Zcash Protocol Specification

Version unavailable (check protocol.ver)

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

November 30, 2022

[1]

Abstract. Zcash is an implementation of the *Decentralized Anonymous Payment scheme* **Zerocash**, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by **Bitcoin** with a ***shielded*** payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (*zk-SNARKs*). It attempted to address the problem of mining centralization by use of the *Equihash* memory-hard proof-of-work algorithm.

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

[†] Electric Coin Company

[1] Jubjub bird image credit: Peter Newell 1902; Daira Hopwood 2018.

1 Introduction

#introduction

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

In this document, technical terms for concepts that play an important rôle in **Zcash** are written in *slanted text*, which links to an index entry. ***Italics*** are used for emphasis and for references between sections of the document. The symbol § precedes section numbers in cross-references.

The key words **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** 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.

The most significant changes from the original **Zerocash** are explained in **§? '??'** on p. **??**.

Changes specific to the **Overwinter** upgrade are highlighted in .

Changes specific to the **Sapling** upgrade following **Overwinter** are highlighted in .

All of these are also changes from **Zerocash**. The name **Sprout** is used for the **Zcash** protocol prior to **Sapling** (both before and after **Overwinter**), and in particular its shielded protocol.

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 the **Zcash** protocol.
- Consensus Changes from **Bitcoin** — how **Zcash** differs from **Bitcoin** at the consensus layer, including the Proof of Work;
- Differences from the **Zerocash** protocol — a summary of changes from the protocol in [**BCGGMTV2014**].
- Appendix: Circuit Design — details of how the **Sapling** circuits are defined as *quadratic constraint programs*.
- Appendix: Batching Optimizations — improvements to the efficiency of validating multiple signatures and verifying multiple proofs.

1.1 Caution

#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

#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.

All value in **Zcash** belongs to some *chain value pool*. There is a single *transparent chain value pool*, and also a *chain value pool* for each *shielded* protocol (**Sprout**). Transfers of *transparent* value work essentially as in **Bitcoin** and have the same privacy properties. Value in a *shielded chain value pool* is carried by *notes*[2], 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*. Once the *transaction* creating a *note* has been mined, the *note* is associated with a fixed *note position* in a tree of *note commitments*, and with a *nullifier***??** unique to that *note*. Computing the *nullifier* requires the associated private *spending key*. It is infeasible to correlate the *note commitment* or *note position* with the corresponding *nullifier* without knowledge of at least this key. An unspent valid *note*, at a given point on the *block chain*, is one for which the *note commitment* has been publically revealed on the *block chain* prior to that point, but the *nullifier* has not.

A *transaction* can contain *transparent* inputs, outputs, and scripts, which all work as in **Bitcoin** [**Bitcoin-Protocol**]. It also can include *JoinSplit 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*.) It is also possible for value to be transferred between *chain value pools*, either *transparent* or *shielded*; this always reveals the amount transferred.

In each *shielded transfer*, 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 and signatures, which prove that all of the following hold except with insignificant probability:

For each *shielded input*,

-
- if the value is nonzero, some revealed *note commitment* exists for this *note*;
- the prover knew the *proof authorizing key* of the *note*;
- the *nullifier* and *note commitment* are computed correctly.

and for each *shielded output*,

-
- the *note commitment* is computed correctly;
- it is infeasible to cause the *nullifier* of the output *note* to collide with the *nullifier* of any other *note*.

For **Sprout**, the *JoinSplit statement* also includes an explicit balance check.

In addition, various measures 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

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

holder of the corresponding *private key*, which in this context is called the *receiving key*. This facility is used to communicate encrypted output *notes* on the *block chain* to their intended recipient, who can use the *receiving key* to scan the *block chain* for *notes* addressed to them and then decrypt those *notes*.

The basis of the privacy properties of **Zcash** is that when a *note* is spent, the spender only proves that some commitment for it had been revealed, without revealing which one. This implies that a spent *note* cannot be linked to the *transaction* in which it was created. That is, from an adversary's point of view the set of possibilities for a given *note* input to a *transaction* —its *note traceability set*— includes ***all*** previous notes that the adversary does not control or know to have been spent.[3] This contrasts with other proposals for private payment systems, such as CoinJoin [**Bitcoin-CoinJoin**] or **CryptoNote** [**vanSaberh2014**], that are based on mixing of a limited number of transactions and that therefore have smaller *note traceability sets*.

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

2 Notation

#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 $f_x(y)$.

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

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

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

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

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

$x : T \mapsto e_x : U$ 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.

$x : T \mapsto_{\notin V} e_x : U$ means $x : T \mapsto e_x : U \cup V$ restricted to the domain $\{x : T \mid e_x \notin V\}$ and range U.

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

$\perp$ is a distinguished value used to indicate unavailable information, a failed decryption or validity check, or an exceptional case.

$T^{[\ell]}$, 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.

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

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

$\mathsf{truncate}_k(S)$ means the sequence formed from the first k elements of S.

`0x` followed by a string of `monospace` hexadecimal digits means the corresponding integer converted from hexadecimal. $[\texttt{0x00}]^\ell$ means the sequence of ℓ zero bytes.

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

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

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

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

$\mathsf{concat}_{\mathbb{B}}(S)$ means the sequence of bits obtained by concatenating the elements of S as bit sequences.

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

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

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

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

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

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

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

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

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

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

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

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

$\sqrt[+]{a}$, where $a : \mathbb{F}_q$, means the positive square root of a in $\mathbb{F}_q$, i.e. in the range $\{0 \,..\, \frac{q-1}{2}\}$. It is only used in cases where the square root must exist.

$\sqrt[?]{a}$, where $a : \mathbb{F}_q$, means an arbitrary square root of a in $\mathbb{F}_q$, or $\perp$ if no such square root exists.

$b \;?\; x : y$ 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 **§? '??'** on p. **??**.

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

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

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

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

The following integer constants will be instantiated in **§? '??'** on p. **??**:

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

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

We use the abbreviation "*ctEdwards*" to refer to *complete twisted Edwards elliptic curves* and coordinates (see **§? '??'** on p. **??**).

