Zcash Protocol Specification

Version 2018.0-beta-15 [Overwinter+Sapling]

Daira Hopwood[†]

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

March 19, 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 *draft specification defines the next upgrade of the* ***Zcash*** *consensus protocol, codenamed* ***Overwinter****, and the subsequent upgrade, codenamed* ***Sapling****. It is a work in progress and should not be used as a reference for the current protocol.*

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** [BCG+2014], with some security fixes and adjustments to terminology, functionality and performance. It bridges the existing ***transparent*** payment scheme used by **Bitcoin** [Naka2008] with a ***shielded*** payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*).

Changes from the original **Zerocash** are explained in §8 *'Differences from the Zerocash paper'* on p. 70, and highlighted in magenta throughout the document. Changes specific to the **Overwinter** upgrade (which are also changes from **Zerocash**) are highlighted in blue. Changes specific to the **Sapling** upgrade following **Overwinter** (which are also changes from **Zerocash**) are highlighted in green. The name **Sprout** is used for the **Zcash** protocol prior to **Sapling** (both before and after **Overwinter**).

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**, **SHOULD NOT**, **MAY**, and **RECOMMENDED** 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;
- Network Upgrades — the strategy for upgrading from **Sprout** to **Overwinter** and then **Sapling**;
- Consensus Changes from **Bitcoin** — how **Zcash** differs from **Bitcoin** at the consensus layer, including the Proof of Work;
- Differences from the **Zerocash** protocol — a summary of changes from the protocol in [BCG+2014].
- Appendix: Circuit Design — details of how the **Sapling** circuit is defined as a Quadratic Arithmetic Program.

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. This overview applies to both **Sprout** and **Sapling**, differences in the cryptographic constructions used notwithstanding.

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 (indirectly) 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*, and a *nullifier*[1] (so that there is a 1:1:1 relation between *notes*, *note commitments*, and *nullifiers*). Computing the *nullifier* requires the associated private *spending key* (or the *nullifier deriving key* for **Sapling** *notes*). It is infeasible to correlate the *note commitment* with the corresponding *nullifier* without knowledge of at least this 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. TODO: The "1:1:1" part isn't correct for **Sapling**.

A *transaction* can contain *transparent* inputs, outputs, and scripts, which all work as in **Bitcoin** [Bitc-Protocol]. It also includes *JoinSplit descriptions*, *Spend descriptions*, and *Output descriptions*. Together these describe *shielded transfers* which take in *shielded input notes*, and/or produce *shielded output notes*. (For **Sprout**, each *JoinSplit description* handles up to two *shielded inputs* and up to two *shielded outputs*. For **Sapling**, each *shielded input* or *shielded output* has its own description.) It is also possible for value to be transferred between the *transparent* and *shielded* domains.

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). A *transaction* also includes computationally sound *zk-SNARK* proofs, which prove that all of the following hold except with insignificant probability:

For each *shielded input*,

- [**Sapling** onward] there is a revealed *value commitment* to the same value as the input *note*;
- some revealed *note commitment* exists for the input *note*;
- the prover knew the *proof authorizing key* of the input *note*;
- the *nullifier* and *note commitment* are computed correctly.

and for each *shielded output*,

- [**Sapling** onward] there is a revealed *value commitment* to the same value as the output *note*;
- the *note commitment* is computed correctly;
- the 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*.

For **Sprout**, the *JoinSplit statement* also includes an explicit balance check. For **Sapling**, the *value commitments* corresponding to the inputs and outputs are checked to balance (together with any net *transparent* input or output) outside the *zk-SNARK*.

In addition, various measures (differing between **Sprout** and **Sapling**) are used to ensure that the *transaction* cannot be modified by a party not authorized to do so.

Outside the *zk-SNARK*, it is 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 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

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

to the *transaction* in which it was created. That is, from an adversary's point of view the set of possibilities for a given *note* input to a *transaction*—its *note traceability set*— includes ***all*** previous notes that the adversary does not control or know to have been spent. This contrasts with other proposals for private payment systems, such as CoinJoin [Bitc-CoinJoin] or **CryptoNote** [vanS2014], that are based on mixing of a limited number of transactions and that therefore have smaller *note traceability sets*.

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

2 Notation

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

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

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

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

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

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

$x \,⦂\, T \mapsto e_x \,⦂\, U$ means the function of type $T \to U$ mapping formal parameter x to e_x (an expression depending on x). The types T and U are always explicit.

$\mathscr{P}\big(T\big)$ means the powerset of T.

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

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

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

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

$\{x \,⦂\, T \mid p(x)\}$ means the subset of x from T for which $p(x)$ holds.

$S \cup T$ 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.

$S \setminus T$ means the set difference obtained by removing elements in T from S, i.e. $\{x \,⦂\, S \mid x \notin T\}$.

`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. $[1]^\ell$ means the sequence of ℓ one 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 [BCG+2014] 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 $\{0\,..\,n-1\}$ (or the unique representative of $a : \mathbb{F}_n^*$ in the range $\{1\,..\,n-1\}$) as $a \bmod n$. Conversely, we denote the element of $\mathbb{F}_n$ corresponding to an integer $k : \mathbb{Z}$ as $k \pmod n$. We also use the latter notation in the context of an equality $k = k' \pmod n$ as shorthand for $k \bmod n = k' \bmod n$, and similarly $k \neq k' \pmod n$ as shorthand for $k \bmod n \neq k' \bmod n$. (When referring to constants such as 0 and 1 it is usually not necessary to make the distinction between field elements and their representatives, since the meaning is normally clear from context.)

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

$a + b$ means the sum of a and b. This may refer to addition of integers, rationals, finite field elements, or group elements (see §4.1.8 ***'Represented Group'*** on p. 21) 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.

$b \;?\; x : y$ means x when $b = 1$, or y when $b = 0$.

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. 21.

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. 36:

> $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, $\ell_{\mathsf{MerkleSprout}}$, $\ell_{\mathsf{MerkleSapling}}$, ℓ_{hSig}, ℓ_{PRF}, ℓ_{rcm}, ℓ_{Seed}, $\ell_{\mathsf{a_{sk}}}$, ℓ_{φ}, ℓ_{sk}, ℓ_{d}, ℓ_{ivk}, $\mathsf{MAX\_MONEY}$, $\mathsf{SlowStartInterval}$, $\mathsf{HalvingInterval}$, $\mathsf{MaxBlockSubsidy}$, $\mathsf{NumFounderAddresses}$, $\mathsf{PoWAveragingWindow}$, $\mathsf{PoWLimit}$, $\mathsf{PoWMedianBlockSpan}$, $\mathsf{PoWDampingFactor}$, $\mathsf{PoWTargetSpacing}$.

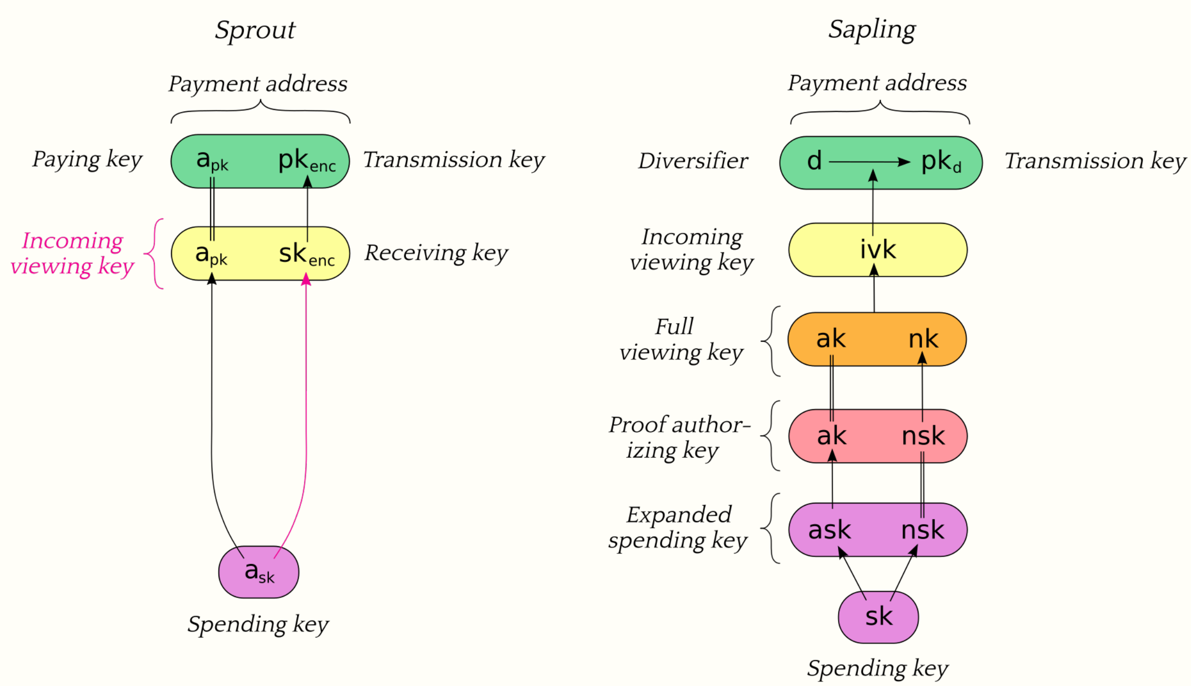
The bit sequence constants $\mathsf{Uncommitted}^{\mathsf{Sprout}} \,⦂\, \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]}$ and $\mathsf{Uncommitted}^{\mathsf{Sapling}} \,⦂\, \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$, and rational constants $\mathsf{FoundersFraction}$, $\mathsf{PoWMaxAdjustDown}$, and $\mathsf{PoWMaxAdjustUp}$ 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*. In **Sprout** this is called $\mathsf{a_{sk}}$ and in **Sapling** it is called sk.

The following diagram depicts the relations between key components in **Sprout** and **Sapling**. Arrows point from a component to any other component(s) that can be derived from it.



[**Sprout**] 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 §4.2.1 ***'Sprout** Key Components'* on p. 24.

[**Sapling** onward] The *spend authorizing key* ask, the *proof authorizing key* $(\mathsf{ak}, \mathsf{nsk})$, the *full viewing key* $(\mathsf{ak}, \mathsf{nk})$, the *incoming viewing key* ivk, and each *diversified payment address* $\mathsf{addr_d} = (\mathsf{d}, \mathsf{pk_d})$ are derived from sk, as described in §4.2.2 ***'Sapling** Key Components'* on p. 24.

The composition of *shielded payment addresses*, *incoming viewing keys*, *full 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* or *full 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.

[**Sapling** onward] **Sapling** provides a mechanism to allow the efficient creation of *diversified payment addresses* with the same spending authority. A group of such addresses shares the same *full viewing key* and *incoming viewing key*, and so creating as many unlinkable addresses as needed does not increase the cost of scanning the *block chain* for relevant *transactions*.

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 ($\mathsf{pk_{enc}}$ or $\mathsf{pk_d}$) need not be publically distributed; it has the same distribution as the *shielded payment address* itself. As mentioned above, limiting the distribution of the *shielded payment address* is important for some use cases. This also helps to reduce reliance of the overall protocol on the security of the cryptosystem used for *note* encryption (see §4.12 ***'In-band secret distribution'*** on p. 34), since an adversary would have to know $\mathsf{pk_{enc}}$ or some $\mathsf{pk_d}$ in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

A *note* (denoted $\mathbf{n}$) can be a **Sprout** *note* or a **Sapling** *note*. In either case it represents that a value v is spendable by the recipient who holds the *spending key* corresponding to a given *shielded payment address*.

A **Sprout** *note* is a tuple $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$, where:

- $\mathsf{a_{pk}} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{PRF}}]}$ is the *paying key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*);
- $\rho \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{PRF}}]}$ is used as input to $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}$ to derive the *nullifier* of the *note*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as defined in §4.1.7 ***'Commitment'*** on p. 21.

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

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

A **Sapling** *note* is a tuple $(\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm})$, where:

- $\mathsf{d} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the *diversifier* of the recipient's *shielded payment address*;
- $\mathsf{pk_d} \mathrel{⦂} \mathbb{J}$ is the *diversified transmission key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as defined in §4.1.7 ***'Commitment'*** on p. 21.

Let $\mathsf{Note}^{\mathsf{Sapling}}$ be the type of a **Sapling** *note*, i.e.

$$\mathsf{Note}^{\mathsf{Sapling}} := \mathbb{B}^{[\ell_{\mathsf{d}}]} \times \mathbb{J} \times \{0 \,..\, \mathsf{MAX\_MONEY}\} \times \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}.$$

Creation of new *notes* is described in §4.6 ***'Sending Notes'*** on p. 27. When *notes* are sent, only a commitment (see §4.1.7 ***'Commitment'*** on p. 21) 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 **Sprout** *note commitment* on a *note* $\mathbf{n} = (\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$ is computed as

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

where $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is instantiated in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 46.

Let DiversifyHash be as defined in §5.4.1.6 'DiversifyHash *Hash Function'* on p. 40.

A **Sapling** *note commitment* on a *note* $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm})$ is computed as

$$\mathsf{g_d} := \mathsf{DiversifyHash}(\mathsf{d})$$

$$\mathsf{NoteCommitment}^{\mathsf{Sapling}}(\mathbf{n}) := \begin{cases} \bot, & \text{if } \mathsf{g_d} = \bot \\ \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}), & \text{otherwise.} \end{cases}$$

where $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ is instantiated in §5.4.7.2 *'Windowed Pedersen commitments'* on p. 46.

Notice that the above definition of a **Sapling** *note* does not have a ρ field. There is in fact a ρ value associated with each **Sapling** *note*, but this only be computed once its position in the *note commitment tree* is known (see §3.3 *'The Block Chain'* on p. 12 and §3.4 *'Transactions and Treestates'* on p. 13). We refer to the combination of a *note* and its *note position* pos, as a *positioned note*.

For a *positioned note*, we can compute the value $\rho : \mathbb{B}^{[\ell_{\mathsf{PRFSapling}}]}$; see §4.10 *'Note Commitments and Nullifiers'* on p. 31.

A *nullifier* (denoted nf) is derived from the ρ value of a *note* and the recipient's *spending key* $\mathsf{a_{sk}}$ or *nullifier deriving key* nk. This computation uses a *Pseudo Random Function* (see §4.1.2 *'Pseudo Random Functions'* on p. 17), as described in §4.10 *'Note Commitments and Nullifiers'* on p. 31.

A *note* is spent by proving knowledge of $(\rho, \mathsf{a_{sk}})$ or $(\rho, \mathsf{ak}, \mathsf{nsk})$ in zero knowledge while publically disclosing its *nullifier* nf, allowing nf to be used to prevent double-spending. In the case of **Sapling**, a *spend authorization signature* is also required, in order to demonstrate knowledge of ask.

3.2.1 Note Plaintexts and Memo Fields

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

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

[**Sapling** onward] The *note plaintext* in each *Output description* is encrypted to the *diversified transmission key* $\mathsf{pk_d}$. Each **Sapling** *note plaintext* (denoted $\mathbf{np}$) consists of $(\mathsf{d}, \mathsf{v}, \mathsf{rcm}, \mathsf{memo})$.

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. 11.

Encodings are given in §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 53.

The result of encryption forms part of a *transmitted notes ciphertext* (see §4.12 *'In-band secret distribution'* on p. 34 for further details).

3.3 The Block Chain

At a given point in time, each *full validator* is aware of a set of candidate *blocks*. These form a tree rooted at the *genesis block*, where each node in the tree refers to its parent via the `hashPrevBlock` *block header* field (see §7.5 *'Block Header'* on p. 63).

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 §7.6.5 ***'Definition of Work'*** on p. 67, 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*.

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

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

To each *transaction* there are associated initial *treestates* for **Sprout** and for **Sapling**.

Each *treestate* consists of:

- a *note commitment tree* (§3.7 ***'Note Commitment Trees'*** on p. 15);
- a *nullifier set* (§3.8 ***'Nullifier Sets'*** on p. 15).

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* (either the **Sprout** tree or the **Sapling** 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*, for each of **Sprout** and **Sapling**, *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* for **Sprout**, explained in the following section. There is no equivalent of interstitial *treestates* for **Sapling**.

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. In **Sprout**, this kind of value transfer was 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 an instance of a *JoinSplit statement* (§4.11.1 ***'JoinSplit Statement (Sprout)'*** on p. 31), 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 **Sprout** *treestate*.

For each of the $\mathrm{N}^{\mathsf{old}}$ *shielded inputs*, a *nullifier* is revealed. This allows detection of double-spends as described in §3.8 ***'Nullifier Sets'*** on p. 15.

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

3.6 Spend Transfers, Output Transfers, and their Descriptions

JoinSplit transfers are not used for **Sapling** *notes*. Instead, there is a separate *Spend transfer* for each *shielded input*, and a separate *Output transfer* for each *shielded output*.

Spend descriptions and *Output descriptions* are data included in a transaction that describe *Spend transfers* and *Output transfers*, respectively.

A *Spend transfer* spends a *note* $\mathbf{n}^{\mathsf{old}}$. Its *Spend description* includes a *Pedersen value commitment* to the value of the *note*. It is associated with an instance of a *Spend statement* (§4.11.2 ***'Spend Statement (Sapling)'*** on p. 32) for which it provides a *zk-SNARK proof*.

An *Output transfer* creates a *note* $\mathbf{n}^{\mathsf{new}}$. Similarly, its *Output description* includes a *Pedersen value commitment* to the *note* value. It is associated with an instance of an *Output statement* (§4.11.3 ***'Output Statement (Sapling)'*** on p. 33) for which it provides a *zk-SNARK proof*.

Each *transaction* has a sequence of *Spend descriptions* and a sequence of *Output descriptions*.

To ensure balance, we use a homomorphic property of *Pedersen commitments* that allows them to be added and subtracted, as elliptic curve points. The result of adding two *Pedersen value commitments*, committing to values v_1 and v_2, is a new *Pedersen value commitment* that commits to $\mathsf{v}_1 + \mathsf{v}_2$. Subtraction works similarly.

Therefore, balance can be enforced by adding all of the *value commitments* for *shielded inputs*, subtracting all of the *value commitments* for *shielded outputs*, and checking that the result commits to a value consistent with the net *transparent* value change (see §4.9 ***'Balance'*** on p. 30 for a full specification). This approach allows all of the *zk-SNARK* statements to be independent of each other, potentially increasing opportunities for precomputation.

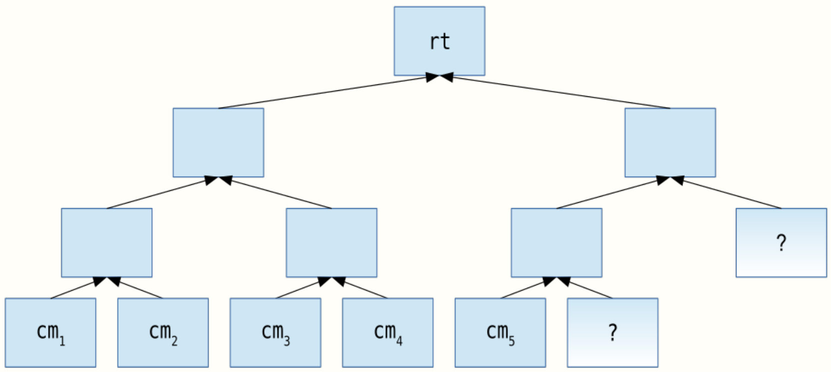
A *Spend description* includes an *anchor*, which refers to the output **Sapling** *treestate* of a previous *block*. It also reveals a *nullifier*, which allows detection of double-spends as described in §3.8 ***'Nullifier Sets'*** on p. 15.

Note: Interstitial *treestates* are not necessary for **Sapling**, because a *Spend transfer* in a given *transaction* cannot spend any of the *shielded outputs* of the same *transaction*. This is not an onerous restriction because, unlike **Sprout** where each *JoinSplit transfer* must balance individually, in **Sapling** it is only necessary for the whole *transaction* to balance.

Consensus rules:

- The *transaction* **MUST** balance as specified in §4.9 *'Balance'* on p. 30.
- The *anchor* of each *Spend description* in a *transaction* **MUST** refer to some earlier *block*'s final **Sapling** *treestate*.

3.7 Note Commitment Trees



TODO: The commitment indices in the above diagram should be zero-based to reflect the *note position*.

The *note commitment tree* is an *incremental Merkle tree* of fixed depth used to store *note commitments* that *JoinSplit transfers* and *Spend 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 this tree is associated with each *treestate*, as described in §3.4 *'Transactions and Treestates'* on p. 13.

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

3.8 Nullifier Sets

Each *full validator* maintains a *nullifier set* logically associated with each *treestate*. As valid *transactions* containing *JoinSplit transfers* or *Spend transfers* are processed, the *nullifiers* revealed in *JoinSplit descriptions* and *Spend 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*.

Note: **Sprout** and **Sapling** *nullifiers* are considered disjoint, even if they have the same bit pattern.

3.9 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. 12.

These calculations are described in §7.7 ***'Calculation of Block Subsidy and Founders' Reward'*** on p. 68.

3.10 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 §7.8 ***'Payment of Founders' Reward'*** on p. 68.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

Let $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell_{\mathsf{MerkleSprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\ell_{\mathsf{MerkleSapling}}$, ℓ_{ivk}, ℓ_{Seed}, ℓ_{hSig}, ℓ_{PRF}, and $\mathrm{N}^{\mathsf{old}}$ be as defined in §5.3 ***'Constants'*** on p. 36.

Let $\ell_{\mathbb{J}}$ be as defined in §5.4.8.3 ***'Jubjub'*** on p. 50.

The functions $\mathsf{MerkleCRH}^{\mathsf{Sprout}} : \{0\,..\,\mathsf{MerkleDepth}^{\mathsf{Sprout}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]}$ and (for **Sapling**), $\mathsf{MerkleCRH}^{\mathsf{Sapling}} : \{0\,..\,\mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ are *hash functions* used in §4.7 ***'Merkle path validity'*** on p. 29. $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ is collision-resistant on all its arguments, and $\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ is collision-resistant except on its first argument. Both of these functions are instantiated in §5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 38.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a collision-resistant *hash function* used in §4.3 ***'JoinSplit Descriptions'*** on p. 25. It is instantiated in §5.4.1.4 '$\mathsf{h}_{\mathsf{Sig}}$ ***Hash Function'*** on p. 39.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{N}^+ \to \mathbb{B}^{[n]}$ is another *hash function*, used in §7.6.1 ***'Equihash'*** on p. 65 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 §5.4.1.9 ***'Equihash Generator'*** on p. 42.

$\mathsf{CRH}^{\mathsf{ivk}} : \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} \to \{0\,..\,2^{\ell_{\mathsf{ivk}}} - 1\}$ is a collision-resistant *hash function* used in §4.2.2 ***'Sapling Key Components'*** on p. 24 to derive an *incoming viewing key* for a **Sapling** *shielded payment address*. It is also used in the *Spend statement* (§4.11.2 ***'Spend Statement (Sapling)'*** on p. 32) to confirm use of the correct key for the *note* being spent. It is instantiated in §5.4.1.5 '$\mathsf{CRH}^{\mathsf{ivk}}$ ***Hash Function'*** on p. 39.

$\mathsf{DiversifyHash} : \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{J}$ is a *hash function* satisfying the Discrete Logarithm Independence property (which implies collision-resistance) described in §4.1.10 ***'Group Hash'*** on p. 22. It is used to derive a *diversified base* from a *diversifier* in §4.2.2 ***'Sapling Key Components'*** on p. 24. It is instantiated in §5.4.1.6 '$\mathsf{DiversifyHash}$ ***Hash Function'*** on p. 40.

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}, $\ell_{\mathsf{PRFSapling}}$, $\mathrm{N}^{\mathsf{old}}$, and $\mathrm{N}^{\mathsf{new}}$ be as defined in §5.3 ***'Constants'*** on p. 36.

Let $\ell_{\mathbb{J}}$ and $r_{\mathbb{J}}$ be as defined in §5.4.8.3 ***'Jubjub'*** on p. 50.

For **Sprout**, four ***independent*** PRF_x are needed:

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

These are used in §4.11.1 ***'JoinSplit Statement (Sprout)'*** on p. 31; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in §4.2.1 ***'Sprout Key Components'*** on p. 24.

For **Sapling**, two additional PRF_x are needed:

$$\begin{aligned}
\mathsf{PRF}^{\mathsf{expand}} &: \mathbb{B}^{[\ell_{\mathsf{sk}}]} \times \{0 \,..\, 255\} &&\to \mathbb{F}_{r_{\mathbb{J}}} \\
\mathsf{PRF}^{\mathsf{nfSapling}} &: \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} &&\to \mathbb{B}^{[\ell_{\mathsf{PRFSapling}}]}
\end{aligned}$$

$\mathsf{PRF}^{\mathsf{expand}}$ is used in §4.2.2 ***'Sapling Key Components'*** on p. 24.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used in §4.11.2 ***'Spend Statement (Sapling)'*** on p. 32.

All of these *Pseudo Random Functions* are instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 43.

