changes to **Sprout**.

2019.0.0 2019-05-01

- Fix a specification error in the *Founders' Reward* calculation during the slow start period.
- Correct an inconsistency in difficulty adjustment between the spec and zcashd implementation for the first PoWAveragingWindow − 1 *blocks* of the *block chain*. This inconsistency was pointed out by NCC Group in their **Blossom** specification audit.

2019.0-beta-39 2019-04-18

- Change author affiliations from "Zerocoin Electric Coin Company" to "Electric Coin Company".
- Add acknowledgement to Mary Maller for the observation that *diversified payment address* unlinkability can be proven in the same way as *key privacy* for ElGamal.

2019.0-beta-38 2019-04-18

- Update `README.rst` to include `Makefile` targets for **Blossom**.
- `Makefile` updates:
  - Fix a typo for the `pvcblossom` target.
  - Update the pinned `git` hashes for `sam2p` and `pdfsizeopt`.

2019.0-beta-37 2019-02-22

- The rule that miners **SHOULD NOT** mine *blocks* that chain to other *blocks* with a *block version number* greater than 4, has been removed. This is because such *blocks* (mined nonconformantly) exist in the current *Mainnet* consensus *block chain*.
- Clarify that *Equihash* is based on a ***variation*** of the Generalized Birthday Problem, and cite [AR2017].
- Update reference [BGG2017] (previously [BGG2016]).
- Add macros and `Makefile` support for building the **Blossom** specification.

2019.0-beta-36 2019-02-09

- Correct isis agora lovecruft's name.

2019.0-beta-35 2019-02-08

- Cite [Gabizon2019] and acknowledge Ariel Gabizon.
- Correct [SBB2019] to [SWB2019].
- The [Gabizon2019] vulnerability affected Soundness of BCTV14 as well as Knowledge Soundness.
- Clarify the history of the [Parno2015] vulnerability and acknowledge Bryan Parno.
- Specify the difficulty adjustment change that occurred on *Testnet* at *block height* 299188.
- Add Eirik Ogilvie-Wigley and Benjamin Winston to acknowledgements.

2019.0-beta-34 2019-02-05

- Disclose a security vulnerability in BCTV14 that affected **Sprout** before activation of the **Sapling** *network upgrade* (see § 5.4.8.1 'BCTV14' on p. 36).
- Rename PHGR13 to BCTV2014.
- Rename reference [BCTV2015] to [BCTV2014a], and [BCTV2014] to [BCTV2014b].

2018.0-beta-33 2018-11-14

- No changes to **Sprout**.

2018.0-beta-32 2018-10-24

- No changes to **Sprout**.

2018.0-beta-31 2018-09-30

- No changes to **Sprout**.
- Add the QED-it report to the acknowledgements.

2018.0-beta-30 2018-09-02

- No changes to **Sprout**.
- Add dates to Change History entries. (These are the dates of the git tags in local, i.e. UK, time.)

2018.0-beta-29 2018-08-15

- No changes to **Sprout**.

2018.0-beta-28 2018-08-14

- No changes to **Sprout**.

2018.0-beta-27 2018-08-12

- Notational changes:
  - Use a superscript $^{(r)}$ to mark the subgroup order, instead of a subscript.
  - Use $\mathbb{G}^{(r)*}$ for the set of $r_{\mathbb{G}}$-order points in $\mathbb{G}$.
  - Mark the subgroup order in pairing groups, e.g. use $\mathbb{G}_1^{(r)}$ instead of $\mathbb{G}_1$.
- Add Charles Rackoff, Rafail Ostrovsky, and Amit Sahai to the acknowledgements section for their work on zero-knowledge proofs.

2018.0-beta-26 2018-08-05

- No changes to **Sprout**.

2018.0-beta-25 2018-08-05

- No changes to **Sprout**.
- `Makefile` changes: name the PDF file for the **Sprout** version of the specification as `sprout.pdf`, and make `protocol.pdf` link to the **Sapling** version.

2018.0-beta-24 2018-07-31

- No changes to **Sprout**.

2018.0-beta-23 2018-07-27

- No changes to **Sprout**.

2018.0-beta-22 2018-07-18

- Update the abstract to clarify that this version of the specification is a historical document.