3 Concepts

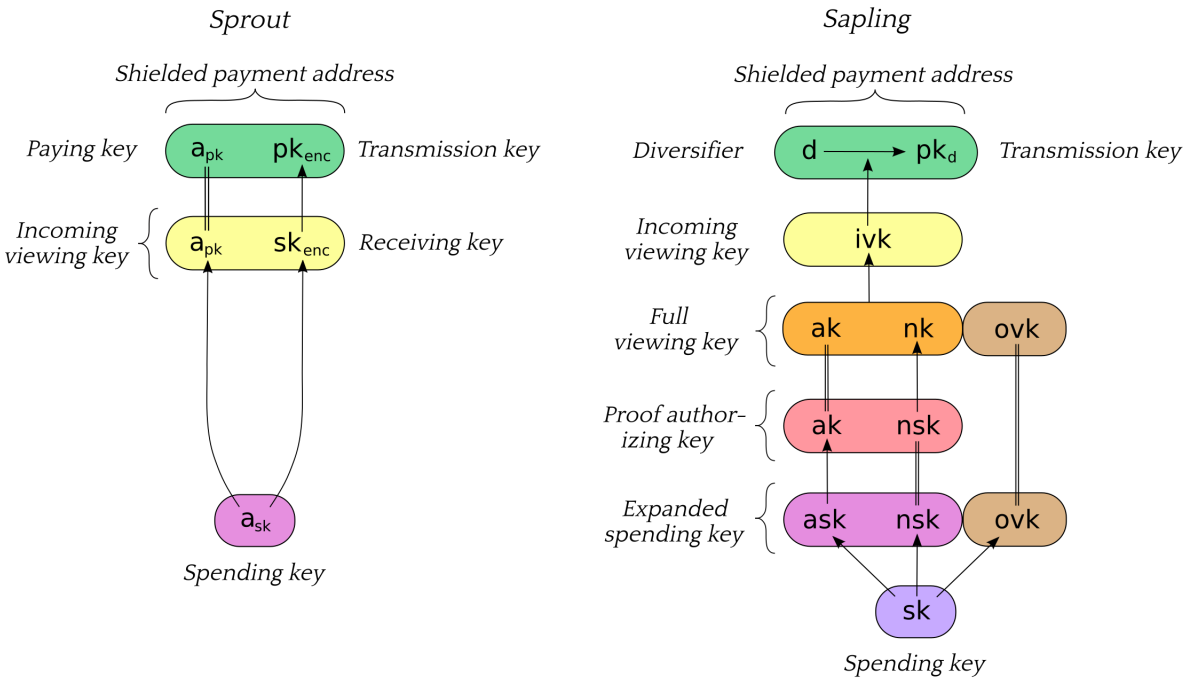
#concepts

3.1 Payment Addresses and Keys

#addressesandkeys

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

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



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

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

Users can accept payment from multiple parties with a single *shielded payment address* and the fact that these payments are destined to the same payee is not revealed on the *block chain*, even to the paying parties. ***However*** if two parties collude to compare a *shielded payment address* they can trivially determine they are the same. In the case that a payee wishes to prevent this they should create a distinct *shielded payment address* for each payer.

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

3.2 Notes

#notes

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

Let MAX_MONEY, $\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

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

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

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

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

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

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

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

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

where $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is instantiated in **§? '??'** on p. **??**.

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

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

The *nullifier* computation uses a *Pseudo Random Function* (see **§? '??'** on p. **??**), as described in **§? '??'** on p. **??**.

A *note* is spent by proving knowledge of $(\rho, \mathsf{a_{sk}})$ in zero knowledge while publically disclosing the *note*'s *nullifier* nf, allowing nf to be used to prevent double-spending.

3.2.1 Note Plaintexts and Memo Fields

#noteptconcept

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

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

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

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

memo represents a 512-byte *memo field* associated with this *note*. The usage of the *memo field* is by agreement between the sender and recipient of the *note*.

Encodings are given in **§? '??'** on p. **??**. The result of encryption forms part of a *transmitted note(s) ciphertext*. For further details, see **§? '??'** on p. **??**.

3.3 The Block Chain

#blockchain

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 **§? '??'** on p. **??**).

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

Each *block* in a *block chain* has a *block height*. The *block height* of the *genesis block* is 0, and the *block height* of each subsequent *block* in the *block chain* increments by 1. Implementations **MUST** support *block heights* up to and including $2^{31} - 1$.

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 **§? '??'** on p. **??**, of all *blocks* in each *valid block chain*, and considers the *valid block chain* with greatest total work to be best. To break ties between leaf *blocks*, a node will prefer the *block* that it received first.

The consensus protocol is designed to ensure that for any given *block height*, the vast majority of well-connected nodes should eventually agree on their *best valid block chain* up to that height. A *full validator*[4] **SHOULD** attempt to obtain candidate *blocks* from multiple sources in order to increase the likelihood that it will find a *valid block chain* that reflects a recent consensus state.

A *network upgrade* is *settled* on a given *network* when there is a social consensus that it has activated with a given *activation block* hash. A *full validator* that potentially risks *Mainnet* funds or displays *Mainnet transaction* information to a user **MUST** do so only for a *block chain* that includes the *activation block* of the most recent *settled network upgrade*, with the corresponding *activation block* hash. Currently, there is social consensus that **NU5** has activated on the **Zcash** *Mainnet* and *Testnet* with the *activation block* hashes given in **§? '??'** on p. **??**.

A *full validator* **MAY** impose a limit on the number of *blocks* it will "roll back" when switching from one *best valid block chain* to another that is not a descendent. For zcashd and zebra this limit is 100 *blocks*.

3.4 Transactions and Treestates

#transactions

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

Each *transaction* has a *transaction ID*. *Transaction IDs* are used to refer to *transactions* in `tx_out` fields, in *leaf nodes* of a *block*'s *transaction* tree rooted at `hashMerkleRoot`, and in other parts of the ecosystem; for example they are shown in *block chain* explorers and can be used in higher-level protocols. The computation of *transaction IDs* is described in **§? '??'** on p. **??**.

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

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

[4] There is reason to follow the requirements in this section also for non-full validators, but those are outside the scope of this protocol specification.

To each *transaction* there are associated initial *treestates* for **Sprout**. *treestate* consists of:

- a *note commitment tree* (**§? '??'** on p. **??**);
- a *nullifier set* (**§? '??'** on p. **??**).