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 $\mathsf{PRF}^{\mathsf{nf}}_x$, $\mathsf{PRF}^{\mathsf{addr}}_x$, PRF^{ρ}_x, and $\mathsf{PRF}^{\mathsf{nfSapling}}_x$ 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} and $\mathsf{PRF}^{\mathsf{nfSapling}}$.

Note: $\mathsf{PRF}^{\mathsf{nf}}$ was called $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash** [BCG+2014].

4.1.3 Authenticated One-Time Symmetric Encryption

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

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

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

Security requirement: Sym must be one-time (INT-CTXT $\wedge$ IND-CPA)-secure. "*One-time*" here means that an honest protocol participant will almost surely encrypt only one message with a given key; however, the attacker may make many adaptive chosen ciphertext queries for a given key. The security notions INT-CTXT and IND-CPA are as defined in [BN2007].

4.1.4 Key Agreement

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

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

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

Let KA.DerivePublic ⦂ KA.Private × KA.Public → KA.Public be a function that derives the KA public key corresponding to a given KA private key and base point.

Let KA.Agree ⦂ KA.Private × KA.Public → KA.SharedSecret be the agreement function.

Optional: Let KA.Base ⦂ KA.Public be a public base point.

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

Security requirements:

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

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

4.1.5 Key Derivation

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

Let KDF $⦂ \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{[\ell_{\mathsf{hSig}}]} \times$ KA.SharedSecret × KA.Public × KA.Public → 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:

TODO: adapt this definition to handle **Sapling**, or maybe just remove it.

Let g := TODO :?

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

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

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

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

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

If the adversary's advantage is insignificant, then the asymmetric encryption scheme constructed from KA, KDF and Sym in §4.12 *'In-band secret distribution'* on p. 34 will be key-private as defined in [BBDP2001].

Note: The given definition only requires ciphertexts to be indistinguishable between *transmission keys* that are outputs of KA.DerivePublic (which includes all keys generated as in §4.2.1 *'**Sprout** Key Components'* on p. 24). If a *transmission key* not in that range is used, it may be distinguishable. This is not considered to be a significant security weakness.

4.1.6 Signature

A signature scheme Sig defines:

- a type of signing keys Sig.Private;
- a type of verifying keys Sig.Public;
- a type of messages Sig.Message;
- a type of signatures Sig.Signature;
- a randomized key pair generation algorithm $\mathsf{Sig.Gen} : () \xrightarrow{\mathrm{R}} \mathsf{Sig.Private} \times \mathsf{Sig.Public}$;
- a randomized signing algorithm $\mathsf{Sig.Sign} : \mathsf{Sig.Private} \times \mathsf{Sig.Message} \xrightarrow{\mathrm{R}} \mathsf{Sig.Signature}$;
- a verifying algorithm $\mathsf{Sig.Verify} : \mathsf{Sig.Public} \times \mathsf{Sig.Message} \times \mathsf{Sig.Signature} \to \mathbb{B}$;

such that for any key pair $(\mathsf{sk}, \mathsf{vk}) \xleftarrow{\mathrm{R}} \mathsf{Sig.Gen}()$, and any $m : \mathsf{Sig.Message}$ and $s : \mathsf{Sig.Signature} \xleftarrow{\mathrm{R}} \mathsf{Sig.Sign}_{\mathsf{sk}}(m)$, $\mathsf{Sig.Verify}_{\mathsf{vk}}(m, s) = 1$.

Zcash uses three 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 JoinSplitSig (instantiated in §5.4.5 *'JoinSplit Signature'* on p. 45), which is used to sign *transactions* that contain at least one *JoinSplit description*;
- [**Sapling** onward] one called SpendAuthSig (instantiated in §5.4.6 *'Spend Authorization Signature'* on p. 45), which is used to sign authorizations of *Spend descriptions*.

The following defines only the security properties needed for JoinSplitSig. Security properties for SpendAuthSig are defined in the next section, §4.1.6.1 *'Signature with Re-Randomizable Keys'* on p. 20.

Security requirement: JoinSplitSig must be Strongly Unforgeable under (non-adaptive) Chosen Message Attack (SU-CMA), as defined for example in [BDEHR2011, Definition 6]. 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.

TODO: Reference a different paper for the security definition. [BDEHR2011] has a flawed security proof; this doesn't affect **Zcash** but it would be better to avoid confusion that it might.

Notes:

- A fresh signature key pair is generated for each *transaction* containing a *JoinSplit description*. Since each key pair is only used for one signature (see §4.8 *'Non-malleability'* on p. 30), 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].

4.1.6.1 Signature with Re-Randomizable Keys

A signature scheme with re-randomizable keys Sig is a signature scheme that additionally defines:

- a type of randomizers Sig.Random;
- a public key randomization algorithm Sig.RandomizePublic : Sig.Public $\times$ Sig.Random $\rightarrow$ Sig.Public;
- a private key randomization algorithm Sig.RandomizePrivate : Sig.Private $\times$ Sig.Random $\rightarrow$ Sig.Private
- a distinguished "identity" randomizer Sig.Id : Sig.Random

such that if (pk : Sig.Public, sk : Sig.Private) is a valid Sig key pair, then:

- (Sig.RandomizePublic(pk, r), Sig.RandomizePrivate(sk, r)) is also a valid Sig key pair for any r : Sig.Random;
- Sig.RandomizePrivate(•, r) : Sig.Private $\rightarrow$ Sig.Private is injective and easily invertible for any r : Sig.Random;
- For ***any*** key pair (pk, sk) returned by Sig.Gen(), the distribution of

  $$(\mathsf{Sig.RandomizePublic}(\mathsf{pk}, r), \mathsf{Sig.RandomizePrivate}(\mathsf{sk}, r)) : r \xleftarrow{R} \mathsf{Sig.Random}$$

  is identical to the distribution of Sig.Gen().
- (Sig.RandomizePublic(pk, Sig.Id), Sig.RandomizePrivate(sk, Sig.Id)) = (pk, sk).

The following security requirement for such signature schemes is based on that given in [FKMSSS2016, section 3]. Note that we require Strong Unforgeability with Re-randomized Keys, not Existential Unforgeability with Re-randomized Keys (the latter is called "Unforgeability under Re-randomized Keys" in [FKMSSS2016, Definition 8]). Unlike the case for JoinSplitSig, we require security under adaptive chosen message attack with multiple messages signed using a given key. (Although each *note* uses a different re-randomized key pair, the same original key pair can be re-randomized for multiple *notes*, and also it can happen that multiple *transactions* spending the same *note* are revealed to an adversary.)

Security requirement: Strong Unforgeability with Re-randomized Keys under adaptive Chosen Message Attack (SURK-CMA)

Let O : Sig.Private $\times$ Sig.Message $\times$ Sig.Random $\rightarrow$ Sig.Signature be a generator of signing oracles.

A signing oracle O_{sk} for private key sk has state Q : $\mathscr{P}$(Sig.Message $\times$ Sig.Signature) initialized to {} that records queried messages and corresponding signatures.

O_{sk} := var $Q \leftarrow \{\}$ in (m : Sig.Message, r : Sig.Random) $\mapsto$

  let σ = $\mathsf{Sig.Sign}_{\mathsf{Sig.RandomizePrivate}(\mathsf{sk}, r)}(m)$

  $Q \leftarrow Q \cup \{(m, \sigma)\}$

  return σ : Sig.Signature.

For random $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{R} \mathsf{Sig.Gen}()$, it must be infeasible for an adversary given pk and a new instance of $\mathsf{O_{sk}}$ to find (m^*, σ^*, r^*) such that $\mathsf{Sig.Verify}_{\mathsf{Sig.RandomizePublic}(\mathsf{pk}, r^*)}(m^*, \sigma^*) = 1$ and $(m^*, \sigma^*) \notin \mathsf{O_{sk}}.Q$.

Notes:

- The requirement for Sig.Id simplifies the definition of SURK-CMA by removing the need for two oracles (since the oracle for original keys, called O_1 in [FKMSSS2016], is a special case of the oracle for randomized keys).
- Since $(\mathsf{Sig.RandomizePublic}(\mathsf{pk}, r), \mathsf{Sig.RandomizePrivate}(\mathsf{sk}, r)) : r \xleftarrow{R} \mathsf{Sig.Random}$ is identically distributed to $\mathsf{Sig.Gen}()$, the combination of a re-randomized public key and signature(s) under that key do not reveal the key from which it was re-randomized.
- Since $\mathsf{Sig.RandomizePrivate}(\bullet, r)$ is injective and easily invertible, knowledge of $\mathsf{Sig.RandomizePrivate}(\mathsf{sk}, r)$ ***and*** r implies knowledge of sk.

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* COMMIT defines a type of inputs COMMIT.Input, a type of commitments COMMIT.Output, and a type of *commitment trapdoors* COMMIT.Trapdoor.

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

Security requirements:

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

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

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 generator $\mathcal{P}_{\mathbb{G}}$ of the subgroup of $\mathbb{G}$ of order $r_{\mathbb{G}}$;
- a bit-length parameter $\ell_{\mathbb{G}} : \mathbb{N}$;
- a representation function $\mathsf{repr}_{\mathbb{G}} : \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]}$;

- an abstraction function $\mathsf{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\}$;

such that $\mathsf{abst}_{\mathbb{G}}$ is the left inverse of $\mathsf{repr}_{\mathbb{G}}$, i.e. for all $P \in \mathbb{G}$, $\mathsf{abst}_{\mathbb{G}}(\mathsf{repr}_{\mathbb{G}}(P)) = P$, and for all S not in the image of $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{abst}_{\mathbb{G}}(S) = \bot$.

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 Hash Extractor

A *hash extractor* for a *represented group* $\mathbb{G}$ is a function $\mathsf{Extract}_{\mathbb{G}} : \mathbb{G} \to T$ for some type T, such that $\mathsf{Extract}_{\mathbb{G}}$ is injective on the subgroup of $\mathbb{G}$ of order $r_{\mathbb{G}}$.

Note: Unlike the representation function $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{Extract}_{\mathbb{G}}$ need not have an efficiently computable left inverse.

4.1.10 Group Hash

Given a represented group $\mathbb{G}$ and a type $\mathsf{CRSType}$, we define a *family of group hashes into* $\mathbb{G}$ as a function

$$\mathsf{GroupHash}^{\mathbb{G}} : \mathsf{CRSType} \times \mathbb{B}^{[\ell]} \to \mathbb{G}$$

with the following security requirement.

Security requirement: Discrete Logarithm Independence

For a randomly selected member $\mathsf{GroupHash}^{\mathbb{G}}_{\mathsf{CRS}}$ of the family, it is infeasible to find a sequence of distinct inputs $m_{1\,..\,n} : \mathbb{B}^{[\ell]}{}^{[n]}$ and a sequence of nonzero scalars $x_{1\,..\,n} : \mathbb{F}^{*}_{r_{\mathbb{G}}}{}^{[n]}$ such that $\sum_{i=1}^{n}\Big([x_i]\,\mathsf{GroupHash}^{\mathbb{G}}_{\mathsf{CRS}}(m_i)\Big) = \mathcal{O}_{\mathbb{G}}$.

Notes:

- This property implies (and is stronger than) collision-resistance, since a collision (m_1, m_2) for $\mathsf{GroupHash}^{\mathbb{G}}_{\mathsf{CRS}}$ trivially gives a discrete logarithm relation with $x_1 = 1$ and $x_2 = -1$.
- An alternative approach is to model $\mathsf{GroupHash}^{\mathbb{G}}_{\mathsf{CRS}}$ as a random oracle, and assume that the Discrete Logarithm Problem is hard in the group. We prefer to avoid the Random Oracle Model and instead make a more specific standard-model assumption, which is effectively no stronger than the assumptions made in the random oracle approach.
- CRS is a *Common Random String*; we choose it verifiably at random, ***after*** fixing the concrete group hash algorithm to be used. If we publish the algorithm and the method of choosing the *Common Random String* before the CRS could be known, then this mitigates the possibility that the group hash algorithm could have been backdoored.

4.1.11 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 $\mathbf{1}_{\mathbb{P}}$;
- 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 $Q : \mathbb{P}_2$, $\hat{e}_{\mathbb{P}}(P, Q) = \mathbf{1}_{\mathbb{P}}$.

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

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

Security requirements:

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

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

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

Zcash uses two *proving systems*:

- PHGR13 (§5.4.9.1 ***'PHGR13'*** on p. 52) is used with the BN-254 pairing (§5.4.8.1 ***'BN-254'*** on p. 47), to prove and verify the **Sprout** *JoinSplit statement* (§4.11.1 ***'JoinSplit Statement (Sprout)'*** on p. 31).
- Groth16 (§5.4.9.2 ***'Groth16'*** on p. 52) is used with the BLS12-381 pairing (§5.4.8.2 ***'BLS12-381'*** on p. 48), to prove and verify the **Sapling** *Spend statement* (§4.11.2 ***'Spend Statement (Sapling)'*** on p. 32) and *Output statement* (§4.11.3 ***'Output Statement (Sapling)'*** on p. 33).

These specializations are referred to as ZKJoinSplit for the **Sprout** *JoinSplit statement*, ZKSpend for the **Sapling** *Spend statement*, and ZKOutput for the **Sapling** *Output statement*.

We omit the key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be the PHGR13 *proving key* and *verifying key* defined in §5.7 ***'Sprout zk-SNARK Parameters'*** on p. 58.

Similarly, we omit the key subscripts on ZKSpend.Prove, ZKSpend.Verify, ZKOutput.Prove, and ZKOutput.Verify, taking them to be the Groth16 *proving keys* and *verifying keys* defined in §5.8 ***'Sapling zk-SNARK Parameters'*** on p. 58.

4.2 Key Components

4.2.1 Sprout Key Components

Let $\mathsf{PRF}^{\mathsf{addr}}$ be a *Pseudo Random Function*, instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 43.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be a *key agreement scheme*, instantiated in §5.4.4.1 ***'Sprout Key Agreement'*** on p. 44.

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

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

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

4.2.2 Sapling Key Components

Let $\mathsf{PRF}^{\mathsf{expand}}$ be a *Pseudo Random Function*, instantiated in §5.4.2 ***'Pseudo Random Functions'*** on p. 43.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be a *key agreement scheme*, instantiated in §5.4.4.3 ***'Sapling Key Agreement'*** on p. 45.

Let $\mathsf{CRH}^{\mathsf{ivk}}$ be a *hash function*, instantiated in §5.4.1.5 '$\mathsf{CRH}^{\mathsf{ivk}}$ ***Hash Function'*** on p. 39.

Let DiversifyHash be a *hash function*, instantiated in §5.4.1.6 'DiversifyHash ***Hash Function'*** on p. 40.

Let $\mathsf{FindGroupHash}^{\mathbb{J}}$ be as defined in §5.4.8.5 ***'Group Hash into Jubjub'*** on p. 51.

Let $\mathcal{G} = \mathsf{FindGroupHash}^{\mathbb{J}}(\texttt{"Zcash\_G\_"}, \texttt{""})$ and let $\mathcal{H} = \mathsf{FindGroupHash}^{\mathbb{J}}(\texttt{"Zcash\_H\_"}, \texttt{""})$.

Let $\mathsf{repr}_{\mathbb{J}}$ be the representation function for the Jubjub *represented group*, instantiated in §5.4.8.3 ***'Jubjub'*** on p. 50.

Let $\mathsf{LEBS2OSP} \colon (\ell \colon \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ be defined as in §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 36.

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

From this *spending key*, the *spend authorizing key* ask and *proof authorizing key* nsk are derived as follows:

$$\mathsf{ask} := \mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(0)$$

$$\mathsf{nsk} := \mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(1)$$

ak, nk, and ivk are then derived as follows:

$$\mathsf{ak} := [\mathsf{ask}]\,\mathcal{G}$$

$$\mathsf{nk} := [\mathsf{nsk}]\,\mathcal{H}$$

$$\mathsf{ivk} := \mathsf{CRH}^{\mathsf{ivk}}\Big(\boxed{256\text{-bit } \mathsf{repr}_{\mathbb{J}}(\mathsf{ak})} \boxed{256\text{-bit } \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})} \Big).$$

As explained in §3.1 ***'Payment Addresses and Keys'*** on p. 10, **Sapling** allows the efficient creation of multiple *diversified payment addresses* with the same spending authority. A group of such addresses shares the same *full viewing key* and *incoming viewing key*.

To create a new *diversified payment address* given an *incoming viewing key* ivk, repeatedly pick a *diversifier* d uniformly at random from $\mathbb{B}^{[\ell_{\mathsf{d}}]}$ until $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$ is not $\bot$. Then calculate:

$$\mathsf{pk_d} := \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{ivk}, \mathsf{g_d}).$$

The resulting *diversified payment address* is $(\mathsf{d}, \mathsf{pk_d})$.

Notes:

- The protocol does not prevent using the *diversifier* d to produce "*vanity*" addresses that start with a meaningful string when encoded in Bech32 (see §5.6.4 ***'Sapling Shielded Payment Addresses'*** on p. 55). Users and writers of software that generates addresses should be aware that this provides weaker privacy properties than a randomly chosen *diversifier*, since a vanity address can obviously be distinguished, and might leak more information than intended as to who created it.
- Similarly, address generators **MAY** encode information in the *diversifier* that can be recovered by the recipient of a payment to determine which *diversified payment address* was used. It is **RECOMMENDED** that such *diversifiers* be randomly chosen unique byte sequences used to index into a database, rather than directly encoding the needed data.

4.3 JoinSplit Descriptions

A *JoinSplit transfer*, as specified in §3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 13, 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 $(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{rt}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{epk}, \mathsf{randomSeed}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}, \pi_{\mathsf{ZKJoinSplit}}, \mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$

where

- $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} \colon \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent value pool*;
- $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} \colon \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent value pool*;

- $\mathsf{rt} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in §3.3 ***'The Block Chain'*** on p. 12, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is the sequence of *nullifiers* for the input *notes*;
- $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} \mathbin{⦂} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} \mathbin{⦂} \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ is a key agreement public key, used to derive the key for encryption of the *transmitted notes ciphertext* (§4.12 ***'In-band secret distribution'*** on p. 34);
- $\mathsf{randomSeed} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{Seed}}]}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is a sequence of tags that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the input *notes*;
- $\pi_{\mathsf{ZKJoinSplit}} \mathbin{⦂} \mathsf{ZKJoinSplit}.\mathsf{Proof}$ is the *zero-knowledge proof* for the *JoinSplit statement*;
- $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}} \mathbin{⦂} \mathsf{Sym}.\mathbf{C}^{[\mathrm{N}^{\mathsf{new}}]}$ is a sequence of ciphertext components for the encrypted output *notes*.

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

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

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

hSigCRH is instantiated in §5.4.1.4 '$\mathsf{h}_{\mathsf{Sig}}$ ***Hash Function'*** on p. 39.

Consensus rules:

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

4.4 Spend Descriptions

A *Spend transfer*, as specified in §3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 14, is encoded in *transactions* as a *Spend description*.

Each *transaction* includes a sequence of zero or more *Spend descriptions*.

Unlike *JoinSplit signatures* of which there is at most one per *transaction*, *each Spend description* is authorized by a signature, called the *spend authorization signature*.

A *Spend description* consists of $(\mathsf{cv}, \mathsf{rt}, \mathsf{nf}, \mathsf{rk}, \pi_{\mathsf{ZKSpend}}, \texttt{spendAuthSig})$

where

- $\mathsf{cv} \mathbin{⦂} \mathsf{ValueCommit}.\mathsf{Output}$ is the *value commitment* to the value of the input *note*;
- $\mathsf{rt} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ is an *anchor*, as defined in §3.3 ***'The Block Chain'*** on p. 12, for the output *treestate* of a previous *block*.
- $\mathsf{nf} \mathbin{⦂} \mathbb{B}^{[\ell_{\mathsf{PRFSapling}}]}$ is the *nullifier* for the input *note*;
- $\mathsf{rk} \mathbin{⦂} \mathsf{SpendAuthSig}.\mathsf{Public}$ is a randomized public key that should be used to verify `spendAuthSig`;
- $\pi_{\mathsf{ZKSpend}} \mathbin{⦂} \mathsf{ZKSpend}.\mathsf{Proof}$ is the *zero-knowledge proof* for the *Spend statement*;
- `spendAuthSig` $\mathbin{⦂} \mathsf{SpendAuthSig}.\mathsf{Signature}$ is a signature authorizing this spend.

Consensus rules:

- Elements of a *Spend description* **MUST** have the types given above.
- The proof π_{ZKSpend} **MUST** be valid given a *primary input* formed from the other fields except `spendAuthSig`. I.e. it must be the case that $\mathsf{ZKSpend.Verify}((\mathsf{cv}, \mathsf{rt}, \mathsf{nf}), \pi_{\mathsf{ZKSpend}}) = 1$.
- The *spend authorization signature* **MUST** be a valid SpendAuthSig signature using rk as the public key, over TODO: ...

4.5 Output Descriptions

An *Output transfer*, as specified in §3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 14, is encoded in *transactions* as an *Output description*.

Each *transaction* includes a sequence of zero or more *Output descriptions*. There are no signatures associated with *Output descriptions*.

An *Output description* consists of $(\mathsf{cv}, \mathsf{cm}, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \pi_{\mathsf{ZKOutput}})$

where

- $\mathsf{cv} ⦂ \mathsf{ValueCommit.Output}$ is the *value commitment* to the value of the output *note*;
- $\mathsf{cm} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ is the *note commitment* for the output *note*;
- $\mathsf{epk} ⦂ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public}$ is a key agreement public key, used to derive the key for encryption of the *transmitted notes ciphertext* (§4.12 ***'In-band secret distribution'*** on p. 34);
- $\mathsf{C}^{\mathsf{enc}} ⦂ \mathsf{Sym}.\mathbf{C}$ is a ciphertext component for the encrypted output *note*.
- $\pi_{\mathsf{ZKOutput}} ⦂ \mathsf{ZKOutput.Proof}$ is the *zero-knowledge proof* for the *Output statement*.

Consensus rules:

- Elements of an *Output description* **MUST** have the types given above.
- The proof π_{ZKOutput} **MUST** be valid given a *primary input* formed from the other fields except $\mathsf{C}^{\mathsf{enc}}$. I.e. it must be the case that $\mathsf{ZKSpend.Verify}((\mathsf{cv}, \mathsf{cm}, \mathsf{epk}), \pi_{\mathsf{ZKOutput}}) = 1$.

4.6 Sending Notes

4.6.1 Sending Notes (Sprout)

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:

$$(\mathtt{joinSplitPrivKey}, \mathtt{joinSplitPubKey}) \xleftarrow{\mathrm{R}} \mathsf{JoinSplitSig.Gen}().$$

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

- Choose $\mathsf{rcm}_i^{\mathsf{new}}$ uniformly at random on $\mathbb{B}^{[\ell_{\mathsf{rcm}}]}$.
- Compute $\rho_i^{\mathsf{new}} = \mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}})$.
- Encrypt the *note* to the recipient *transmission key* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i}$, as described in §4.12 ***'In-band secret distribution'*** on p. 34, giving the ciphertext component $\mathsf{C}^{\mathsf{enc}}_i$.

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

After generating all of the *JoinSplit descriptions*, the sender obtains the dataToBeSigned (§4.8 *'Non-malleability'* on p. 30), and signs it with the private *JoinSplit signing key*:

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

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

4.6.2 Dummy Notes (Sprout)

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

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

- Generate a new random *spending key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}$ and derive its *paying key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}$.
- Set $\mathsf{v}^{\mathsf{old}}_i := 0$.
- Choose $\mathsf{\rho}^{\mathsf{old}}_i$ uniformly at random on $\mathbb{B}^{[\ell_{\mathsf{PRF}}]}$.
- Choose $\mathsf{rcm}^{\mathsf{old}}_i$ uniformly at random on $\mathbb{B}^{[\ell_{\mathsf{rcm}}]}$.
- Compute $\mathsf{nf}^{\mathsf{old}}_i := \mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(\mathsf{\rho}^{\mathsf{old}}_i)$.
- 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*.

4.6.3 Sending Notes (Sapling)

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

Let OutputIndex be the type $\{0 .. 2^{32} - 1\}$.

