Zcash Protocol Specification
Version 2018.0-beta-16

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

April 21, 2018

Abstract. **Zcash** is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash**, with security fixes and adjustments to terminology, functionality and performance. 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 attempts 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 and explains its differences from **Zerocash** and **Bitcoin**.

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

Contents 1

[†] Zerocoin Electric Coin Company

1 Introduction

Zcash is an implementation of the *Decentralized Anonymous Payment* scheme **Zerocash** [BCGGMTV2014], with some security fixes and adjustments to terminology, functionality and performance. 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 §7 ***'Differences from the Zerocash paper'*** on p. 46, 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 the *note* has been mined, it 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 contains 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 non-zero value, some revealed *note commitment* exists for that *note*.
- The prover knew the private *spending keys* of the input *notes*.
- The *nullifiers* and *note commitments* are computed correctly.
- The private *spending keys* of the input *notes* are cryptographically linked to a signature over the whole *transaction*, in such a way that the *transaction* cannot be modified by a party who did not know these private keys.
- Each output *note* is generated in such a way that 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 §7.1 '*Transaction Structure*' on p. 46 for differences.

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

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. $\{0, 1\}$.

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

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 $f_x(y)$.

$T^{[\ell]}$, 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 $\mathbb{BY}^{[k]}$ means the set of sequences of k bytes.

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

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

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

$S \cup T$ means the set union of S and T, or the type corresponding to it.

$S \cap T$ means the set intersection of S and T.

`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.

Where there is a need to make the distinction, we denote the unique representative of $a : \mathbb{F}_n$ in the range $\{0 \mathinner{..} n-1\}$ (or the unique representative of $a : \mathbb{F}_n^*$ in the range $\{1 \mathinner{..} n-1\}$) 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. 16) 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 or (equal-length) bit sequences 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 the appropriate 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. 16.

The convention of including a superscript $\star$ in 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. 25:

$\mathsf{MerkleDepth}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, ℓ_{value}, ℓ_{Merkle}, ℓ_{hSig}, ℓ_{PRF}, ℓ_{r}, ℓ_{Seed}, $\ell_{\mathsf{a_{sk}}}$, ℓ_{φ}, $\mathsf{MAX\_MONEY}$, $\mathsf{SlowStartInterval}$, $\mathsf{HalvingInterval}$, $\mathsf{MaxBlockSubsidy}$, $\mathsf{NumFounderAddresses}$, $\mathsf{PoWLimit}$, $\mathsf{PoWAveragingWindow}$, $\mathsf{PoWMedianBlockSpan}$, $\mathsf{PoWDampingFactor}$, and $\mathsf{PoWTargetSpacing}$.

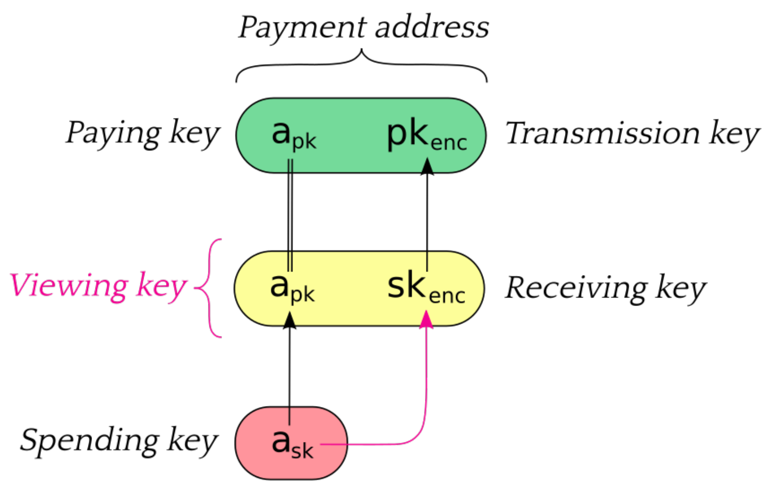
The bit sequence constant $\mathsf{Uncommitted} : \mathbb{B}^{[\ell_{\mathsf{Merkle}}]}$, and rational constants $\mathsf{FoundersFraction}$, $\mathsf{PoWMaxAdjustDown}$, and $\mathsf{PoWMaxAdjustUp}$ will also be defined in that section.

3 Concepts

3.1 Payment Addresses and Keys

Users who wish to receive payments under this scheme first generate a random *spending key* $\mathsf{a_{sk}}$.

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



The *receiving key* $\mathsf{sk_{enc}}$, the *incoming viewing key* $\mathsf{ivk} = (\mathsf{a_{pk}}, \mathsf{sk_{enc}})$, and the *shielded payment address* $\mathsf{addr_{pk}} = (\mathsf{a_{pk}}, \mathsf{pk_{enc}})$ are derived from $\mathsf{a_{sk}}$, as described in §4.2 ***'Key Components'*** on p. 17.

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 refer to the key used to encrypt a message in an asymmetric encryption scheme as the "*public key*". However, the public key used as the *transmission key* component of an address ($\mathsf{pk_{enc}}$) need not be publically distributed; it has the same distribution as the *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 §4.11 ***'In-band secret distribution'*** on p. 22), since an adversary would have to know $\mathsf{pk_{enc}}$ in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

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

Let $\mathsf{MAX\_MONEY}$ and ℓ_{PRF} be as defined in §5.3 ***'Constants'*** on p. 25.

Let $\mathsf{COMM}^{\mathsf{Sprout}}$ be as defined in §5.4.6.1 ***'Note Commitments'*** on p. 30.

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

- $a_{pk} : \mathbb{B}^{[\ell_{PRF}]}$ is the *paying key* of the recipient's *shielded payment address*;
- $v : \{0 .. \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*);
- $\rho : \mathbb{B}^{[\ell_{PRF}]}$ is used as input to $\mathsf{PRF}^{nf}_{a_{sk}}$ to derive the *nullifier* of the *note*;
- $r : \mathsf{COMM}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as defined in §4.1.7 ***'Commitment'*** on p. 15.

Let Note be the type of a *note*, i.e.

$$\mathsf{Note} := \mathbb{B}^{[\ell_{PRF}]} \times \{0 .. \mathsf{MAX\_MONEY}\} \times \mathbb{B}^{[\ell_{PRF}]} \times \mathsf{COMM}^{\mathsf{Sprout}}.\mathsf{Trapdoor}.$$

Creation of new *notes* is described in §4.4 ***'Sending Notes'*** on p. 19. When *notes* are sent, only a commitment (see §4.1.7 ***'Commitment'*** on p. 15) 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 *zero-knowledge proof* when the *note* is spent, to check that it exists on the *block chain*.

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

$$\mathsf{NoteCommitment}(\mathbf{n}) = \mathsf{COMM}^{\mathsf{Sprout}}_r(a_{pk}, v, \rho),$$

where $\mathsf{COMM}^{\mathsf{Sprout}}$ is instantiated in §5.4.6.1 ***' Note Commitments'*** on p. 30.

A *nullifier* (denoted nf) is derived from the ρ value of a *note* and the recipient's *spending key* a_{sk}. This computation uses a *Pseudo Random Function* (see §4.1.2 ***'Pseudo Random Functions'*** on p. 12), as described in §4.9 ***'Note Commitments and Nullifiers'*** on p. 21.

A *note* is spent by proving knowledge of (ρ, a_{sk}) in zero knowledge while publically disclosing its *nullifier* nf, allowing nf to be used to prevent double-spending.

3.2.1 Note Plaintexts and Memo Fields

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

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

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

Other fields are as defined in §3.2 ***'Notes'*** on p. 8.

Encodings are given in §5.5 ***'Encodings of Note Plaintexts and Memo Fields'*** on p. 32. The result of encryption forms part of a *transmitted note(s) ciphertext*. For further details, see §4.11 ***'In-band secret distribution'*** on p. 22.

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 §6.3 ***'Block Header'*** on p. 39).

A path from the root toward the leaves of the tree consisting of a sequence of one or 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 §6.4.5 ***'Definition of Work'*** on p. 43, of all *blocks* in each 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 value pool*, 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 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. 11);
- a *nullifier set* (§3.7 ***'Nullifier Sets'*** on p. 11).

Validation state associated with *transparent transfers*, 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 $\mathrm{N}^{\mathsf{old}}$ *notes* $\mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$ and *transparent* input $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$, and creates $\mathrm{N}^{\mathsf{new}}$ *notes* $\mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ and *transparent* output $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$. It is associated with a *JoinSplit statement* instance (§4.10.1 ***'JoinSplit Statement'*** on p. 21), for which it provides a *zk-SNARK proof*.

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

The total $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ value adds to, and the total $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ value subtracts from the *transparent value pool* of the containing *transaction*.

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

For each of the $\mathrm{N}^{\mathsf{old}}$ *shielded inputs*, a *nullifier* is revealed. This allows detection of double-spends as described in §3.7 ***'Nullifier Sets'*** on p. 11.

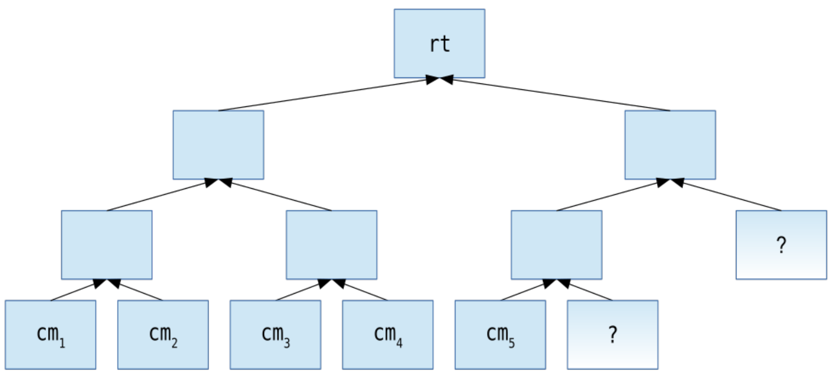
For each *JoinSplit description* in a *transaction*, an interstitial output *treestate* is constructed which adds the *note commitments* and *nullifiers* specified in that *JoinSplit description* to the input *treestate* referred to by its *anchor*. This interstitial output *treestate* is available for use as the *anchor* of subsequent *JoinSplit descriptions* in the same *transaction*.