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

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

In a given *block chain*, *treestates* are chained as follows:

- The input *treestate* of the first *block* is the empty *treestate*.
- The input *treestate* of the first *transaction* of a *block* is the final *treestate* of the immediately preceding *block*.
- The input *treestate* of each subsequent *transaction* in a *block* is the output *treestate* of the immediately preceding *transaction*.
- The final *treestate* of a *block* is the output *treestate* of its last *transaction*.

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

3.5 JoinSplit Transfers and Descriptions

#joinsplit

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

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

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

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

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

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

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

The input and output values of each *JoinSplit transfer* **MUST** balance exactly. This is not a consensus rule since it cannot be checked directly; it is enforced by the **Balance** rule of the *JoinSplit statement*.

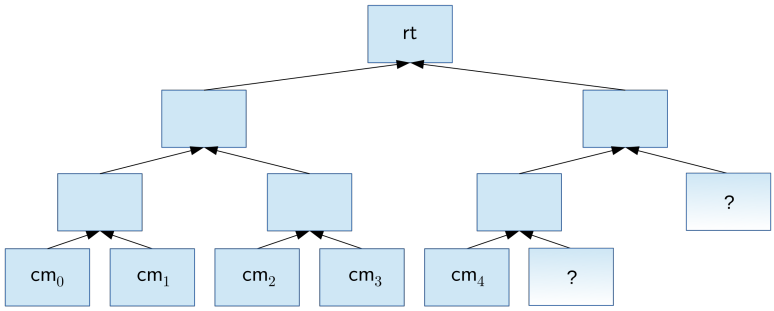
Consensus rules:

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

3.6 Note Commitment Trees

#merkletree

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.



A *note commitment tree* is an *incremental Merkle tree* of fixed depth used to store *note commitments* that *JoinSplit transfers* produce. Just as the *UTXO (unspent transaction output) set* used in **Bitcoin**, it is used to express the existence of value and the capability to spend it. However, unlike the *UTXO set*, it is ***not*** the job of this tree to protect against double-spending, as it is append-only.

A *root* of a *note commitment tree* is associated with each *treestate* (**§? '??'** on p. **??**).

Each *node* in the *incremental Merkle tree* is associated with a *hash value* of size $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$ bits. The *layer* numbered h, counting from *layer* 0 at the *root*, has 2^h *nodes* with *indices* 0 to $2^h - 1$ inclusive. The *hash value* associated with the *node* at *index* i in *layer* h is denoted $\mathsf{M}^{\mathsf{h}}_i$.

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

Consensus rules:

- A *block* **MUST NOT** add **Sprout** *note commitments* that would result in the **Sprout** *note commitment tree* exceeding its capacity of $2^{\mathsf{MerkleDepth}^{\mathsf{Sprout}}}$ *leaf nodes*.

3.7 Nullifier Sets

#nullifierset

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

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

3.8 Block Subsidy and Founders' Reward

#subsidyconcepts

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

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

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

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

The calculations are described in **§? '??'** on p. **??**.

3.9 Coinbase Transactions

#coinbasetransactions

A *transaction* that has a single *transparent input* with a null `prevout` field, is called a *coinbase transaction*. Every *block* has a single *coinbase transaction* as the first *transaction* in the *block*. The purpose of this *coinbase transaction* is to collect and spend any *miner subsidy*, and *transaction fees* paid by other *transactions* included in the *block*.

As described in **§? '??'** on p. **??**, the *coinbase transaction* **MUST** also pay the *Founders' Reward*.

3.10 Mainnet and Testnet

#networks

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

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

Mainnet genesis block: `00040fe8ec8471911baa1db1266ea15dd06b4a8a5c453883c000b031973dce08`

Mainnet **NU5** *activation block*: `0000000000d723156d9b65ffcf4984da7a19675ed7e2f06d9e5d5188af087bf8`

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

Testnet genesis block: `05a60a92d99d85997cce3b87616c089f6124d7342af37106edc76126334a2c38`

Testnet **NU5** *activation block*: `0006d75c60b3093d1b671ff7da11c99ea535df9927c02e6ed9eb898605eb7381`

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

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

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

4 Abstract Protocol

#abstractprotocol

> *'We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of [their] powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worth-while to point out that the purpose of abstracting is* **not** *to be vague, but to create a new semantic level in which one can be absolutely precise.'*
>
> — Edsger Dijkstra, "The Humble Programmer" [**EWD-340**]

Abstraction is an incredibly important idea in the design of any complex system. Without abstraction, we would not be able to design anything as ambitious as a computer, or a cryptographic protocol. Were we to attempt it, the computer would be hopelessly unreliable or the protocol would be insecure, if they could be completed at all.

The aim of abstraction is primarily to limit how much a human working on a piece of a system has to keep in mind at one time, in order to apprehend the connections of that piece to the remainder. The work could be to extend or maintain the system, to understand its security or other properties, or to explain it to others.

In this specification, we make use wherever possible of abstractions that have been developed by the cryptography community to model cryptographic primitives: *Pseudo Random Functions*, *commitment schemes*, *signature schemes*, etc. Each abstract primitive has associated syntax (its interface as used by the rest of the system) and security properties, as documented in this part. Their instantiations are documented in part **§? '??'** on p. **??**.

In some cases this syntax or these security requirements have been extended to meet the needs of the **Zcash** protocol. For example, some of the PRFs used in **Zcash** need to be *collision-resistant*, which is not part of the usual security requirement for a PRF; some *signature schemes* need to support additional functionality and security properties; and so on. Also, security requirements are sometimes intentionally stronger than what is known to be needed, because the stronger property is simpler or less error-prone to work with, and/or because it has been studied in the cryptographic literature in more depth.

We explicitly ***do not claim***, however, that all of these instantiations satisfying their documented syntax and security requirements would be sufficient for security or correctness of the overall **Zcash** protocol, or that it is always necessary. The claim is only that it helps to understand the protocol; that is, that analysis or extension is simplified by making use of the abstraction. In other words, ***a good way to understand*** the use of that primitive in the protocol is to model it as an instance of the given abstraction. And furthermore, if the instantiated primitive does not in fact satisfy the requirements of the abstraction, then this is an error that should be corrected –whether or not it leads to a vulnerability– since that would compromise the facility to understand its use in terms of the abstraction.