2018.0-beta-21 2018-06-22

- Remove the consensus rule "If `nJoinSplit` > 0, the *transaction* **MUST NOT** use *SIGHASH types* other than SIGHASH_ALL.", which was never implemented.
- Add section on signature hashing.
- Briefly describe the changes to computation of *SIGHASH transaction hashes*.
- Clarify that interstitial *treestates* form a tree for each *transaction* containing *JoinSplit descriptions*.
- Correct the description of P2PKH addresses in § 5.6.1.1 ***'Transparent Addresses'*** on p. 38 — they use a hash of a compressed, not an uncompressed ECDSA key representation.
- Clarify the wording of the caveat[3] about the claimed security of shielded *transactions*.
- Correct the definition of set difference ($S \setminus T$).
- Add a note concerning malleability of *zk-SNARK proofs*.
- Clarify attribution of the **Zcash** protocol design.
- Acknowledge Alex Biryukov and Dmitry Khovratovich as the designers of *Equihash*.
- Acknowledge Shafi Goldwasser, Silvio Micali, Oded Goldreich, Rosario Gennaro, Bryan Parno, Jon Howell, Craig Gentry, Mariana Raykova, and Jens Groth for their work on zero-knowledge proving systems.
- Acknowledge Tomas Sander and Amnon Ta–Shma for [ST1999].
- Acknowledge Kudelski Security's audit.

2018.0-beta-20 2018-05-22

- Add Michael Dixon and Andrew Poelstra to acknowledgements.
- Minor improvements to cross-references.

2018.0-beta-19 2018-04-23

- No changes to **Sprout**.

2018.0-beta-18 2018-04-23

- No changes to **Sprout**.

2018.0-beta-17 2018-04-21

- No changes to **Sprout**.

2018.0-beta-16 2018-04-21

- Explicitly note that outputs from *coinbase transactions* include *Founders' Reward* outputs.
- The point represented by $\underline{R}$ in an Ed25519 signature is checked to not be of small order; this is not the same as checking that it is of prime order ℓ.
- Specify support for [BIP-111] (the `NODE_BLOOM` service bit) in peer-to-peer network protocol version 170004.
- Give references [Vercauter2009] and [AKLGL2010] for the optimal ate pairing.
- Give references for BN [BN2005] curves.
- Define $\mathsf{KA}^{\mathsf{Sprout}}$.DerivePublic for Curve25519.
- Caveat the claim about *note traceability set* in § 1.2 ***'High-level Overview'*** on p. 5 and link to [Peterson2017] and [Quesnelle2017].

- Do not require a generator as part of the specification of a *represented group*; instead, define it in the *represented pairing* or scheme using the group.
- Refactor the abstract definition of a *signature scheme* to allow derivation of *validating keys* independent of key pair generation.
- Add acknowledgements for Brian Warner, Mary Maller, and the Least Authority audit.
- `Makefile` improvements.

2018.0-beta-15 2018-03-19

- Clarify the bit ordering of SHA-256.
- Drop `_t` from the names of representation types.
- Remove functions from the **Sprout** specification that it does not use.
- Change the `Makefile` to avoid multiple reloads in PDF readers while rebuilding the PDF.
- Spacing and pagination improvements.

2018.0-beta-14 2018-03-11

- Only cosmetic changes to **Sprout**.

2018.0-beta-13 2018-03-11

- Only cosmetic changes to **Sprout**.

2018.0-beta-12 2018-03-06

- No changes to **Sprout**.

2018.0-beta-11 2018-02-26

- No changes to **Sprout**.

2018.0-beta-10 2018-02-26

- Split the descriptions of SHA-256 and SHA256Compress into their own sections. Specify SHA256Compress more precisely.
- Add Tracy Hu to acknowledgements.
- Move bit/byte/integer conversion primitives into § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 28.

2018.0-beta-9 2018-02-10

- Specify the coinbase maturity rule, and the rule that *coinbase transactions* cannot contain *JoinSplit descriptions*.

2018.0-beta-8 2018-02-08

- No changes to **Sprout**.

2018.0-beta-7 2018-02-07

- Specify the 100000-byte limit on *transaction* size. (The implementation in zcashd was as intended.)
- Specify that 0xF6 followed by 511 zero bytes encodes an empty *memo field*.
- Reference security definitions for *Pseudo Random Functions*.
- Rename clamp to bound and ActualTimespanClamped to ActualTimespanBounded in the difficulty adjustment algorithm, to avoid a name collision with Curve25519 scalar "clamping".
- Change uses of the term *full node* to *full validator*. A *full node* by definition participates in the peer-to-peer network, whereas a *full validator* just needs a copy of the *block chain* from somewhere. The latter is what was meant.

2018.0-beta-6 2018-01-31

- No changes to **Sprout**.

2018.0-beta-5 2018-01-30

- Specify more precisely the requirements on Ed25519 *validating keys* and signatures.

2018.0-beta-4 2018-01-25

- No changes to **Sprout**.

2018.0-beta-3 2018-01-22

- Explain how the chosen fix to Faerie Gold avoids a potential "roadblock" attack.