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 *treestate* of a previous *block*.
- The *anchor* of each *JoinSplit description* in a *transaction* **MUST** refer to either some earlier *block*'s final *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*; see §3.4 ***'Transactions and Treestates'*** on p. 10.

Each *node* in the *incremental Merkle tree* is associated with a *hash value* of size ℓ_{Merkle} 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.

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*. It 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. 9.

These calculations are described in §6.5 ***'Calculation of Block Subsidy and Founders' Reward'*** on p. 43.

3.9 Coinbase Transactions

The first *transaction* in a block must be a *coinbase transaction*, which should collect and spend any *miner subsidy* and *transaction fees* paid by *transactions* included in this *block*. The *coinbase transaction* must also pay the *Founders' Reward* as described in §6.6 ***'Payment of Founders' Reward'*** on p. 44.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

Let MerkleDepth, ℓ_{Merkle}, ℓ_{Seed}, ℓ_{hSig}, and $\mathrm{N}^{\mathsf{old}}$ be as defined in §5.3 ***'Constants'*** on p. 25.

$\mathsf{MerkleCRH} : \mathbb{B}^{[\ell_{\mathsf{Merkle}}]} \times \mathbb{B}^{[\ell_{\mathsf{Merkle}}]} \to \mathbb{B}^{[\ell_{\mathsf{Merkle}}]}$ is a collision-resistant *hash function* used in §4.6 ***'Merkle path validity'*** on p. 20. It is instantiated in §5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 26.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a collision-resistant *hash function* used in §4.3 ***'JoinSplit Descriptions'*** on p. 18. It is instantiated in §5.4.1.4 '$\mathsf{h_{Sig}}$ ***Hash Function'*** on p. 27.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{N}^+ \to \mathbb{B}^{[n]}$ is another *hash function*, used in §6.4.1 ***'Equihash'*** on p. 41 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. 27.

4.1.2 Pseudo Random Functions

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

Let $\ell_{\mathsf{a_{sk}}}$, ℓ_{φ}, ℓ_{hSig}, ℓ_{PRF}, $\mathrm{N}^{\mathsf{old}}$, and $\mathrm{N}^{\mathsf{new}}$ be as defined in §5.3 ***'Constants'*** on p. 25.

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

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

These are used in §4.10.1 ***'JoinSplit Statement'*** on p. 21; $\mathsf{PRF^{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in §4.2 ***'Key Components'*** on p. 17.

They are instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 28.

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

Note: $\mathsf{PRF}^{\mathsf{nf}}$ was called $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash** [BCGGMTV2014].

4.1.3 Authenticated One-Time Symmetric Encryption

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

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

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

Security requirement: Sym must be one-time (INT-CTXT $\wedge$ IND-CPA)-secure [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 $\mathsf{KA.FormatPrivate} : \mathbb{B}^{[\ell_{\mathsf{PRF}}]} \to \mathsf{KA.Private}$ be a function to convert a bit string of length ℓ_{PRF} to a KA private key.

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

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

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

Security requirement: In addition to adaptive security of the key agreement and KDF, the following security property is required:

Let $\mathsf{g} := \mathsf{KA.Base}$.

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

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

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

1. Choose esk uniformly at random from $\mathsf{KA.Private}$.
2. Let $\mathsf{epk} := \mathsf{KA.DerivePublic}(\mathsf{esk}, \mathsf{g})$.
3. For $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$, let $\mathsf{K}_i := \mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{KA.Agree}(\mathsf{esk}, \mathsf{pk}^j_{\mathsf{enc}}), \mathsf{epk}, \mathsf{pk}^j_{\mathsf{enc}}))$.
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 $\mathsf{KA.DerivePublic}$ (which includes all keys generated as in §4.2 ***'Key Components'*** on p. 17). 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 $\mathsf{Sig.Private}$;
- a type of verifying keys $\mathsf{Sig.Public}$;
- a type of messages $\mathsf{Sig.Message}$;
- a type of signatures $\mathsf{Sig.Signature}$;
- a randomized signing key generation algorithm $\mathsf{Sig.GenPrivate} : () \xrightarrow{\mathbb{R}} \mathsf{Sig.Private}$;
- an injective verifying key derivation algorithm $\mathsf{Sig.DerivePublic} : \mathsf{Sig.Private} \to \mathsf{Sig.Public}$;
- a randomized signing algorithm $\mathsf{Sig.Sign} : \mathsf{Sig.Private} \times \mathsf{Sig.Message} \xrightarrow{\mathbb{R}} \mathsf{Sig.Signature}$;
- a verifying algorithm $\mathsf{Sig.Verify} : \mathsf{Sig.Public} \times \mathsf{Sig.Message} \times \mathsf{Sig.Signature} \to \mathbb{B}$;

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

Zcash uses two signature schemes:

- one used for signatures that can be verified by script operations such as `OP_CHECKSIG` and `OP_CHECKMULTISIG` as in **Bitcoin**;
- one called $\mathsf{JoinSplitSig}$ (instantiated in §5.4.5 ***'JoinSplit Signature'*** on p. 29), which is used to sign *transactions* that contain at least one *JoinSplit description*.

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.

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 §4.7 ***'Non-malleability'*** on p. 20), 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].

4.1.7 Commitment

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

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

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

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

Security requirements:

- **Computational hiding:** For all x, x' $:$ COMM.Input, the distributions $\{\ \mathsf{COMM}_r(x) \mid r \xleftarrow{\mathbb{R}} \mathsf{COMM.Trapdoor}\ \}$ and $\{\ \mathsf{COMM}_r(x') \mid r \xleftarrow{\mathbb{R}} \mathsf{COMM.Trapdoor}\ \}$ are computationally indistinguishable.
- **Computational binding:** It is infeasible to find x, x' $:$ COMM.Input and r, r' $:$ 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 $:$ COMM.Input and r, r' $:$ COMM.Trapdoor such that $r \neq r'$ and $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(x)$, this would not by itself contradict the computational binding security requirement.

Let ℓ_{r}, ℓ_{Merkle}, ℓ_{PRF}, and ℓ_{value} be as defined in §5.3 ***'Constants'*** on p. 25.

Define $\mathsf{COMM}^{\mathsf{Sprout}}.\mathsf{Trapdoor} := \mathbb{B}^{[\ell_{\mathsf{r}}]}$ and $\mathsf{COMM}^{\mathsf{Sprout}}.\mathsf{Output} := \mathbb{B}^{[\ell_{\mathsf{Merkle}}]}$.

Zcash uses a *note commitment scheme*

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

instantiated in §5.4.6.1 ***'Note Commitments'*** on p. 30.

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

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 $\mathsf{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\perp\}$, such that $\mathsf{abst}_{\mathbb{G}}$ is the left inverse of $\mathsf{repr}_{\mathbb{G}}$, i.e. for all $P \in \mathbb{G}$, $\mathsf{abst}_{\mathbb{G}}(\mathsf{repr}_{\mathbb{G}}(P)) = P$, and for all S not in the image of $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{abst}_{\mathbb{G}}(S) = \perp$.

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* $\mathbb{P}$ consists of:

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

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*.

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

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

Security requirements:

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

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

These definitions are derived from those in [BCTV2014, Appendix C], adapted to state concrete security for a fixed circuit, rather than asymptotic security for arbitrary circuits. (ZK.`Prove` corresponds to P, ZK.`Verify` corresponds to V, and 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 $\mathsf{ZK.}\mathtt{Verify}_{\mathsf{vk}}(x, \pi) = 1$ 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 $\mathsf{ZK.}\mathtt{Verify}_{\mathsf{vk}}(x, \pi) = 1$ without ***there existing*** an *auxiliary input* w such that $(x, w) \in \mathsf{ZK.SatisfyingInputs}$.

It is possible to replay proofs, but informally, a proof for a given (x, w) gives no information that helps to find a proof for other (x, w). TODO: Clarify this and/or switch to a proving system that provides Simulation Extractability.

The *proving system* is instantiated in §5.4.8.1 ***'PHGR13'*** on p. 31. ZKJoinSplit refers to this *proving system* with the BN-254 pairing, specialized to the *JoinSplit statement* given in §4.10.1 ***'JoinSplit Statement'*** on p. 21. 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 ***'zk-SNARK Parameters'*** on p. 35.

4.2 Key Components

Let $\mathsf{PRF}^{\mathsf{addr}}$ be a *Pseudo Random Function*, instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 28.

Let KA be a *key agreement scheme*, instantiated in §5.4.4.1 ***'Key Agreement'*** on p. 28.

A new *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:

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

4.3 JoinSplit Descriptions

A *JoinSplit transfer*, as specified in §3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 10, 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 verification key and signature.