In this respect the abstractions play a similar rôle to that of a type system (which we also use): they add a form of redundancy to the specification that helps to express the intent.

Each property is a claim that may be incorrect (or that may be insufficiently precisely stated to determine whether it is correct). An example of an incorrect security claim occurs in the **Zerocash** protocol [**BCGGMTV2014**]: the instantiation of the *note commitment* scheme used in **Zerocash** failed to be *binding* at the intended security level (see **§? '??'** on p. **??**).

Another hazard that we should be aware of is that abstractions can be "leaky": an instantiation may impose conditions on its correct or secure use that are not captured by the abstraction's interface and semantics. Ideally, the abstraction would be changed to explicitly document these conditions, or the protocol changed to rely only on the original abstraction.

An abstraction can also be incomplete (not quite the same thing as being leaky): it intentionally –usually for simplicity– does not model an aspect of behaviour that is important to security or correctness. An example would be resistance to side-channel attacks; this specification says little about side-channel defence, among many other implementation concerns.

4.1 Abstract Cryptographic Schemes

#abstractschemes

4.1.1 Hash Functions

#abstracthashes

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

The following *hash functions* are used in **§? '??'** on p. **??**:

$$\mathsf{MerkleCRH}^{\mathsf{Sprout}} : \{0 \,..\, \mathsf{MerkleDepth}^{\mathsf{Sprout}} - 1\} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$$
$$\mathsf{MerkleCRH}^{\mathsf{Sapling}} : \{0 \,..\, \mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1\} \times \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]} \to \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}.$$

$\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ is *collision-resistant* except on its first argument.

These functions are instantiated in **§? '??'** on p. **??**.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a *collision-resistant hash function* used in **§? '??'** on p. **??**. It is instantiated in **§? '??'** on p. **??**.

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

4.1.2 Pseudo Random Functions

#abstractprfs

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

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

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

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

These are used in **§? '??'** on p. **??**; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in **§? '??'** on p. **??**.

$\mathsf{PRF}^{\mathsf{expand}}$ is used in the following places:

-
-
- in [**ZIP-32**], with first byte in {0x80};
- in [**ZIP-316**], with first byte 0xD0.

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

Security requirements:

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

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

4.1.3 Symmetric Encryption

#abstractsym

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

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

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

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

4.1.4 Key Agreement

#abstractkeyagreement

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

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

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

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

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

Let $\mathsf{KA.Base} : \mathsf{KA.Public}$ be a public base point.

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

Security requirements:

- KA.FormatPrivate must preserve sufficient entropy from its input to be used as a secure KA *private key.*
- The key agreement and the KDF defined in the next section must together satisfy a suitable adaptive security assumption along the lines of [**Bernstein2006**] or [**ABR1999**].

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

4.1.5 Key Derivation

#abstractkdf

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.

$\mathsf{KDF}^{\mathsf{Sprout}}$ takes as input an output index in $\{1..\mathrm{N}^{\mathsf{new}}\}$, the value $\mathsf{h}_{\mathsf{Sig}}$, the shared Diffie–Hellman secret sharedSecret, the *ephemeral public key* epk, and the recipient's public *transmission key* $\mathsf{pk}_{\mathsf{enc}}$. It is suitable for use with $\mathsf{KA}^{\mathsf{Sprout}}$ and derives keys for Sym.Encrypt.

$$\mathsf{KDF}^{\mathsf{Sprout}} : \{1..\mathrm{N}^{\mathsf{new}}\} \times \mathbb{B}^{[\ell_{\mathsf{hSig}}]} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{SharedSecret} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public} \times \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public} \to \mathsf{Sym}.\mathbf{K}$$

Security requirements:

- The asymmetric encryption scheme in **§? '??'** on p. **??**, constructed from $\mathsf{KA}^{\mathsf{Sprout}}$, $\mathsf{KDF}^{\mathsf{Sprout}}$ and Sym, is required to be IND-CCA2-secure and *key-private*.
-

Key privacy is defined in [**BBDP2001**].

4.1.6 Signature

#abstractsig

A *signature scheme* Sig defines:

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

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

Zcash uses *signature schemes*:

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

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

The following security property is needed for JoinSplitSig.

Security requirement: JoinSplitSig must be Strongly Unforgeable under (non-adaptive) Chosen Message Attack (SU-CMA), as defined for example in [**BDEHR2011**].[5] This allows an adversary to obtain signatures on chosen messages, and then requires it to be infeasible for the adversary to forge a previously unseen valid (message, signature) pair without access to the *signing key*.

Non-normative notes:

- We need separate *signing key* generation and *validating key* derivation algorithms, rather than the more conventional combined key pair generation algorithm $\mathsf{Sig.Gen} : () \xrightarrow{R} \mathsf{Sig.Private} \times \mathsf{Sig.Public}$, to support the key derivation in **§? '??'** on p. **??**.

  The definitions of schemes with additional features in **§? '??'** on p. **??** and in **§? '??'** on p. **??** also become simpler.
- 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 **§? '??'** on p. **??**), 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.

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

- SU-CMA security requires it to be infeasible for the adversary, not knowing the *private key*, to forge a distinct signature on a previously seen message. That is, *JoinSplit signatures* are intended to be *nonmalleable* in the sense of [**BIP-62**].
- The terminology used in this specification is that we "validate" signatures, and "verify" *zk-SNARK proofs*.

4.1.7 Commitment

#abstractcommit

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

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

A *commitment scheme* COMM defines a type of inputs COMM.Input, a type of commitments COMM.Output, a type of *commitment trapdoors* COMM.Trapdoor, and a *trapdoor* generator $\mathsf{COMM.GenTrapdoor} : () \xrightarrow{\mathrm{R}} \mathsf{COMM.Trapdoor}$.

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

Security requirements:

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

Notes:

- COMM.GenTrapdoor need not produce the uniform distribution on COMM.Trapdoor. In that case, it is incorrect to choose a *trapdoor* from the latter distribution.
- If it were only feasible to find $x :$ COMM.Input and $r, r' :$ COMM.Trapdoor such that $r \neq r'$ and $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(x)$, this would not contradict the computational *binding* security requirement.