2017.0-beta-2.9 2017-12-17

- Refer to $\mathsf{sk}_{\mathsf{enc}}$ as a *receiving key* rather than as a viewing key.
- Updates for *incoming viewing key* support.

2017.0-beta-2.8 2017-12-02

- Correct the non-normative note describing how to check the order of π_B.

2017.0-beta-2.7 2017-07-10

- Fix an off-by-one error in the specification of the *Equihash* algorithm binding condition. (The implementation in zcashd was as intended.)
- Correct the types and consensus rules for *transaction version numbers* and *block version numbers*. (Again, the implementation in zcashd was as intended.)
- Clarify the computation of h_i in a *JoinSplit statement*.

2017.0-beta-2.6 2017-05-09

- Be more precise when talking about curve points and pairing groups.

2017.0-beta-2.5 2017-03-07

- Clarify the consensus rule preventing double-spends.
- Clarify what a *note commitment* opens to in § 8.8 ***'Omission in Zerocash security proof'*** on p. 56.
- Correct the order of arguments to COMM in § 5.4.6.1 ***'Sprout Note Commitments'*** on p. 34.
- Correct a statement about indistinguishability of *JoinSplit descriptions*.
- Change the *Founders' Reward* addresses, for *Testnet* only, to reflect the hard-fork upgrade described in [Zcash-Issue2113].

2017.0-beta-2.4 2017-02-25

- Explain a variation on the Faerie Gold attack and why it is prevented.
- Generalize the description of the InternalH attack to include finding collisions on $(\mathsf{a_{pk}}, \rho)$ rather than just on ρ.
- Rename $\mathsf{enforce}_i$ to $\mathsf{enforceMerklePath}_i$.

2017.0-beta-2.3 2017-02-12

- Specify the security requirements on the SHA256Compress function in order for the scheme in § 5.4.6.1 ***'Sprout Note Commitments'*** on p. 34 to be a secure commitment.
- Specify $\mathbb{G}_2$ more precisely.
- Explain the use of interstitial *treestates* in chained *JoinSplit transfers*.

2017.0-beta-2.2 2017-02-11

- Give definitions of computational binding and computational hiding for commitment schemes.
- Give a definition of statistical zero knowledge.
- Reference the white paper on MPC parameter generation [BGG2017].

2017.0-beta-2.1 2017-02-06

- ℓ_{Merkle} is a bit length, not a byte length.
- Specify the maximum *block* size.

2017.0-beta-2 2017-02-04

- Add abstract and keywords.
- Fix a typo in the definition of *nullifier* integrity.
- Make the description of *block chains* more consistent with upstream **Bitcoin** documentation (referring to "best" chains rather than using the concept of a *block chain view*).
- Define how nodes select a *best valid block chain*.

2016.0-beta-1.13 2017-01-20

- Specify the difficulty adjustment algorithm.
- Clarify some definitions of fields in a *block header*.
- Define $\mathsf{PRF}^{\mathsf{addr}}$ in § 4.2 ***'Key Components'*** on p. 19.

2016.0-beta-1.12 2017-01-09

- Update the hashes of proving and verifying keys for the final Sprout parameters.
- Add cross references from *shielded payment address* and *spending key* encoding sections to where the key components are specified.
- Add acknowledgements for Filippo Valsorda and Zaki Manian.

2016.0-beta-1.11 2016-12-19

- Specify a check on the order of π_B in a *zk-SNARK proof*.
- Note that due to an oversight, the **Zcash** *genesis block* does not follow [BIP-34].

2016.0-beta-1.10 2016-10-30

- Update reference to the *Equihash* paper [BK2016]. (The newer version has no algorithmic changes, but the section discussing potential ASIC implementations is substantially expanded.)
- Clarify the discussion of proof size in "Differences from the **Zerocash** paper".

2016.0-beta-1.9 2016-10-28

- Add *Founders' Reward* addresses for *Mainnet*.
- Change *"protected"* terminology to *"shielded"*.

2016.0-beta-1.8 2016-10-04

- Revise the lead bytes for *transparent* P2SH and P2PKH addresses, and reencode the *Testnet Founders' Reward* addresses.
- Add a section on which BIPs apply to **Zcash**.
- Specify that `OP_CODESEPARATOR` has been disabled, and no longer affects *SIGHASH transaction hashes*.
- Change the representation type of `vpub_old` and `vpub_new` to `uint64`. (This is not a consensus change because the type of $v_{\mathsf{pub}}^{\mathsf{old}}$ and $v_{\mathsf{pub}}^{\mathsf{new}}$ was already specified to be $\{0 \,..\, \mathsf{MAX\_MONEY}\}$; it just better reflects the implementation.)
- Correct the representation type of the *block* `nVersion` field to `uint32`.