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

where

- $\mathsf{v^{old}_{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent value pool*;
- $\mathsf{v^{new}_{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent value pool*;
- $\mathsf{rt} ⦂ \mathbb{B}^{[\ell_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in §3.3 ***'The Block Chain'*** on p. 9, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf}^{\mathsf{old}}_{1..N^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRF}}][N^{\mathsf{old}}]}$ is the sequence of *nullifiers* for the input *notes*;
- $\mathsf{cm}^{\mathsf{new}}_{1..N^{\mathsf{new}}} ⦂ \mathsf{COMM^{Sprout}.Output}^{[N^{\mathsf{new}}]}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} ⦂ \mathsf{KA.Public}$ is a key agreement public key, used to derive the key for encryption of the *transmitted notes ciphertext* (§4.11 ***'In-band secret distribution'*** on p. 22);
- $\mathsf{randomSeed} ⦂ \mathbb{B}^{[\ell_{\mathsf{Seed}}]}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h}_{1..N^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRF}}][N^{\mathsf{old}}]}$ is a sequence of tags that bind $\mathsf{h_{Sig}}$ to each $\mathsf{a_{sk}}$ of the input *notes*;
- $\pi_{\mathsf{ZKJoinSplit}} ⦂ \mathsf{ZKJoinSplit.Proof}$ is a *zk proof* with *primary input* $(\mathsf{rt}, \mathsf{nf}^{\mathsf{old}}_{1..N^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..N^{\mathsf{new}}}, \mathsf{v^{old}_{pub}}, \mathsf{v^{new}_{pub}}, \mathsf{h_{Sig}}, \mathsf{h}_{1..N^{\mathsf{old}}})$ for the *JoinSplit statement* defined in §4.10.1 ***'JoinSplit Statement'*** on p. 21;
- $\mathsf{C}^{\mathsf{enc}}_{1..N^{\mathsf{new}}} ⦂ \mathsf{Sym}.\mathbf{C}^{[N^{\mathsf{new}}]}$ is a sequence of ciphertext components for the encrypted output *notes*.

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

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

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

hSigCRH is instantiated in §5.4.1.4 ***'$\mathsf{h_{Sig}}$ Hash Function'*** on p. 27.

Consensus rules:

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

4.4 Sending Notes

In order to send *shielded* value, the sender constructs a *transaction* containing one or more *JoinSplit descriptions*. This 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 $\mathsf{randomSeed}$ uniformly at random on $\mathbb{B}^{[\ell_{\mathsf{Seed}}]}$, and selects the input *notes*. At this point there is sufficient information to compute $\mathsf{h_{Sig}}$, as described in the previous section. The sender also chooses φ uniformly at random on $\mathbb{B}^{[\ell_\varphi]}$. Then it creates each output *note* with index $i : \{1..\mathrm{N}^{\mathsf{new}}\}$ as follows:

- Choose uniformly random $\mathsf{r}_i^{\mathsf{new}} \xleftarrow{R} \mathsf{COMM}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$.
- Compute $\rho_i^{\mathsf{new}} = \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}})$.
- Compute $\mathsf{cm}_i^{\mathsf{new}} = \mathsf{COMM}^{\mathsf{Sprout}}_{\mathsf{r}_i^{\mathsf{new}}}(\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},i}, \mathsf{v}_i^{\mathsf{new}}, \rho_i^{\mathsf{new}})$.
- Let $\mathbf{np}_i = (\mathsf{v}_i^{\mathsf{new}}, \rho_i^{\mathsf{new}}, \mathsf{r}_i^{\mathsf{new}}, \mathsf{memo}_i)$.

$\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ are then encrypted to the recipient *transmission keys* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, giving the *transmitted notes ciphertext* $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$, as described in §4.11 *'In-band secret distribution'* on p. 22.

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 $\mathsf{dataToBeSigned} : \mathbb{BY}^{[\mathbb{N}]}$ as described in §4.7 *'Non-malleability'* on p. 20, 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 network.

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*.

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

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

A *dummy* 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 $\mathsf{MerkleDepth}$ (defined in §5.3 ***'Constants'*** on p. 25).

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

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

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

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

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

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

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

where

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

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

4.7 Non-malleability

Bitcoin defines several *SIGHASH types* that cover various parts of a transaction. In **Zcash**, all of these *SIGHASH types* are extended to cover the **Zcash**-specific fields `nJoinSplit`, `vJoinSplit`, and (if present) `joinSplitPubKey`, described in §6.1 ***'Encoding of Transactions'*** on p. 36. They ***do not*** cover the field `joinSplitSig`.

Consensus rule: If `nJoinSplit` > 0, the *transaction* **MUST NOT** use *SIGHASH types* other than SIGHASH_ALL.

Let $\mathsf{dataToBeSigned}$ be the hash of the *transaction* using the SIGHASH_ALL *SIGHASH type*. This ***excludes*** all of the `scriptSig` fields in the non-**Zcash**-specific parts of the *transaction*.

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

$\mathsf{JoinSplitSig}$ is instantiated in §5.4.5 ***'JoinSplit Signature'*** on p. 29.

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

Let $\mathsf{h_{Sig}}$ be computed as specified in §4.3 *'JoinSplit Descriptions'* on p. 18.

Let $\mathsf{PRF^{pk}}$ be as defined in §4.1.2 *'Pseudo Random Functions'* on p. 12.

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

The correctness of $\mathsf{h}_{1..\mathrm{N^{old}}}$ is enforced by the *JoinSplit statement* given in §4.10.1 *'Non-malleability'* on p. 22. This ensures that a holder of all of the $\mathsf{a^{old}_{sk,1..N^{old}}}$ for every *JoinSplit description* in the *transaction* has authorized the use of the private signing key corresponding to `joinSplitPubKey` to sign this *transaction*.

4.8 Balance

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

Unlike original **Zerocash** [BCGGMTV2014], **Zcash** does not have a distinction between Mint and Pour operations. The addition of $\mathsf{v^{old}_{pub}}$ 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. 18, either $\mathsf{v^{old}_{pub}}$ or $\mathsf{v^{new}_{pub}}$ **MUST** be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v^{old}_{pub}}$ and $\mathsf{v^{new}_{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v^{old}_{pub}}, \mathsf{v^{new}_{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

4.9 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. 11).

Each *note* has a ρ component.

Let $\mathsf{PRF^{nf}}$ be as instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 28.

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

4.10 Zk-SNARK Statement

4.10.1 JoinSplit Statement

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

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

the prover knows an *auxiliary input*:

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

where:

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

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforceMerklePath}_i = 1$: $(\mathsf{path}_i, \mathsf{pos}_i)$ is a valid *Merkle tree path* (see §4.6 ***'Merkle path validity'*** on p. 20) of depth MerkleDepth from $\mathsf{NoteCommitment}(\mathbf{n}^{\mathsf{old}}_i)$ to the *anchor* rt.

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

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

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

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

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

For details of the form and encoding of proofs, see §5.4.8.1 ***'PHGR13'*** on p. 31.

4.11 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 r. A *memo field* (§3.2.1 ***'Note Plaintexts and Memo Fields'*** on p. 9) is also transmitted.

To transmit these secrets securely to a recipient ***without*** requiring an out-of-band communication channel, the *transmission key* $\mathsf{pk_{enc}}$ 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 $\mathrm{N}^{\mathsf{new}}$ *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 *encryption scheme* instantiated in §5.4.3 ***'Authenticated One-Time Symmetric Encryption'*** on p. 28;
- let KDF be the *Key Derivation Function* instantiated in §5.4.4.2 ***'Key Derivation'*** on p. 29;
- let KA be the *key agreement scheme* instantiated in §5.4.4.1 ***'Key Agreement'*** on p. 28;
- let $\mathsf{h_{Sig}}$ be the value computed for this *JoinSplit description* in §4.3 ***'JoinSplit Descriptions'*** on p. 18.

4.11.1 Encryption

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

Let $\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ be the *note plaintexts* as defined in §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 32.

Then to encrypt:

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

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

Note: It is technically possible to replace $\mathsf{C}^{\mathsf{enc}}_i$ 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 *note* received out-of-band, which are not addressed in this document.

4.11.2 Decryption

Let $\mathsf{ivk} = (\mathsf{a_{pk}}, \mathsf{sk_{enc}})$ be the recipient's *incoming viewing key*, and let $\mathsf{pk_{enc}}$ be the corresponding *transmission key* derived from $\mathsf{sk_{enc}}$ as specified in §4.2 *'Key Components'* on p. 17.

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

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

let $\mathsf{sharedSecret}_i = \mathsf{KA.Agree}(\mathsf{sk_{enc}}, \mathsf{epk})$

let $\mathsf{K}^{\mathsf{enc}}_i = \mathsf{KDF}(i, \mathsf{h_{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk_{enc}})$

return $\mathtt{DecryptNote}(\mathsf{K}^{\mathsf{enc}}_i, \mathsf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i, \mathsf{a_{pk}})$.

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

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

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

extract $\mathbf{np}_i = (\mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i, \mathsf{memo}_i)$ from $\mathsf{P}^{\mathsf{enc}}_i$

if $\mathsf{NoteCommitment}((\mathsf{a_{pk}}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{r}^{\mathsf{new}}_i)) \neq \mathsf{cm}^{\mathsf{new}}_i$, return $\bot$, else return $\mathbf{np}_i$.

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

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 *block chain* is extended by new *transactions*. Also, *block chain* reorganizations can cause a node to switch to a different best *block chain* that does not contain the *transaction* in which a *note* was output.

See §7.7 *'In-band secret distribution'* on p. 49 for further discussion of the security and engineering rationale behind this encryption scheme.

4.12 Block Chain Scanning

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

Let $\mathsf{ivk} = (\mathsf{a_{pk}}, \mathsf{sk_{enc}})$ be the *incoming viewing key* corresponding to $\mathsf{a_{sk}}$, and let $\mathsf{pk_{enc}}$ be the associated *transmission key*, as specified in §4.2 *'Key Components'* on p. 17.

Initialize $\mathsf{ReceivedSet} : \mathscr{P}\big(\mathsf{Note} \times \mathsf{memo}\big) = \{\}$.