For each *Output description* with index $\mathsf{idx} ⦂ \mathsf{OutputIndex}$, the sender selects a value $\mathsf{v}^{\mathsf{new}}_{\mathsf{idx}}$ and a destination **Sapling** *shielded payment address* $(\mathsf{d}, \mathsf{pk_d})$, and then performs the following steps:

1. Check that $\mathsf{pk_d}$ is a valid compressed representation of an Edwards point on the *Jubjub curve* and this point is not of small order (i.e. $\mathsf{abst}_{\mathbb{J}}(\mathsf{pk_d}) \neq \bot$ and $[8]\,\mathsf{abst}_{\mathbb{J}}(\mathsf{pk_d}) \neq \mathcal{O}_{\mathbb{J}}$).
2. Calculate $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$ and check that $\mathsf{g_d} \neq \bot$.
3. Choose esk uniformly at random on $\{0 .. r_{\mathbb{J}} - 1\}$.
4. Choose independent random commitment trapdoors:

$$\mathsf{rcv}^{\mathsf{new}}_{\mathsf{idx}} ⦂ \mathsf{ValueCommit.Trapdoor}$$
$$\mathsf{rcm}^{\mathsf{new}}_{\mathsf{idx}} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}\mathsf{.Trapdoor}$$

5. Calculate

$$\mathsf{cv}^{\mathsf{new}}_{\mathsf{idx}} := \mathsf{ValueCommit}_{\mathsf{rcv}^{\mathsf{new}}_{\mathsf{idx}}}(\mathsf{v}^{\mathsf{new}}_{\mathsf{idx}})$$
$$\mathsf{cm}^{\mathsf{new}}_{\mathsf{idx}} := \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}_{\mathsf{idx}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}^{\mathsf{new}}_{\mathsf{idx}})$$
$$\mathsf{epk} := \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}, \mathsf{g_d})$$
$$\mathsf{sharedSecret} := \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{esk}, \mathsf{pk_d}).$$

6. Let $\mathsf{K} := \mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{idx}, \mathsf{sharedSecret}, \mathsf{epk})$.
7. Let P be the raw encoding of the *note plaintext* $(\mathsf{d}, \mathsf{v}^{\mathsf{new}}_{\mathsf{idx}}, \mathsf{rcm}^{\mathsf{new}}_{\mathsf{idx}}, \mathsf{memo})$.
   (See §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 53.)
8. Encrypt P using the IETF version of AEAD_CHACHA20_POLY1305, with empty associated data, all zero 96-bit nonce, and 256-bit key K, giving C.
9. Generate a proof π_{ZKOutput} for the *Output circuit* described below.
10. Return $(\mathsf{cv}^{\mathsf{new}}_{\mathsf{idx}}, \mathsf{cm}^{\mathsf{new}}_{\mathsf{idx}}, \mathsf{epk}, \mathsf{C}, \pi_{\mathsf{ZKOutput}})$.

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.

The encoded *transaction* is submitted to the network.

TODO: The actual encryption should be split into a subsection of §4.12 *'In-band secret distribution'* on p. 34 as it is for **Sprout**.

TODO: Receiving a **Sapling** note.

4.7 Merkle path validity

Let MerkleDepth be $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ for the **Sprout** *note commitment tree*, or $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ for the **Sapling** *note commitment tree*. These constants are defined in §5.3 *'Constants'* on p. 36.

Similarly, let MerkleCRH be $\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ for **Sprout**, or $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ for **Sapling**.

The following discussion applies independently to the **Sprout** and **Sapling** *note commitment trees*.

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* 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* $\mathsf{Uncommitted}^{\mathsf{Sprout}}$ or $\mathsf{Uncommitted}^{\mathsf{Sapling}}$. It is assumed to be infeasible to find a preimage *note* $\mathbf{n}$ such that $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}) = \mathsf{Uncommitted}^{\mathsf{Sprout}}$. (No similar assumption is needed for **Sapling** because we use a representation for $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ that cannot occur as an output of $\mathsf{NoteCommitment}^{\mathsf{Sapling}}$.)

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

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

where

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

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

4.8 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 §7.1 ***'Encoding of Transactions'*** on p. 59. They ***do not*** cover the field `joinSplitSig`.

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

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

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

JoinSplitSig is instantiated in §5.4.5 ***'JoinSplit Signature'*** on p. 45.

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 $\mathsf{h}_{\mathsf{Sig}}$ be computed as specified in §4.3 ***'JoinSplit Descriptions'*** on p. 25.

Let $\mathsf{PRF}^{\mathsf{pk}}$ be as defined in §4.1.2 ***'Pseudo Random Functions'*** on p. 17.

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

The correctness of $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ is enforced by the *JoinSplit statement* given in §4.11.1 ***'Non-malleability'*** on p. 32. This ensures that a holder of all of the $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$ for every *JoinSplit description* in the *transaction* has authorized the use of the private signing key corresponding to `joinSplitPubKey` to sign this *transaction*.

[**Sapling** onward] TODO: Specify the *spend authorization signature*.

4.9 Balance

A *JoinSplit transfer* can be seen, from the perspective of the *transaction*, as an input and an output simultaneously.

$\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ takes value from the *transparent value pool* and $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$ adds value to the *transparent value pool*. As a result, $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ is treated like an ***output*** value, whereas $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$ is treated like an ***input*** value.

Unlike original **Zerocash** [BCG+2014], **Zcash** does not have a distinction between Mint and Pour operations. The addition of $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ 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. 25, either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ **MUST** be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

TODO: Add details of balance checking for **Sapling** *transactions*.

4.10 Note Commitments and Nullifiers

A *transaction* that contains one or more *JoinSplit descriptions* or *Spend 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 §3.8 *'Nullifier Sets'* on p. 15).

In **Sprout**, each *note* has a ρ component.

In **Sapling**, each *positioned note* has an associated ρ value which is computed from its *note commitment* cm and *note position* pos as follows:

$$\rho := \mathsf{MixingPedersenHash}(\mathsf{cm}, \mathsf{pos}).$$

$\mathsf{MixingPedersenHash}$ is defined in §5.4.1.8 *'Mixing Pedersen Hash Function'* on p. 42.

Let $\mathsf{PRF}^{\mathsf{nf}}$ and $\mathsf{PRF}^{\mathsf{nfSapling}}$ be as instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 43.

For a **Sprout** *note*, the *nullifier* is derived as $\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho)$.

For a **Sapling** *note*, the *nullifier* is derived as $\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}}(\rho)$.

4.11 Zk-SNARK Statements

4.11.1 JoinSplit Statement (Sprout)

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

$$\begin{aligned}
&(\mathsf{rt} ⦂ \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]},\\
&\ \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]},\\
&\ \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} ⦂ \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]},\\
&\ \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} ⦂ \{0\,..\,2^{64}-1\},\\
&\ \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} ⦂ \{0\,..\,2^{64}-1\},\\
&\ \mathsf{h_{Sig}} ⦂ \mathbb{B}^{[\ell_{\mathsf{hSig}}]},\\
&\ \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}),
\end{aligned}$$

the prover knows an *auxiliary input*:

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

where:

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

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforceMerklePath}_i = 1$: path_i is a valid *Merkle tree path* (see §4.7 *'Merkle path validity'* on p. 29) of depth $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ from $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}_i^{\mathsf{old}})$ to the *anchor* rt.

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

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

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

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

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

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

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

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

For details of the form and encoding of proofs, see §5.4.9.1 *'PHGR13'* on p. 52.

4.11.2 Spend Statement (Sapling)

Let $\mathcal{G}$ be as defined in §4.2.2 ***'Sapling Key Components'*** on p. 24.

A valid instance of π_{ZKSpend} assures that given a *primary input*:

$(\mathsf{rt} ⦂ \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]},$
$\mathsf{cv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit.Output},$
$\mathsf{nf}^{\mathsf{old}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRFSapling}}]},$
$\mathsf{rk}^{\mathsf{old}\star} ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]}),$

the prover knows an *auxiliary input*:

$(\mathsf{path} ⦂ \mathbb{B}^{[\ell_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sapling}}]} \times \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}-1\},$
$\mathsf{g_d}^\star ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]},$
$\mathsf{pk_d}^\star ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]},$
$\mathsf{v}^{\mathsf{old}} ⦂ \{0\,..\,2^{64}-1\},$
$\mathsf{rcv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit.Trapdoor},$
$\mathsf{cm}^{\mathsf{old}} ⦂ \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]},$
$\mathsf{rcm}^{\mathsf{old}} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor},$
$\mathsf{ar} ⦂ \{0\,..\,2^{252}-1\},$
$\mathsf{ak}^\star ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]},$
$\mathsf{nsk} ⦂ \{0\,..\,2^{252}-1\})$

such that the following conditions hold:

Note commitment integrity $\mathsf{pack}(\mathsf{cm}^{\mathsf{old}}) = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{old}}}(\mathsf{g_d}^*, \mathsf{pk_d}^*, \mathsf{v}^{\mathsf{old}})$.

TODO: define pack.

Merkle path validity path is a valid *Merkle tree path*, as defined in §4.7 ***'Merkle path validity'*** on p. 29, of depth $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ from $\mathsf{cm}^{\mathsf{old}}$ to the *anchor* rt.

Value commitment integrity $\mathsf{cv}^{\mathsf{old}} = \mathsf{ValueCommit}_{\mathsf{rcv}^{\mathsf{old}}}(\mathsf{v}^{\mathsf{old}})$.

Point validity checks $\mathsf{rk}^{\mathsf{old}}, \mathsf{ak}, \mathsf{g_d} \in \mathbb{J}$ and are not of small order, i.e. $[8]\,\mathsf{rk}^{\mathsf{old}} \neq \mathcal{O}_{\mathbb{J}}$ and $[8]\,\mathsf{ak} \neq \mathcal{O}_{\mathbb{J}}$ and $[8]\,\mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$.

Nullifier integrity $\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}}(\rho)$ where

$\mathsf{nk} = [\mathsf{nsk}]\,\mathcal{H}$

$\rho = \mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})$.

Spend authority $\mathsf{rk}^{\mathsf{old}} = \mathsf{ak} + [\mathsf{ar}]\,\mathcal{G}$ where

$\mathsf{rk}^{\mathsf{old}} : \mathbb{J} = \mathsf{abst}_{\mathbb{J}}\left(\mathsf{rk}^{\mathsf{old}*}\right)$

$\mathsf{ak} : \mathbb{J} = \mathsf{abst}_{\mathbb{J}}\left(\mathsf{ak}^*\right)$.

Diversified address integrity $\mathsf{pk_d} = [\mathsf{ivk}]\,\mathsf{g_d}$ where

$\mathsf{ivk} = \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}^*, \mathsf{nk}^*)$

$\mathsf{g_d} = \mathsf{abst}_{\mathbb{J}}\left(\mathsf{g_d}^*\right)$.

For details of the form and encoding of *Spend statement* proofs, see §5.4.9.2 ***'Groth16'*** on p. 52.

4.11.3 Output Statement (Sapling)

A valid instance of π_{ZKOutput} assures that given a *primary input*:

$(\mathsf{cv}^{\mathsf{new}} : \mathsf{ValueCommit.Output},$
$\mathsf{cm}^{\mathsf{new}} : \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Output},$
$\mathsf{epk} : \mathbb{J}),$

the prover knows an *auxiliary input*:

$(\mathsf{g_d}^* : \mathbb{B}^{[\ell_{\mathbb{J}}]},$
$\mathsf{pk_d}^* : \mathbb{B}^{[\ell_{\mathbb{J}}]},$
$\mathsf{v}^{\mathsf{new}} : \{0\,..\,2^{64}-1\},$
$\mathsf{rcv}^{\mathsf{new}} : \mathsf{ValueCommit.Trapdoor},$
$\mathsf{rcm}^{\mathsf{new}} : \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor},$
$\mathsf{esk} : \{0\,..\,2^{252}-1\})$

such that the following conditions hold:

Note commitment integrity $\mathsf{pack}(\mathsf{cm}^{\mathsf{new}}) = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}}(\mathsf{g_d}^*, \mathsf{pk_d}^*, \mathsf{v}^{\mathsf{new}})$.

TODO: define pack.

Value commitment integrity $\mathsf{cv}^{\mathsf{new}} = \mathsf{ValueCommit}_{\mathsf{rcv}^{\mathsf{new}}}(\mathsf{v}^{\mathsf{new}})$.

Point validity checks $\mathsf{g_d} \in \mathbb{J}$ and is not of small order, i.e. $[8]\,\mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$, where

$\mathsf{g_d} = \mathsf{abst}_{\mathbb{J}}(\mathsf{g_d}^{*})$.

Ephemeral public key integrity $\mathsf{epk} = [\mathsf{esk}]\,\mathsf{g_d}$ where

$\mathsf{epk} = \mathsf{abst}_{\mathbb{J}}(\mathsf{epk}^{*})$.

For details of the form and encoding of *Output statement* proofs, see §5.4.9.2 *'Groth16'* on p. 52.

4.12 In-band secret distribution

The secrets that need to be transmitted to a recipient of funds in order for them to later spend, are v, ρ, rcm, and in the case of **Sapling** d and $\mathsf{pk_d}$. A *memo field* (§3.2.1 *'Note Plaintexts and Memo Fields'* on p. 12) is also transmitted.

In order to the transmit these secrets securely to a recipient ***without*** requiring an out-of-band communication channel, the *transmission key* $\mathsf{pk_{enc}}$ or $\mathsf{pk_d}$ 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*.

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. 44.

Let $\mathsf{KDF}^{\mathsf{Sprout}}$ and $\mathsf{KDF}^{\mathsf{Sapling}}$ be the *Key Derivation Functions* instantiated in §5.4.4.2 *'**Sprout** Key Derivation'* on p. 44.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ and $\mathsf{KA}^{\mathsf{Sapling}}$ be the *key agreement schemes* instantiated in §5.4.4 *'Key Agreement and Derivation'* on p. 44.

[**Sprout**] Let $\mathsf{h_{Sig}}$ be the value computed for this *JoinSplit description* in §4.3 *'JoinSplit Descriptions'* on p. 25.

4.12.1 Encryption (Sprout)

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. 53.

Then to encrypt:

- Generate a new $\mathsf{KA}^{\mathsf{Sprout}}$ (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 $\mathsf{sharedSecret}_i := \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Agree}(\mathsf{esk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
  - Let $\mathsf{K}^{\mathsf{enc}}_i := \mathsf{KDF}^{\mathsf{Sprout}}(i, \mathsf{h_{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i})$.
  - Let $\mathsf{C}^{\mathsf{enc}}_i := \mathsf{Sym}.\mathsf{Encrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathsf{P}^{\mathsf{enc}}_i)$.

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

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

4.12.2 Decryption by a Recipient (Sprout)

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. 24.

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

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

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

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

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

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

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

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

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

if $\mathsf{NoteCommit}^{\mathsf{Sprout}}_{(}(\mathsf{a_{pk}}, \mathsf{v}_i^{\mathsf{new}}, \rho_i^{\mathsf{new}}, \mathsf{rcm}_i^{\mathsf{new}})) \neq \mathsf{cm}_i^{\mathsf{new}}$, 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 [BCG+2014, 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 §8.7 ***'In-band secret distribution'*** on p. 74 for further discussion of the security and engineering rationale behind this encryption scheme.

5 Concrete Protocol

5.1 Caution

TODO: Explain the kind of things that can go wrong with linkage between abstract and concrete protocol. E.g. §8.5 *'Internal hash collision attack and fix'* on p. 72

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*.

The following functions convert between sequences of bits, sequences of bytes, and integers:

- $\mathsf{I2LEBSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^{\ell}-1\} \to \mathbb{B}^{[\ell]}$, such that $\mathsf{I2LEBSP}_{\ell}(x)$ is the sequence of ℓ bits representing x in little-endian order;
- $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^{\ell}-1\} \to \mathbb{B}^{[\ell]}$ such that $\mathsf{I2BEBSP}_{\ell}(x)$ is the sequence of ℓ bits representing x in big-endian order.
- $\mathsf{LEOS2IP} : (k : \mathbb{N}) \times \mathbb{BY}^{[k]} \to \{0\,..\,256^{k}-1\}$ such that $\mathsf{LEOS2IP}_{k}(S)$ is the integer represented in little-endian order by the byte sequence S of length k.
- $\mathsf{LEBS2OSP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ defined as follows: pad the input on the right with $8 \cdot \mathsf{ceiling}(\ell/8) - \ell$ zero bits so that its length is a multiple of 8 bits. Then convert each group of 8 bits to a byte value with the *least* significant bit first, and concatenate the resulting bytes in the same order as the groups.

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, or for the output of $\mathsf{LEBS2OSP}_{\ell}$).

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

5.3 Constants

Define:

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

$\mathsf{MerkleDepth}^{\mathsf{Sapling}} : \mathbb{N} := 32$

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

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

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

$\ell_{\mathsf{MerkleSapling}} : \mathbb{N} := 255$

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

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

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

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

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

$\ell_{\varphi} : \mathbb{N} := 252$

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

$\ell_{\mathsf{d}} : \mathbb{N} := 88$

$\ell_{\mathsf{ivk}} : \mathbb{N} := 251$

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

$\mathsf{Uncommitted}^{\mathsf{Sapling}} : \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} := \mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}(1)$

$\mathsf{MAX\_MONEY} : \mathbb{N} := 2.1 \cdot 10^{15}$ (*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 $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$.

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

[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*, SHA256Compress. This operates on a single 512-bit block and ***excludes*** the padding step specified in [NIST2015, section 5.1].

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

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

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

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

5.4.1.2 BLAKE2 Hash Function

BLAKE2 is defined by [ANWW2013]. **Zcash** uses both the BLAKE2b and BLAKE2s variants.

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

BLAKE2b is used to instantiate hSigCRH, EquihashGen, and $\mathsf{KDF}^{\mathsf{Sprout}}$. From **Overwinter** onward, it is used to compute *SIGHASH transaction hashes* as specified in [ZIP-143]. For **Sapling**, it is also used to instantiate $\mathsf{KDF}^{\mathsf{Sapling}}$, and in the EdJubjub *signature scheme* which instantiates SpendAuthSig.

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

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

$\mathsf{BLAKE2s}\text{-}\ell(p, x)$ refers to unkeyed BLAKE2s-ℓ in sequential mode, with an output digest length of $\ell/8$ bytes, 8-byte personalization string p, and input x.

BLAKE2s is used to instantiate $\mathsf{PRF}^{\mathsf{expand}}$, $\mathsf{PRF}^{\mathsf{nfSapling}}$, $\mathsf{CRH}^{\mathsf{ivk}}$, and $\mathsf{GroupHash}^{\mathbb{J}}$.

$$\mathsf{BLAKE2s}\text{-}\ell : \mathbb{BY}^{[8]} \times \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{BY}^{[\ell/8]}$$

5.4.1.3 Merkle Tree Hash Function

$\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ and $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ are used to hash *incremental Merkle tree hash values* for **Sprout** and **Sapling** respectively.

$\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ Hash Function

Let SHA256Compress be as specified in §5.4.1.1 *'SHA-256 and SHA256Compress Hash Functions'* on p. 37.

$\mathsf{MerkleCRH}^{\mathsf{Sprout}} : \{0 \,..\, \mathsf{MerkleDepth}^{\mathsf{Sprout}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]}$ is defined as follows:

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

Notes:

- The layer argument does not affect the output.
- 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 $\mathsf{SHA256Compress}(x) = [0]^{256}$.

$\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ Hash Function

Let PedersenHash be as specified in §5.4.1.7 *'Pedersen Hash Function'* on p. 40.

$\mathsf{MerkleCRH}^{\mathsf{Sapling}} : \{0 \,..\, \mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ is defined as follows:

$$\mathsf{MerkleCRH}^{\mathsf{Sapling}}(\mathsf{layer}, \mathsf{left}, \mathsf{right}) := \mathsf{PedersenHash}(\text{“}\mathtt{Zcash\_PH}\text{”}, l \,||\, \mathsf{left} \,||\, \mathsf{right})$$
$$\text{where } l = \mathsf{I2LEBSP}_6(\mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1 - \mathsf{layer}).$$

Security requirement: $\mathsf{PedersenHash}$ must be collision-resistant.

Note: The prefix l provides domain separation between inputs at different layers of the *note commitment tree*. It is distinct from the prefix used in $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ as noted in §5.4.7.2 ***'Windowed Pedersen commitments'*** on p. 46.

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

$\mathsf{hSigCRH}$ is used to compute the value $\mathsf{h_{Sig}}$ in §4.3 ***'JoinSplit Descriptions'*** on p. 25.

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

where

$\mathsf{hSigInput} :=$

256-bit $\mathsf{randomSeed}$	256-bit $\mathsf{nf}^{\mathsf{old}}_{1}$ ...	256-bit $\mathsf{nf}^{\mathsf{old}}_{\mathrm{N}^{\mathsf{old}}}$	256-bit $\mathtt{joinSplitPubKey}$

.

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38.

Security requirement: $\mathsf{BLAKE2b\text{-}256}\big(\text{“}\mathtt{ZcashComputehSig}\text{”}, x\big)$ must be collision-resistant on x.

5.4.1.5 $\mathsf{CRH^{ivk}}$ Hash Function

$\mathsf{CRH^{ivk}}$ is used to derive the *incoming viewing key* ivk for a **Sapling** *shielded payment address*. For its use when generating an address see §4.2.2 ***'Sapling Key Components'*** on p. 24, and for its use in the *Spend statement* see §4.11.2 ***'Spend Statement (Sapling)'*** on p. 32.

It is defined as follows:

$$\mathsf{CRH^{ivk}}(\mathsf{ak}, \mathsf{nk}) := \mathsf{LEOS2IP}_{256}\big(\mathsf{BLAKE2s\text{-}256}\big(\text{“}\mathtt{Zcashivk}\text{”},\ \mathsf{crhInput}\big)\big) \bmod 2^{\ell_{\mathsf{ivk}}}$$

where

$\mathsf{crhInput} :=$

256-bit $\mathsf{LEBS2OSP}_{256}\big(\mathsf{ak}^*\big)$	256-bit $\mathsf{LEBS2OSP}_{256}\big(\mathsf{nk}^*\big)$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38.

Security requirement: $\mathsf{LEOS2IP}_{256}\big(\mathsf{BLAKE2s\text{-}256}\big(\text{“}\mathtt{Zcashivk}\text{”}, x\big)\big) \bmod 2^{\ell_{\mathsf{ivk}}}$ must be collision-resistant on a 64-byte input x. Note that this does not follow from collision-resistance of $\mathsf{BLAKE2s\text{-}256}$ (and the best possible concrete security is that of a 251-bit hash rather than a 256-bit hash), but it is a reasonable assumption given the design, structure, and cryptanalysis to date of $\mathsf{BLAKE2s}$.

Note: The variable output digest length feature of $\mathsf{BLAKE2s}$ does not support arbitrary bit lengths, otherwise that would have been used rather than external truncation. However, the protocol-specific personalization string together with truncation achieve essentially the same effect as using that feature.

5.4.1.6 DiversifyHash Hash Function

DiversifyHash is used to derive a *diversified base* from a *diversifier* in §4.2.2 ***'Sapling Key Components'*** on p. 24.

Let $\mathsf{GroupHash}^{\mathbb{J}}$ and U be as defined in §5.4.8.5 ***'Group Hash into Jubjub'*** on p. 51.

Define

$$\mathsf{DiversifyHash}(\mathsf{d}) := \mathsf{GroupHash}^{\mathbb{J}}_{U}(\texttt{"Zcash\_gd"}, \mathsf{LEBS2OSP}_{\ell_{\mathsf{d}}}(\mathsf{d}))$$

Security requirement: DiversifyHash must satisfy the Discrete Logarithm Independence property described in §4.1.10 ***'Group Hash'*** on p. 22. TODO: make this more precise.

5.4.1.7 Pedersen Hash Function

PedersenHash is an algebraic hash function with collision resistance (for fixed input length) derived from assumed hardness of the Discrete Logarithm Problem on the *Jubjub curve*. It is based on the work of David Chaum, Ivan Damgård, Jeroen van de Graaf, Jurjen Bos, George Purdy, Eugène van Heijst and Birgit Pfitzmann in [CDG1987], [BCP1988] and [CvHP1991], and of Mihir Bellare, Oded Goldreich, and Shafi Goldwasser in [BGG1995], with optimizations for efficient instantiation in *zk-SNARK circuits* by Sean Bowe and Daira Hopwood.

PedersenHash is used in the *incremental Merkle tree* over *note commitments* (§3.7 ***'Note Commitment Trees'*** on p. 15) and in the definition of *Pedersen commitments* (§5.4.7.2 ***'Windowed Pedersen commitments'*** on p. 46).

Let $\mathbb{J}$ be as defined in §5.4.8.3 ***'Jubjub'*** on p. 50.

Let $\mathsf{Extract}_{\mathbb{J}}$ be as defined in §5.4.8.4 ***'Hash Extractor for Jubjub'*** on p. 50.

Let $\mathsf{FindGroupHash}^{\mathbb{J}}$ be as defined in §5.4.8.5 ***'Group Hash into Jubjub'*** on p. 51.

Let $c := 63$.

Define $\mathcal{I} : \mathbb{B}^{\mathbb{Y}[8]} \times \mathbb{N} \to \mathbb{J}$ by:

$$\mathcal{I}^D_i := \mathsf{FindGroupHash}^{\mathbb{J}}\left(D, \boxed{\text{32-bit } \mathsf{floor}\left(\tfrac{i-1}{c}\right)}\right).$$

Define $\mathsf{PedersenHashToPoint}(D : \mathbb{B}^{\mathbb{Y}[8]}, M : \mathbb{B}^{[\mathbb{N}^+]})$ as follows:

Pad M to a multiple of 3 bits by appending zero bits, giving M'.

Let $n = \mathsf{ceiling}\left(\frac{\mathsf{length}(M')}{3 \cdot c}\right)$.

Split M' into n *"segments"* $M_{1\,..\,n}$ so that $M' = \mathsf{concat}_{\mathbb{B}}(M_{1\,..\,n})$, and each of $M_{1\,..\,n-1}$ is of length $3 \cdot c$ bits. (M_n may be shorter.)