2016.0-beta-1.7 2016-10-02

- Clarify the consensus rule for payment of the *Founders' Reward*, in response to an issue raised by the NCC audit.

2016.0-beta-1.6 2016-09-26

- Fix an error in the definition of the sortedness condition for *Equihash*: it is the sequences of indices that are sorted, not the sequences of hashes.
- Correct the number of bytes in the encoding of `solutionSize`.
- Update the section on encoding of *transparent* addresses. (The precise prefixes are not decided yet.)
- Clarify why BLAKE2b-ℓ is different from truncated BLAKE2b-512.
- Clarify a note about SU-CMA security for signatures.
- Add a note about $\mathsf{PRF}^{\mathsf{nfSprout}}$ corresponding to $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash**.
- Add a paragraph about key length in § 8.7 ***'In-band secret distribution'*** on p. 55.
- Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

2016.0-beta-1.5 2016-09-22

- Update the *Founders' Reward* address list.
- Add some clarifications based on Eli Ben-Sasson's review.

2016.0-beta-1.4 2016-09-19

- Specify the *block subsidy*, *miner subsidy*, and the *Founders' Reward*.
- Specify *coinbase transaction* outputs to *Founders' Reward* addresses.
- Improve notation (for example "·" for multiplication and "$T^{[\ell]}$" for sequence types) to avoid ambiguity.

2016.0-beta-1.3 2016-09-16

- Correct the omission of `solutionSize` from the *block header* format.
- Document that `compactSize` encodings must be canonical.
- Add a note about conformance language in the introduction.
- Add acknowledgements for Solar Designer, Ling Ren and Alison Stevenson, and for the NCC Group and Coinspect security audits.

2016.0-beta-1.2 2016-09-11

- Remove GeneralCRH in favour of specifying hSigCRH and EquihashGen directly in terms of BLAKE2b-ℓ.
- Correct the security requirement for EquihashGen.

2016.0-beta-1.1 2016-09-05

- Add a specification of abstract signatures.
- Clarify what is signed in the "Sending Notes" section.
- Specify ZK parameter generation as a randomized algorithm, rather than as a distribution of parameters.

2016.0-beta-1 2016-09-04

- Major reorganization 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 $\mathsf{h}_{\mathsf{Sig}}$.
- Fix the lead bytes in *shielded payment address* and *spending key* encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$.
- 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 2016-05-20

- Change main font to Quattrocento.

2016.0-alpha-3 2016-05-09

- Change version numbering convention (no other changes).

2.0-alpha-3 2016-05-06

- 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 2016-04-21

- 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 § 8.8 ***'Omission in Zerocash security proof'*** on p. 56).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in § 4.11.1 ***'JoinSplit Statement'*** on p. 25 about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

2.0-alpha-1 2016-03-30

- First version intended for public review.

11 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) (↑ p15, 56).

[AKLGL2010] Diego Aranha, Koray Karabina, Patrick Longa, Catherine Gebotys, and Julio López. *Faster Explicit Formulas for Computing Pairings over Ordinary Curves*. Cryptology ePrint Archive: Report 2010/526. Last revised September 12, 2011. URL: `https://eprint.iacr.org/2010/526` (visited on 2018-04-03) (↑ p35, 64).

[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) (↑ p30).

[AR2017] Leo Alcock and Ling Ren. "A Note on the Security of Equihash". In: *CCSW '17. Proceedings of the 2017 Cloud Computing Security Workshop (Dallas, TX, USA, November 3, 2017); post-workshop of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. URL: `http://sci-hub.tw/10.1145/3140649.3140652` (visited on 2019-01-09) (↑ p47, 62).

[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. (↑ p55).

[BCCGLRT2014] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. *The Hunting of the SNARK*. Cryptology ePrint Archive: Report 2014/580. Received July 24, 2014. URL: `https://eprint.iacr.org/2014/580` (visited on 2020-08-01) (↑ p18, 59).

[BCGGMTV2014] 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. (↑ p5, 6, 7, 14, 24, 25, 27, 53, 54, 55, 56).

[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 2013*, pages 90–108; IACR, 2013. (↑ p36).

[BCTV2014a] 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 February 5, 2019. URL: `https://eprint.iacr.org/2013/879` (visited on 2019-02-08) (↑ p36, 37, 62).

[BCTV2014a-old] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture (May 19, 2015 version)*. Cryptology ePrint Archive: Report 2013/879. Version: 20150519:172604. URL: `https://eprint.iacr.org/2013/879/20150519:172604` (visited on 2019-02-08) (↑ p36).

[BCTV2014b] 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) (↑ p19, 62).