Initialize $\mathsf{SpentSet} : \mathscr{P}\big(\mathsf{Note}\big) = \{\}$.

Initialize $\mathsf{NullifierMap} : \mathbb{B}^{[\ell_{\mathsf{PRF}}]} \to \mathsf{Note}$ to the empty mapping.

For each *transaction* tx,

For each *JoinSplit description* in tx,

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

For i in $1..\mathrm{N}^{\mathsf{new}}$,

Attempt to decrypt the *transmitted note ciphertext* component $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}_i)$ using the algorithm in §4.11.2 *'Decryption'* on p. 23. If this succeeds giving $\mathbf{np}$:

Extract $\mathbf{n}$ and memo from $\mathbf{np}$ (taking the $\mathsf{a_{pk}}$ field of the *note* to be $\mathsf{a_{pk}}$ from ivk).

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

Calculate the nullifier nf of $\mathbf{n}$ using $\mathsf{a_{sk}}$ as described in §3.2 *'Notes'* on p. 8.

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

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 $\mathsf{NullifierMap}$, add $\mathsf{NullifierMap}(\mathsf{nf}_i)$ to $\mathsf{SpentSet}$.

Return $(\mathsf{ReceivedSet}, \mathsf{SpentSet})$.

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. §7.5 *'Internal hash collision attack and fix'* on p. 47

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:

$\mathsf{MerkleDepth} ⦂ \mathbb{N} := 29$

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

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

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

$\ell_{\mathsf{Merkle}} ⦂ \mathbb{N} := 256$

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

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

$\ell_{\mathsf{r}} ⦂ \mathbb{N} := 256$

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

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

$\ell_{\varphi} ⦂ \mathbb{N} := 252$

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

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

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

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

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

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

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

$$\mathsf{PoWLimit} ⦂ \mathbb{N} := \begin{cases} 2^{243} - 1, & \text{for the production network} \\ 2^{251} - 1, & \text{for the test network} \end{cases}$$

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

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

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

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

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

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

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

5.4.1.1 SHA-256 and SHA256Compress Hash Functions

SHA-256 is defined by [NIST2015].

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

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

[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".)

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

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

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

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

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

5.4.1.2 BLAKE2 Hash Function

BLAKE2 is defined by [ANWW2013]. **Zcash** uses only the $\mathsf{BLAKE2b}$ variant.

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

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

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

Note: $\mathsf{BLAKE2b}\text{-}\ell$ is not the same as $\mathsf{BLAKE2b\text{-}512}$ truncated to ℓ bits, because the digest length is encoded in the parameter block.

5.4.1.3 Merkle Tree Hash Function

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

Let $\mathsf{SHA256Compress}$ be as specified in §5.4.1.1 ***'SHA-256 and SHA256Compress Hash Functions'*** on p. 26.

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

$$\mathsf{MerkleCRH}(\mathsf{left}, \mathsf{right}) := \mathsf{SHA256Compress}\left(\begin{array}{|c|c|}\hline 256\text{-bit left} & 256\text{-bit right} \\ \hline\end{array}\right).$$

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

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

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. 18.

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

where

$\mathsf{hSigInput} :=$

256-bit randomSeed	256-bit $\mathsf{nf}^{\mathsf{old}}_1$ ...	256-bit $\mathsf{nf}^{\mathsf{old}}_{\mathrm{N}^{\mathsf{old}}}$	256-bit joinSplitPubKey

BLAKE2b-256(p, x) is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 26.

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 §6.4.1 ***'Equihash'*** on p. 41.

Let powtag :=

64-bit “ZcashPoW”	32-bit n	32-bit k

Let $\mathsf{powcount}(g)$:=

32-bit g

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

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

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

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

Indices of bits in T are 1-based.

BLAKE2b-$\ell(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 26.

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 (§6.4.1 ***'Equihash'*** on p. 41), 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

$\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nf}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ}, described in §4.1.2 ***'Pseudo Random Functions'*** on p. 12, are all instantiated using the *SHA-256 compression function* defined in §5.4.1.1 ***'SHA-256 and SHA256Compress Hash Functions'*** on p. 26:

$\mathsf{PRF}^{\mathsf{addr}}_{x}(t) := \mathsf{SHA256Compress}$	1	1	0	0	252-bit x	8-bit t	$[0]^{248}$
$\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho) := \mathsf{SHA256Compress}$	1	1	1	0	252-bit $\mathsf{a_{sk}}$	256-bit ρ	
$\mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}}) := \mathsf{SHA256Compress}$	0	i-1	0	0	252-bit $\mathsf{a_{sk}}$	256-bit $\mathsf{h_{Sig}}$	
$\mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}}) := \mathsf{SHA256Compress}$	0	i-1	1	0	252-bit φ	256-bit $\mathsf{h_{Sig}}$	

Security requirements:

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

Note: The first four bits –i.e. the most significant four bits of the first byte– are used to separate distinct uses of $\mathsf{SHA256Compress}$, 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 ***'Note Commitments'*** on p. 30.

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

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

Note: The "IETF" definition of AEAD_CHACHA20_POLY1305 from [RFC-7539] is used; this 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 Key Agreement

KA is a *key agreement scheme* as specified in §4.1.4 ***'Key Agreement'*** on p. 13.

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

Let KA.Public and KA.SharedSecret be the type of Curve25519 public keys (i.e. $\mathbb{B}^{\mathbb{Y}[32]}$), and let KA.Private be the type of Curve25519 secret keys.

Let $\mathsf{Curve25519}(\underline{n}, \underline{q})$ be the result of point multiplication of the Curve25519 public key represented by the byte sequence $\underline{q}$ by the Curve25519 secret key represented by the byte sequence $\underline{n}$, as defined in [Bernstein2006, section 2].

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

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

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

5.4.4.2 Key Derivation

KDF is a *Key Derivation Function* as specified in §4.1.5 ***'Key Derivation'*** on p. 13.

It is instantiated using BLAKE2b-256 as follows:

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

where:

kdftag :=

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

kdfinput :=

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

.

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 26.

5.4.5 JoinSplit Signature

JoinSplitSig is a *signature scheme* as specified in §4.1.6 ***'Signature'*** on p. 14.

It is instantiated as Ed25519 [BDLSY2012], with the additional requirements that:

- $\underline{S}$ **MUST** represent an integer less than the prime $\ell = 2^{252} + 27742317777372353535851937790883648493$;
- $\underline{R}$ **MUST** represent a point on the Ed25519 curve of order at least ℓ.

If these requirements are not met then the signature is considered invalid. Note that it is ***not*** required that the encoding of the y-coordinate in $\underline{R}$ is less than $2^{255} - 19$; also the order of the point represented by $\underline{R}$ is permitted to be greater than ℓ.

Ed25519 is defined as using SHA-512 internally.

A valid Ed25519 public key is defined as a point of order ℓ on the Ed25519 curve, in the encoding specified by [BDLSY2012]. Again, it is ***not*** required that the encoding of the y-coordinate of the public key is less than $2^{255} - 19$.

The encoding of a signature is:

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

where $\underline{R}$ and $\underline{S}$ are as defined in [BDLSY2012]. The encoding of a public key is as defined in [BDLSY2012].

5.4.6 Commitment schemes

5.4.6.1 Note Commitments

The commitment scheme $\mathsf{COMM}^{\mathsf{Sprout}}$ specified in §4.1.7 ***'Commitment'*** on p. 15 is instantiated using SHA-256 as follows:

$$\mathsf{COMM}^{\mathsf{Sprout}}_{\mathsf{r}}(\mathsf{a_{pk}}, \mathsf{v}, \rho) := \mathsf{SHA\text{-}256}\left(\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{\text{256-bit } \mathsf{a_{pk}}}\boxed{\text{64-bit } \mathsf{v}}\boxed{\text{256-bit } \rho}\boxed{\text{256-bit } \mathsf{r}} \right)$$

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

Security requirements:

- The *SHA-256 compression function* must be collision-resistant.
- The *SHA-256 compression function* must be a PRF when keyed by the bits corresponding to the position of r 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 $\mathbb{G}_1$ be the group of 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 $\mathbb{G}_2$ be the subgroup of order r in the sextic twist $E_{\mathbb{G}_2}$ of $\mathbb{G}_1$ 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 $\mathbb{G}_T$ be the subgroup of $r_{\mathbb{G}}{}^{\text{th}}$ roots of unity in $\mathbb{F}^*_{q_{\mathbb{G}}{}^{12}}$.

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

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

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

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

$\mathcal{P}_{\mathbb{G}_1}$ and $\mathcal{P}_{\mathbb{G}_2}$ are generators of the order-$r_{\mathbb{G}}$ subgroups of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

Define $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0 .. 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 24.

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

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

0	0	0	0	0	0	1	1-bit $\tilde{y}$	256-bit $\mathsf{I2BEBSP}_{256}(x)$

.

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

- 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:

- The use of big-endian order by $\mathsf{I2BEBSP}$ is different from the encoding of most other integers in this protocol. The encodings for $\mathbb{G}_{1,2}^*$ are consistent with the definition of $\mathsf{EC2OSP}$ for compressed curve points in [IEEE2004, section 5.5.6.2]. The LSB compressed form (i.e. $\mathsf{EC2OSP\text{-}XL}$) is used for points in $\mathbb{G}_1^*$, and the SORT compressed form (i.e. $\mathsf{EC2OSP\text{-}XS}$) for points in $\mathbb{G}_2^*$.
- The points at infinity $\mathcal{O}_{\mathbb{G}_{1,2}}$ never occur in proofs and have no defined encodings in this protocol.
- Testing $y > y'$ for the compression of $\mathbb{G}_2^*$ points is equivalent to testing whether $(a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})$ in lexicographic order.
- Algorithms for decompressing points from the above encodings are given in [IEEE2000, Appendix A.12.8] for $\mathbb{G}_1^*$, and [IEEE2004, Appendix A.12.11] for $\mathbb{G}_2^*$.
- 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 $\mathbb{G}_2^*$, by checking that $r_\mathbb{G} \cdot P = \mathcal{O}_{\mathbb{G}_2}$.