Return $\sum_{i=1}^{n} [\langle M_i \rangle]\, \mathcal{I}^D_i : \mathbb{J}$.

where $\langle \bullet \rangle : \mathbb{B}^{[3 \cdot \{1\,..\,c\}]} \to \{-\frac{r_{\mathbb{J}}-1}{2}\,..\,\frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$ is defined as:

Let $k_i = \mathsf{length}(M_i)/3$.

Split M_i into 3-bit *"chunks"* $m_{1\,..\,k_i}$ so that $M_i = \mathsf{concat}_{\mathbb{B}}(m_{1\,..\,k_i})$.

Write each m_j as $[s^j_0, s^j_1, s^j_2]$, and let $\mathsf{enc}(m_j) = (1 - 2 \cdot s^j_2) \cdot (1 + s^j_0 + 2 \cdot s^j_1) : \mathbb{Z}$.

Let $\langle M_i \rangle = \sum_{j=1}^{k_i} \mathsf{enc}(m_j) \cdot 2^{4 \cdot (j-1)}$.

Finally, define $\mathsf{PedersenHash} \colon \mathbb{B}^{\mathbb{Y}^{[8]}} \times \mathbb{B}^{[\mathbb{N}^+]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ by:

$$\mathsf{PedersenHash}(D, M) := \mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}(\mathsf{Extract}_{\mathbb{J}}(\mathsf{PedersenHashToPoint}(D, M))).$$

See §A.3.3.9 *'Pedersen hash'* on p. 99 for rationale and efficient circuit implementation of these functions.

Security requirement: PedersenHash and PedersenHashToPoint are required to be collision-resistant between inputs of fixed length, for a given personalization input D. No other security properties commonly associated with *hash functions* are needed.

Theorem 5.4.1. *The encoding function* $\langle \bullet \rangle$ *is injective.*

Proof. We first check that the range of $\sum_{j=1}^{k_i} \mathsf{enc}(m_j) \cdot 2^{4 \cdot (j-1)}$ is a subset of the allowable range $\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$.

The range of this expression is a subset of $\{-\Delta \,..\, \Delta\} \setminus \{0\}$ where $\Delta = 4 \cdot \sum_{i=0}^{c-1} 2^{4 \cdot i} = 4 \cdot \frac{2^{4 \cdot c}}{15}$.

When $c = 63$, we have

$$4 \cdot \frac{2^{4 \cdot c}}{15} = \texttt{0x444444444444444444444444444444444444444444444444444444444444444}$$

$$\frac{r_{\mathbb{J}}-1}{2} = \texttt{0x73EDA753299D7D483339D80809A1D8053341049E6640841684B872F6B7B965B}$$

so the required condition is met. This implies that there is no "wrap around" and so $\sum_{j=1}^{k_i} \mathsf{enc}(m_j) \cdot 2^{4 \cdot (j-1)}$ may be treated as an integer expression.

enc is injective. In order to prove that $\langle \bullet \rangle$ is injective, consider $\langle \bullet \rangle^{\Delta} \colon \mathbb{B}^{[3 \cdot \{1 \,..\, c\}]} \to \{0 \,..\, 2 \cdot \Delta\}$ such that $\langle M_i \rangle^{\Delta} = \langle M_i \rangle + \Delta$. With k_i and m_j defined as above, we have $\langle M_i \rangle^{\Delta} = \sum_{j=1}^{k_i} \mathsf{enc}'(m_j) \cdot 2^{4 \cdot (j-1)}$ where $\mathsf{enc}'(m_j) = \mathsf{enc}(m_j) + 4$ is in $\{0 \,..\, 8\}$ and enc' is injective. Express this sum in hexadecimal; then each m_j affects only one hex digit, and it is easy to see that $\langle \bullet \rangle^{\Delta}$ is injective. Therefore so is $\langle \bullet \rangle$. □

Since the security proof from [BGG1995, Appendix A] depends only on the encoding being injective and its range not including zero, the proof can be adapted straightforwardly to show that PedersenHashToPoint is collision-resistant under the same assumptions and security bounds. Because $\mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}$ and $\mathsf{Extract}_{\mathbb{J}}$ are injective, it follows that PedersenHash is equally collision-resistant.

Theorem 5.4.2. $\mathsf{Uncommitted}^{\mathsf{Sapling}} = \mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}(1)$ *is not in the range of* PedersenHash.

Proof. By the definition of PedersenHash, $\mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}(1)$ can be in the range of PedersenHash only if there exist $D \colon \mathbb{B}^{\mathbb{Y}^{[8]}}$ and $M \colon \mathbb{B}^{[\mathbb{N}^+]}$ such that $\mathsf{Extract}_{\mathbb{J}}(\mathsf{PedersenHashToPoint}(D, M)) = 1$. The latter can only be the affine-Edwards u-coordinate of a point in $\mathbb{J}$. We show that there are no points in $\mathbb{J}$ with affine-Edwards u-coordinate 1. Suppose for a contradiction that $(u, \mathsf{v}) \in \mathbb{J}$ for $u = 1$ and some $\mathsf{v} \colon \mathbb{F}_{r_{\mathbb{J}}}$. By writing the curve equation as $\mathsf{v}^2 = (1 - a_{\mathbb{J}} \cdot u^2)/(1 - d_{\mathbb{J}} \cdot u^2)$, and noting that $1 - d_{\mathbb{J}} \cdot u^2 \neq 0$, we have $\mathsf{v}^2 = (1 - a_{\mathbb{J}})/(1 - d_{\mathbb{J}})$. The right-hand-side is a nonsquare in $\mathbb{F}_{r_{\mathbb{J}}}$, so there are no solutions for v (contradiction). □

5.4.1.8 Mixing Pedersen Hash Function

A mixing *Pedersen hash* is used to compute ρ from cm and pos in §4.10 ***'Note Commitments and Nullifiers'*** on p. 31. It takes as input a *Pedersen commitment* P, and hashes it with another input x.

Let $\mathcal{J} = \mathsf{FindGroupHash}^{\mathbb{J}}(\text{"}\mathtt{Zcash\_J\_}\text{"}, \text{""})$.

We define $\mathsf{MixingPedersenHash} ⦂ \mathbb{J} \times \{0\,..\,r_{\mathbb{J}} - 1\} \to \mathbb{J}$ by:

$$\mathsf{MixingPedersenHash}(P, x) := P + [x]\,\mathcal{J}.$$

Security requirement: The function

$$(r, M, x) ⦂ \{0\,..\,r_{\mathbb{J}} - 1\} \times \mathbb{B}^{[\mathbb{N}^+]} \times \{0\,..\,r_{\mathbb{J}} - 1\} \mapsto \mathsf{MixingPedersenHash}(\mathsf{WindowedPedersenCommit}_r(M), x) ⦂ \mathbb{J}$$

must be collision-resistant on (r, M, x).

See §A.3.3.10 ***'Mixing Pedersen hash'*** on p. 100 for efficient circuit implementation of this function.

5.4.1.9 Equihash Generator

$\mathsf{EquihashGen}_{n,k}$ is a specialized *hash function* that maps an input and an index to an output of length n bits. It is used in §7.6.1 ***'Equihash'*** on p. 65.

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

Let powcount(g) := | 32-bit g | .

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

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

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

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

Indices of bits in T are 1-based.

$\mathsf{BLAKE2b\text{-}}\ell(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38.

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

Note: When EquihashGen is evaluated for sequential indices, as in the Equihash solving process (§7.6.1 ***'Equihash'*** on p. 65), 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. 17, are all instantiated using the *SHA-256 compression function* defined in §5.4.1.1 ***'SHA-256 and SHA256Compress Hash Functions'*** on p. 37:

$$\mathsf{PRF}^{\mathsf{addr}}_{x}(t) := \mathsf{SHA256Compress}\left(\begin{array}{|c|c|c|c|c|c|c|}\hline 1 & 1 & 0 & 0 & \text{252-bit } x & \text{8-bit } t & [0]^{248} \\ \hline\end{array}\right)$$

$$\mathsf{PRF}^{\mathsf{nf}}_{\mathsf{a_{sk}}}(\rho) := \mathsf{SHA256Compress}\left(\begin{array}{|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 0 & \text{252-bit } \mathsf{a_{sk}} & \text{256-bit } \rho \\ \hline\end{array}\right)$$

$$\mathsf{PRF}^{\mathsf{pk}}_{\mathsf{a_{sk}}}(i, \mathsf{h_{Sig}}) := \mathsf{SHA256Compress}\left(\begin{array}{|c|c|c|c|c|c|}\hline 0 & i\text{-}1 & 0 & 0 & \text{252-bit } \mathsf{a_{sk}} & \text{256-bit } \mathsf{h_{Sig}} \\ \hline\end{array}\right)$$

$$\mathsf{PRF}^{\rho}_{\varphi}(i, \mathsf{h_{Sig}}) := \mathsf{SHA256Compress}\left(\begin{array}{|c|c|c|c|c|c|}\hline 0 & i\text{-}1 & 1 & 0 & \text{252-bit } \varphi & \text{256-bit } \mathsf{h_{Sig}} \\ \hline\end{array}\right)$$

Security requirements:

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

Note: The first four bits –i.e. the most significant four bits of the first byte– are used to separate distinct uses of SHA256Compress, ensuring that the functions are independent. As well as the inputs shown here, bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function; see §5.4.7.1 ***'Sprout Note Commitments'*** on p. 46.

(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. In fact since **Sapling** switches to non-SHA256Compress-based cryptographic primitives, these extensions are unlikely to be necessary.)

Let LEOS2IP be as defined in §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 36.

$\mathsf{PRF}^{\mathsf{expand}}$ is used in §4.2.2 ***'Sapling Key Components'*** on p. 24 to derive the *spend authorizing key* ask and the *proof authorizing key* nsk.

It is instantiated using the BLAKE2b *hash function* defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38:

$$\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(t) := \mathsf{LEOS2IP}_{512}\Big(\mathsf{BLAKE2b\text{-}512}\Big(\texttt{"Zcash\_ExpandSeed"}, \begin{array}{|c|c|}\hline \mathsf{LEBS2OSP}_{256}(\mathsf{sk}) & \text{8-bit } t \\ \hline\end{array}\Big)\Big) \pmod{r_{\mathbb{J}}}$$

Security requirement: $\mathsf{BLAKE2b\text{-}512}\Big(\texttt{"Zcash\_ExpandSeed"}, \begin{array}{|c|c|}\hline \mathsf{LEBS2OSP}_{256}(\mathsf{sk}) & \text{8-bit } t \\ \hline\end{array}\Big)$ must be a PRF for output range $\mathbb{BY}^{[64]}$ when keyed by the bits corresponding to sk, with input in the bits corresponding to t. In that case it follows that $\mathsf{PRF}^{\mathsf{expand}}$ is also a PRF for output range $\mathbb{F}_{r_{\mathbb{J}}}$, since $\mathsf{LEOS2IP}_{512} : \mathbb{BY}^{[64]} \to \{0 \,..\, 2^{512} - 1\}$ is injective and 2^{512} is large compared to $r_{\mathbb{J}}$.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used to derive the *nullifier* for a **Sapling** *note*. It is instantiated using the BLAKE2s *hash function* defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38:

$$\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}}(\rho) := \mathsf{BLAKE2s\text{-}256}\Big(\texttt{"Zcash\_nf"}, \begin{array}{|c|c|}\hline \mathsf{LEBS2OSP}_{256}(\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})) & \mathsf{LEBS2OSP}_{256}(\mathsf{repr}_{\mathbb{J}}(\rho)) \\ \hline\end{array}\Big).$$

Security requirement: $\mathsf{BLAKE2s\text{-}256}\Big(\texttt{"Zcash\_nf"}, \boxed{\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big)} \boxed{\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\rho)\big)}\Big)$ must be a collision-resistant PRF for output range $\mathbb{BY}^{[32]}$ when keyed by the bits corresponding to nk, with input in the bits corresponding to ρ.

5.4.3 Authenticated One-Time Symmetric Encryption

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

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

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

Note: The "IETF" definition of AEAD_CHACHA20_POLY1305 from [RFC-7539] is used; this has a 32-bit block count and a 96-bit nonce, rather than a 64-bit block count and 64-bit nonce as in the original definition of ChaCha20.

5.4.4 Key Agreement and Derivation

5.4.4.1 Sprout Key Agreement

The *key agreement scheme* specified in §4.1.4 ***'Key Agreement'*** on p. 18 is instantiated using Curve25519 [Bern2006] as follows.

Let $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ and $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{SharedSecret}$ be the type of Curve25519 public keys (i.e. a sequence of 32 bytes), and let $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$ be the type of Curve25519 secret keys.

Let $\mathsf{Curve25519}(\underline{n}, \underline{q})$ be the result of point multiplication of the Curve25519 public key represented by the byte sequence $\underline{q}$ by the Curve25519 secret key represented by the byte sequence $\underline{n}$, as defined in [Bern2006, section 2].

Let $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Base} := \underline{9}$ be the public byte sequence representing the Curve25519 base point.

Let $\mathsf{clamp}_{\mathsf{Curve25519}}(\underline{x})$ take a 32-byte sequence $\underline{x}$ as input and return a byte sequence representing a Curve25519 private key, with bits "clamped" as described in [Bern2006, section 3]: "clear bits 0, 1, 2 of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte." Here the bits of a byte are numbered such that bit b has numeric weight 2^b.

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

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

5.4.4.2 Sprout Key Derivation

The *Key Derivation Function* specified in §4.1.5 ***'Key Derivation'*** on p. 18 is instantiated using BLAKE2b-256 as follows:

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

where:

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

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

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38.

5.4.4.3 Sapling Key Agreement

The *key agreement scheme* specified in §4.1.4 ***'Key Agreement'*** on p. 18 is instantiated using Diffie-Hellman with cofactor multiplication on Jubjub as follows.

Let $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public}$ and $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret}$ be the type of compressed Jubjub points CompressedEdwardsJubjub, and let $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private}$ be the type of Jubjub secret keys. TODO: expand this

5.4.4.4 Sapling Key Derivation

The $\mathsf{KDF}^{\mathsf{Sapling}}$ *Key Derivation Function* is specified in §4.1.5 ***'Key Derivation'*** on p. 18.

It is instantiated using BLAKE2b-256 as follows:

$$\mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{idx}, \mathsf{sharedSecret}, \mathsf{epk}) := \mathsf{BLAKE2b\text{-}256}\big(\texttt{"Zcash\_SaplingKDF"}, \mathsf{kdfinput}\big).$$

where:

$$\mathsf{kdfinput} := \boxed{\text{32-bit idx}}\boxed{\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{sharedSecret})\big)}\boxed{\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})\big)}.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 ***'BLAKE2 Hash Function'*** on p. 38.

5.4.5 JoinSplit Signature

JoinSplitSig is specified in §4.1.6 ***'Signature'*** on p. 19.

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 of order ℓ on the Ed25519 curve;

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 $2^{255} - 19$.

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 Spend Authorization Signature

SpendAuthSig is a signature scheme with re-randomizable keys specified in §4.1.6.1 ***'Signature with Re-Randomizable Keys'*** on p. 20.

It is instantiated as EdJubjub, which is defined as EdDSA [BJLSY2015] over the *Jubjub curve* which these additional constraints: TODO: ...

[FKMSSS2016]

5.4.7 Commitment schemes

5.4.7.1 Sprout Note Commitments

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ specified in §4.1.7 ***‘Commitment’*** on p. 21 is instantiated using SHA-256 as follows:

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

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 rcm in the second block of SHA-256 input, with input to the PRF in the remaining bits of the block and the chaining variable.

5.4.7.2 Windowed Pedersen commitments

We construct *“windowed” Pedersen commitments* by reusing the *Pedersen hash* construction from §5.4.1.7 ***‘Pedersen Hash Fu*** on p. 40, and adding a randomized point on the *Jubjub curve* (see §5.4.8.3 ***‘Jubjub’*** on p. 50):

$$\mathsf{WindowedPedersenCommit}_r(s) := \mathsf{PedersenHashToPoint}(\texttt{“Zcash\_PH”}, s) + [r]\,\mathsf{FindGroupHash}^{\mathbb{J}}(\texttt{“Zcash\_PH”}, \texttt{“r”})$$

See §A.3.5 ***‘Windowed Pedersen Commitment’*** on p. 101 for rationale and efficient circuit implementation of this function.

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ specified in §4.1.7 ***‘Commitment’*** on p. 21 is instantiated using WindowedPedersenCom as follows:

$$\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g_d}^*, \mathsf{pk_d}^*, \mathsf{v}) := \mathsf{WindowedPedersenCommit}_{\mathsf{rcm}}([1]^6 \,||\, \mathsf{g_d}^* \,||\, \mathsf{pk_d}^* \,||\, \mathsf{I2LEBSP}_{64}(\mathsf{v})).$$

Security requirements:

- WindowedPedersenCommit must be a computationally binding and at least computationally hiding *commitment scheme*.
- $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ must be a computationally binding and at least computationally hiding *commitment scheme*.

(They are in fact unconditionally hiding *commitment schemes*.)

Note: The prefix $[1]^6$ distinguishes the use of WindowedPedersenCommit in $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ from the layer prefix used in $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ (see §5.4.1.3 ***‘Merkle Tree Hash Function’*** on p. 38). The latter is a 6-bit little-endian encoding of an integer in $\{0 \mathinner{..} \mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1\}$, and so cannot collide with $[1]^6$ because $\mathsf{MerkleDepth}^{\mathsf{Sapling}} < 64$.

5.4.7.3 Homomorphic Pedersen commitments

The windowed Pedersen commitments defined in the preceding section are highly efficient, but they do not support the homomorphic property we need when instantiating ValueCommit (see §4.9 ***'Balance'*** on p. 30 and §3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 14).

In order to support this property, we also define *"homomorphic" Pedersen commitments* as follows:

$$\mathsf{HomomorphicPedersenCommit}_{\mathsf{rcv}}(D, \mathsf{v}) := [\mathsf{v}]\,\mathsf{FindGroupHash}^{\mathbb{J}}\big(D, \text{“}\mathtt{v}\text{”}\big) + [\mathsf{rcv}]\,\mathsf{FindGroupHash}^{\mathbb{J}}\big(D, \text{“}\mathtt{r}\text{”}\big)$$

See §A.3.6 ***'Homomorphic Pedersen Commitment'*** on p. 101 for rationale and efficient circuit implementation of this function.

The commitment scheme ValueCommit specified in §4.1.7 ***'Commitment'*** on p. 21 is instantiated using HomomorphicPedersenComm as follows:

$$\mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v}) := \mathsf{HomomorphicPedersenCommit}_{\mathsf{rcv}}(\text{“}\mathtt{Zcash\_cv}\text{”}, \mathsf{v}).$$

Security requirements:

- HomomorphicPedersenCommit must be a computationally binding and at least computationally hiding *commitment scheme*, for a given personalization input D.
- ValueCommit must be a computationally binding and at least computationally hiding *commitment scheme*.

(They are in fact unconditionally hiding *commitment schemes*.)

5.4.8 Represented Groups and Pairings

5.4.8.1 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 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 $r_{\mathbb{G}}{}^{\mathrm{th}}$ roots of unity in $\mathbb{F}^*_{q_{\mathbb{G}}{}^{12}}$.

Let $\hat{e}_{\mathbb{G}}$ be the optimized ate pairing 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 $\mathcal{P}_{\mathbb{G}_1} : \mathbb{G}_1^* := (1, 2)$.

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

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

Define I2BEBSP $\mathrel{⦂} (\ell \mathrel{⦂} \mathbb{N}) \times \{0 \mathinner{..} 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

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

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

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

.

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

- Define FE2IP $\mathrel{⦂} \mathbb{F}_{q_\mathbb{G}}[t]/(t^2+1) \to \{0 \mathinner{..} {q_\mathbb{G}}^2 - 1\}$ such that FE2IP$(a_{w,1} \cdot t + a_{w,0}) = a_{w,1} \cdot q + a_{w,0}$.
- Let $x =$ FE2IP(x_P), $y =$ FE2IP(y_P), and $y' =$ 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 I2BEBSP$_{512}(x)$

.

Non-normative notes:

- The use of big-endian order by I2BEBSP is different from the encoding of most other integers in this protocol. The encodings for $\mathbb{G}_{1,2}^*$ are consistent with the definition of EC2OSP for compressed curve points in [IEEE2004, section 5.5.6.2]. The LSB compressed form (i.e. EC2OSP-XL) is used for points in $\mathbb{G}_1^*$, and the SORT compressed form (i.e. EC2OSP-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.

5.4.8.2 BLS12-381

The *represented pairing* BLS12-381 is defined in this section. Parameters are taken from [Bowe2017].

Let $q_{\mathbb{S}} := 4002409555221667393417789825735904156556882819939007885332058136124031650490837864442687629129015664037894272559787$.

Let $r_{\mathbb{S}} := 52435875175126190479447740508185965837690552500527637822603658699938581184513$.

Let $u_{\mathbb{S}} := -15132376222941642752$.

Let $b_{\mathbb{S}} := 4$.

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

Let $\mathbb{S}_1$ be the group of points on a Barreto–Lynn–Scott curve $E_{\mathbb{S}_1}$ over $\mathbb{F}_{q_{\mathbb{S}}}$ with equation $y^2 = x^3 + b_{\mathbb{S}}$. This curve has embedding degree 12 with respect to $r_{\mathbb{S}}$.

Let $\mathbb{S}_2$ be the subgroup of order $r_{\mathbb{S}}$ in the sextic twist $E_{\mathbb{S}_2}$ of $\mathbb{S}_1$ over $\mathbb{F}_{{q_{\mathbb{S}}}^2}$ with equation $y^2 = x^3 + 4(i + 1)$, where $i \typecolon \mathbb{F}_{{q_{\mathbb{S}}}^2}$.

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

Let $\mathbb{S}_T$ be the subgroup of ${r_{\mathbb{S}}}^{\text{th}}$ roots of unity in $\mathbb{F}^*_{{q_{\mathbb{S}}}^{12}}$.

Let $\hat{e}_{\mathbb{S}}$ be the optimized ate pairing of type $\mathbb{S}_1 \times \mathbb{S}_2 \rightarrow \mathbb{S}_T$.

For $i \typecolon \{1\,..\,2\}$, let $\mathcal{O}_{\mathbb{S}_i}$ be the point at infinity in $\mathbb{S}_i$, and let $\mathbb{S}^*_i := \mathbb{S}_i \setminus \{\mathcal{O}_{\mathbb{S}_i}\}$.

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

$$\begin{aligned}\text{Let } \mathcal{P}_{\mathbb{S}_2} \typecolon \mathbb{S}^*_2 := (&1155973203298638710799100402139228578392581286182119253091740315145239180563 4 \cdot t + \\ &1085704699902305713594457076223282948137075635957851808699051999328565585278 1, \\ &408236787586343368133220340314543556831685132759340120810574107621412009353 1 \cdot t + \\ &849565392312343141760497324748927243841819058726360014877028064930695810193 0).\end{aligned}$$

$\mathcal{P}_{\mathbb{S}_1}$ and $\mathcal{P}_{\mathbb{S}_2}$ are generators of $\mathbb{S}_1$ and $\mathbb{S}_2$ respectively.

Define $\mathsf{I2BEBSP} \typecolon (\ell \typecolon \mathbb{N}) \times \{0\,..\,2^\ell - 1\} \rightarrow \mathbb{B}^{[\ell]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

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

- The field elements x_P and $y_P \typecolon \mathbb{F}_{q_{\mathbb{S}}}$ are represented as integers x and $y \typecolon \{0\,..\,q_{\mathbb{S}} - 1\}$.
- Let $\tilde{y} = \begin{cases} 1, & \text{if } y > q_{\mathbb{S}} - y \\ 0, & \text{otherwise.} \end{cases}$
- P is encoded as

1	0	1-bit $\tilde{y}$	381-bit $\mathsf{I2BEBSP}_{381}(x)$

.

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

- Define $\mathsf{FE2IPP} \typecolon \mathbb{F}_{q_{\mathbb{S}}}[t]/(t^2 + 1) \rightarrow \{0\,..\,q_{\mathbb{S}} - 1\}^{[2]}$ such that $\mathsf{FE2IPP}(a_{w,1} \cdot t + a_{w,0}) = [a_{w,1}, a_{w,0}]$.
- Let $x = \mathsf{FE2IPP}(x_P)$, $y = \mathsf{FE2IPP}(y_P)$, and $y' = \mathsf{FE2IPP}(-y_P)$.
- Let $\tilde{y} = \begin{cases} 1, & \text{if } y > y' \text{ lexicographically} \\ 0, & \text{otherwise.} \end{cases}$
- P is encoded as

1	0	1-bit $\tilde{y}$	381-bit $\mathsf{I2BEBSP}_{381}(x_1)$	384-bit $\mathsf{I2BEBSP}_{384}(x_2)$

.