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

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

Sprout uses a *note commitment scheme*

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

instantiated in **§? '??'** on p. **??**.

4.1.8 Represented Group

#abstractgroup

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

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

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

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

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

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

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

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

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

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

4.1.9 Represented Pairing

#abstractpairing

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

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

4.1.10 Zero-Knowledge Proving System

#abstractzk

A *zero-knowledge proving system* is a cryptographic protocol that allows proving a particular *statement*, dependent on *primary* and *auxiliary inputs*, in zero knowledge — that is, without revealing information about the *auxiliary inputs* other than that implied by the *statement*. The type of *zero-knowledge proving system* needed by **Zcash** is a *preprocessing zk-SNARK* [**BCCGLRT2014**].

A *preprocessing zk-SNARK* instance ZK defines:

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

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

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 $\mathcal{E}_{\mathcal{A}}$ such that if $\mathcal{E}_{\mathcal{A}}(\mathsf{vk}, \mathsf{pk})$ 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:

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

These definitions are derived from those in [**BCTV2014b**], 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.

Non-normative notes:

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

4.2 Key Components

#keycomponents

4.2.1 Sprout Key Components

#sproutkeycomponents

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

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

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

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

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

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

4.3 JoinSplit Descriptions

#joinsplitdesc

A *JoinSplit transfer*, as specified in **§? '??'** on p. **??**, 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 $\mathsf{JoinSplitSig}$ public *validating key* and signature.

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

Let $\mathsf{hSigCRH}$ be as defined in **§? '??'** on p. **??**.

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

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

Let Sym be as defined in **§? '??'** on p. **??**.

Let $\mathsf{ZKJoinSplit}$ be as defined in **§? '??'** on p. **??**.

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

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

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

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

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

Consensus rules:

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

4.4 Sending Notes

#send

4.4.1 Sending Notes (Sprout)

#sproutsend

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

Let JoinSplitSig be as specified in ***§? '??'*** on p. **??**.

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

Let ℓ_{Seed} and $\ell^{\mathsf{Sprout}}_{\varphi}$ be as specified in ***§? '??'*** on p. **??**.

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

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

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

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

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

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

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

After generating all of the *JoinSplit descriptions*, the sender obtains $\mathsf{dataToBeSigned} : \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ as described in ***§? '??'*** on p. **??**, and signs it with the private *JoinSplit signing key*:

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

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

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

4.5 Dummy Notes

#dummynotes

4.5.1 Dummy Notes (Sprout)

#sproutdummynotes

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

Let $\ell_{\mathsf{a_{sk}}}$ and $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$ be as defined in **§? *'??'*** on p. **??**.

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ be as defined in **§? *'??'*** on p. **??**.

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

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

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

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

4.6 Merkle Path Validity

#merklepath

Let MerkleDepth be $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ for the **Sprout** *note commitment tree*. These constants are defined in **§? *'??'*** on p. **??**.

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

The following discussion applies independently to the **Sprout** *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 $\mathsf{M}^{\mathsf{h}}_i$ 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* $\mathsf{Uncommitted}^{\mathsf{Sprout}}$. It is assumed to be infeasible to find a preimage *note* $\mathbf{n}$ such that $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}) = \mathsf{Uncommitted}^{\mathsf{Sprout}}$.

The *nodes* at *layers* 0 to $\mathsf{MerkleDepth} - 1$ 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 $0 \leq i < 2^h$,

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

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

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

where

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

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

4.7 SIGHASH Transaction Hashing

#sighash

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

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

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

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

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

4.8 Non-malleability (Sprout)

#sproutnonmalleability

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

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

JoinSplitSig is instantiated in **§? '??'** on p. **??**.

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

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

Let $\mathsf{PRF}^{\mathsf{pk}}$ be as defined in **§? '??'** on p. **??**.

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

4.9 Balance (Sprout)

#joinsplitbalance

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

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

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

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

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

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

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

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

4.10 Note Commitments and Nullifiers

#commitmentsandnullifiers

A *transaction* that contains one or more *JoinSplit descriptions*, when entered into the *block chain*, appends to the *note commitment tree* with all constituent *note commitments*.

All of the constituent *nullifiers* are also entered into the *nullifier set* of the associated *treestate*. A *transaction* is not valid if it would have added a *nullifier* to the *nullifier set* that already exists in the set (see **§? '??'** on p. **??**).

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

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ be as instantiated in **§? '??'** on p. **??**.

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

Security requirement: For each shielded protocol, the requirements on *nullifier* derivation are as follows:

- The derived *nullifier* must be determined completely by the fields of the *note*, in a way that can be checked in the corresponding statement that controls spends (i.e. the *JoinSplit statement*).
- Under the assumption that ρ values are unique, it must not be possible to generate two *notes* with distinct *note commitments* but the same *nullifier*. (See **§? '??'** on p. **??** for further discussion.)

- Given a set of *nullifiers* of ***a priori*** unknown *notes*, they must not be linkable to those *notes* with probability greater than expected by chance, even to an adversary with the corresponding *incoming viewing keys* (but not *full viewing keys*), and even if the adversary may have created the *notes*.

4.11 Zk-SNARK Statements

#snarkstatements

4.11.1 JoinSplit Statement (Sprout)

#joinsplitstatement

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, ℓ_{value}, $\ell_{\mathsf{a_{sk}}}$, $\ell^{\mathsf{Sprout}}_{\varphi}$, ℓ_{hSig}, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$ be as defined in **§? '??'** on p. **??**.
Let $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ} be as defined in **§? '??'** on p. **??**.
Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**, and let $\mathsf{Note}^{\mathsf{Sprout}}$ and $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

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

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

the prover knows an *auxiliary input*:

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

where:

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

such that the following conditions hold:

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

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

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

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

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

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

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

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

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

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

4.12 In-band secret distribution (Sprout) #sproutinband

In **Sprout**, the secrets that need to be transmitted to a recipient of funds in order for them to later spend, are v, ρ, and rcm. A *memo field* (**§? '??'** on p. **??**) is also transmitted.