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 PHGR13

Zcash uses *zk-SNARKs* generated by its fork of *libsnark* [Zcash-libsnark] with the *proving system* described in [BCTV2015], which is a refinement of the systems in [PHGR2013] and [BCGTV2013].

A PHGR13 proof consists of a tuple $(\pi_A : \mathbb{G}_1^*,\ \pi'_A : \mathbb{G}_1^*,\ \pi_B : \mathbb{G}_2^*,\ \pi'_B : \mathbb{G}_1^*,\ \pi_C : \mathbb{G}_1^*,\ \pi'_C : \mathbb{G}_1^*,\ \pi_K : \mathbb{G}_1^*,\ \pi_H : \mathbb{G}_1^*)$. It is computed as described in [BCTV2015, Appendix B], using the pairing parameters specified in §5.4.7.1 ***'BN-254'*** on p. 30.

Note: Many details of the *proving system* are beyond the scope of this protocol document. For example, the *quadratic arithmetic program* verifying the *JoinSplit statement*, or its expression as a *Rank 1 Constraint System*, are not specified in this document. In practice it will be necessary to use the specific proving and verification keys generated for the **Zcash** production *block chain* (see §5.7 ***'zk-SNARK Parameters'*** on p. 35), and a *proving system* implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

Encoding of PHGR13 Proofs A PHGR13 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 [BCTV2015, Appendix B], the verifier **MUST** check, for the encoding of each element, that:

- the lead byte is of the required form;
- the remaining bytes encode a big-endian representation of an integer in $\{0 \mathinner{..} q_{\mathbb{S}}-1\}$ or (in the case of π_B) $\{0 \mathinner{..} {q_{\mathbb{S}}}^2-1\}$;
- the encoding represents a point in $\mathbb{G}_1^*$ or (in the case of π_B) $\mathbb{G}_2^*$, 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. 9, 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 $(\mathsf{v}, \rho, \mathsf{r}, \mathsf{memo})$.

memo is a 512-byte *memo field* associated with this *note*.

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

- a UTF-8 human-readable string [Unicode], padded by appending zero bytes; or
- an arbitrary sequence of 512 bytes starting with a byte value of 0xF5 or greater, which is therefore not a valid UTF-8 string.

In the former case, wallet software is expected to strip any trailing zero bytes and then display the resulting UTF-8 string to the recipient user, where applicable. Incorrect UTF-8-encoded byte sequences **SHOULD** be displayed as replacement characters (U+FFFD).

In the latter case, the contents of the *memo field* **SHOULD NOT** be displayed. A start byte of 0xF5 is reserved for use by automated software by private agreement. A start byte of 0xF6 followed by 511 0x00 bytes means "no memo". A start byte of 0xF6 followed by anything else, or a start byte of 0xF7 or greater, are reserved for use in future **Zcash** protocol extensions.

Other fields are as defined in §3.2 *'Notes'* on p. 8.

The encoding of a *note plaintext* consists of:

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

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

5.6 Encodings of Addresses and Keys

This section describes how **Zcash** encodes *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].

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

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

5.6.1 Transparent 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 the production network. (Addresses on the test network 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 public key hash

- Two bytes [`0x1C`, `0xB8`], indicating this version of the raw encoding of a P2PKH address on the production network. (Addresses on the test network use [`0x1D`, `0x25`] instead.)
- 20 bytes specifying a public key hash, which is a RIPEMD-160 hash [RIPEMD160] of a SHA-256 hash [NIST2015] of an uncompressed 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 the production network, 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].

5.6.2 Transparent Private Keys

These are encoded in the same way as in **Bitcoin** [Bitcoin-Base58], for both the production and test networks.

5.6.3 Shielded Payment Addresses

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{pk_{enc}}$ is a KA.Public key (see §5.4.4.1 ***'Key Agreement'*** on p. 28), for use with the encryption scheme defined in §4.11 ***'In-band secret distribution'*** on p. 22. These components are derived from a *spending key* as described in §4.2 ***'Key Components'*** on p. 17.

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

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

- Two bytes [0x16, 0x9A], indicating this version of the raw encoding of a **Zcash** *shielded payment address* on the production network. (Addresses on the test network 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 the production network, the lead bytes and encoded length cause the first two characters of the Base58Check encoding to be fixed as **"zc"**. For the test network, the first two characters are fixed as **"zt"**.

5.6.4 Incoming Viewing Keys

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{sk_{enc}}$ is a KA.Private key (see §5.4.4.1 ***'Key Agreement'*** on p. 28), for use with the encryption scheme defined in §4.11 ***'In-band secret distribution'*** on p. 22. These components are derived from a *spending key* as described in §4.2 ***'Key Components'*** on p. 17.

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

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

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

$\mathsf{sk_{enc}}$ **MUST** be "clamped" using KA.FormatPrivate as specified in §4.2 ***'Key Components'*** on p. 17. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{sk_{enc}} \neq \mathsf{KA.FormatPrivate}(\mathsf{sk_{enc}})$.

KA.FormatPrivate is defined in §5.4.4.1 ***'Key Agreement'*** on p. 28.

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

5.6.5 Spending Keys

A *spending key* consists of $\mathsf{a_{sk}}$, which is a sequence of 252 bits (see §4.2 *'Key Components'* on p. 17).

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

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

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

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

Notes:

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

5.7 zk-SNARK Parameters

For the **Zcash** production *block chain* and testnet, the SHA-256 hashes of the *proving key* and *verifying key* for the **Zcash** *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 [BGG2016].

6 Consensus Changes from Bitcoin

6.1 Encoding of Transactions

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

Version	Bytes	Name	Data Type	Description
≥ 1	4	`header`	`uint32`	Contains: · `fOverwintered` flag (bit 31) · `version` (bits 30 .. 0) – *transaction version*.
≥ 1	*Varies*	`tx_in_count`	`compactSize uint`	Number of *transparent* inputs in this *transaction*.
≥ 1	*Varies*	`tx_in`	`tx_in`	*Transparent* inputs, encoded as in **Bitcoin**.
≥ 1	*Varies*	`tx_out_count`	`compactSize uint`	Number of *transparent* outputs in this *transaction*.
≥ 1	*Varies*	`tx_out`	`tx_out`	*Transparent* outputs, encoded as in **Bitcoin**.
≥ 1	4	`lock_time`	`uint32`	A Unix epoch time (UTC) or *block height*, encoded as in **Bitcoin**.
≥ 2	*Varies*	`nJoinSplit`	`compactSize uint`	The number of *JoinSplit descriptions* in `vJoinSplit`.
≥ 2	1802· `nJoinSplit`	`vJoinSplit`	`JoinSplitDescription [nJoinSplit]`	A sequence of *JoinSplit descriptions* using PHGR13 proofs, each encoded as in §6.2 ***'Encoding of JoinSplit Descriptions'*** on p. 38.
≥ 2 †	32	`joinSplitPubKey`	`char[32]`	An encoding of a JoinSplitSig public verification key.
≥ 2 †	64	`joinSplitSig`	`char[64]`	A signature on a prefix of the *transaction* encoding, to be verified using `joinSplitPubKey`.

† The `joinSplitPubKey` and `joinSplitSig` fields are present if and only if `version` ≥ 2 and `nJoinSplit` > 0. The encoding of `joinSplitPubKey` and the data to be signed are specified in §4.7 ***'Non-malleability'*** on p. 20.

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 `version` $= 1$ or `nJoinSplit` $= 0$, then `tx_in_count` **MUST NOT** be 0.
- A *transaction* with one or more inputs from *coinbase transactions* **MUST** have no *transparent* outputs (i.e. `tx_out_count` **MUST** be 0). Note that inputs from *coinbase transactions* include *Founders' Reward* outputs.
- If `version` ≥ 2 and `nJoinSplit` > 0, then:
  - `joinSplitPubKey` **MUST** represent a valid Ed25519 public key encoding (§5.4.5 ***'JoinSplit Signature'*** on p. 29).
  - `joinSplitSig` **MUST** represent a valid signature under `joinSplitPubKey` of dataToBeSigned, as defined in §4.7 ***'Non-malleability'*** on p. 20.

- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions*.
- A *transaction* **MUST NOT** spend an output of a *coinbase transaction* (necessarily a *transparent* output) from a *block* less than 100 *blocks* prior to the spend. Note that outputs of *coinbase transactions* include *Founders' Reward* outputs.
- TODO: Other rules inherited from **Bitcoin**.

In addition, consensus rules associated with each *JoinSplit description* (§ 6.2 *'Encoding of JoinSplit Descriptions'* on p. 38) **MUST** 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 either 1 or 2 is not currently defined. Miners **MUST NOT** create *blocks* containing such *transactions*.
- 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*.
- 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*.