[BDEHR2011] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. *On the Security of the Winternitz One-Time Signature Scheme (full version)*. Cryptology ePrint Archive: Report 2011/191. Received April 13, 2011. URL: `https://eprint.iacr.org/2011/191` (visited on 2016-09-05) (↑ p16).

[BDJR2000] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. *A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation*. September 2000. URL: https://cseweb.ucsd.edu/~mihir/papers/sym-enc.html (visited on 2018-02-07). An extended abstract appeared in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (Miami Beach, Florida, USA, October 20–22, 1997)*, pages 394–403; IEEE Computer Society Press, 1997; ISBN 0-8186-8197-7. (↑ p14).

[BDLSY2012] 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. (↑ p33, 34).

[Bernstein2005] Daniel Bernstein. "Understanding brute force". In: *ECRYPT STVL Workshop on Symmetric Key Encryption, eSTREAM report 2005/036*. April 25, 2005. URL: https://cr.yp.to/papers.html#bruteforce (visited on 2016-09-24). Document ID: 73e92f5b71793b498288efe81fe55dee. (↑ p56).

[Bernstein2006] 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, 2006)*. Springer-Verlag, February 9, 2006. URL: http://cr.yp.to/papers.html#curve25519 (visited on 2016-08-14). Document ID: 4230efdfa673480fc079449d90f322c0. (↑ p15, 32, 39, 40, 55).

[BFIJSV2010] Olivier Blazy, Georg Fuchsbauer, Malika Izabachène, Amandine Jambert, Hervé Sibert, and Damien Vergnaud. *Batch Groth–Sahai*. Cryptology ePrint Archive: Report 2010/040. Last revised February 3, 2010. URL: https://eprint.iacr.org/2010/040 (visited on 2020-10-17) (↑ p58).

[BGG-mpc] Sean Bowe, Ariel Gabizon, and Matthew Green. *GitHub repository 'zcash/mpc': zk-SNARK parameter multi-party computation protocol*. URL: https://github.com/zcash/mpc (visited on 2017-01-06) (↑ p40).

[BGG2017] Sean Bowe, Ariel Gabizon, and Matthew Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive: Report 2017/602. Last revised June 25, 2017. URL: https://eprint.iacr.org/2017/602 (visited on 2019-02-10) (↑ p37, 40, 62, 67).

[BIP-11] Gavin Andresen. *M-of-N Standard Transactions*. Bitcoin Improvement Proposal 11. Created October 18, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-13] Gavin Andresen. *Address Format for pay-to-script-hash*. Bitcoin Improvement Proposal 13. Created October 18, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki (visited on 2020-07-13) (↑ p38, 52).

[BIP-14] Amir Taaki and Patrick Strateman. *Protocol Version and User Agent*. Bitcoin Improvement Proposal 14. Created November 10, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-16] Gavin Andresen. *Pay to Script Hash*. Bitcoin Improvement Proposal 16. Created January 3, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-30] Pieter Wuille. *Duplicate transactions*. Bitcoin Improvement Proposal 30. Created February 22, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-31] Mike Hearn. *Pong message*. Bitcoin Improvement Proposal 31. Created April 11, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-32] Pieter Wuille. *Hierarchical Deterministic Wallets*. Bitcoin Improvement Proposal 32. Created February 11, 2012. Last updated January 15, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (visited on 2020-07-13) (↑ p39, 60).

[BIP-34] Gavin Andresen. *Block v2, Height in Coinbase*. Bitcoin Improvement Proposal 34. Created July 6, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki (visited on 2020-07-13) (↑ p43, 52, 68).

[BIP-35] Jeff Garzik. *mempool message*. Bitcoin Improvement Proposal 35. Created August 16, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-37] Mike Hearn and Matt Corallo. *Connection Bloom filtering*. Bitcoin Improvement Proposal 37. Created October 24, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-61] Gavin Andresen. *Reject P2P message*. Bitcoin Improvement Proposal 61. Created June 18, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-62] Pieter Wuille. *Dealing with malleability*. Bitcoin Improvement Proposal 62. Withdrawn November 17, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki (visited on 2020-07-13) (↑ p16).

[BIP-65] Peter Todd. *OP_CHECKLOCKTIMEVERIFY*. Bitcoin Improvement Proposal 65. Created October 10, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki (visited on 2020-07-13) (↑ p52).

[BIP-66] Pieter Wuille. *Strict DER signatures*. Bitcoin Improvement Proposal 66. Created January 10, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki (visited on 2020-07-13) (↑ p52).

[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 2020-07-13) (↑ p43).