Non-normative notes:

- The encodings for $\mathbb{S}^*_{1,2}$ are specific to **Zcash**.
- The points at infinity $\mathcal{O}_{\mathbb{S}_{1,2}}$ never occur in proofs and have no defined encodings in this protocol.
- Algorithms for decompressing points from the encodings of $\mathbb{S}^*_{1,2}$ are defined analogously to those for $\mathbb{G}^*_{1,2}$ in §5.4.8.1 ***'BN-254'*** on p. 47, taking into account that the SORT compressed form (not the LSB compressed form) is used for $\mathbb{G}^*_1$.
- A rational point $P \neq \mathcal{O}_{\mathbb{S}_2}$ on the curve $E_{\mathbb{S}_2}$ can be verified to be of order $r_{\mathbb{S}}$, and therefore in $\mathbb{S}^*_2$, by checking that $r_{\mathbb{S}} \cdot P = \mathcal{O}_{\mathbb{S}_2}$.

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

5.4.8.3 Jubjub

The *represented group* Jubjub is defined in this section.

Let $q_{\mathbb{J}} := r_{\mathbb{S}}$, as defined in §5.4.8.2 ***'BLS12-381'*** on p. 48.

Let $r_{\mathbb{J}} := 6554484396890773809930967563523245729705921265872317281365359162392183254199$.

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

Let $a_{\mathbb{J}} := -1$.

Let $d_{\mathbb{J}} := -10240/10241 \pmod{q_{\mathbb{J}}}$.

Let $\mathbb{J}$ be the group of points (u, v) on a twisted Edwards curve $E_{\mathbb{J}}$ over $\mathbb{F}_{q_{\mathbb{J}}}$ with equation $a_{\mathbb{J}} \cdot u^2 + v^2 = 1 + d_{\mathbb{J}} \cdot u^2 \cdot v^2$. The zero point with coordinates $(0, 1)$ is denoted $\mathcal{O}_{\mathbb{J}}$. $\mathbb{J}$ has order $8 \cdot r_{\mathbb{J}}$.

Let $\ell_{\mathbb{J}} := 256$.

Define $\mathsf{I2LEBSP} : (\ell : \mathbb{N}) \times \{0 \,..\, 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 36.

Define $\mathsf{repr}_{\mathbb{J}} : \mathbb{J} \to \mathbb{B}^{[\ell_{\mathbb{J}}]}$ such that $\mathsf{repr}_{\mathbb{J}}(u, v) = \mathsf{I2LEBSP}_{256}(v + 2^{255} \cdot \tilde{u})$, where $\tilde{u} = u \bmod 2$.

Let $\mathsf{abst}_{\mathbb{J}} : \mathbb{B}^{[\ell_{\mathbb{J}}]} \to \mathbb{J} \cup \{\bot\}$ be the left inverse of $\mathsf{repr}_{\mathbb{J}}$ such that if S is not in the range of $\mathsf{repr}_{\mathbb{J}}$, then $\mathsf{abst}_{\mathbb{J}}(S) = \bot$.

Non-normative notes:

- The encoding of a compressed twisted Edwards point used here is consistent with that used in EdDSA [BJLSY2015] for public keys and the R element of a signature.
- Algorithms for decompressing points from the encoding of $\mathbb{J}$ are given in [BJLSY2015, "Encoding and parsing curve points"

When computing square roots in $\mathbb{F}_{q_{\mathbb{J}}}$ 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.

This specification requires "strict" parsing as defined in [BJLSY2015, "Encoding and parsing integers"].

Note that algorithms elsewhere in this specification that use Jubjub may impose other conditions on points, for example that they are not the zero point, or are in the large prime-order subgroup.

5.4.8.4 Hash Extractor for Jubjub

Let $\mathcal{U}((u, v)) = u$ and let $\mathcal{V}((u, v)) = v$.

Let $\mathsf{Extract}_{\mathbb{J}} : \mathbb{J} \to \mathbb{F}_{q_{\mathbb{J}}}$ be $\mathcal{U}$.

Let G be the subgroup of $\mathbb{J}$ of order $r_{\mathbb{J}}$ (an odd prime).

Facts: The point $(0, 1) = \mathcal{O}_{\mathbb{J}}$, and the point $(0, -1)$ has order 2 in $\mathbb{J}$.

Lemma. *Let $P = (u, v) \in G$. Then $(u, -v) \notin G$.*

Proof. If $P = \mathcal{O}_{\mathbb{J}}$ then $(u, -v) = (0, -1) \notin G$. Else, P is of odd-prime order. Note that $v \neq 0$. (If $v = 0$ then $a \cdot u^2 = 1$, and so applying the doubling formula gives $[2]\,P = (0, -1)$, then $[4]\,P = (0, 1) = \mathcal{O}_{\mathbb{J}}$; contradiction since then P would not be of odd-prime order.) Therefore, $-v \neq v$. Now suppose $(u, -v) = Q$ is a point in G. Then by applying the doubling formula we have $[2]\,Q = -[2]\,P$. But also $[2]\,(-P) = -[2]\,P$. Therefore either $Q = -P$ (then $\mathcal{V}(Q) = \mathcal{V}(-P)$; contradiction since $-v \neq v$), or doubling is not injective on G (contradiction since G is of odd order [KvE2013]). □

Theorem 5.4.3. *$\mathcal{U}$ is injective on G.*

Proof. By writing the curve equation as $v^2 = (1 - a \cdot u^2)/(1 - d \cdot u^2)$, and noting that the potentially exceptional case $1 - d \cdot u^2 = 0$ does not occur for a complete twisted Edwards curve, we see that for a given u there can be at most two possible solutions for v, and that if there are two solutions they can be written as v and $-v$. In that case by the Lemma, at most one of (u, v) and $(u, -v)$ is in G. Therefore, $\mathcal{U}$ is injective on points in G. □

5.4.8.5 Group Hash into Jubjub

TODO: Define CRS using the MPC randomness beacon.

Let BLAKE2s-256 be as defined in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 38.

Let LEOS2IP be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

Let $\mathsf{abst}_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 50.

Let $D : \mathbb{BY}^{[8]}$ be an 8-byte domain separator, and let $M : \mathbb{BY}^{[\mathbb{N}]}$ be the hash input.

The hash $\mathsf{GroupHash}^{\mathbb{J}}_{\mathsf{CRS}}(D, M)$ is calculated as follows:

$P := \mathsf{abst}_{\mathbb{J}}\big(\mathsf{LEOS2IP}_{256}\big(\mathsf{BLAKE2s\text{-}256}\big(D,\ \mathsf{CRS} \,||\, M\big)\big)\big)$

If $P = \bot$ then return $\bot$.

$Q := [8]\,P$

If $Q = \mathcal{O}_{\mathbb{J}}$ then return $\bot$, else return Q.

Define $\mathsf{first} : (\mathbb{N} \to T \cup \{\bot\}) \to T \cup \{\bot\}$ so that $\mathsf{first}(f) = f(i)$ where i is the least integer in $\{0 \,..\, 255\}$ such that $f(i) \neq \bot$, or $\bot$ if no such i exists.

Let $\mathsf{FindGroupHash}^{\mathbb{J}}(D, M) = \mathsf{first}(i : \mathbb{N} \mapsto \mathsf{GroupHash}^{\mathbb{J}}_{\mathsf{CRS}}(D, M \,||\, [i]) : \mathbb{J})$.

Notes:

- The BLAKE2s-256 chaining variable after processing CRS may be precomputed.
- For random input, $\mathsf{FindGroupHash}^{\mathbb{J}}$ returns $\bot$ with probability approximately 2^{-256}. In the **Zcash** protocol, uses of $\mathsf{FindGroupHash}^{\mathbb{J}}$ never return $\bot$.

5.4.9 Zero-Knowledge Proving Systems

5.4.9.1 PHGR13

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

A proof consists of a tuple $(\pi_A \,{:}\, \mathbb{G}_1^*,\ \pi'_A \,{:}\, \mathbb{G}_1^*,\ \pi_B \,{:}\, \mathbb{G}_2^*,\ \pi'_B \,{:}\, \mathbb{G}_1^*,\ \pi_C \,{:}\, \mathbb{G}_1^*,\ \pi'_C \,{:}\, \mathbb{G}_1^*,\ \pi_K \,{:}\, \mathbb{G}_1^*,\ \pi_H \,{:}\, \mathbb{G}_1^*)$. It is computed using the parameters above as described in [BCTV2015, Appendix B].

Note: Many details of the *proving system* are beyond the scope of this protocol document. For example, the *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 ***'Sprout** zk-SNARK Parameters'* on p. 58), 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:

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

The resulting proof size is 296 bytes.

In addition to the steps to verify a proof given in [BCTV2015, Appendix B], the verifier **MUST** check, for the encoding of each element, that:

- the lead byte is of the required form;
- the remaining bytes encode a big-endian representation of an integer in $\{0\,..\,q_{\mathbb{S}}-1\}$ or (in the case of π_B) $\{0\,..\,q_{\mathbb{S}}^2-1\}$;
- the encoding represents a point in $\mathbb{G}_1^*$ or (in the case of π_B) $\mathbb{G}_2^*$, including checking that it is of order $r_{\mathbb{G}}$ in the latter case.

5.4.9.2 Groth16

Sapling uses *zk-SNARKs* generated by the *bellman* library, with the *proving system* described in [Grot2016].

A proof consists of a tuple $(\pi_A \,{:}\, \mathbb{S}_1^*,\ \pi_B \,{:}\, \mathbb{S}_2^*,\ \pi_C \,{:}\, \mathbb{S}_1^*)$. It is computed using the parameters above as described in [Grot2016].

Note: The *quadratic arithmetic programs* verifying the *Spend statement* and *Output statement* are described in Appendix A *'Circuit Design'* on p. 90. However, many other details of the *proving system* are beyond the scope of this protocol document. For example, the expressions of the *Spend statement* and *Output statement* as *Rank 1 Constraint Systems* 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 §5.8 ***'Sapling** zk-SNARK Parameters'* on p. 58), and a *proving system* implementation that is interoperable with the *bellman* library used by **Zcash**, to ensure compatibility.

Encoding of Groth16 Proofs A Groth16 proof is encoded by concatenating the encodings of its elements:

384-bit π_A	768-bit π_B	384-bit π_C

The resulting proof size is 192 bytes.

In addition to the steps to verify a proof given in [Grot2016], the verifier **MUST** check, for the encoding of each element, that:

- the leading bitfield is of the required form;
- the remaining bits encode a big-endian representation of an integer in $\{0 \mathinner{..} q_{\mathbb{S}}-1\}$ or (in the case of π_B) two integers in that range;
- the encoding represents a point in $\mathbb{S}_1^*$ or (in the case of π_B) $\mathbb{S}_2^*$, including checking that it is of order $r_{\mathbb{S}}$ 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. 12, 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 **Sprout** *note plaintext* (denoted **np**) consists of $(\mathsf{v}, \rho, \mathsf{rcm}, \mathsf{memo})$.

[**Sapling** onward] The *note plaintext* in each *Output description* is encrypted to the *diversified transmission key* $\mathsf{pk_d}$. Each **Sapling** *note plaintext* (denoted **np**) consists of $(\mathsf{d}, \mathsf{v}, \mathsf{rcm}, \mathsf{memo})$.

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. 11.

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

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

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

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

8-bit `0x01`	88-bit d	64-bit v	256-bit rcm	memo (512 bytes)

- A byte, `0x01`, indicating this version of the encoding of a **Sapling** *note plaintext*.
- 11 bytes specifying d.
- 8 bytes specifying v.
- 32 bytes specifying rcm.
- 512 bytes specifying memo.

5.6 Encodings of Addresses and Keys

This section describes how **Zcash** encodes *shielded payment addresses*, *incoming viewing keys*, and *spending keys*.

Addresses and keys can be encoded as a byte sequence; this is called the *raw encoding*. This byte sequence can then be further encoded using Base58Check. The Base58Check layer is the same as for upstream **Bitcoin** addresses [Bitc-Base58].

For **Sapling**-specific key and address formats, Bech32 [BIP-173] is used instead of Base58Check.

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) [BIP-13] or P2PKH (Pay to Public Key Hash) [Bitc-P2PKH] addresses.

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 [Bitc-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** [Bitc-Base58], for both the production and test networks.

5.6.3 Sprout Shielded Payment Addresses

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{pk_{enc}}$ is a $\mathsf{KA^{Sprout}.Public}$ key (see §5.4.4.1 ***'Sprout Key Agreement'*** on p. 44), for use with the encryption scheme defined in §4.12 ***'In-band secret distribution'*** on p. 34. These components are derived from a *spending key* as described in §4.2.1 ***'Sprout Key Components'*** on p. 24.

The raw encoding of a **Sprout** *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 **Sprout** *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 [Bern2006].

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 Sapling Shielded Payment Addresses

A **Sapling** *shielded payment address* consists of $\mathsf{d} \colon \mathbb{B}^{[\ell_{\mathsf{d}}]}$ and $\mathsf{pk_d} \colon \mathsf{KA^{Sapling}.Public}$.

d is a sequence of 11 bytes. $\mathsf{pk_d}$ is an encoding of a $\mathsf{KA^{Sapling}.Public}$ key (see §5.4.4.3 ***'Sapling Key Agreement'*** on p. 45), for use with the encryption scheme defined in §4.12 ***'In-band secret distribution'*** on p. 34. These components are derived as described in §4.2.2 ***'Sapling Key Components'*** on p. 24.

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

$\mathsf{LEBS2OSP}_{88}(\mathsf{d})$	$\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d})\big)$

- 11 bytes specifying d.
- 32 bytes specifying the compressed Edwards encoding of $\mathsf{pk_d}$ (see §5.4.8.3 ***'Jubjub'*** on p. 50).

When decoding the representation of $\mathsf{pk_d}$, the address is not valid if $\mathsf{abst}_{\mathbb{J}}$ returns $\bot$.

For addresses on the production network, the *Human-Readable Part* is **"zs"**. For addresses on the test network, the *Human-Readable Part* is **"ztestsapling"**.

5.6.5 Sprout Incoming Viewing Keys

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{sk_{enc}}$ is a $\mathsf{KA^{Sprout}.Private}$ key (see §5.4.4.1 ***'Sprout Key Agreement'*** on p. 44), for use with the encryption scheme defined in §4.12 ***'In-band secret distribution'*** on p. 34. These components are derived from a *spending key* as described in §4.2.1 ***'Sprout Key Components'*** on p. 24.

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 [Bern2006].

$\mathsf{sk_{enc}}$ **MUST** be "clamped" using $\mathsf{KA^{Sprout}.FormatPrivate}$ as specified in §4.2.1 ***'Sprout Key Components'*** on p. 24. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{sk_{enc}} \neq \mathsf{KA^{Sprout}.FormatPrivate}(\mathsf{sk_{enc}})$.

$\mathsf{KA^{Sprout}.FormatPrivate}$ is defined in §5.4.4.1 ***'Sprout Key Agreement'*** on p. 44.

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`".

5.6.6 Sapling Incoming Viewing Keys

A **Sapling** *incoming viewing key* consists of $\mathsf{ivk} \,⦂\, \mathsf{KA^{Sprout}.Private}$ (see §5.4.4.3 ***'Sapling Key Agreement'*** on p. 45).

ivk is a $\mathsf{KA^{Sprout}.Private}$ key for use with the encryption scheme defined in §4.12 ***'In-band secret distribution'*** on p. 34. It is derived as described in §4.2.2 ***'Sapling Key Components'*** on p. 24.

The raw encoding of an *incoming viewing key* consists of:

256-bit ivk

- 32 bytes (little-endian) specifying ivk.

ivk **MUST** be in the range $\{0 \,..\, 2^{\ell_{\mathsf{ivk}}} - 1\}$ as specified in §4.2.2 ***'Sapling Key Components'*** on p. 24. That is, a decoded *incoming viewing key* **MUST** be considered invalid if ivk is not in this range.

For *incoming viewing keys* on the production network, the *Human-Readable Part* is "**`zivks`**". For *incoming viewing keys* on the test network, the *Human-Readable Part* is "**`zivktestsapling`**".

5.6.7 Sapling Full Viewing Keys

A **Sapling** *full viewing key* consists of $\mathsf{ak} \,⦂\, \mathbb{J}$ and $\mathsf{nk} \,⦂\, \mathbb{J}$.

ak and nk are points on the *Jubjub curve* (see §5.4.8.3 ***'Jubjub'*** on p. 50). They are derived as described in §4.2.2 ***'Sapling Key Components'*** on p. 24.

The raw encoding of a *full viewing key* consists of:

$\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})\big)$	$\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big)$

- 32 bytes specifying the compressed Edwards encoding of ak (see §5.4.8.3 ***'Jubjub'*** on p. 50).
- 32 bytes specifying the compressed Edwards encoding of nk.

When decoding this representation, the key is not valid if $\mathsf{abst}_{\mathbb{J}}$ returns $\bot$ for either point.

For *incoming viewing keys* on the production network, the *Human-Readable Part* is **"zviews"**. For *incoming viewing keys* on the test network, the *Human-Readable Part* is **"zviewtestsapling"**.

5.6.8 Sprout Spending Keys

A **Sprout** *spending key* consists of $\mathsf{a_{sk}}$, which is a sequence of 252 bits (see §4.2.1 ***'Sprout** Key Components'* on p. 24).

The raw encoding of a **Sprout** *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"**.

5.6.9 Sapling Spending Keys

A **Sapling** *spending key* consists of $\mathsf{sk} ⦂ \mathbb{B}^{[\ell_{\mathsf{sk}}]}$ (see §4.2.1 ***'Sprout** Key Components'* on p. 24).

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

$\mathsf{LEBS2OSP}_{256}(\mathsf{sk})$

- 32 bytes specifying sk.

For *spending keys* on the production network, the *Human-Readable Part* is **"secret-spending-key-main"**. For *spending keys* on the test network, the *Human-Readable Part* is **"secret-spending-key-test"**.

5.7 Sprout zk-SNARK Parameters

For the **Zcash** production *block chain* and testnet, the SHA-256 hashes of the *proving key* and *verifying key* for the **Sprout** *JoinSplit statement*, encoded in *libsnark* format, are:

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

These parameters were obtained by a multi-party computation described in [GitHub-mpc] and [BGG2016].

5.8 Sapling zk-SNARK Parameters

The SHA-256 hashes of the *proving key* and *verifying key* for the **Sapling** *Spend statement*, encoded in *bellman* format, are:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx sapling-spend-proving.key
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx sapling-spend-verifying.key
```

The SHA-256 hashes of the *proving key* and *verifying key* for the **Sapling** *Output statement*, encoded in *bellman* format, are:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx sapling-output-proving.key
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx sapling-output-verifying.key
```

These parameters were obtained by a multi-party computation described in TODO: .

6 Network Upgrades

Zcash launched with a protocol revision that we call **Sprout**. At the time of writing, two upgrades are planned: **Overwinter**, and **Sapling**. This section summarizes the planned strategy for upgrading from **Sprout** to **Overwinter** and then **Sapling**.

The upgrade mechanism is described in [ZIP-200]. The specifications of the **Overwinter** upgrade are described in [ZIP-201], [ZIP-202], [ZIP-203], and [ZIP-143].

Overwinter and **Sapling** will each be introduced as a *"bilateral consensus rule change"*. In this kind of upgrade,

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

Full support for each upgrade is indicated by a minimum version of the peer-to-peer protocol. At the planned upgrade *block height*, nodes that support a given upgrade will disconnect from (and will not reconnect to) nodes with a protocol version lower than this minimum. See [ZIP-201] for how this applies to the **Overwinter** upgrade.

This ensures that upgrade-supporting nodes transition cleanly from the old protocol to the new protocol. Nodes that do not support the upgrade will find themselves on a network that uses the old protocol and is fully partitioned from the upgrade-supporting network.

This allows us to specify arbitrary protocol changes that take effect at a given *block height*. Note, however, that a *block chain* reorganization across the upgrade *block height* is possible. In the case of such a reorganization, *blocks* at a height before the upgrade *block height* will still be created and validated according to the pre-upgrade rules, and upgrade-supporting nodes **MUST** allow for this.

7 Consensus Changes from Bitcoin

7.1 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 number*.
≥ 3	4	`nVersionGroupId`	`uint32`	Version group ID (nonzero).
≥ 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**.
≥ 3	4	`nExpiryHeight`	`uint32`	A *block height* in the range $\{1\,..\,499999999\}$ after which the *transaction* will expire, or 0 to disable expiry ([ZIP-203]).
≥ 4	*Varies*	`nShieldedSpend`	`compactSize uint`	The number of *Spend descriptions* in `vShieldedSpend`.
≥ 4	384· `nShieldedSpend`	`vShieldedSpend`	`SpendDescription [nShieldedSpend]`	A sequence of *Spend descriptions*, each encoded as in §7.3 ***'Encoding of Spend Descriptions'*** on p. 62.
≥ 4	*Varies*	`nShieldedOutput`	`compactSize uint`	The number of *Output descriptions* in `vShieldedOutput`.
≥ 4	580· `nShieldedOutput`	`vShieldedOutput`	`OutputDescription [nShieldedOutput]`	A sequence of *Output descriptions*, each encoded as in §7.4 ***'Encoding of Output Descriptions'*** on p. 62.
≥ 2	*Varies*	`nJoinSplit`	`compactSize uint`	The number of *JoinSplit descriptions* in `vJoinSplit`.
≥ 2	1802· `nJoinSplit`	`vJoinSplit`	`JoinSplitDescription [nJoinSplit]`	A sequence of *JoinSplit descriptions*, each encoded as in §7.2 ***'Encoding of JoinSplit Descriptions'*** on p. 61.
≥ 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 §4.8 ***'Non-malleability'*** on p. 30.

Consensus rules:

- The *transaction version number* **MUST** be greater than or equal to 1.
- [**Sprout** only] The `fOverwintered` flag **MUST NOT** be set.
- [**Overwinter** onward] The `fOverwintered` flag **MUST** be set.
- [**Overwinter** onward] The *version group ID* **MUST** be recognized.
- [**Overwinter** only, pre-**Sapling**] The *transaction version number* **MUST** be 3, and the *version group ID* **MUST** be `0x03C48270`.
- [**Sapling** onward] The *transaction version number* and *version group ID* **MUST** be either (3, `0x03C48270`) or (4, TODO : **Sapling** *version group ID*).
- [**Sprout** only] If `version` $= 1$ or `nJoinSplit` $= 0$, then `tx_in_count` **MUST NOT** be 0.
- [**Sapling** onward] At least one of `tx_in_count`, `nShieldedSpend`, and `nJoinSplit` **MUST** be nonzero.
- A *transaction* with one or more inputs from *coinbase transactions* **MUST** have no *transparent* outputs (i.e. `tx_out_count` **MUST** be 0).
- If `nJoinSplit` > 0, then `joinSplitSig` **MUST** represent a valid signature over dataToBeSigned as defined in §4.8 ***'Non-malleability'*** on p. 30.
- If `nJoinSplit` > 0, then `joinSplitPubKey` **MUST** represent a valid Ed25519 public key encoding as specified in §5.4.5 ***'JoinSplit Signature'*** on p. 45.
- [**Sprout** only] The encoded size of the *transaction* **MUST** be less than or equal to 100000 bytes.
- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions*, *Spend descriptions*, or *Output 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.
- [**Overwinter** onward] `nExpiryHeight` **MUST** be less than or equal to 499999999.
- [**Overwinter** onward] If a *transaction* is not a *coinbase transaction* and its `nExpiryHeight` field is nonzero, then it **MUST NOT** be mined at a *block height* greater than its `nExpiryHeight`.
- TODO: Other rules inherited from **Bitcoin**.

In addition, consensus rules associated with each *JoinSplit description* (§7.2 ***'Encoding of JoinSplit Descriptions'*** on p. 61), each *Spend description* (§7.3 ***'Encoding of Spend Descriptions'*** on p. 62), and each *Output description* (§7.4 ***'Encoding of Output Descriptions'*** on p. 62) **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.
- The semantics of *transactions* with *transaction version number* not equal to 1, 2, 3, or 4 is not currently defined. Miners **MUST NOT** create *blocks* before the **Overwinter** *activation block height* containing *transactions* with version other than 1 or 2.
- The exclusion of *transactions* with *transaction version number* ***greater than*** 2 is not a consensus rule before **Overwinter** activation. Such *transactions* may exist in the *block chain* and **MUST** be treated identically to version 2 *transactions*.
- [**Overwinter** onward] Once **Overwinter** has activated, limits on the maximum *transaction version number* are consensus rules.

- 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.
- TODO: Describe interpretation of `fOverwintered` and `version`.
- 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, or the related BIPs 9, 112 and 113.

The changes relative to **Bitcoin** version 1 *transactions* as described in [Bitc-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.

[**Sprout** only] Software that creates *transactions* **SHOULD** use version 1 for *transactions* with no *JoinSplit descriptions*.

7.2 Encoding of JoinSplit Descriptions

An abstract *JoinSplit description*, as described in §3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 13, 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 **Sprout** *note commitment tree* at some *block height* in the past, or the *root* produced by a previous *JoinSplit transfer* in this *transaction*.
64	`nullifiers`	`char[32]` $[\mathrm{N}^{\mathsf{old}}]$	A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$.
64	`commitments`	`char[32]` $[\mathrm{N}^{\mathsf{new}}]$	A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$.
32	`ephemeralKey`	`char[32]`	A Curve25519 public key epk.
32	`randomSeed`	`char[32]`	A 256-bit seed that must be chosen independently at random for each *JoinSplit description*.
64	`vmacs`	`char[32]` $[\mathrm{N}^{\mathsf{old}}]$	A sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the *JoinSplit description*.
296	`zkproof`	`char[296]`	An encoding of the *zero-knowledge proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see §5.4.9.1 ***'PHGR13'*** on p. 52).
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 vmacs field encodes $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ which are computed as described in §4.8 *'Non-malleability'* on p. 30.