6.2 Encoding of JoinSplit Descriptions

An abstract *JoinSplit description*, as described in §3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 10, 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 $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ that the *JoinSplit transfer* removes from the *transparent value pool*.
8	`vpub_new`	`uint64`	A value $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ that the *JoinSplit transfer* inserts into the *transparent value pool*.
32	`anchor`	`char[32]`	A *root* rt of the *note commitment tree* at some *block height* in the past, or the *root* produced by a previous *JoinSplit transfer* in this *transaction*.
64	`nullifiers`	`char[32]` $[\mathrm{N}^{\mathsf{old}}]$	A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$.
64	`commitments`	`char[32]` $[\mathrm{N}^{\mathsf{new}}]$	A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$.
32	`ephemeralKey`	`char[32]`	A Curve25519 public key epk.
32	`randomSeed`	`char[32]`	A 256-bit seed that must be chosen independently at random for each *JoinSplit description*.
64	`vmacs`	`char[32]` $[\mathrm{N}^{\mathsf{old}}]$	A sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ binding $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the *JoinSplit description*, computed as described in §4.7 ***'Non-malleability'*** on p. 20.
296	`zkproof`	`char[296]`	An encoding of the *zero-knowledge proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see §5.4.8.1 ***'PHGR13'*** on p. 31).
1202	`encCiphertexts`	`char[601]` $[\mathrm{N}^{\mathsf{new}}]$	A sequence of ciphertext components for the encrypted output *notes*, $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$.

The `ephemeralKey` and `encCiphertexts` fields together form the *transmitted notes ciphertext*, which is computed as described in §4.11 ***'In-band secret distribution'*** on p. 22.

Consensus rules applying to a *JoinSplit description* are given in §4.3 ***'JoinSplit Descriptions'*** on p. 18.

6.3 Block Header

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

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`	`char[32]`	A *SHA-256d* hash in internal byte order of the previous *block*'s *header*. This ensures no previous *block* can be changed without also changing this *block*'s *header*.
32	`hashMerkleRoot`	`char[32]`	A *SHA-256d* hash in internal byte order. The merkle root is derived from the hashes of all *transactions* included in this *block*, ensuring that none of those *transactions* can be modified without modifying the *header*.
32	`hashReserved`	`char[32]`	A reserved field which should be ignored.
4	`nTime`	`uint32`	The *block time* 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`	`char[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 uint`	The size of an Equihash solution in bytes (always 1344).
1344	`solution`	`char[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 §6.4.3 *'Difficulty adjustment'* on p. 42, and let PoWMedianBlockSpan be the constant defined in §5.3 *'Constants'* on p. 25.

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 §6.4.2 *'Difficulty filter'* on p. 42.
- `solution` **MUST** represent a valid Equihash solution as defined in §6.4.1 *'Equihash'* on p. 41.
- `nTime` **MUST** be strictly greater than the median time of the previous PoWMedianBlockSpan *blocks*.
- 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*, and **SHOULD NOT** mine other blocks that chain to them.
- 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 uint`, 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 $\mathsf{PoWMedianBlockSpan}$ *blocks*. The Bitcoin Developer Reference [Bitcoin-Block] was previously in error on this point, but has now been corrected.

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 `char[32]`.
- The maximum *block* size has been doubled to 2000000 bytes.

6.4 Proof of Work

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

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

- The `solution` field encodes a *valid Equihash solution* according to §6.4.1 ***'Equihash'*** on p. 41.
- The *block header* satisfies the difficulty check according to §6.4.2 ***'Difficulty filter'*** on p. 42.

6.4.1 Equihash

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

The Equihash parameters for the production and test networks are $n = 200, k = 9$.

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

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

Let powheader :=

<table>
<tr><td>32-bit <code>nVersion</code></td><td colspan="2">256-bit <code>hashPrevBlock</code></td><td>256-bit <code>hashMerkleRoot</code></td></tr>
<tr><td colspan="2">256-bit <code>hashReserved</code></td><td>32-bit <code>nTime</code></td><td>32-bit <code>nBits</code></td><td>256-bit <code>nNonce</code></td></tr>
</table>

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. 27.

Define $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 24.

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

<table>
<tr><td colspan="21">$\mathsf{I2BEBSP}_{21}(68)$</td><td colspan="21">$\mathsf{I2BEBSP}_{21}(41)$</td><td colspan="21">$\mathsf{I2BEBSP}_{21}(2^{21} - 1)$</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="8">8-bit 0</td><td colspan="8">8-bit 2</td><td colspan="8">8-bit 32</td><td colspan="8">8-bit 0</td><td colspan="8">8-bit 10</td><td colspan="8">8-bit 127</td><td colspan="8">8-bit 255</td><td colspan="7">$\cdots$</td></tr>
</table>

and so the first 7 bytes of `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).

6.4.2 Difficulty filter

Let ToTarget be as defined in §6.4.4 *'nBits conversion'* on p. 43.

Difficulty is defined in terms of a *target threshold*, which is adjusted for each *block* according to the algorithm defined in §6.4.3 *'Difficulty adjustment'* on p. 42.

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`).

6.4.3 Difficulty adjustment

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 2.5-minute block time. Unlike **Bitcoin**, the difficulty adjustment occurs after every block.

The constants PoWLimit, PoWAveragingWindow, PoWMaxAdjustDown, PoWMaxAdjustUp, PoWDampingFactor, and PoWTargetSpacing are instantiated in §5.3 *'Constants'* on p. 25.

Let ToCompact and ToTarget be as defined in §6.4.4 *'nBits conversion'* on p. 43.

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 §6.3 *'Block Header'* on p. 39.

Define:

$$\mathsf{mean}(S) := \left(\sum_{i=1}^{\mathsf{length}(S)} S_i\right) \big/ \mathsf{length}(S).$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note: The convention used for the height parameters to MedianTime, ActualTimespan, ActualTimespanDamped, ActualTimespanBounded, MeanTarget, Threshold, and ThresholdBits is that these functions use only information from *blocks* ***preceding*** the given *block height*.

6.4.4 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:

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

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

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

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

6.4.5 Definition of Work

As explained in §3.3 ***'The Block Chain'*** on p. 9, 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 §6.4.4 ***'nBits conversion'*** on p. 43.

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

6.5 Calculation of Block Subsidy and Founders' Reward

§3.8 ***'Block Subsidy and Founders' Reward'*** on p. 12 defines the *block subsidy*, *miner subsidy*, and *Founders' Reward*. Their amounts in *zatoshi* are calculated from the *block height* using the formulae below. The constants SlowStartInterval, HalvingInterval, MaxBlockSubsidy, and FoundersFraction are instantiated in §5.3 ***'Constants'*** on p. 25.

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

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

$$\mathsf{Halving}(\mathsf{height}) := \mathsf{floor}\left(\frac{\mathsf{height} - \mathsf{SlowStartShift}}{\mathsf{HalvingInterval}}\right)$$

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

$$\mathsf{FoundersReward}(\mathsf{height}) := \begin{cases} \mathsf{BlockSubsidy}(\mathsf{height}) \cdot \mathsf{FoundersFraction}, & \text{if } \mathsf{height} < \mathsf{SlowStartShift} + \mathsf{HalvingInterval} \\ 0, & \text{otherwise} \end{cases}$$

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

6.6 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 the production network, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ 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 the test network, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ 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 the test network only, the addresses from index 4 onward have been changed from what was implemented at launch. This reflects an upgrade on the test network, starting from *block height* 53127. [Zcash-Issue2113]

Each address representation in FounderAddressList denotes a *transparent* P2SH multisig address.

Let SlowStartShift be defined as in the previous section.

Define:

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

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

Let RedeemScriptHash(height) be the standard redeem script hash, as defined in [Bitcoin-Multisig], for the P2SH multisig address with Base58Check representation given by $\mathsf{FounderAddressList}_{\,\mathsf{FounderAddressIndex}(\mathsf{height})}$.

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

Notes:

- No *Founders' Reward* is required to be paid for $\mathsf{height} \geq \mathsf{SlowStartShift} + \mathsf{HalvingInterval}$ (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 $\mathsf{height} \in \{1 \,..\, \mathsf{SlowStartShift} + \mathsf{HalvingInterval} - 1\}$ 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.

6.7 Changes to the Script System

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

6.8 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].

[BIP-34] applies to all blocks other than the **Zcash** *genesis block* (for which the "height in coinbase" was inadvertently omitted).

[BIP-13] applies with the changes to address version bytes described in §5.6.1 ***'Transparent Addresses'*** on p. 33.

[BIP-111] applies from 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).

7 Differences from the Zerocash paper

7.1 Transaction Structure

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

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

This allows for the possibility of chaining transfers of *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 *JoinSplit transfer* output *notes* 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.

TODO: Describe changes to signing.

7.2 Memo Fields

Zcash adds a *memo field* sent from the creator of a *JoinSplit description* to the recipient of each output *note*. This feature is described in more detail in §5.5 ***'Encodings of Note Plaintexts and Memo Fields'*** on p. 32.

7.3 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 $\mathrm{N}^{\mathsf{old}}$ *shielded notes* as input, and produces up to $\mathrm{N}^{\mathsf{new}}$ *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 §7.1 ***'Transaction Structure'*** on p. 46) consists only of *JoinSplit transfers*. A *JoinSplit transfer* is a Pour operation generalized to take a *transparent* UTXO as input, allowing *JoinSplit transfers* to subsume the functionality of Mints. An advantage of this is that a **Zcash** *transaction* that takes input from an UTXO can produce up to $\mathrm{N}^{\mathsf{new}}$ output *notes*, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the *JoinSplit transfer*: an unused (zero-value) input is indistinguishable from an input that takes value from a *note*.

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

7.4 Faerie Gold attack and fix

When a *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* ($\mathsf{a_{sk}}$) and ρ. The *note commitment* is derived from the recipient address component $\mathsf{a_{pk}}$, the value v, and the commitment trapdoor r, as well as ρ. However nothing prevents creating multiple *notes* with different v and r (hence different *note commitments*) but the same ρ.