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

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

For both encryption and decryption,

- let Sym be the scheme instantiated in **§? '??'** on p. **??**;
- let $\mathsf{KDF}^{\mathsf{Sprout}}$ be the *Key Derivation Function* instantiated in **§? '??'** on p. **??**;
- let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in **§? '??'** on p. **??**;
- let $\mathsf{h_{Sig}}$ be the value computed for this *JoinSplit description* in **§? '??'** on p. **??**.

4.12.1 Encryption (Sprout) #sproutencrypt

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in **§? '??'** on p. **??**.

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

Let $\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ be **Sprout** *note plaintexts* defined in **§? '??'** on p. **??**.

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 $\mathsf{P}_i^{\mathsf{enc}}$ be the *raw encoding* of $\mathbf{np}_i$.
  - Let $\mathsf{sharedSecret}_i = \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Agree}(\mathsf{esk}, \mathsf{pk}_{\mathsf{enc},i})$.
  - Let $\mathsf{K}_i^{\mathsf{enc}} = \mathsf{KDF}^{\mathsf{Sprout}}(i, \mathsf{h_{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk}_{\mathsf{enc},i})$.
  - Let $\mathsf{C}_i^{\mathsf{enc}} = \mathsf{Sym}.\mathsf{Encrypt}_{\mathsf{K}_i^{\mathsf{enc}}}(\mathsf{P}_i^{\mathsf{enc}})$.

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

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 **Sprout** *note* received out-of-band, which are not addressed in this document.

4.12.2 Decryption (Sprout) #sproutdecrypt

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 **§? '??'** on p. **??**.

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

Then for each $i \in \{1..\mathrm{N}^{\mathsf{new}}\}$, the recipient will attempt to decrypt that ciphertext component $(\mathsf{epk}, \mathsf{C}_i^{\mathsf{enc}})$ 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 $\mathtt{DecryptNoteSprout}(\mathsf{K}_i^{\mathsf{enc}}, \mathsf{C}_i^{\mathsf{enc}}, \mathsf{cm}_i, \mathsf{a_{pk}})$.

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

$\quad \text{let } \mathsf{P}_i^{\mathsf{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K}_i^{\mathsf{enc}}}(\mathsf{C}_i^{\mathsf{enc}})$
$\quad \text{if } \mathsf{P}_i^{\mathsf{enc}} = \bot, \text{ return } \bot$
$\quad \text{extract } \mathbf{np}_i = (\mathsf{leadByte}_i : \mathbb{B}^{\mathbb{Y}}, \mathsf{v}_i : \{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\}, \rho_i : \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]}, \mathsf{rcm}_i : \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}, \mathsf{memo}_i : \mathbb{B}^{\mathbb{Y}[512]})$
$\quad \text{from } \mathsf{P}_i^{\mathsf{enc}}$
$\quad \text{let } \mathbf{n}_i = (\mathsf{a}_{\mathsf{pk}}, \mathsf{v}_i, \rho_i, \mathsf{rcm}_i)$
$\quad \text{if } \mathsf{leadByte}_i \neq \texttt{0x00} \text{ or } \mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}_i) \neq \mathsf{cm}_i, \text{ return } \bot$
$\quad \text{return } (\mathbf{n}_i, \mathsf{memo}_i).$

Notes:

- The decryption algorithm corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in [**BCGGMTV2014**].
- To test whether a *note* is unspent in a particular *block chain* also requires the *spending key* a_{sk}; the coin is unspent if and only if $\mathsf{nf} = \mathsf{PRF}_{\mathsf{a}_{\mathsf{sk}}}^{\mathsf{nfSprout}}(\rho)$ is not in the *nullifier set* for that *block chain*.
- A *note* can change from being unspent to spent as a node's view of the *best valid block chain* is extended by new *transactions*. Also, *block chain reorganizations* can cause a node to switch to a different *best valid block chain* that does not contain the *transaction* in which a *note* was output.

See **§? '??'** on p. **??** for further discussion of the security and engineering rationale behind this encryption scheme.

4.13 Block Chain Scanning (Sprout) #sproutscan

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

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

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

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

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

let mutable $\mathsf{ReceivedSet} : \mathscr{P}\big(\mathsf{Note}^{\mathsf{Sprout}} \times \mathbb{B}^{\mathbb{Y}^{[512]}}\big) \leftarrow \{\}$
let mutable $\mathsf{SpentSet} : \mathscr{P}\big(\mathsf{Note}^{\mathsf{Sprout}}\big) \leftarrow \{\}$
let mutable $\mathsf{NullifierMap} : \mathbb{B}^{[\ell_{\mathsf{PRF}}^{\mathsf{Sprout}}]} \to \mathsf{Note}^{\mathsf{Sprout}} \leftarrow$ the empty mapping
for each *transaction* tx:
  for each *JoinSplit description* in tx:
    let $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$ be the *transmitted notes ciphertext* of the *JoinSplit description*
    for i in $1..\mathrm{N}^{\mathsf{new}}$:
      Attempt to decrypt the *transmitted notes ciphertext* component $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}_i)$ using ivk with the algorithm in **§? '??'** on p. **??**. If this succeeds with $(\mathbf{n}, \mathsf{memo})$:
        Add $(\mathbf{n}, \mathsf{memo})$ to ReceivedSet.
        Calculate the nullifier nf of $\mathbf{n}$ using $\mathsf{a_{sk}}$ as described in **§? '??'** on p. **??**.
        Add the mapping $\mathsf{nf} \to \mathbf{n}$ to NullifierMap.
    let $\mathsf{nf}_{1..\mathrm{N}^{\mathsf{old}}}$ be the *nullifiers* of the *JoinSplit description*
    for i in $1..\mathrm{N}^{\mathsf{old}}$:
      if nf_i is present in NullifierMap, add $\mathsf{NullifierMap}(\mathsf{nf}_i)$ to SpentSet
return (ReceivedSet, SpentSet).

5 Concrete Protocol #concreteprotocol

5.1 Integers, Bit Sequences, and Endianness #endian

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;
- $\mathsf{I2LEOSP} : (\ell : \mathbb{N}) \times \{0 .. 2^\ell - 1\} \to \mathbb{B}^{\mathbb{Y}^{[\mathsf{ceiling}(\ell/8)]}}$, such that $\mathsf{I2LEBSP}_\ell(x)$ is the sequence of $\mathsf{ceiling}(\ell/8)$ bytes 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.
- $\mathsf{LEBS2IP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \{0 .. 2^\ell - 1\}$ such that $\mathsf{LEBS2IP}_\ell(S)$ is the integer represented in little-endian order by the bit sequence S of length ℓ.