The ephemeralKey and encCiphertexts fields together form the *transmitted notes ciphertext*, which is computed as described in §4.12 *'In-band secret distribution'* on p. 34.

Consensus rules applying to a *JoinSplit description* are given in §4.3 *'JoinSplit Descriptions'* on p. 25.

7.3 Encoding of Spend Descriptions

Let LEBS2OSP be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

An abstract *Spend description*, as described in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 14, is encoded in a *transaction* as an instance of a `SpendDescription` type as follows:

Bytes	Name	Data Type	Description
32	`cv`	`char[32]`	A *value commitment* to the value of the input *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{cv})$.
32	`anchor`	`char[32]`	A *root* of the **Sapling** *note commitment tree* at some *block height* in the past, $\mathsf{LEBS2OSP}_{256}(\mathsf{rt})$.
32	`nullifier`	`char[32]`	The *nullifier* of the input *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{nf})$.
32	`rk`	`char[32]`	The randomized public key for `spendAuthSig`, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{rk})\big)$.
192	`zkproof`	`char[192]`	An encoding of the *zero-knowledge proof* π_{ZKSpend} (see §5.4.9.2 *'Groth16'* on p. 52).
64	`spendAuthSig`	`char[64]`	A signature authorizing this spend.

Consensus rules applying to a *Spend description* are given in §4.4 *'Spend Descriptions'* on p. 26.

7.4 Encoding of Output Descriptions

Let LEBS2OSP be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

An abstract *Output description*, described in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 14, is encoded in a *transaction* as an instance of an `OutputDescription` type as follows:

Bytes	Name	Data Type	Description
32	`cv`	`char[32]`	A *value commitment* to the value of the output *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{cv})$.
32	`cm`	`char[32]`	The *note commitment* for the output *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{cm})$.
32	`ephemeralKey`	`char[32]`	An encoding of a Jubjub public key epk (see §5.4.4.3 *'**Sapling** Key Agreement'* on p. 45).
580	`encCiphertext`	`char[580]`	A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{enc}}$.
192	`zkproof`	`char[192]`	An encoding of the *zero-knowledge proof* π_{ZKOutput} (see §5.4.9.2 *'Groth16'* on p. 52).

The ephemeralKey and encCiphertext fields together form the *transmitted note ciphertext*, which is computed as described in §4.12 *'In-band secret distribution'* on p. 34.

Consensus rules applying to an *Output description* are given in §4.5 *'Output Descriptions'* on p. 27.

7.5 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` / `hashFinalSaplingRoot`	`char[32]`	[**Sprout** only] A reserved field which should be ignored. [**Sapling** onward] A *root* (TODO: specify bit sequence to byte sequence conversion) of the **Sapling** *note commitment tree* corresponding to the final **Sapling** *treestate* of this *block*.
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**. [Bitc-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 §7.6.3 *'Difficulty adjustment'* on p. 66, and let PoWMedianBlockSpan be the constant defined in §5.3 *'Constants'* on p. 36.

Consensus rules:

- The *block version number* **MUST** be greater than or equal to 4.
- For a *block* at *block height* height, `nBits` **MUST** be equal to ThresholdBits(height).
- The *block* **MUST** pass the difficulty filter defined in §7.6.2 *'Difficulty filter'* on p. 66.
- `solution` **MUST** represent a valid Equihash solution as defined in §7.6.1 *'Equihash'* on p. 65.

- `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.
- [**Sapling** onward] `hashFinalSaplingRoot` **MUST** be the *root* of the **Sapling** *note commitment tree* for the final **Sapling** *treestate* of this *block*.
- 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 PoWMedianBlockSpan *blocks*. The Bitcoin Developer Reference [Bitc-Block] was previously in error on this point, but has now been corrected.
- There are no changes to the *block version number* or format for **Overwinter**.
- Although the *block version number* does not change for **Sapling**, the previously reserved (and ignored) field `hashReserved` has been repurposed for `hashFinalSaplingRoot`. There are no other format changes.

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

- *Block versions* less than 4 are not supported.
- The `hashReserved` (or `hashFinalSaplingRoot`), `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.

7.6 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 §7.6.1 *'Equihash'* on p. 65.
- The *block header* satisfies the difficulty check according to §7.6.2 *'Difficulty filter'* on p. 66.

7.6.1 Equihash

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

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

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

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

Let powheader :=

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

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

EquihashGen is instantiated in §5.4.1.9 *'Equihash Generator'* on p. 42.

Define $\mathsf{I2BEBSP} : (u : \mathbb{N}) \times \{0\,..\,2^u - 1\} \to \mathbb{B}^{[u]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.

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

$\mathsf{I2BEBSP}_{21}(68)$	$\mathsf{I2BEBSP}_{21}(41)$	$\mathsf{I2BEBSP}_{21}(2^{21}-1)$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

8-bit 0	8-bit 2	8-bit 32	8-bit 0	8-bit 10	8-bit 127	8-bit 255	$\cdots$

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 using little-endian ordering of bits in the solution encoding would require bit-reversal (as opposed to only shifting).

7.6.2 Difficulty filter

Let ToTarget be as defined in §7.6.4 ***'nBits conversion'*** on p. 67.

Difficulty is defined in terms of a *target threshold*, which is adjusted for each *block* according to the algorithm defined in §7.6.3 ***'Difficulty adjustment'*** on p. 66.

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

7.6.3 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. 36.

Let ToCompact and ToTarget be as defined in §7.6.4 ***'nBits conversion'*** on p. 67.

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

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

Block header fields are specified in §7.5 ***'Block Header'*** on p. 63.

Define:

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

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

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

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

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

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

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

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

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

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

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

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

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*.

7.6.4 nBits conversion

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

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

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

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

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

7.6.5 Definition of Work

As explained in §3.3 ***'The Block Chain'*** on p. 12, a node chooses the "best" *block chain* visible to it by finding the chain of valid *blocks* with the greatest total work.

Let ToTarget be as defined in §7.6.4 ***'nBits conversion'*** on p. 67.

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

7.7 Calculation of Block Subsidy and Founders' Reward

§3.9 ***'Block Subsidy and Founders' Reward'*** on p. 16 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. 36.

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

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

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

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

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

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

7.8 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. [ZcashIssue-2113]

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 [Bitc-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.

7.9 Changes to the Script System

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

7.10 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. 54.

8 Differences from the Zerocash paper

8.1 Transaction Structure

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

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

This allows for the possibility of chaining transfers of *shielded* value in a single **Zcash** *transaction*, e.g. to spend a *shielded note* that has just been created. (In **Zcash**, we refer to value stored in UTXOs as *transparent*, and value stored in *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.

8.2 Memo Fields

Zcash adds a *memo field* sent from the creator of a *JoinSplit description* to the recipient of each output *note*. This feature is described in more detail in §5.5 ***'Encodings of Note Plaintexts and Memo Fields'*** on p. 53.

8.3 Unification of Mints and Pours

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

- a "Mint" transaction takes value from *transparent* UTXOs as input and produces a new *shielded note* as output.
- a "Pour" transaction takes up to $\mathrm{N}^{\mathsf{old}}$ *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.

[**Sprout** only] In **Zcash**, the sequence of operations added to a *transaction* (see §8.1 ***‘Transaction Structure’*** on p. 70) 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.

[**Sapling** onward] In **Sapling**, there are still no “Mint” transactions. Instead of *JoinSplit transfers*, there are *Spend transfers* and *Output transfers*. These make use of *Pedersen value commitments* to represent the shielded values that are transferred. Because these commitments are additively homomorphic (using elliptic curve addition), it is possible to check that all *Spend transfers* and *Output transfers* balance; see §4.9 ***‘Balance’*** on p. 30 for detail. This reduces the granularity of the circuit, allowing a substantial performance improvement (orthogonal to other **Sapling** circuit improvements) when the numbers of *shielded* inputs and outputs are significantly different. This comes at the cost of revealing the exact number of *shielded* inputs and outputs, but dummy (zero-valued) outputs are still possible.

8.4 Faerie Gold attack and fix

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

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by Receive (as defined in [BCG+2014, Figure 2]), but only one of which can be spent.

We call this a “Faerie Gold” attack — referring to various Celtic legends in which faeries pay mortals in what appears to be gold, but which soon after reveals itself to be leaves, gorse blossoms, gingerbread cakes, or other less valuable things [LG2004].

This attack does not violate the security definitions given in [BCG+2014]. The issue could be framed as a problem either with the definition of Completeness, or the definition of Balance:

- The Completeness property asserts that a validly received *note* can be spent provided that its *nullifier* does not appear on the ledger. This does not take into account the possibility that distinct *notes*, which are validly received, could have the same *nullifier*. That is, the security definition depends on a protocol detail –*nullifiers*– that is not part of the intended abstract security property, and that could be implemented incorrectly.
- The Balance property only asserts that an adversary cannot obtain ***more*** funds than they have minted or received via payments. It does not prevent an adversary from causing others’ funds to decrease. In a Faerie Gold attack, an adversary can cause spending of a *note* to reduce (to zero) the effective value of another *note* for which the attacker does not know the *spending key*, which violates an intuitive conception of global balance.

These problems with the security definitions need to be repaired, but doing so is outside the scope of this specification. Here we only describe how **Zcash** addresses the immediate attack.

It would be possible to address the attack by requiring that a recipient remember all of the ρ values for all *notes* they have ever received, and reject duplicates (as proposed in [GGM2016]). However, this requirement would interfere with the intended **Zcash** feature that a holder of a *spending key* can recover access to (and be sure that they are able to spend) all of their funds, even if they have forgotten everything but the *spending key*.

[**Sprout**] 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 §4.6.2 ***'Dummy Notes (Sprout)'*** on p. 28). 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. 25. ($\mathsf{h_{Sig}}$ 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.)

[**Sprout**] 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 §4.11.1 ***'Uniqueness of*** ρ^{new}_i***'*** on p. 32 in the *JoinSplit statement*.

[**Sprout**] 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. 17.

[**Sprout**] Crucially, "*nullifier* integrity" (§4.11.1 ***'Nullifier integrity'*** on p. 32) is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note*. 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.

[**Sprout**] *Nullifier* integrity also prevents a "roadblock attack" in which the attacker 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 attacker'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.

[**Sapling** onward] In **Sapling**, uniqueness of ρ is ensured by making it dependent on the position of the *note commitment* in the **Sapling** *note commitment tree*. Specifically, $\rho = \mathsf{cm} + [\mathsf{pos}]\,\mathcal{J}$, where $\mathcal{J}$ is a generator independent of the generators used in $\mathsf{NoteCommit}^{\mathsf{Sapling}}$. Therefore, ρ commits uniquely to the *note* and its position, and this commitment is collision-resistant by the same argument used to prove collision resistance of *Pedersen hashes*. Note that it is possible for two distinct **Sapling** *positioned notes* (having different ρ values and *nullifiers*, but different *note positions*) to have the same *note commitment*, but this causes no security problem. Roadblock attacks are not possible because a given *note position* does not repeat for outputs of different *transactions* in the same *block chain*.

8.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM_{rcm}}$ and $\mathsf{COMM_s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, v, and ρ. However, the instantiation of $\mathsf{COMM_{rcm}}$ 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 hash evaluation for the commitment: SHA-256 for **Sprout**, and $\mathsf{PedersenHash}$ for **Sapling**. The motivation for the nested construction in **Zerocash** was to allow Mint transactions to be publically verified without requiring a *zero-knowledge proof* (as described under step 3 in [BCG+2014, section 1.3]). Since **Zcash** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers* (for **Sprout**), or *Spend transfers* and *Output transfers* (for **Sapling**), and each transfer always uses a *zero-knowledge proof*, **Zcash** does not require the nesting. A side benefit is that this reduces the cost of computing the *note commitments*: for **Sprout** it reduces the number of $\mathsf{SHA256Compress}$ evaluations needed to compute each *note commitment* from three to two, saving a total of four $\mathsf{SHA256Compress}$ evaluations in the *JoinSplit statement*.

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

[**Sapling** onward] In **Sapling**, *Pedersen commitments* are used instead of $\mathsf{SHA256Compress}$. These commitments are statistically hiding, and so "everlasting anonymity" is supported for **Sapling** notes under the same conditions as in **Zerocash** (by the protocol, not necessarily by zcashd).

8.6 Changes to PRF inputs and truncation

The format of inputs to the PRFs instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 43 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 [BCG+2014, Appendix D].

In more detail:

- In the argument relating $\mathbf{H}$ and ∂_2, it is stated that in ∂_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 $\mathsf{COMM_{rcm}}$, which further complicates the analysis.

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

[**Sprout**] For resistance to Faerie Gold attacks as described in §8.4 *'Faerie Gold attack and fix'* on p. 71, **Zcash** depends on collision resistance of $\mathsf{hSigCRH}$ (instantiated using $\mathsf{BLAKE2b\text{-}256}$) and PRF^{ρ} (instantiated using $\mathsf{SHA256Compress}$). 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.

[**Sprout**] Since the PRFs are instantiated using $\mathsf{SHA256Compress}$ 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 $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nf}}$, and $\mathsf{PRF}^{\mathsf{pk}}$, and to φ (which does not exist in **Zerocash**) for PRF^{ρ}, and so those values have been reduced to 252 bits. This is preferable to requiring reasoning about truncation, and 252 bits is quite sufficient for security of these cryptovalues.

Sapling uses *Pedersen hashes* and $\mathsf{BLAKE2s}$ where **Sprout** used $\mathsf{SHA256Compress}$. *Pedersen hashes* can be efficiently instantiated for arbitrary input lengths. $\mathsf{BLAKE2s}$ has an input block size of 512 bits, and uses a finalization flag rather than padding of the last input block; it also supports domain separation via a personalization parameter distinct from the input. Therefore, there is no need for truncation in the inputs to any of these hashes. TODO: check, especially $\mathsf{CRH}^{\mathsf{ivk}}$ which has truncated output.

8.7 In-band secret distribution

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

The motivations for this change were as follows:

- The **Zerocash** paper did not specify the curve to be used. We believe that Curve25519 has significant side-channel resistance, performance, implementation complexity, and robustness advantages over most other available curve choices, as explained in [Bern2006]. For **Sapling**, the *Jubjub curve* was designed according to a similar design process following the "Safe curves" criteria [BL-SafeCurves] [GitHub-jubjub]. This retains Curve25519's advantages while keeping *shielded payment address* sizes short, because the same public key material supports both encryption and spend authentication.
- 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 (and Jubjub) key agreement is defined in a way that avoids these concerns due to the curve structure and the "clamping" of private keys (or explicit cofactor multiplication and point validation for **Sapling**).
- 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.
- [**Sprout**] 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 $\mathsf{a^{old}_{sk,1..N^{old}}}$, so an adversary cannot modify it in a ciphertext from someone else's transaction for use in a chosen-ciphertext attack without detection. In **Sapling**, there is no equivalent to $\mathsf{h_{Sig}}$. TODO: Explain why this is ok.
- [**Sprout**] The scheme used by **Sprout** includes an optimization that reuses the same ephemeral key (with different nonces) for the two ciphertexts encrypted in each *JoinSplit description*.

The security proofs of [ABR1999] can be adapted straightforwardly to the resulting scheme. Although DHAES as defined in that paper does not pass the recipient public key or a public seed to the *hash function* H, this does not impair the proof because we can consider H to be the specialization of our KDF to a given recipient key and seed. (Passing the recipient public key to the KDF could in principle compromise key privacy, but not confidentiality of encryption.) [**Sprout**] 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 [Bern2005] in the multi-user setting [Zave2012]. 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: for **Sprout**, $\mathsf{a_{sk}}$ is 252 bits, and $\mathsf{sk_{enc}}$ is no shorter than $\mathsf{a_{sk}}$.

8.8 Omission in Zerocash security proof

The abstract **Zerocash** protocol requires $\mathsf{PRF^{addr}}$ 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 [BCG+2014, Appendix D.3]. For the "$\mathcal{A}$ violates Condition I" case, the proof says:

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

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

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 **Sprout**, which ***are*** collision-resistant assuming that $\mathsf{SHA256Compress}$ 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 $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}$, together with constraint 1(b) of the *JoinSplit statement* (see §4.11.1 ***'Spend authority'*** on p. 32), implies distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},2}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

8.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a_{pk}}, \mathsf{pk_{enc}}), \mathsf{v}, \rho, \mathsf{rcm}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines a **Sprout** *note* as $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$. 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 $\mathsf{NoteCommit^{Sprout}}$ in **Sprout**. $\mathsf{pk_{enc}}$ is also not needed as part of a *note*: it is not an input to $\mathsf{NoteCommit^{Sprout}}$ nor is it constrained by the **Zerocash** `POUR` *statement* or the **Zcash** *JoinSplit statement*. cm can be computed from the other fields. (The definition of *notes* for **Sapling** is different again.)
- The length of proof encodings given in the paper is 288 bytes. [**Sprout**] This differs from the 296 bytes specified in §5.4.9.1 ***'PHGR13'*** on p. 52, 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 $\{0 \,..\, \mathsf{MAX\_MONEY}\}$; in **Zerocash** it is $\{0 \,..\, 2^{64} - 1\}$. (The *JoinSplit statement* still only directly enforces that the sum of amounts in a given *JoinSplit transfer* is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

9 Acknowledgements

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

The authors would like to thank everyone with whom they have discussed the **Zerocash** protocol design; in addition to the inventors, this includes Mike Perry, Isis Lovecruft, Leif Ryge, Andrew Miller, Zooko Wilcox, Samantha Hulsey, Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, 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, and no doubt others.

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

The Faerie Gold attack was found by Zooko Wilcox; subsequent analysis of variations on the attack was performed by Daira Hopwood and Sean Bowe. The internal hash collision attack was found by Taylor Hornby. The error in the **Zerocash** proof of Balance relating to collision-resistance of $\mathsf{PRF}^{\mathsf{addr}}$ was found by Daira Hopwood. The errors in the proof of Ledger Indistinguishability mentioned in §8.6 ***'Changes to PRF inputs and truncation'*** on p. 73 were also found by Daira Hopwood.

The design of **Sapling** is primarily due to Matthew Green, Ian Miers, Daira Hopwood, Sean Bowe, and Jack Grigg.

10 Change History

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.
- Updates to transaction format and consensus rules for Overwinter and Sapling.
- Add specification of the *Output statement*.
- Change $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ from 29 to 32.
- Updates to **Sapling** construction, changing how the *nullifier* is computed and separating it from the *randomized spend verifying key* (rk).
- Clarify conversions between bit and byte sequences for sk, $\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})$, and $\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$.
- 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**.
- Simplify $\mathsf{FindGroupHash}^{\mathbb{J}}$ to use a single-byte index.
- Changes to diversification for *Pedersen hashes* and *Pedersen commitments*.
- Improve security definitions for signatures.

2018.0-beta-13

- Only cosmetic changes to **Sprout**.
- Change how (ask, nsk) are derived from the *spending key* sk to ensure they are on the full range of $\mathbb{F}_{r_{\mathbb{J}}}$.
- Change $\mathsf{PRF}^{\mathsf{nr}}$ to produce output computationally indistinguishable from uniform on $\mathbb{F}_{r_{\mathbb{J}}}$.
- Change $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ to be a u-coordinate for which there is no point on the curve.
- Appendix A updates:
  - categorize components into larger sections
  - fill in the [de]compression and validation algorithm
  - more precisely state the assumptions for inputs and outputs
  - delete not-all-one component which is no longer needed
  - factor out xor into its own component
  - specify [un]packing more precisely; separate it from boolean constraints
  - optimize checking for non-small order
  - notation in variable-base multiplication algorithm.

2018.0-beta-12

- No changes to **Sprout**.
- Add references to **Overwinter** ZIPs and update the section on **Overwinter**/**Sapling** transitions.
- Add a section on re-randomizable signatures.
- Add definition of $\mathsf{PRF}^{\mathsf{nr}}$.
- Work-in-progress on **Sapling** statements.
- Rename *"raw"* to *"homomorphic" Pedersen commitments*.
- Add packing modulo the field size and range checks to Appendix A.
- Update the algorithm for variable-base scalar multiplication to what is implemented by sapling-crypto.

2018.0-beta-11

- No changes to **Sprout**.
- Add sections on *Spend descriptions* and *Output descriptions*.
- Swap order of cv and rt in a *Spend description* for consistency.
- Fix off-by-one error in the range of ivk.

2018.0-beta-10

- Split the descriptions of SHA-256 and SHA256Compress, and of BLAKE2, into their own sections. Specify SHA256Compress more precisely.
- Add Tracy Hu to acknowledgements (for the idea of explicitly encoding the root of the **Sapling** *note commitment tree* in *block headers*).
- Move bit/byte/integer conversion primitives into §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 36.
- Refer to **Overwinter** and **Sapling** just as "upgrades" in the abstract, not as the next "minor version" and "major version".

- $\mathsf{PRF}^{\mathsf{nr}}$ must be collision-resistant.
- Correct an error in the *Pedersen hash* specification.
- Use a named variable, c, for chunks per segment in the *Pedersen hash* specification, and change its value from 61 to 63. Add a proof justifying this value of c.
- Specify *Pedersen commitments*.
- Notation changes.
- Generalize the *distinct-x criterion* (Theorem A.3.3 on p. 95) to allow negative indices.

2018.0-beta-9

- Specify the coinbase maturity rule, and the rule that *coinbase transactions* cannot contain *JoinSplit descriptions*, *Spend descriptions*, or *Output descriptions*.
- Delay lifting the 100000-byte *transaction* size limit from **Overwinter** to **Sapling**.
- Improve presentation of the proof of injectivity for $\mathsf{Extract}_{\mathbb{J}}$.
- Specify $\mathsf{GroupHash}^{\mathbb{J}}$.
- Specify *Pedersen hashes*.

2018.0-beta-8

- No changes to **Sprout**.
- Add instantiation of $\mathsf{CRH}^{\mathsf{ivk}}$.
- Add instantiation of a hash extractor for Jubjub.
- Make the background lighter and the **Sapling** green darker, for contrast.

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* and *Pseudo Random Generators*.
- Rename clamp to bound and $\mathsf{ActualTimespanClamped}$ to $\mathsf{ActualTimespanBounded}$ 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.
- Add an explanation of how **Sapling** prevents Faerie Gold and roadblock attacks.
- **Sapling** work in progress.

2018.0-beta-6

- No changes to **Sprout**.
- **Sapling** work in progress, mainly on Appendix A *'Circuit Design'* on p. 90.

2018.0-beta-5

- Specify more precisely the requirements on Ed25519 public keys and signatures.
- **Sapling** work in progress.

2018.0-beta-4

- No changes to **Sprout**.
- Update key components diagram for **Sapling**.

2018.0-beta-3

- Explain how the chosen fix to Faerie Gold avoids a potential "roadblock" attack.
- Update some explanations of changes from **Zerocash** for **Sapling**.
- Add a description of the *Jubjub curve*.
- Add an acknowledgement to George Tankersley.
- Add an appendix on the design of the **Sapling** circuits at the *quadratic arithmetic program* level.

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.
- Refer to Network Upgrade 0 as **Overwinter**.

2017.0-beta-2.8

- Correct the non-normative note describing how to check the order of π_B.
- Initial version of draft **Sapling** protocol specification.

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 §8.8 *'Omission in **Zerocash** security proof'* on p. 75.
- Correct the order of arguments to COMM in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 46.
- 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 [ZcashIssue-2113].

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 §5.4.7.1 *'**Sprout** Note Commitments'* on p. 46 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 §4.2.1 ‘***Sprout** Key Components*’ on p. 24.

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 BLAKE2b-ℓ is different from truncated BLAKE2b-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 §8.7 ***'In-band secret distribution'*** on p. 74.
- 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 GeneralCRH in favour of specifying hSigCRH and EquihashGen directly in terms of BLAKE2b-ℓ.
- Correct the security requirement for EquihashGen.

2016.0-beta-1.1

- 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 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 §8.8 ***'Omission in Zerocash security proof'*** on p. 75).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in §4.11.1 ***'Merkle path validity'*** on p. 32 about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

2.0-alpha-1

- First version intended for public review.

11 References

[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) (↑ p18, 74).

[AGRRT2017] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive: Report 2016/492. Received May 21, 2016. January 5, 2017. URL: `https://eprint.iacr.org/2016/492` (visited on 2018-01-12) (↑ p102).

[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) (↑ p38, 101).

[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. (↑ p19, 74).

[BBJLP2008] Daniel Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. *Twisted Edwards Curves*. Cryptology ePrint Archive: Report 2008/013. Received January 8, 2008. March 13, 2008. URL: `https://eprint.iacr.org/2008/013` (visited on 2018-01-12) (↑ p96).