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by $\mathsf{Receive}$ (as defined in [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. 19). The *nullifiers* are used as input to $\mathsf{hSigCRH}$ to derive a public value $\mathsf{h_{Sig}}$ which uniquely identifies the transaction, as described in §4.3 ***'JoinSplit Descriptions'*** on p. 18. ($\mathsf{h_{Sig}}$ was already used in **Zerocash** in a way that requires it to be unique in order to maintain indistinguishability of *JoinSplit descriptions*; adding the *nullifiers* to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the *transaction* creator is an adversary.)

The ρ value for each output *note* is then derived from a random private seed φ and $\mathsf{h_{Sig}}$ using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of ρ for each output *note* is enforced by §4.10.1 ***'Uniqueness of*** ρ_i^{new}***'*** on p. 22 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 $\mathsf{hSigCRH}$ 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 $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho)$, this is only possible if the adversary finds a collision (across both inputs) on $\mathsf{PRF}^{\mathsf{nf}}$, which is assumed to be infeasible — see §4.1.2 ***'Pseudo Random Functions'*** on p. 12.

Crucially, "*nullifier* integrity" is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note* (§4.10.1 ***'Nullifier integrity'*** on p. 22). 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.

7.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, v, and ρ. However, the instantiation of $\mathsf{COMM_r}$ and $\mathsf{COMM_s}$ in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of $\mathsf{a_{pk}}$ and ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct pairs $(\mathsf{a_{pk}}, \rho)$ and $(\mathsf{a_{pk}}', \rho')$ with colliding outputs of the truncated hash, and therefore the same *note commitment*. This would have allowed such an attacker to break the Balance property by double-spending *notes*, potentially creating arbitrary amounts of currency for themself [HW2016].

Zcash uses a simpler construction with a single SHA-256 evaluation for the commitment. The motivation for the nested construction in **Zerocash** was to allow Mint transactions to be publically verified without requiring a *zero-knowledge proof* ([BCGGMTV2014, section 1.3, under step 3]). Since **Zcash** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers*, and each transfer always uses a *zero-knowledge 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.

7.6 Changes to PRF inputs and truncation

The format of inputs to the PRFs instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 28 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 $\mathsf{PRF}^{\mathsf{nf}}$ in **Zcash**). Also, $\mathsf{h_{Sig}}$ is truncated from 256 to 253 bits in the input to $\mathsf{PRF}^{\mathsf{pk}}$. These truncations are not taken into account in the security proofs.

Both truncations affect the validity of the proof sketch for Lemma D.2 in the proof of Ledger Indistinguishability in [BCGGMTV2014, Appendix D].

In more detail:

- In the argument relating $\mathbf{H}$ and $\eth_2$, it is stated that in $\eth_2$, "for each $i \in \{1, 2\}, \mathsf{sn}_i := \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$ for a random (and not previously used) ρ". It is also argued that "the calls to $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$ are each by definition unique". The latter assertion depends on the fact that ρ is "not previously used". However, the argument is incorrect because the truncated input to $\mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}$, i.e. $[\rho]_{254}$, may repeat even if ρ does not.
- In the same argument, it is stated that "with overwhelming probability, $\mathsf{h_{Sig}}$ is unique". In fact what is required to be unique is the truncated input to $\mathsf{PRF}^{\mathsf{pk}}$, i.e. $[\mathsf{h_{Sig}}]_{253} = [\mathsf{CRH}(\mathsf{pk_{sig}})]_{253}$. In practice this value will be unique under a plausible assumption on CRH provided that $\mathsf{pk_{sig}}$ is chosen randomly, but no formal argument for this is presented.

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

As further evidence that it is essential for the proofs to explicitly take any such truncations into account, consider a slightly modified protocol in which ρ is truncated in the input to $\mathsf{COMM_r}$ but not in the input to $\mathsf{PRF}^{\mathsf{sn}}$. 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 §7.4 ***'Faerie Gold attack and fix'*** on p. 46, **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 $\mathsf{h_{Sig}}$ to the uses of ρ.

Since the PRFs are instantiated using SHA256Compress which has an input block size of 512 bits (of which 256 bits are used for the PRF input and 4 bits are used for domain separation), it was necessary to reduce the size of the

PRF key to 252 bits. The key is set to $\mathsf{a_{sk}}$ in the case of $\mathsf{PRF^{addr}}$, $\mathsf{PRF^{nf}}$, and $\mathsf{PRF^{pk}}$, and to φ (which does not exist in **Zerocash**) for PRF^{ρ}, and so those values have been reduced to 252 bits. This is preferable to requiring reasoning about truncation, and 252 bits is quite sufficient for security of these cryptovalues.

7.7 In-band secret distribution

Zerocash specified ECIES (referencing Certicom's SEC 1 standard) as the encryption scheme used for the in-band secret distribution. This has been changed to a 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 notes ciphertext* unless $\mathsf{pk_{enc}}$ is known or guessed.
- The KDF also takes a public seed $\mathsf{h_{Sig}}$ as input. This can be modeled as using a different "randomness extractor" for each *JoinSplit transfer*, which limits degradation of security with the number of *JoinSplit transfers*. This facilitates security analysis as explained in [DGKM2011] — see section 7 of that paper for a security proof that can be applied to this construction under the assumption that single-block BLAKE2b-256 is a "weak PRF". Note that $\mathsf{h_{Sig}}$ is authenticated, by the *zk-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 **Zcash** 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 $\mathsf{a_{sk}}$.

7.8 Omission in Zerocash security proof

The abstract **Zerocash** protocol requires $\mathsf{PRF^{addr}}$ 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 $\mathsf{PRF^{addr}}$ such that $\mathsf{a^1_{sk}}$ and $\mathsf{a^2_{sk}}$ 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 $\mathsf{cm^{old}_1} = \mathsf{cm^{old}_2}$, then the fact that $\mathsf{sn^{old}_1} \neq \mathsf{sn^{old}_2}$ implies that the witness a contains two distinct openings of $\mathsf{cm^{old}_1}$ (the first opening contains $(\mathsf{a^{old}_{sk,1}}, \rho^{\mathsf{old}}_1)$, while the second opening contains $(\mathsf{a^{old}_{sk,2}}, \rho^{\mathsf{old}}_2)$). This violates the binding property of the commitment scheme COMM."

In fact the openings do not contain $\mathsf{a^{old}_{sk,}}_i$; they contain $\mathsf{a^{old}_{\mathbf{pk},}}_i$. (In **Zcash** $\mathsf{cm}^{\mathsf{old}}_i$ opens directly to $(\mathsf{a^{old}_{\mathbf{pk},}}_i, \mathsf{v}^{\mathsf{old}}_i, \rho^{\mathsf{old}}_i)$, and in **Zerocash** it opens to $(\mathsf{v}^{\mathsf{old}}_i, \mathsf{COMM_s}(\mathsf{a^{old}_{\mathbf{pk},}}_i, \rho^{\mathsf{old}}_i))$.)

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

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

The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of $\mathsf{PRF^{addr}}$ (on both its arguments) to argue that distinctness of $\mathsf{a^{old}_{sk,1}}$ and $\mathsf{a^{old}_{sk,2}}$, together with constraint 1(b) of the *JoinSplit statement* (see §4.10.1 ***'Spend authority'*** on p. 22), implies distinctness of $\mathsf{a^{old}_{pk,1}}$ and $\mathsf{a^{old}_{pk,2}}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

7.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a_{pk}}, \mathsf{pk_{enc}}), \mathsf{v}, \rho, \mathsf{r}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines it as $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{r})$. The instantiation of $\mathsf{COMM_s}$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of $\mathsf{COMM^{Sprout}}$ in **Zcash**. $\mathsf{pk_{enc}}$ is also not needed as part of a *note*: it is not an input to $\mathsf{COMM^{Sprout}}$ 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 ***'PHGR13'*** on p. 31, 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.)

8 Acknowledgements

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

The authors would like to thank everyone with whom they have discussed the **Zerocash** protocol design; in addition to the inventors, this includes Mike Perry, Isis Lovecruft, Leif Ryge, Andrew Miller, Zooko Wilcox, Samantha Hulsey, Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, Alison Stevenson, John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, Jack Gavigan, Filippo Valsorda, Zaki Manian, George Tankersley, Tracy Hu, Brian Warner, Mary Maller, and no doubt others.

Zcash has benefited from security audits performed by NCC Group, Coinspect, and Least Authority.

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 §7.6 ***'Changes to PRF inputs and truncation'*** on p. 48 were also found by Daira Hopwood.

9 Change History

2018.0-beta-16

- 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 network protocol version 170004.
- Give references [Vercauter2009] and [AKLGL2010] for the optimal ate pairing.
- Give references for BN [BN2005] curves.
- Define KA.DerivePublic for Curve25519.
- Caveat the claim about *note traceability set* in §1.2 ***'High-level Overview'*** on p. 4 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 verifying keys independent of key pair generation.
- Add acknowledgements for Brian Warner, Mary Maller, and the Least Authority audit.
- `Makefile` improvements.

2018.0-beta-15

- 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

- Only cosmetic changes to **Sprout**.

2018.0-beta-13

- Only cosmetic changes to **Sprout**.

2018.0-beta-12

- No changes to **Sprout**.

2018.0-beta-11

- No changes to **Sprout**.

2018.0-beta-10

- 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. 24.

2018.0-beta-9

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

2018.0-beta-8

- No changes to **Sprout**.

2018.0-beta-7

- 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

- No changes to **Sprout**.

2018.0-beta-5

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

2018.0-beta-4

- No changes to **Sprout**.