- $\mathsf{LEOS2IP} : (\ell : \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{B}^{\mathbb{Y}[\ell/8]} \to \{0 \,..\, 2^\ell - 1\}$ such that $\mathsf{LEOS2IP}_\ell(S)$ is the integer represented in little-endian order by the byte sequence S of length $\ell/8$.
- $\mathsf{LEBS2OSP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{B}^{\mathbb{Y}[\mathsf{ceiling}(\ell/8)]}$ 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.
- $\mathsf{LEOS2BSP} : (\ell : \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{B}^{\mathbb{Y}[\mathsf{ceiling}(\ell/8)]} \to \mathbb{B}^{[\ell]}$ defined as follows: convert each byte to a group of 8 bits with the ***least*** significant bit first, and concatenate the resulting groups in the same order as the bytes.

5.2 Bit layout diagrams

#bitlayout

We sometimes use bit layout diagrams, in which 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

#constants

Define:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Concrete Cryptographic Schemes

#concreteschemes

5.4.1 Hash Functions

#concretehashes

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

#concretesha

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

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

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

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

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

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

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

That is, the input to SHA256Compress is what [**NIST2015**] 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 MerkleCRH$^{\text{Sprout}}$.

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

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

Ed25519 uses SHA-512:

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

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

5.4.1.2 BLAKE2 Hash Functions

#concreteblake2

BLAKE2 is defined by [**ANWW2013**].

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

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

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

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

5.4.1.3 Merkle Tree Hash Function

#merklecrh

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

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

#sproutmerklecrh

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

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

SHA256Compress is defined in ***§? '??'*** on p. **??**.

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

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.

5.4.1.4 $\mathsf{h}_{\mathsf{Sig}}$ Hash Function

#hsigcrh

hSigCRH is used to compute the value $\mathsf{h}_{\mathsf{Sig}}$ in ***§? '??'*** on p. **??**.

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

where

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

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

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

5.4.1.5 Equihash Generator

#equihashgen

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

Let powtag := $\boxed{64\text{-bit } \textbf{“ZcashPoW”} \mid 32\text{-bit } n \mid 32\text{-bit } k}$.

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

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

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

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

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

Indices of bits in T are 1-based.

BLAKE2b-$\ell(p, x)$ is defined in **§? '??'** on p. **??**.

Security requirement: BLAKE2b-$\ell(\mathsf{powtag}, x)$ 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 (**§? '??'** on p. **??**), 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

#concreteprfs

Let SHA256Compress be as given in **§? '??'** on p. **??**.

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

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

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

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

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

Security requirements:

- SHA256Compress must be *collision-resistant*.
- SHA256Compress must be a *PRF* when keyed by the bits corresponding to x, $\mathsf{a_{sk}}$ or φ in the above diagrams, with input in the remaining bits.

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

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

5.4.3 Symmetric Encryption

#concretesym

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

Let the *authenticated one-time symmetric encryption scheme* $\mathsf{Sym.Encrypt}_{\mathsf{K}}(\mathsf{P})$ 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 $\perp$ indicating failure to decrypt.

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

5.4.4 Key Agreement And Derivation

#concretekaandkdf

5.4.4.1 Sprout Key Agreement

#concretesproutkeyagreement

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

It is instantiated as Curve25519 key agreement, described in [**Bernstein2006**], 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. $\mathbb{B}^{\mathbb{Y}^{[32]}}$), 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 [**Bernstein2006**].

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

Let $\mathsf{clamp}_{\mathsf{Curve25519}}(\underline{x})$ take a 32-byte sequence $\underline{x}$ as input and return a byte sequence representing a Curve25519 *private key*, with bits "clamped" as described in [**Bernstein2006**]: "clear bits $0, 1, 2$ of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte." Here the bits of a byte are numbered such that bit b has numeric weight 2^b.

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

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

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

5.4.4.2 Sprout Key Derivation

#concretesproutkdf

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

It is instantiated using BLAKE2b-256 as follows:

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

where:

$\mathsf{kdftag} :=$

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

$\mathsf{kdfinput} :=$

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

.

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

5.4.5 Ed25519

#concreteed25519

Ed25519 is a *signature scheme* as specified in **§? '??'** on p. **??**. It is used to instantiate JoinSplitSig as described in **§? '??'** on p. **??**.

Let PreCanopyExcludedPointEncodings $: \mathscr{P}\big(\mathbb{B}^{\mathbb{Y}^{[32]}}\big) = \{$

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

$\}$.

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

Let $a = -1$.

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

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

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

Define the notation $\sqrt[?]{\bullet}$ as in **§? '??'** on p. **??**.

Define I2LEOSP, LEOS2BSP, and LEBS2IP as in **§? '??'** on p. **??**.

Define $\mathsf{reprBytes}_{\mathsf{Ed25519}} : \mathsf{Ed25519} \to \mathbb{B}^{\mathbb{Y}^{[32]}}$ such that $\mathsf{reprBytes}_{\mathsf{Ed25519}}((x, y)) = \mathsf{I2LEOSP}_{256}\big((y \bmod p) + 2^{255} \cdot \tilde{x}\big)$, where $\tilde{x} = x \bmod 2$.[6]

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

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

Note: This definition of point decoding differs from that of [**RFC-8032**]. In the latter there is an additional step "`If x = 0, and x_0 = 1, decoding fails.`", which rejects the encodings $\{$

[0x01, 0x00, 0x80],
[0xee, 0xff],
[0xec, 0xff]

$\}$.
In this specification, the first two of these are accepted as encodings of $(0, 1)$, and the third is accepted as an encoding of $(0, -1)$.

Ed25519 is defined as in [**BDLSY2012**], using SHA-512 as the internal *hash function*, with the additional requirements below. A valid Ed25519 *validating key* is defined as a sequence of 32 bytes encoding a point on the Ed25519 curve.