[BCG+2014] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version)*. URL: `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf` (visited on 2016-08-06). A condensed version appeared in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland) 2014*, pages 459–474; IEEE, 2014. (↑ p6, 7, 8, 17, 30, 32, 35, 71, 72, 73, 75).

[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. (↑ p52).

[BCP1988] Jurgen Bos, David Chaum, and George Purdy. “A Voting Scheme”. Unpublished. Presented at the rump session of CRYPTO ’88 (Santa Barbara, California, USA, August 21–25, 1988); does not appear in the proceedings. (↑ p40).

[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) (↑ p24).

[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) (↑ p52).

[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) (↑ p19).

[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. (↑ p17).

[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. (↑ p45).

[Bern2005] 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. (↑ p74).

[Bern2006] Daniel Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *Public Key Cryptography - PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography (New York, NY, USA, April 24–26, 2006)*. Springer-Verlag, February 9, 2006. URL: `http://cr.yp.to/papers.html#curve25519` (visited on 2016-08-14). Document ID: 4230efdfa673480fc079449d90f322c0. (↑ p18, 44, 55, 56, 74).

[BGG1995] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. “Incremental Cryptography: The Case of Hashing and Signing”. In: *Advances in Cryptology - CRYPTO ’94. Proceedings of the 14th Annual International Cryptology Conference (Santa Barbara, California, USA, August 21–25, 1994)*. Ed. by Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer, October 20, 1995, pages 216–233. ISBN: 978-3-540-48658-9. DOI: `10.1007/3-540-48658-5_22`. URL: `https://cseweb.ucsd.edu/~mihir/papers/inc1.pdf` (visited on 2018-02-09) (↑ p40, 41, 99).

[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) (↑ p58, 80).

[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) (↑ p70).

[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) (↑ p54, 70).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p55).

[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) (↑ p70, 81).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p20).

[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) (↑ p70).

[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) (↑ p70).

[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) (↑ p61).

[BIP-173] Pieter Wuille and Greg Maxwell. *Base32 address format for native v0-16 witness outputs*. Bitcoin Improvement Proposal 173. Last revised September 24, 2017. URL: `https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki` (visited on 2018-01-22) (↑ p54).

[Bitc-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: `https://en.bitcoin.it/wiki/Base58Check_encoding` (visited on 2016-01-26) (↑ p54, 55).

[Bitc-Block] *Block Headers — Bitcoin Developer Reference*. URL: `https://bitcoin.org/en/developer-reference#block-headers` (visited on 2017-04-25) (↑ p64).

[Bitc-CoinJoin] *CoinJoin — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/CoinJoin (visited on 2016-08-17) (↑ p8).

[Bitc-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#raw-transaction-format (visited on 2016-03-15) (↑ p61).

[Bitc-Multisig] *P2SH multisig (definition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2sh-multisig (visited on 2016-08-19) (↑ p69).

[Bitc-nBits] *Target nBits — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#target-nbits (visited on 2016-08-13) (↑ p63, 67).

[Bitc-P2PKH] *P2PKH (definition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2pkh (visited on 2016-08-24) (↑ p54).

[Bitc-P2SH] *P2SH (definition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2sh (visited on 2016-08-24) (↑ p54).

[Bitc-Protocol] *Protocol documentation — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 2016-10-02) (↑ p7).

[BJLSY2015] Daniel Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *EdDSA for more curves*. Technical Report. July 4, 2015. URL: https://cr.yp.to/papers.html#eddsa (visited on 2018-01-22) (↑ p45, 50).

[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) (↑ p8, 64, 81).

[BL-SafeCurves] Daniel Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. URL: https://safecurves.cr.yp.to (visited on 2018-01-29) (↑ p74, 88).

[BL2017] Daniel Bernstein and Tanja Lange. *Montgomery curves and the Montgomery ladder*. Cryptology ePrint Archive: Report 2017/293. Received March 30, 2017. URL: https://eprint.iacr.org/2017/293 (visited on 2017-11-26) (↑ p90, 95, 96).

[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) (↑ p17).

[Bowe2017] Sean Bowe. *ebfull/pairing source code, BLS12-381 – README.md as of commit e726600*. URL: https://github.com/ebfull/pairing/tree/e72660056e00c93d6b054dfb08ff34a1c67cb799/src/bls12_381 (visited on 2017-07-16) (↑ p48).

[CDG1987] David Chaum, Ivan Damgård, and Jeroen van de Graaf. "Multiparty computations ensuring privacy of each party's input and correctness of the result". In: *Advances in Cryptology - CRYPTO '87. Proceedings of the 14th Annual International Cryptology Conference (Santa Barbara, California, USA, August 16–20, 1987)*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, January 1988, pages 87–119. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_7. URL: https://www.researchgate.net/profile/Jeroen_Van_de_Graaf/publication/242379939_Multiparty_computations_ensuring_secrecy_of_each_party%27s_input_and_correctness_of_the_output (visited on 2018-03-01) (↑ p40).

[CvHP1991] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. *Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer*. February 1991. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.8570 (visited on 2018-02-17). DOI: 10.1.1.34.8570. An extended abstract appeared in *Advances in Cryptology - CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference (Santa Barbara, California, USA, August 11–15, 1991)*; Ed. by Joan Feigenbaum; Vol. 576, Lecture Notes in Computer Science, pages 470–484; Springer, 1992; ISBN 978-3-540-55188-1. (↑ p40, 99).

[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) (↑ p74).

[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) (↑ p66).

[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) (↑ p8).

[FKMSSS2016] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. *Efficient Unlinkable Sanitizable Signatures from Signatures with Re-Randomizable Keys*. Cryptology ePrint Archive: Report 2012/159. Last revised February 11, 2016. URL: `https://eprint.iacr.org/2015/395` (visited on 2018-03-03). An extended abstract appeared in *Public Key Cryptography – PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography (Taipei, Taiwan, March 6–9, 2016), Proceedings, Part 1*; Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang; Vol. 9614, Lecture Notes in Computer Science, pages 301–330; Springer, 2016; ISBN 978-3-662-49384-7. (↑ p20, 21, 45).

[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) (↑ p71).

[GitHub-jubjub] Daira Hopwood. *GitHub repository 'daira/jubjub': Supporting evidence for security of the Jubjub curve to be used in Zcash*. URL: `https://github.com/daira/jubjub` (visited on 2018-02-18). Based on code written for SafeCurves [BL-SafeCurves] by Daniel Bernstein and Tanja Lange. (↑ p74).

[GitHub-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) (↑ p58).

[Grot2016] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive: Report 2016/260. Last revised May 31, 2016. URL: `https://eprint.iacr.org/2016/260` (visited on 2017-08-03) (↑ p52, 53).

[HW2016] Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the Zcash Protocol*. Zcash blog. April 25, 2016. URL: `https://z.cash/blog/fixing-zcash-vulns.html` (visited on 2016-06-22) (↑ p72).

[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) (↑ p48).

[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) (↑ p48, 74, 75).

[KvE2013] Kaa1el and Hagen von Eitzen. *If a group G has odd order, then the square function is injective (answer)*. Mathematics Stack Exchange. URL: `https://math.stackexchange.com/a/522277/185422` (visited on 2018-02-08). Version: 2013-10-11. (↑ p51).

[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 (↑ p71).

[libsnark-fork] *libsnark: C++ library for zkSNARK proofs (Zcash fork)*. URL: `https://github.com/zcash/zcash/tree/master/src/snark` (visited on 2018-02-04) (↑ p52).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: `https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html` (visited on 2016-02-01) (↑ p74).

[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 (23–25 August 2010, Atlanta, GA, USA, 23–25 August 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) (↑ p74).

[Naka2008] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. October 31, 2008. URL: `https://bitcoin.org/en/bitcoin-paper` (visited on 2016-08-14) (↑ p6).

[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) (↑ p37, 54).

[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) (↑ p52).

[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) (↑ p6).

[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). (↑ p44).

[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) (↑ p54).

[Unicode] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2016. URL: `http://www.unicode.org/versions/latest/` (visited on 2016-08-31) (↑ p53).

[vanS2014] Nicolas van Saberhagen. *CryptoNote v 2.0*. Date disputed. URL: `https://cryptonote.org/whitepaper.pdf` (visited on 2016-08-17) (↑ p8).

[WG2016] Zooko Wilcox and Jack Grigg. *Why Equihash?* Zcash blog. April 15, 2016. URL: `https://z.cash/blog/why-equihash.html` (visited on 2016-08-05) (↑ p64).

[Zave2012] 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) (↑ p74).

[ZcashIssue-2113] Simon Liu. *GitHub repository ‘zcash/zcash’: Issue 2113*. URL: `https://github.com/zcash/zcash/issues/2113` (visited on 2017-02-20) (↑ p69, 80).

[ZIP-143] Jack Grigg and Daira Hopwood. *Transaction Signature Verification for Overwinter*. Zcash Improvement Proposal 143. Created December 27, 2017. URL: `https://github.com/zcash/zips/blob/master/zip-0143.rst` (visited on 2018-03-01) (↑ p38, 58).

[ZIP-200] Jack Grigg. *Network Upgrade Mechanism*. Zcash Improvement Proposal 200. Created January 8, 2018. URL: `https://github.com/zcash/zips/blob/master/zip-0200.rst` (visited on 2018-03-01) (↑ p58).

[ZIP-201] Simon Liu. *Network Peer Management for Overwinter*. Zcash Improvement Proposal 201. Created January 15, 2018. URL: `https://github.com/zcash/zips/blob/master/zip-0201.rst` (visited on 2018-03-01) (↑ p58).

[ZIP-202] Simon Liu. *Version 3 Transaction Format for Overwinter*. Zcash Improvement Proposal 202. Created January 10, 2018. URL: `https://github.com/zcash/zips/blob/master/zip-0202.rst` (visited on 2018-03-01) (↑ p58).

[ZIP-203] Jay Graber. *Transaction Expiry*. Zcash Improvement Proposal 203. Created January 9, 2018. URL: `https://github.com/zcash/zips/blob/master/zip-0203.rst` (visited on 2018-03-01) (↑ p58, 59).

Appendices

A Circuit Design

A.1 Quadratic Arithmetic Programs

Sapling defines two circuits, Spend and Output, each implementing an abstract statement described in §4.11.2 ***'Spend Statement (Sapling)'*** on p. 32 and §4.11.3 ***'Output Statement (Sapling)'*** on p. 33 respectively. At the next lower level, each circuit is defined in terms of a *quadratic arithmetic program*, detailed in this section. The description given here is necessary to compute witness elements for the circuit.

Let $\mathbb{F}_{r_\mathbb{S}}$ be the finite field over which Jubjub is defined, as given in §5.4.8.3 ***'Jubjub'*** on p. 50.

A *quadratic arithmetic program* consists of a set of constraints over variables in $\mathbb{F}_{r_\mathbb{S}}$, each of the form:

$$(A) \times (B) = (C)$$

where (A), (B), and (C) are *linear combinations* of variables and constants in $\mathbb{F}_{r_\mathbb{S}}$.

Here $\times$ and $\cdot$ both represent multiplication in the field $\mathbb{F}_{r_\mathbb{S}}$, but we use $\times$ for multiplications corresponding to gates of the circuit, and $\cdot$ for multiplications by constants in the terms of a *linear combination*.

A.2 Elliptic curve background

The circuit makes use of a twisted Edwards curve, Jubjub, and also a Montgomery curve that is birationally equivalent to Jubjub. From here on we omit "twisted" when referring to the Edwards Jubjub curve or coordinates. Following the notation in [BL2017] we use (u, v) for affine coordinates on the Edwards curve, and (x, y) for affine coordinates on the Montgomery curve.

The Montgomery curve has parameters $A_\mathbb{M} = 40962$ and $B_\mathbb{M} = 1$. We use an affine representation of this curve with the formula:

$$B_\mathbb{M} \cdot y^2 = x^3 + A_\mathbb{M} \cdot x^2 + x$$

Usually, elliptic curve arithmetic over prime fields is implemented using some form of projective coordinates, in order to reduce the number of expensive inversions required. In the circuit, it turns out that a division can be implemented at the same cost as a multiplication, i.e. one constraint. Therefore it is beneficial to use affine coordinates for both curves.

We define the following types representing affine Edwards and Montgomery coordinates respectively:

$$\mathsf{AffineEdwardsJubjub} := (u \,⦂\, \mathbb{F}_{r_\mathbb{S}}) \times (v \,⦂\, \mathbb{F}_{r_\mathbb{S}}) : a_\mathbb{J} \cdot u^2 + v^2 = 1 + d_\mathbb{J} \cdot u^2 \cdot v^2$$
$$\mathsf{AffineMontJubjub} := (x \,⦂\, \mathbb{F}_{r_\mathbb{S}}) \times (y \,⦂\, \mathbb{F}_{r_\mathbb{S}}) : B_\mathbb{M} \cdot y^2 = x^3 + A_\mathbb{M} \cdot x^2 + x$$

We also define a type representing compressed, ***not necessarily valid***, Edwards coordinates:

$$\mathsf{CompressedEdwardsJubjub} := (\tilde{u} \,⦂\, \mathbb{B}) \times (v \,⦂\, \mathbb{F}_{r_\mathbb{S}})$$

See §5.4.8.3 ***'Jubjub'*** on p. 50 for how this type is represented as a byte sequence in external encodings.

We use affine Montgomery arithmetic in parts of the circuit because it is more efficient, in terms of the number of constraints, than affine Edwards arithmetic.

An important consideration when using Montgomery arithmetic is that the addition formula is not complete, that is, there are cases where it produces the wrong answer. We must ensure that these cases do not arise.

We will need the theorem below about y-coordinates of points on Montgomery curves.

Fact: ${A_\mathbb{M}}^2 - 4$ is a nonsquare in $\mathbb{F}_{r_\mathbb{J}}$.

Theorem A.2.1. *Let $P = (x, y)$ be a point other than $(0, 0)$ on a Montgomery curve over $\mathbb{F}_r$ with parameter A, such that $A^2 - 4$ is a nonsquare in $\mathbb{F}_r$. Then $y \neq 0$.*

Proof. Substituting $y = 0$ into the Montgomery curve equation gives $0 = x^3 + A \cdot x^2 + x = x \cdot (x^2 + A \cdot x + 1)$. So either $x = 0$ or $x^2 + A \cdot x + 1 = 0$. Since $P \neq (0, 0)$, the case $x = 0$ is excluded. In the other case, complete the square for $x^2 + A \cdot x + 1 = 0$ to give the equivalent $(2 \cdot x + A)^2 = A^2 - 4$. The left-hand side is a square, so if the right-hand side is a nonsquare, then there are no solutions for x. □

A.3 Circuit Components

Each of the following sections describes how to implement a particular component of the circuit, and counts the number of constraints required. Some components make use of others; the order of presentation is "bottom-up".

It is important for security to ensure that variables intended to be of boolean type are boolean-constrained; and for efficiency that they are boolean-constrained only once. We explicitly state for the boolean inputs and outputs of each component whether they are boolean-constrained by the component, or are assumed to have been boolean-constrained separately.

Affine coordinates for elliptic curve points are assumed to represent points on the relevant curve, unless otherwise specified.

In this section, variables have type $\mathbb{F}_{r_\mathbb{S}}$ unless otherwise specified. In contrast to most of this document, we use zero-based indexing in order to more closely match the implementation.

A.3.1 Operations on individual bits

A.3.1.1 Boolean constraints

A boolean constraint $b \in \mathbb{B}$ can be implemented as:

$$\big(1 - b\big) \times \big(b\big) = \big(0\big)$$

A.3.1.2 Selection constraints

A selection constraint $b \mathbin{?} x : y = z$, where $b \mathrel{⦂} \mathbb{B}$ has been boolean-constrained, can be implemented as:

$$\big(b\big) \;\times\; \big(y - x\big) \;=\; \big(y - z\big)$$

A.3.1.3 Nonzero constraints

Since only nonzero elements of $\mathbb{F}_{r_\mathbb{S}}$ have a multiplicative inverse, the assertion $a \neq 0$ can be implemented by witnessing the inverse, $a_{\mathsf{inv}} = a^{-1} \pmod{r_\mathbb{S}}$:

$$\big(a_{\mathsf{inv}}\big) \;\times\; \big(a\big) \;=\; \big(1\big)$$

A global optimization allows to use a single inverse computation outside the circuit for any number of nonzero constraints. Suppose that we have n variables (or *linear combinations*) that are supposed to be nonzero: $a_{0\,..\,n-1}$. Multiply these together (using $n-1$ constraints) to give $a^* = \prod_{i=0}^{n-1} a_i$; then, constrain a^* to be nonzero. This works because the product a^* is nonzero if and only if all of $a_{0\,..\,n-1}$ are nonzero.

A.3.1.4 Exclusive-or constraints

An exclusive-or operation $a \oplus b = c$, where $a, b \mathrel{⦂} \mathbb{B}$ are already boolean-constrained, can be implemented in one constraint as:

$$\big(2 \cdot a\big) \;\times\; \big(b\big) \;=\; \big(a + b - c\big)$$

This automatically boolean-constrains c. Its correctness can be seen by checking the truth table of (a, b).

A.3.2 Operations on multiple bits

A.3.2.1 [Un]packing modulo $r_\mathbb{S}$

Let $n \mathrel{⦂} \mathbb{N}^+$ be a constant. The operation of converting a field element, $a \mathrel{⦂} \mathbb{F}_{r_\mathbb{S}}$, to a sequence of boolean variables $b_{0\,..\,n-1} \mathrel{⦂} \mathbb{B}^{[n]}$ such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i \pmod{r_\mathbb{S}}$, is called "*unpacking*". The inverse operation is called "*packing*".

In the *quadratic arithmetic program* these are the same operation (but see the note about canonical representation below). We assume that the variables $b_{0\,..\,n-1}$ are boolean-constrained separately.

We have $a \bmod r_\mathbb{S} = \left(\displaystyle\sum_{i=0}^{n-1} b_i \cdot 2^i\right) \bmod r_\mathbb{S} = \left(\displaystyle\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_\mathbb{S})\right) \bmod r_\mathbb{S}$.

This can be implemented in one constraint:

$$\left(\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_\mathbb{S})\right) \;\times\; \big(1\big) \;=\; \big(a\big)$$

Notes:

- The bit length n is not limited by the field element size.
- Since the constraint has only a trivial multiplication, it is possible to eliminate it by merging it into the boolean constraint of one of the output bits, expressing that bit as a linear combination of the others and a. However, this optimization requires substitutions that would interfere with the modularity of the circuit implementation (for a saving of only one constraint per unpacking operation), and so we do not use it for the **Sapling** circuit. TODO: Do we want to use it internally to the BLAKE2s implementation where modularity is not significantly affected?
- In the case $n = 255$, for $a < 2^{255} - r_{\mathbb{S}}$ there are two possible representations of $a : \mathbb{F}_{r_{\mathbb{S}}}$ as a sequence of 255 bits, corresponding to $\mathsf{I2LEBSP}_{255}(a)$ and $\mathsf{I2LEBSP}_{255}(a+r_{\mathbb{S}})$. This is a potential hazard, but it may or may not be necessary to force use of the canonical representation $\mathsf{I2LEBSP}_{255}(a)$, depending on the context in which the [un]packing operation is used. We therefore do not consider this to be part of the [un]packing operation itself.

A.3.2.2 Range check

Let $n : \mathbb{N}^+$ be a constant, and let $a = \sum_{i=0}^{n-1} a_i \cdot 2^i : \mathbb{N}$. Suppose we want to constrain $a \leq c$ for some ***constant*** $c = \sum_{i=0}^{n-1} c_i \cdot 2^i : \mathbb{N}$.

Without loss of generality we can assume that $c_{n-1} = 1$, because if it were not then we would decrease n accordingly.

Note that since a and c are provided in binary representation, their bit length n is not limited by the field element size. We ***do not*** assume that the bits $a_{0\,..\,n-1}$ are already boolean-constrained.

Suppose c has k bits set to 1, and let $j_{0\,..\,k-1}$ be the indices of those bits in ascending order. Let t be the minimum of $k-1$ and the number of trailing 1 bits in c.

Let $\Pi_{j_{k-1}} = a_{j_{k-1}}$. For $z \in \{t\,..\,k-2\}$, constrain:

$$\big(\Pi_{j_{z+1}}\big) \times \big(a_{j_z}\big) = \big(\Pi_{j_z}\big)$$

For $i \in \{0\,..\,n-1\}$:

- if $c_i = 0$, constrain $\big(1 - \Pi_{j_z} - a_i\big) \times \big(a_i\big) = \big(0\big)$ where j_z is the least element of j greater than i;
- if $c_i = 1$, boolean-constrain a_i as in §A.3.1.1 ***'Boolean constraints'*** on p. 91.

Note that the constraints corresponding to zero bits of c are ***in place of*** boolean constraints on bits of a_i.

This costs $n + k - 1 - t$ constraints.

TODO: Explain why this works (see `https://github.com/zcash/zcash/issues/2234#issuecomment-338930637`).

A.3.3 Elliptic curve operations

A.3.3.1 Checking that affine Edwards coordinates are on the curve

To check that (u, v) is a point on the Edwards curve, use:

$$\big(u\big) \times \big(u\big) = \big(uu\big)$$
$$\big(v\big) \times \big(v\big) = \big(vv\big)$$
$$\big(d_{\mathbb{J}} \cdot uu\big) \times \big(vv\big) = \big(a_{\mathbb{J}} \cdot uu + vv - 1\big)$$

A.3.3.2 Edwards [de]compression and validation

Define $\mathsf{DecompressValidate} \colon \mathsf{CompressedEdwardsJubjub} \to \mathsf{AffineEdwardsJubjub}$ as follows:

$\mathsf{DecompressValidate}(\tilde{u}, v)$:

// Prover supplies the u-coordinate.

Let $u \colon \mathbb{F}_{r_\mathbb{S}}$.

// §A.3.3.1 ***'Checking that affine Edwards coordinates are on the curve'*** on p. 93.

Check that (u, v) is a point on the Edwards curve.

// §A.3.2.1 ***'[Un]packing modulo $r_\mathbb{S}$'*** on p. 92.

Unpack u to $\sum_{i=0}^{254} u_i \cdot 2^i$, equating $\tilde{u}$ with u_0.

// §A.3.2.2 ***'Range check'*** on p. 93.

Check that $\sum_{i=0}^{254} u_i \cdot 2^i \leq r_\mathbb{S} - 1$.

Return (u, v).

This costs 3 constraints for the curve equation check, 1 constraint for the unpacking, and $255 + 133 - 1$ constraints for the range check (which includes boolean-constraining $u_{0\,..\,254}$), for a total of 391 constraints.

The same *quadratic arithmetic program* be used for compression and decompression.

Note: The point-on-curve check could be omitted if (u, v) were already known to be on the curve. However, the **Sapling** circuit never omits it; this provides a redundant consistency check on the elliptic curve arithmetic in some cases.

A.3.3.3 Edwards $\leftrightarrow$ Montgomery conversion

Define $\mathsf{EdwardsToMont} \colon \mathsf{AffineEdwardsJubjub} \to \mathsf{AffineMontJubjub}$ as follows:

$$\mathsf{EdwardsToMont}(u, v) = \left(\frac{1 + v}{1 - v}, \sqrt{-40964} \cdot \frac{1 + v}{(1 - v) \cdot u}\right) \qquad [1 - v \neq 0 \text{ and } u \neq 0]$$

Define $\mathsf{MontToEdwards} \colon \mathsf{AffineMontJubjub} \to \mathsf{AffineEdwardsJubjub}$ as follows:

$$\mathsf{MontToEdwards}(x, y) = \left(\sqrt{-40964} \cdot \frac{x}{y}, \frac{x - 1}{x + 1}\right) \qquad [x + 1 \neq 0 \text{ and } y \neq 0]$$

Either of these conversions can be implemented by the same *quadratic arithmetic program*:

$$\begin{aligned}
\big(y\big) \;\times\; \big(u\big) \;&=\; \big(\sqrt{-40964} \cdot x\big) \\
\big(x + 1\big) \;\times\; \big(v\big) \;&=\; \big(x - 1\big)
\end{aligned}$$

The above conversions should only be used if the input is guaranteed to be a point on the relevant curve. If that is the case, the theorems below enumerate all exceptional inputs that may violate the side-conditions.

Theorem A.3.1. *Let (u, v) be an affine point on a complete twisted Edwards curve. Then the only points with $u \neq 0$ or $v \neq 0$ are $(0, 1) = \mathcal{O}_\mathbb{J}$; $(0, -1)$ of order 2; and $\left(\pm 1/\sqrt{a_\mathbb{J}}, 0\right)$ of order 4.*

Proof. Straightforward from the curve equation. (The fact that the points $\big(\pm 1/\sqrt{a_{\mathbb{J}}}, 0\big)$ are of order 4 can be inferred by applying the doubling formula.) □

Theorem A.3.2. *Let (x, y) be an affine point on a Montgomery curve over $\mathbb{F}_r$ with parameter A such that $A^2 - 4$ is a nonsquare in $\mathbb{F}_r$, that is birationally equivalent to a complete twisted Edwards curve. Then $x + 1 \neq 0$, and the only point (x, y) with $y = 0$ is $(0, 0)$ of order 2.*

Proof. That the only point with $y = 0$ is $(0, 0)$ is proven by Theorem A.2.1 on p. 91.