2018.0-beta-3

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

2017.0-beta-2.9

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

2017.0-beta-2.8

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

2017.0-beta-2.7

- 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

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

2017.0-beta-2.5

- Clarify the consensus rule preventing double-spends.
- Clarify what a *note commitment* opens to in §7.8 *'Omission in **Zerocash** security proof'* on p. 50.
- Correct the order of arguments to COMM in §5.4.6.1 *'Note Commitments'* on p. 30.
- Correct a statement about indistinguishability of *JoinSplit descriptions*.
- Change the *Founders' Reward* addresses, for the test network only, to reflect the hard-fork upgrade described in [Zcash-Issue2113].

2017.0-beta-2.4

- 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

- Specify the security requirements on the *SHA-256 compression* function in order for the scheme in §5.4.6.1 '***Note Commitments***' on p. 30 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

- 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 [BGG2016].

2017.0-beta-2.1

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

2017.0-beta-2

- 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 chain.

2016.0-beta-1.13

- 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. 17.

2016.0-beta-1.12

- 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

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

2016.0-beta-1.10

- 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

- Add *Founders' Reward* addresses for the production network.
- Change "*protected*" terminology to "*shielded*".

2016.0-beta-1.8

- 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 signature hashes.
- Change the representation type of `vpub_old` and `vpub_new` to `uint64`. (This is not a consensus change because the type of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ 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

- 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

- 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 $\mathsf{BLAKE2b}\text{-}\ell$ is different from truncated $\mathsf{BLAKE2b}\text{-}512$.
- Clarify a note about SU-CMA security for signatures.
- Add a note about $\mathsf{PRF}^{\mathsf{nf}}$ corresponding to $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash**.
- Add a paragraph about key length in §7.7 ***'In-band secret distribution'*** on p. 49.
- Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

2016.0-beta-1.5

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

2016.0-beta-1.4

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

2016.0-beta-1.3

- Correct the omission of `solutionSize` from the *block header* format.
- Document that `compactSize uint` 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

- Remove $\mathsf{GeneralCRH}$ in favour of specifying $\mathsf{hSigCRH}$ and $\mathsf{EquihashGen}$ directly in terms of $\mathsf{BLAKE2b}\text{-}\ell$.
- Correct the security requirement for $\mathsf{EquihashGen}$.

2016.0-beta-1.1

- 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

- 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 $\mathsf{Ed25519}$, with consequent changes to the computation of $\mathsf{h_{Sig}}$.
- 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 $\mathsf{v^{old}_{pub}}$ and $\mathsf{v^{new}_{pub}}$.

- Clarify cryptographic security requirements and added definitions relating to the in-band secret distribution.
- Add various citations: the “Fixing Vulnerabilities in the Zcash Protocol” and “Why Equihash?” blog posts, several crypto papers for security definitions, the **Bitcoin** whitepaper, the **CryptoNote** whitepaper, and several references to **Bitcoin** documentation.
- Reference the extended version of the **Zerocash** paper rather than the Oakland proceedings version.
- Add *JoinSplit transfers* to the Concepts section.
- Add a section on Coinbase Transactions.
- Add acknowledgements for Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, and Jake Tarren.
- Fix a `Makefile` compatibility problem with the escaping behaviour of `echo`.
- Switch to `biber` for the bibliography generation, and add backreferences.
- Make the date format in references more consistent.
- Add visited dates to all URLs in references.
- Terminology changes.

2016.0-alpha-3.1

- Change main font to Quattrocento.

2016.0-alpha-3

- Change version numbering convention (no other changes).

2.0-alpha-3

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

2.0-alpha-2

- Change from truncated BLAKE2b-512 to BLAKE2b-256.
- Clarify endianness, and that uses of BLAKE2b are unkeyed.
- Minor correction to what *SIGHASH types* cover.
- Add “as intended for the **Zcash** release of summer 2016" to title page.
- Require $\mathsf{PRF}^{\mathsf{addr}}$ to be collision-resistant (see §7.8 *‘Omission in **Zerocash** security proof’* on p. 50).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in §4.10.1 *‘Merkle path validity’* on p. 22 about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

2.0-alpha-1

- First version intended for public review.

10 References

[ABR1999] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem*. Cryptology ePrint Archive: Report 1999/007. Received March 17, 1999. September 1998. URL: `https://eprint.iacr.org/1999/007` (visited on 2016-08-21) (↑ p13, 49).

[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) (↑ p30, 51).

[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) (↑ p26).

[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. (↑ p49).

[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. (↑ p4, 5, 6, 13, 21, 22, 23, 46, 47, 48, 50).

[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. (↑ p31).

[BCTV2014] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves (extended version)". In: *Advances in Cryptology - CRYPTO 2014*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pages 276–294. URL: `https://www.cs.tau.ac.il/~tromer/papers/scalablezk-20140803.pdf` (visited on 2016-09-01) (↑ p17).

[BCTV2015] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Cryptology ePrint Archive: Report 2013/879. Last revised May 19, 2015. URL: `https://eprint.iacr.org/2013/879` (visited on 2016-08-21) (↑ p31, 32).

[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) (↑ p15).

[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. (↑ p13).

[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. (↑ p29).

[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. (↑ p49).

[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. (↑ p13, 28, 29, 34, 49).

[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) (↑ p35).

[BGG2016] Sean Bowe, Ariel Gabizon, and Matthew Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. November 24, 2016. URL: https://github.com/zcash/mpc/blob/master/whitepaper.pdf (visited on 2017-02-11) (↑ p35, 54).

[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 2016-10-02) (↑ p45).

[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 2016-09-24) (↑ p33, 45).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[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 2016-09-24) (↑ p33).

[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 2016-10-02) (↑ p45, 55).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[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 2016-09-05) (↑ p15).

[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 2016-10-02) (↑ p45).

[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 2016-10-02) (↑ p45).

[BIP-68] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. *Relative lock-time using consensus-enforced sequence numbers*. Bitcoin Improvement Proposal 68. Last revised November 21, 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki` (visited on 2016-09-02) (↑ p37).

[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 2018-04-02) (↑ p45, 51).

[Bitcoin-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: `https://en.bitcoin.it/wiki/Base58Check_encoding` (visited on 2016-01-26) (↑ p33).

[Bitcoin-Block] *Block Headers — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#block-headers` (visited on 2017-04-25) (↑ p40).

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

[Bitcoin-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#raw-transaction-format` (visited on 2016-03-15) (↑ p37).

[Bitcoin-Multisig] *P2SH multisig (definition) — Bitcoin Developer Guide*. URL: `https://bitcoin.org/en/developer-guide#term-p2sh-multisig` (visited on 2016-08-19) (↑ p45).

[Bitcoin-nBits] *Target nBits — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#target-nbits` (visited on 2016-08-13) (↑ p39, 43).

[Bitcoin-P2PKH] *P2PKH (definition) — Bitcoin Developer Guide*. URL: `https://bitcoin.org/en/developer-guide#term-p2pkh` (visited on 2016-08-24) (↑ p33).

[Bitcoin-P2SH] *P2SH (definition) — Bitcoin Developer Guide*. URL: `https://bitcoin.org/en/developer-guide#term-p2sh` (visited on 2016-08-24) (↑ p33).

[Bitcoin-Protocol] *Protocol documentation — Bitcoin Wiki*. URL: `https://en.bitcoin.it/wiki/Protocol_documentation` (visited on 2016-10-02) (↑ p5).

[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) (↑ p6, 40, 55).

[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) (↑ p30, 51).

[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) (↑ p13).

[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) (↑ p49).

[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) (↑ p42).

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

[GGM2016] Christina Garman, Matthew Green, and Ian Miers. *Accountable Privacy for Decentralized Anonymous Payments*. Cryptology ePrint Archive: Report 2016/061. Last revised January 24, 2016. URL: https://eprint.iacr.org/2016/061 (visited on 2016-09-02) (↑ p47).

[HW2016] Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the Zcash Protocol*. Zcash blog. April 26, 2016. URL: https://blog.z.cash/fixing-zcash-vulns/ (visited on 2018-04-15). Updated December 26, 2017. (↑ p48).

[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) (↑ p31).

[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) (↑ p31, 49, 50).

[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 (↑ p46).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html (visited on 2016-02-01) (↑ p49).

[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) (↑ p15).

[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) (↑ p49).

[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) (↑ p4).

[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 2018-02-14) (↑ p26, 33).

[Peterson2017] Paige Peterson. *Transaction Linkability*. Zcash blog. January 25, 2017. URL: https://blog.z.cash/transaction-linkability/ (visited on 2018-04-15) (↑ p5, 51).

[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) (↑ p31).

[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) (↑ p5, 51).

[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://tools.ietf.org/html/rfc2119` (visited on 2016-09-14) (↑ p4).

[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://tools.ietf.org/html/rfc7539` (visited on 2016-09-02). As modified by verified errata at `https://www.rfc-editor.org/errata_search.php?rfc=7539` (visited on 2016-09-02). (↑ p28).

[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) (↑ p33).

[Unicode] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2016. URL: `http://www.unicode.org/versions/latest/` (visited on 2016-08-31) (↑ p32).

[vanSaberh2014] Nicolas van Saberhagen. *CryptoNote v 2.0*. Date disputed. URL: `https://cryptonote.org/whitepaper.pdf` (visited on 2016-08-17) (↑ p5).

[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. (↑ p30, 51).

[WG2016] Zooko Wilcox and Jack Grigg. *Why Equihash?* Zcash blog. April 15, 2016. URL: `https://blog.z.cash/why-equihash/` (visited on 2018-04-15). Updated December 14, 2017. (↑ p40).

[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) (↑ p49).

[Zcash-Issue2113] Simon Liu. *GitHub repository 'zcash/zcash': Issue 2113*. URL: `https://github.com/zcash/zcash/issues/2113` (visited on 2017-02-20) (↑ p45, 53).

[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) (↑ p31).