[BIP-111] Matt Corallo and Peter Todd. *NODE_BLOOM service bit*. Bitcoin Improvement Proposal 111. Created August 20, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0111.mediawiki (visited on 2020-07-13) (↑ p52, 64).

[Bitcoin-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Base58Check_encoding (visited on 2020-07-13) (↑ p38, 39).

[Bitcoin-Block] *Block Headers — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/block_chain.html#block-headers (visited on 2020-07-13) (↑ p46).

[Bitcoin-CoinJoin] *CoinJoin — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/CoinJoin (visited on 2020-07-13) (↑ p6).

[Bitcoin-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/transactions.html#raw-transaction-format (visited on 2020-07-13) (↑ p43).

[Bitcoin-Multisig] *Transactions: Multisig — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#multisig (visited on 2020-07-13) (↑ p51).

[Bitcoin-nBits] *Target nBits — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/block_chain.html#target-nbits (visited on 2020-07-13) (↑ p45, 49).

[Bitcoin-P2PKH] *Transactions: P2PKH Script Validation — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#p2pkh-script-validation (visited on 2020-07-13) (↑ p38).

[Bitcoin-P2SH] *Transactions: P2SH Scripts — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#pay-to-script-hash-p2sh (visited on 2020-07-13) (↑ p38).

[Bitcoin-Protocol] *Protocol documentation — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 2020-07-13) (↑ p6).

[Bitcoin-SigHash] *Signature Hash Types — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#signature-hash-types (visited on 2020-07-13) (↑ p23).

[BK2016] Alex Biryukov and Dmitry Khovratovich. *Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem (full version)*. Cryptology ePrint Archive: Report 2015/946. Last revised October 27, 2016. URL: https://eprint.iacr.org/2015/946 (visited on 2016-10-30) (↑ p7, 47, 68).

[BN2005] Paulo Barreto and Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*. Cryptology ePrint Archive: Report 2005/133. Last revised February 28, 2006. URL: https://eprint.iacr.org/2005/133 (visited on 2018-04-20) (↑ p35, 64).

[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) (↑ p15).

[CVE-2019-7167] Common Vulnerabilities and Exposures. *CVE-2019-7167*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7167 (visited on 2019-02-05) (↑ p37).

[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) (↑ p56).

[DigiByte-PoW] DigiByte Core Developers. *DigiSpeed 4.0.0 source code, functions GetNextWorkRequiredV3/4 in src/main.cpp as of commit 178e134*. URL: https://github.com/digibyte/digibyte/blob/178e1348a67d9624db328062397fde0de03fe388/src/main.cpp#L1587 (visited on 2017-01-20) (↑ p48).

[DSDCOPS2001] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Guiseppe Persiano, and Amit Sahai. "Robust Non-Interactive Zero Knowledge". In: *Advances in Cryptology - CRYPTO 2001. Proceedings of the 21st Annual International Cryptology Conference (Santa Barbara, California, USA, August 19–23, 2001)*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pages 566–598. ISBN: 978-3-540-42456-7. DOI: 10.1007/3-540-44647-8_33. URL: https://www.iacr.org/archive/crypto2001/21390566.pdf (visited on 2018-05-28) (↑ p19, 23).

[ECCZF2019] Electric Coin Company and Zcash Foundation. *Zcash Trademark Donation and License Agreement*. November 6, 2019. URL: https://www.zfnd.org/about/contracts/2019_ECC_ZFND_TM_agreement.pdf (visited on 2020-07-05) (↑ p13).

[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) (↑ p7).

[Gabizon2019] Ariel Gabizon. *On the security of the BCTV Pinocchio zk-SNARK variant*. Draft. February 5, 2019. URL: https://github.com/arielgabizon/bctv/blob/master/bctv.pdf (visited on 2019-02-07) (↑ p36, 37, 58, 62).

[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) (↑ p53).

[GWC2019] Ariel Gabizon, Zachary Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive: Report 2019/953. Last revised September 3, 2020. URL: https://eprint.iacr.org/2019/953 (visited on 2021-01-28) (↑ p58).

[Hamdon2018] Elise Hamdon. *Sapling Activation Complete*. Electric Coin Company blog. June 28, 2018. URL: https://electriccoin.co/blog/sapling-activation-complete/ (visited on 2021-01-10) (↑ p41).

[HW2016] Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the Zcash Protocol*. Electric Coin Company blog. April 26, 2016. URL: https://electriccoin.co/blog/fixing-zcash-vulns/ (visited on 2019-08-27). Updated December 26, 2017. (↑ p54).

[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) (↑ p36).

[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) (↑ p36, 56, 57).

[KYMM2018] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. *An Empirical Analysis of Anonymity in Zcash*. Preprint, to be presented at the 27th Usenix Security Syposium (Baltimore, Maryland, USA, August 15–17, 2018). May 8, 2018. URL: https://smeiklej.com/files/usenix18.pdf (visited on 2018-06-05) (↑ p6).