If $x+1 = 0$, then subtituting $x = -1$ into the Montgomery curve equation gives $B_{\mathbb{M}} \cdot y^2 = x^3 + A_{\mathbb{M}}.x^2 + x = A_{\mathbb{M}} - 2$. So in that case $y^2 = (A_{\mathbb{M}} - 2)/B_{\mathbb{M}}$. The right-hand-side is equal to the parameter d of a particular complete twisted Edwards curve birationally equivalent to the Montgomery curve (see [BL2017, section 4.3.5]). For all complete twisted Edwards curves, d is nonsquare, so this equation has no solutions for y, hence $x + 1 \neq 0$. □

(The complete twisted Edwards curve referred to in the proof is an isomorphic y-coordinate rescaling of the *Jubjub curve*.)

A.3.3.4 Affine-Montgomery arithmetic

The incomplete affine-Montgomery addition formulae given in [BL2017, section 4.3.2] are:

$$\begin{aligned}
x_3 &= B_{\mathbb{M}} \cdot \lambda^2 - A_{\mathbb{M}} - x_1 - x_2 \\
y_3 &= (x_1 - x_3) \cdot \lambda^2 - y_1 \\
\text{where } \lambda &= \begin{cases} \dfrac{3 \cdot x_1^2 + 2 \cdot A_{\mathbb{M}} \cdot x_1 + 1}{2 \cdot B_{\mathbb{M}} \cdot y_1}, & \text{if } x_1 = x_2 \\[1ex] \dfrac{y_2 - y_1}{x_2 - x_1}, & \text{otherwise.} \end{cases}
\end{aligned}$$

The following theorem helps to determine when these incomplete addition formulae can be safely used:

Theorem A.3.3. *Let Q be a point of odd-prime order s on a Montgomery curve $E_{A_{\mathbb{M}}, B_{\mathbb{M}}}/\mathbb{F}_{r_{\mathbb{S}}}$. Let $k_{1\,..\,2}$ be integers in $\{-\frac{s-1}{2} \,..\, \frac{s-1}{2}\} \setminus \{0\}$. Let $P_i = [k_i]\,Q = (x_i, y_i)$ for $i \in \{1\,..\,2\}$, with $k_1 \neq \pm k_2$. Then the non-unified addition constraints*

$$\begin{aligned}
\big(x_2 - x_1\big) &\times \big(\lambda\big) = \big(y_2 - y_1\big) \\
\big(B_{\mathbb{M}} \cdot \lambda\big) &\times \big(\lambda\big) = \big(A_{\mathbb{M}} + x_1 + x_2 + x_3\big) \\
\big(x_1 - x_3\big) &\times \big(\lambda\big) = \big(y_3 + y_1\big)
\end{aligned}$$

implement the affine-Montgomery addition $P_1 + P_2 = (x_3, y_3)$ for all such $P_{1\,..\,2}$.

Proof. The given constraints are equivalent to the Montgomery addition formulae under the side condition $x_1 \neq x_2$. (Note that neither P_i can be the zero point since $k_{1\,..\,2} \neq 0 \pmod{s}$.) Assume for a contradiction that $x_1 = x_2$. For any $P_1 = [k_1]\,Q$, there can be only one other point $-P_1$ with the same x-coordinate. (This follows from the fact that the curve equation determines $\pm y$ as a function of x.) But $-P_1 = [-1]\,[k_1]\,Q = [-k_1]\,Q$. Since $k \colon \{-\frac{s-1}{2} \,..\, \frac{s-1}{2}\} \mapsto [k]\,Q \colon \mathbb{J}$ is injective and $k_{1\,..\,2}$ are in $\{-\frac{s-1}{2} \,..\, \frac{s-1}{2}\}$, then $k_2 = \pm k_1$ (contradiction). □

The conditions of this theorem are called the *distinct-x criterion*.

In particular, if $k_{1\,..\,2}$ are integers in $\{1 \,..\, \frac{s-1}{2}\}$ then it is sufficient to require $k_1 \neq k_2$, since that implies $k_1 \neq \pm k_2$.

Affine-Montgomery doubling can be implemented as:

$$\begin{aligned}
&(x) \times (x) = (xx) \\
&(2 \cdot B_{\mathbb{M}} \cdot y) \times (\lambda) = (3 \cdot xx + 2 \cdot A_{\mathbb{M}} \cdot x + 1) \\
&(B_{\mathbb{M}} \cdot \lambda) \times (\lambda) = (A_{\mathbb{M}} + 2 \cdot x + x_3) \\
&(x - x_3) \times (\lambda) = (y_3 + y)
\end{aligned}$$

This doubling formula is valid when $y \neq 0$, which is the case when (x, y) is not the point $(0, 0)$ (the only point of order 2), as proven in Theorem A.2.1 on p. 91.

A.3.3.5 Affine-Edwards arithmetic

Formulae for affine-Edwards addition are given in [BBJLP2008, section 6]. With a change of variable names to match our convention, the formulae for $(u_1, v_1) + (u_2, v_2) = (u_3, v_3)$ are:

$$\begin{aligned}
u_3 &= \frac{u_1 \cdot v_2 + v_1 \cdot u_2}{1 + d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2} \\
v_3 &= \frac{v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2}{1 - d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2}
\end{aligned}$$

We use an optimized implementation found by Daira Hopwood making use of an observation by Bernstein and Lange in [BL2017, last paragraph of section 4.5.2]:

$$\begin{aligned}
&(u_1 + v_1) \times (v_2 - a_{\mathbb{J}} \cdot u_2) = (T) \\
&(u_1) \times (v_2) = (A) \\
&(v_1) \times (u_2) = (B) \\
&(d_{\mathbb{J}} \cdot A) \times (B) = (C) \\
&(1 + C) \times (u_3) = (A + B) \\
&(1 - C) \times (v_3) = (T - A + a_{\mathbb{J}} \cdot B)
\end{aligned}$$

The correctness of this implementation can be seen by expanding $T - A + a_{\mathbb{J}} \cdot B$:

$$\begin{aligned}
T - A + a_{\mathbb{J}} \cdot B &= (u_1 + v_1) \cdot (v_2 - a_{\mathbb{J}} \cdot u_2) - u_1 \cdot v_2 + a_{\mathbb{J}} \cdot v_1 \cdot u_2 \\
&= v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2 + u_1 \cdot v_2 - a_{\mathbb{J}} \cdot v_1 \cdot u_2 - u_1 \cdot v_2 + a_{\mathbb{J}} \cdot v_1 \cdot u_2 \\
&= v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2
\end{aligned}$$

The above addition formulae are "unified", that is, they can also be used for doubling. Affine-Edwards doubling $[2]\,(u, v) = (u_3, v_3)$ can also be implemented slightly more efficiently as:

$$\begin{aligned}
&(u + v) \times (v - a_{\mathbb{J}} \cdot u) = (T) \\
&(u) \times (v) = (A) \\
&(d_{\mathbb{J}} \cdot A) \times (A) = (C) \\
&(1 + C) \times (u_3) = (2 \cdot A) \\
&(1 - C) \times (v_3) = (T + (a_{\mathbb{J}} - 1) \cdot A)
\end{aligned}$$

This implementation is obtained by specializing the addition formulae to $(u, v) = (u_1, v_1) = (u_2, v_2)$ and observing that $u \cdot v = A = B$.

A.3.3.6 Affine-Edwards nonsmall-order check

In order to avoid small-subgroup attacks, we check that certain points used in the circuit are not of small order. In practice the **Sapling** circuit uses this in combination with a check that the coordinates are on the curve (§A.3.3.1 ***'Checking that affine Edwards coordinates are on the curve'*** on p. 93), so we combine the two operations.

The *Jubjub curve* has a large prime-order subgroup with a cofactor of 8. To check for a point P of order 8 or less, we double twice (as in §A.3.3.5 ***'Affine-Edwards arithmetic'*** on p. 96) and check that the resulting u-coordinate is not 0 (as in §A.3.1.3 ***'Nonzero constraints'*** on p. 92).

On a twisted Edwards curve, only the zero point $\mathcal{O}_{\mathbb{J}}$, and the unique point of order 2 at $(0, -1)$ have zero u-coordinate. So this u-coordinate check rejects both $\mathcal{O}_{\mathbb{J}}$ and the point of order 2, and no other points.

The first doubling can be merged with the curve point check to avoid recomputing C or T. The second doubling does not need to compute T or the v-coordinate of the result; also, the u-coordinate of the result is zero if-and-only-if the intermediate value A is zero.

// Curve equation check.

$\big(u\big) \times \big(u\big) = \big(uu\big)$

$\big(v\big) \times \big(v\big) = \big(vv\big)$

$\big(d_{\mathbb{J}} \cdot uu\big) \times \big(vv\big) = \big(a_{\mathbb{J}} \cdot uu + vv - 1\big)$

// First doubling; subsitute $C = d_{\mathbb{J}} \cdot uu \cdot vv = a_{\mathbb{J}} \cdot uu + vv - 1$ and $T + (a_{\mathbb{J}} - 1) \cdot A = vv - a_{\mathbb{J}} \cdot uu$.

$\big(u\big) \times \big(v\big) = \big(A_1\big)$

$\big(a_{\mathbb{J}} \cdot uu + vv\big) \times \big(u_1\big) = \big(2 \cdot A_1\big)$

$\big(2 - a_{\mathbb{J}} \cdot uu - vv\big) \times \big(v_1\big) = \big(vv - a_{\mathbb{J}} \cdot uu\big)$

// Second doubling and non-zero check.

$\big(u_1\big) \times \big(v_1\big) = \big(A_2\big)$

// u-coordinate is zero if-and-only-if A_2 is zero.

$\big(A_{\mathsf{inv}}\big) \times \big(A_2\big) = \big(1\big)$

The total cost, including the curve check, is $3 + 3 + 2 = 8$ constraints.

Notes:

- This ***does not*** ensure that the point is in the prime-order subgroup.
- If the point P is used as the base of a variable-base scalar multiplication using the algorithm of §A.3.3.8 ***'Variable-base affine-Edwards scalar multiplication'*** on p. 98, then $[4]\, P$ will be calculated as Base_2. Then $\mathcal{U}(\mathsf{Base}_2) \neq 0$ can be checked using a single constraint (saving 4 constraints). The **Sapling** circuit does not use this optimization.

A.3.3.7 Fixed-base affine-Edwards scalar multiplication

If the base point B is fixed for a given scalar multiplication $[k]\,B$, we can fully precompute window tables for each window position.

It is most efficient to use 3-bit fixed windows. Since the length of $r_{\mathbb{J}}$ is 252 bits, we need 84 windows.

Express k in base 8, i.e. $k = \sum_{i=0}^{83} k_i \cdot 8^i$.

Then $[k]\,B = \sum_{i=0}^{83} w_{(B,\,i,\,k_i)}$, where $w_{(B,\,i,\,k_i)} = [k_i \cdot 8^i]\,B$.

We precompute all of $w_{(B,\,i,\,s)}$ for $i \in \{0\,..\,83\}, s \in \{0\,..\,7\}$.

To look up a given window entry $w_{(B,\,i,\,s)} = (u_s, v_s)$, where $s = 4 \cdot s_2 + 2 \cdot s_1 + s_0$, we use:

$$\begin{aligned}
&\big(s_1\big) \times \big(s_0\big) = \big(s_\&\big)\\
&\big(s_2\big) \times \big(-\,u_0 \cdot s_\& + u_0 \cdot s_1 + u_0 \cdot s_0 - u_0 + u_1 \cdot s_\& - u_1 \cdot s_0 + u_2 \cdot s_\& - u_2 \cdot s_1 - u_3 \cdot s_\&\\
&\qquad\qquad + u_4 \cdot s_\& - u_4 \cdot s_1 - u_4 \cdot s_0 + u_4 - u_5 \cdot s_\& + u_5 \cdot s_0 - u_6 \cdot s_\& + u_6 \cdot s_1 + u_7 \cdot s_\&\big) =\\
&\quad\big(u_s - u_0 \cdot s_\& + u_0 \cdot s_1 + u_0 \cdot s_0 - u_0 + u_1 \cdot s_\& - u_1 \cdot s_0 + u_2 \cdot s_\& - u_2 \cdot s_1 - u_3 \cdot s_\&\big)\\
&\big(s_2\big) \times \big(-\,v_0 \cdot s_\& + v_0 \cdot s_1 + v_0 \cdot s_0 - v_0 + v_1 \cdot s_\& - v_1 \cdot s_0 + v_2 \cdot s_\& - v_2 \cdot s_1 - v_3 \cdot s_\&\\
&\qquad\qquad + v_4 \cdot s_\& - v_4 \cdot s_1 - v_4 \cdot s_0 + v_4 - v_5 \cdot s_\& + v_5 \cdot s_0 - v_6 \cdot s_\& + v_6 \cdot s_1 + v_7 \cdot s_\&\big) =\\
&\quad\big(v_s - v_0 \cdot s_\& + v_0 \cdot s_1 + v_0 \cdot s_0 - v_0 + v_1 \cdot s_\& - v_1 \cdot s_0 + v_2 \cdot s_\& - v_2 \cdot s_1 - v_3 \cdot s_\&\big)
\end{aligned}$$

This costs 3 constraints for each of 84 window lookups, plus 6 constraints for each of 83 Edwards additions (as in §A.3.3.5 ***'Affine-Edwards arithmetic'*** on p. 96), for a total of 750 constraints.

Note: It would be more efficient to use arithmetic on the Montgomery curve, as in §A.3.3.9 ***'Pedersen hash'*** on p. 99. However since there are only three instances of fixed-base scalar multiplication in the *Spend circuit* and two in the *Output circuit*[2], the additional complexity was not considered justified for **Sapling**.

A.3.3.8 Variable-base affine-Edwards scalar multiplication

When the base point B is not fixed, the method in the preceding section cannot be used. Instead we use a naïve double-and-add method.

Given $k = \sum_{i=0}^{250} k_i \cdot 2^i$, we calculate $R = [k]\,B$ using:

// $\mathsf{Base}_i = [2^i]\,B$

let $\mathsf{Base}^u_0 = \mathcal{U}(B)$

let $\mathsf{Base}^v_0 = B_v$

let $\mathsf{Acc}^u_0 = k_0 \;?\; B^u : 0$

let $\mathsf{Acc}^v_0 = k_0 \;?\; B^v : 1$

for i from 1 up to 250:

  let $\mathsf{Base}_i = [2]\,\mathsf{Base}_{i-1}$

  // select Base_i or $\mathcal{O}_{\mathbb{J}}$ depending on the bit k_i

[2] A Pedersen commitment uses fixed-base scalar multiplication as a subcomponent.

let $\mathsf{Addend}^u_i = k_i\ ?\ \mathsf{Base}^u_i : 0$

let $\mathsf{Addend}^v_i = k_i\ ?\ \mathsf{Base}^v_i : 1$

let $\mathsf{Acc}_i = \mathsf{Acc}_{i-1} + \mathsf{Addend}^i$

let $R = \mathsf{Acc}_{250}$.

This costs 5 constraints for each of 250 Edwards doublings, 6 constraints for each of 250 Edwards additions, and 2 constraints for each of 251 point selections, for a total of 3252 constraints.

Note: It would be more efficient to use 2-bit fixed windows, and/or to use arithmetic on the Montgomery curve in a similar way to §A.3.3.9 ***'Pedersen hash'*** on p. 99. However since there are only two instances of variable-base scalar multiplication in the *Spend circuit* and one in the *Output circuit*, the additional complexity was not considered justified for **Sapling**.

A.3.3.9 Pedersen hash

The specification of the *Pedersen hashes* used in **Sapling** is given in §5.4.1.7 ***'Pedersen Hash Function'*** on p. 40. It is based on the scheme from [CvHP1991, section 5.2] –for which a tighter security reduction to the Discrete Logarithm Problem was given in [BGG1995]– but tailored to allow several optimizations in the circuit implementation.

Pedersen hashes are the single most commonly used primitive in the **Sapling** circuits. $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ *Pedersen hash* instances are used in the *Spend circuit* to check a Merkle path to the *note commitment* of the *note* being spent. We also reuse the *Pedersen hash* implementation to construct the *commitment scheme* $\mathsf{NoteCommit}^{\mathsf{Sapling}}$.

This motivates considerable attention to optimizing this circuit implementation of this primitive, even at the cost of complexity.

First, we use a windowed scalar multiplication algorithm with signed digits. Each 3-bit message chunk corresponds to a window; the chunk is encoded as an integer from the set $\mathsf{Digits} = \{-4 \,..\, 4\} \setminus \{0\}$. This allows a more efficient lookup of the window entry for each chunk than if the set $\{1 \,..\, 8\}$ had been used, because a point can be conditionally negated using only a single constraint.

Next, we optimize the cost of point addition by allowing as many additions as possible to be performed on the Montgomery curve. An incomplete Montgomery addition costs 3 constraints, in comparison with an Edwards addition which costs 6 constraints.

However, we cannot do all additions on the Montgomery curve because the Montgomery addition is incomplete. In order to be able to prove that exceptional cases do not occur, we need to ensure that the *distinct-x criterion* from §A.3.3.4 ***'Affine-Montgomery arithmetic'*** on p. 95 is met. This requires splitting the input into segments (each using an independent generator), calculating an intermediate result for each segment, and then converting to the Edwards curve and summing the intermediate results using Edwards addition. If the resulting point is R, then (abstracting away the changes of curve) this calculation can be written as:

$$\mathsf{PedersenHashToPoint}(D, M) = \sum_{j=1}^{N} [\langle M_j \rangle]\, \mathcal{I}^D_j$$

where $\langle\cdot\rangle$ and $\mathcal{I}^D_j$ are defined as in §5.4.1.7 ***'Pedersen Hash Function'*** on p. 40.

We have to prove that:

- the *distinct-x criterion* is met for all Montgomery additions within a segment;
- the Montgomery-to-Edwards conversions can be implemented without exceptional cases.

The proof of Theorem 5.4.1 on p. 41 showed that all indices of addition inputs are in the range $\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$.

Because the $\mathcal{I}^D_j$ (which are outputs of $\mathsf{GroupHash}^{\mathbb{J}}$) are all of prime order, and $\langle M_j \rangle \neq 0 \pmod{r_{\mathbb{J}}}$, it is guaranteed that all of the terms $[\langle M_j \rangle]\, \mathcal{I}^D_j$ to be converted to Edwards form are of prime order. From Theorem A.3.2 on p. 95, we can infer that the conversions will not encounter exceptional cases.

We also need to show that the indices of addition inputs are all distinct disregarding sign.

Theorem A.3.4. *For all disjoint nonempty subsets S and S' of $\{1\,..\,c\}$, and for all $m \in \mathbb{B}^{[3]\,[c]}$,*

$$\sum_{j \in S} \mathsf{enc}(m_j) \cdot 2^{4 \cdot (j-1)} \neq \pm \sum_{j' \in S'} \mathsf{enc}(m_{j'}) \cdot 2^{4 \cdot (j'-1)}$$

Proof. TODO: ...

□

When these hashes are used in the circuit, the first two windows of the input are fixed and can be optimized (for example, in the Merkle tree hashes they represent the layer number). This is done by precomputing the sum of the relevant two points, and adding them to the intermediate result for the remainder of the first segment. This requires 3 constraints for a single Montgomery addition rather than .. constraints for 2 window lookups and 2 additions.

Taking into account this optimization, the cost of a Pedersen hash over ℓ bits, with the first 6 bits fixed, is ... constraints. In particular, for the Merkle tree hashes $\ell = 516$, so the cost is ... constraints.

A.3.3.10 Mixing Pedersen hash

A mixing *Pedersen hash* is used to compute ρ from cm and pos in §4.10 ***'Note Commitments and Nullifiers'*** on p. 31. It takes as input a *Pedersen commitment* P, and hashes it with another input x.

Let $\mathcal{J}$ be as defined in §5.4.1.8 ***'Mixing Pedersen Hash Function'*** on p. 42.

We define $\mathsf{MixingPedersenHash} : \{0\,..\,r_{\mathbb{J}} - 1\} \times \mathbb{J} \to \mathbb{J}$ by:

$$\mathsf{MixingPedersenHash}(P, x) := P + [x]\, \mathcal{J}.$$

This costs TODO: ... for the scalar multiplication, and 6 constraints for the Edwards addition, for a total of TODO: ... constraints.

A.3.4 Merkle path check

Checking a Merkle authentication path, as described in §4.7 ***'Merkle path validity'*** on p. 29, requires to:

- boolean-constrain the path bit specifying whether the previous node is a left or right child;
- conditionally swap the previous-layer and sibling hashes (as $\mathbb{F}_r$ elements) depending on the path bit;
- unpack the previous-layer and sibling hashes to 255-bit sequences;
- compute the Merkle hash.

The unpacking need not be canonical in the sense discussed in §A.3.2.1 ***'[Un]packing modulo $r_{\mathbb{S}}$'*** on p. 92; that is, it is ***not*** necessary to ensure that the previous-layer or sibling bit-sequence inputs represent integers in the range $\{0\,..\,r_{\mathbb{S}} - 1\}$. Since the root of the Merkle tree is calculated outside the circuit using the canonical representations, and since the *Pedersen hashes* are collision-resistant on arbitrary bit-sequence inputs, an attempt by an adversarial prover to use a non-canonical input would result in the wrong root being calculated, and the overall path check would fail.

Note that the leaf node input of the authentication path is given as a bit sequence, not as a field element.

For each layer, the cost is $1 + 2 \cdot 255$ boolean constraints, 2 constraints for the conditional swap (implemented as two selection constraints), and todo... for the Merkle hash, for a total of TODO: ... constraints.

Note: The conditional swap $(a_0, a_1) \mapsto (c_0, c_1)$ could be implemented in only one constraint by substituting $c_1 = a_0 + a_1 - c_0$ into the uses of c_1. The **Sapling** circuit does not use this optimization.

A.3.5 Windowed Pedersen Commitment

We construct *windowed Pedersen commitments* by reusing the Pedersen hash implementation, and adding a randomized point:

$$\mathsf{WindowedPedersenCommit}_r(s) = \mathsf{PedersenHashToPoint}(\texttt{"Zcash\_PH"}, s) + [r]\, \mathsf{FindGroupHash}^{\mathbb{J}}(\texttt{"Zcash\_PH"}, \texttt{"r"})$$

This can be implemented in:

- $\ldots \cdot \ell + \ldots$ constraints for the Pedersen hash on $\ell = \mathsf{length}(s)$ bits (again assuming that the first 6 bits are fixed);
- 750 constraints for the fixed-base scalar multiplication;
- 6 constraints for the final Edwards addition

for a total of $\ldots \cdot \ell + 756$ constraints.

A.3.6 Homomorphic Pedersen Commitment

The *windowed Pedersen commitments* defined in the preceding section are highly efficient, but they do not support the homomorphic property we need when instantiating ValueCommit (see §4.9 ***'Balance'*** on p. 30 and §3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 14).

In order to support this property, we also define *homomorphic Pedersen commitments* as follows:

$$\mathsf{HomomorphicPedersenCommit}_{\mathsf{rcv}}(D, \mathsf{v}) = [\mathsf{v}]\, \mathsf{FindGroupHash}^{\mathbb{J}}(D, \texttt{"v"}) + [\mathsf{rcv}]\, \mathsf{FindGroupHash}^{\mathbb{J}}(D, \texttt{"r"})$$

In the case that we need for ValueCommit, v has 64 bits[3]. This can be straightforwardly implemented in ... constraints.

A.3.7 BLAKE2s hashes

BLAKE2s is defined in [ANWW2013]. Its main subcomponent is a "G function", defined as follows:

$G \colon \ldots \to \ldots$

$G(\ldots) = \ldots$

A 32-bit exclusive-or can be implemented in 32 constraints, one for each bit position $a \oplus b = c$ as in §A.3.1.4 ***'Exclusive-or constraints'*** on p. 92.

Additions not involving a message word require 33 constraints:

...

Additions of message words require one extra constraint each, i.e. $a + b + m = c$ is implemented by declaring 34 boolean variables, and ...

There are $10 \cdot 4 \cdot 2$ such message word additions.

[3] It would be sufficient to use 51 bits, which accomodates the range $\{0 \,..\, \mathsf{MAX\_MONEY}\}$, but the **Sapling** circuit uses 64.

Each G evaluation requires 260 constraints. There are $10 \cdot 8$ instances of G:

...

There are also 8 output exclusive-ors.

The total cost is 21136 constraints. This includes boolean-constraining the hash output bits, but not the input bits.

Note: It should be clear that BLAKE2s is very expensive in the circuit compared to elliptic curve operations. This is primarily because it is inefficient to use $\mathbb{F}_{r_{\mathbb{S}}}$ elements to represent single bits. However Pedersen hashes do not have the necessary cryptographic properties for the two cases where the *Spend circuit* uses BLAKE2s. While it might be possible to use variants of functions with low circuit cost such as MiMC [AGRRT2017], it was felt that they had not yet received sufficient cryptanalytic attention to confidently use them for **Sapling**.

A.4 The SaplingSpend circuit

...

A.5 The SaplingOutput circuit

...