[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 (↑ p53).

[libsodium] *libsodium documentation*. URL: https://libsodium.org/ (visited on 2020-03-02) (↑ p34).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html (visited on 2016-02-01) (↑ p55).

[LM2017] Philip Lafrance and Alfred Menezes. *On the security of the WOTS-PRF signature scheme*. Cryptology ePrint Archive: Report 2017/938. Last revised February 5, 2018. URL: https://eprint.iacr.org/2017/938 (visited on 2018-04-16) (↑ p16).

[MAEÁ2010] 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 (Atlanta, Georgia, USA, August 23–25, 2010)*. IEEE, 2010, pages 1–4. ISBN: 978-1-4244-7407-3. 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) (↑ p55).

[Nakamoto2008] 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) (↑ p5).

[NIST2015] NIST. *FIPS 180-4: Secure Hash Standard (SHS)*. August 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: https://csrc.nist.gov/publications/detail/fips/180/4/final (visited on 2021-03-08) (↑ p29, 30, 38).

[Parno2015] Bryan Parno. *A Note on the Unsoundness of vnTinyRAM's SNARK*. Cryptology ePrint Archive: Report 2015/437. Received May 6, 2015. URL: https://eprint.iacr.org/2015/437 (visited on 2019-02-08) (↑ p36, 37, 58, 62).

[Peterson2017] Paige Peterson. *Transaction Linkability*. Electric Coin Company blog. January 25, 2017. URL: https://electriccoin.co/blog/transaction-linkability/ (visited on 2019-08-27) (↑ p6, 64).

[PHGR2013] Bryan Parno, Jon Howell, Craig Gentry, 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) (↑ p36).

[Quesnelle2017] Jeffrey Quesnelle. *On the linkability of Zcash transactions*. arXiv:1712.01210 [cs.CR]. December 4, 2017. URL: https://arxiv.org/abs/1712.01210 (visited on 2018-04-15) (↑ p6, 64).

[RFC-2119] Scott Bradner. *Request for Comments 7693: Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF). March 1997. URL: https://www.rfc-editor.org/rfc/rfc2119.html (visited on 2016-09-14) (↑ p5).

[RFC-7539] Yoav Nir and Adam Langley. *Request for Comments 7539: ChaCha20 and Poly1305 for IETF Protocols*. Internet Research Task Force (IRTF). May 2015. URL: https://www.rfc-editor.org/rfc/rfc7539.html (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). (↑ p32).

[RFC-8032] Simon Josefsson and Ilari Liusvaara. *Request for Comments 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)*. Internet Engineering Task Force (IETF). January 2017. URL: https://www.rfc-editor.org/rfc/rfc8032.html (visited on 2020-07-06). As modified by errata at https://www.rfc-editor.org/errata_search.php?rfc=8032 (visited on 2020-07-06). (↑ p34).

[RIPEMD160] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. *RIPEMD-160, a strengthened version of RIPEMD*. URL: http://homes.esat.kuleuven.be/~bosselae/ripemd160.html (visited on 2016-09-24) (↑ p38).

[ST1999] Tomas Sander and Amnon Ta-Shma. "Auditable, Anonymous Electronic Cash". In: *Advances in Cryptology - CRYPTO '99. Proceedings of the 19th Annual International Cryptology Conference (Santa Barbara, California, USA, August 15–19, 1999)*. Ed. by Michael Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pages 555–572. ISBN: 978-3-540-66347-8. DOI: 10.1007/3-540-48405-1_35. URL: https://link.springer.com/content/pdf/10.1007/3-540-48405-1_35.pdf (visited on 2018-06-05) (↑ p58, 64).

[SWB2019] Josh Swihart, Benjamin Winston, and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. February 5, 2019. URL: https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/ (visited on 2019-08-27) (↑ p37, 62).

[Swihart2018] Josh Swihart. *Overwinter Activated Successfully*. Electric Coin Company blog. June 26, 2018. URL: https://electriccoin.co/blog/overwinter-activated-successfully/ (visited on 2021-01-10) (↑ p41).

[Unicode] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2016. URL: http://www.unicode.org/versions/latest/ (visited on 2016-08-31) (↑ p37).

[vanSaberh2014] Nicolas van Saberhagen. *CryptoNote v 2.0*. Date disputed. URL: https://cryptonote.org/whitepaper.pdf (visited on 2016-08-17) (↑ p6).

[Vercauter2009] Frederik Vercauteren. *Optimal pairings*. Cryptology ePrint Archive: Report 2008/096. Last revised March 7, 2008. URL: https://eprint.iacr.org/2008/096 (visited on 2018-04-06). A version of this paper appeared in *IEEE Transactions of Information Theory*, Vol. 56, pages 455–461; IEEE, 2009. (↑ p35, 64).

[WCBTV2015] Zooko Wilcox, Alessandro Chiesa, Eli Ben-Sasson, Eran Tromer, and Madars Virza. *A Bug in libsnark*. Least Authority blog. May 16, 2015. URL: https://leastauthority.com/blog/a_bug_in_libsnark/ (visited on 2019-08-27) (↑ p36).

[WG2016] Zooko Wilcox and Jack Grigg. *Why Equihash?* Electric Coin Company blog. April 15, 2016. URL: https://electriccoin.co/blog/why-equihash/ (visited on 2019-08-27). Updated August 21, 2019. (↑ p47).

[Zaverucha2012] Gregory M. Zaverucha. *Hybrid Encryption in the Multi-User Setting*. Cryptology ePrint Archive: Report 2012/159. Received March 20, 2012. URL: https://eprint.iacr.org/2012/159 (visited on 2016-09-24) (↑ p56).

[Zcash-Blossom] Electric Coin Company. *Blossom*. December 11, 2019. URL: https://z.cash/upgrade/blossom/ (visited on 2021-01-10) (↑ p41).

[Zcash-Canopy] Electric Coin Company. *Canopy*. November 18, 2020. URL: https://z.cash/upgrade/canopy/ (visited on 2021-01-10) (↑ p41).

[Zcash-Heartwd] Electric Coin Company. *Heartwood*. July 16, 2020. URL: https://z.cash/upgrade/heartwood/ (visited on 2021-01-10) (↑ p41).

[Zcash-Issue2113] Simon Liu. *GitHub repository 'zcash/zcash': Issue 2113*. URL: https://github.com/zcash/zcash/issues/2113 (visited on 2017-02-20) (↑ p51, 67).

[Zcash-libsnark] *libsnark: C++ library for zkSNARK proofs (Zcash fork)*. URL: `https://github.com/zcash/zcash/tree/master/src/snark` (visited on 2018-02-04) (↑ p36).

[ZIP-32] Jack Grigg and Daira Hopwood. *Shielded Hierarchical Deterministic Wallets*. Zcash Improvement Proposal 32. URL: `https://zips.z.cash/zip-0032` (visited on 2019-08-28) (↑ p60).

[ZIP-76] Jack Grigg and Daira Hopwood. *Transaction Signature Validation before Overwinter*. Zcash Improvement Proposal 76 (in progress). (↑ p23, 52).

[ZIP-200] Jack Grigg. *Network Upgrade Mechanism*. Zcash Improvement Proposal 200. Created January 8, 2018. URL: `https://zips.z.cash/zip-0200` (visited on 2019-08-28) (↑ p41).

[ZIP-201] Simon Liu. *Network Peer Management for Overwinter*. Zcash Improvement Proposal 201. Created January 15, 2018. URL: `https://zips.z.cash/zip-0201` (visited on 2019-08-28) (↑ p41).

[ZIP-205] Daira Hopwood. *Deployment of the Sapling Network Upgrade*. Zcash Improvement Proposal 205. Created October 8, 2018. URL: `https://zips.z.cash/zip-0205` (visited on 2019-08-28) (↑ p41, 49).

[ZIP-206] Daira Hopwood. *Deployment of the Blossom Network Upgrade*. Zcash Improvement Proposal 206. Created July 29, 2019. URL: `https://zips.z.cash/zip-0206` (visited on 2019-08-28) (↑ p41).

[ZIP-209] Sean Bowe. *Prohibit Negative Shielded Value Pool Balances*. Zcash Improvement Proposal 209. Created February 25, 2019. URL: `https://zips.z.cash/zip-0209` (visited on 2020-11-05) (↑ p23, 59).

[ZIP-250] Daira Hopwood. *Deployment of the Heartwood Network Upgrade*. Zcash Improvement Proposal 250. Created February 28, 2020. URL: `https://zips.z.cash/zip-0250` (visited on 2020-03-20) (↑ p41).

[ZIP-251] Daira Hopwood. *Deployment of the Canopy Network Upgrade*. Zcash Improvement Proposal 251. Created February 28, 2020. URL: `https://zips.z.cash/zip-0251` (visited on 2020-03-24) (↑ p41).

[ZIP-302] Jay Graber and Jack Grigg. *Standardized Memo Field Format*. Zcash Improvement Proposal 302. Reserved. URL: `https://github.com/zcash/zips/pull/105` (visited on 2020-02-13) (↑ p37, 38, 60).

Index