[6] Here we use the (x, y) naming of coordinates in [**BDLSY2012**], which is different from the (u, v) naming used for coordinates of *ctEdwards curves* in **§? '??'** on p. **??** and in **§? '??'** on p. **??**.

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

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

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

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

The encoding of an Ed25519 signature is:

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

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

Notes:

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

Non-normative note: The exclusion of PreCanopyExcludedPointEncodings from $\underline{R}$ is due to a quirk of version 1.0.15 of the libsodium library [**libsodium**] which was initially used to implement Ed25519 signature validation in zcashd. (The `ED25519_COMPAT` compile-time option was not set.) The intent was to exclude points of order less than ℓ; however, not all such points were covered.

5.4.6 Commitment schemes

#concretecommit

5.4.6.1 Sprout Note Commitments

#concretesproutnotecommit

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

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

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

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

Security requirements:

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

5.4.7 Represented Groups and Pairings

#concretepairing

5.4.7.1 BN-254

#bnpairing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Define I2BEBSP $: (\ell : \mathbb{N}) \times \{0 .. 2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in **§? '??'** on p. **??**.

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

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

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

.

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

- Define FE2IP $: \mathbb{F}_{q_{\mathbb{G}}}[t]/(t^2 + 1) \to \{0 .. {q_{\mathbb{G}}}^2 - 1\}$ 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:

- Only the $r_{\mathbb{G}}$-order subgroups $\mathbb{G}_{2,T}^{(r)}$ are used in the protocol, not their containing groups $\mathbb{G}_{2,T}$. Points in $\mathbb{G}_2^{(r)*}$ are ***always*** checked to be of order $r_{\mathbb{G}}$ when decoding from external representation. (The group of rational points $\mathbb{G}_1$ on $E_{\mathbb{G}_1}/\mathbb{F}_{q_{\mathbb{G}}}$ is of order $r_{\mathbb{G}}$ so no subgroup checks are needed in that case, and elements of $\mathbb{G}_T^{(r)}$ are never represented externally.) The (r) superscripts on $\mathbb{G}_{1,2,T}^{(r)}$ are used for consistency with notation elsewhere in this specification.
- The points at infinity $\mathcal{O}_{\mathbb{G}_{1,2}}$ never occur in proofs and have no defined encodings in this protocol.
- A rational point $P \neq \mathcal{O}_{\mathbb{G}_2}$ on the curve $E_{\mathbb{G}_2}$ can be verified to be of order $r_{\mathbb{G}}$, and therefore in $\mathbb{G}_2^{(r)*}$, by checking that $r_{\mathbb{G}} \cdot P = \mathcal{O}_{\mathbb{G}_2}$.
- The use of big-endian order by I2BEBSP is different from the encoding of most other integers in this protocol. The encodings for $\mathbb{G}_{1,2}^{(r)*}$ are consistent with the definition of EC2OSP for compressed curve points in [**IEEE2004**]. The LSB compressed form (i.e. EC2OSP-XL) is used for points in $\mathbb{G}_1^{(r)*}$, and the SORT compressed form (i.e. EC2OSP-XS) for points in $\mathbb{G}_2^{(r)*}$.
- Testing $y > y'$ for the compression of $\mathbb{G}_2^{(r)*}$ points is equivalent to testing whether $(a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})$ in lexicographic order.
- Algorithms for decompressing points from the above encodings are given in [**IEEE2000**] for $\mathbb{G}_1^{(r)*}$, and [**IEEE2004**] for $\mathbb{G}_2^{(r)*}$.

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

5.4.8 Zero-Knowledge Proving Systems

#concretezk

5.4.8.1 BCTV14

#bctv

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

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

Note: Many details of the *proving system* are beyond the scope of this protocol document. For example, the *quadratic constraint program* verifying the *JoinSplit statement*, or its translation to a *Quadratic Arithmetic Program* [**BCTV2014a**], are not specified in this document. In 2015, Bryan Parno found a bug in this translation, which is corrected by the *libsnark* implementation[7] [**WCBTV2015**] [**Parno2015**] [**BCTV2014a**]. In practice it will be necessary to use the specific proving and verifying keys that were generated for the **Zcash** production *block chain*, given in **§? '??'** on p. **??**, together with a *proving system* implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

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

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

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

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

Encoding of BCTV14 **Proofs** #bctvencoding

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

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

The resulting proof size is 296 bytes.

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

5.5 Encodings of Note Plaintexts and Memo Fields #notept

As explained in **§? '??'** on p. **??**, 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:

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

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

The usage of the *memo field* is by agreement between the sender and recipient of the *note*. Non-consensus constraints on the *memo field* contents are specified in [**ZIP-302**].

Other fields are as defined in **§? '??'** on p. **??**.

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

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

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

5.6 Encodings of Addresses and Keys

#addressandkeyencoding

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*. For **Sprout** *shielded payment addresses*, this byte sequence can then be further encoded using *Base58Check*. The *Base58Check* layer is the same as for upstream **Bitcoin** addresses [**Bitcoin-Base58**].

5.6.1 Transparent Encodings

#transparentencodings

5.6.1.1 Transparent Addresses

#transparentaddrencoding

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

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

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

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

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

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

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

Notes:

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

5.6.1.2 Transparent Private Keys

#transparentkeyencoding

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

5.6.2 Sprout Encodings

#sproutencodings

5.6.2.1 Sprout Payment Addresses

#sproutpaymentaddrencoding

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

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

a_{pk} is a SHA256Compress output. $\mathsf{pk}_{\mathsf{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ key, for use with the encryption scheme defined in **§? '??'** on p. **??**. These components are derived from a *spending key* as described in **§? '??'** on p. **??**.

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

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

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

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

5.6.2.2 Sprout Incoming Viewing Keys

#sproutinviewingkeyencoding

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in **§? '??'** on p. **??**.

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

a_{pk} is a SHA256Compress output. $\mathsf{sk}_{\mathsf{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$ key, for use with the encryption scheme defined in **§? '??'** on p. **??**. These components are derived from a *spending key* as described in **§? '??'** on p. **??**.

The *raw encoding* of a **Sprout** *incoming viewing key* consists of:

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

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

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

$\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{FormatPrivate}$ is defined in **§? '??'** on p. **??**.

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

5.6.2.3 Sprout Spending Keys

#sproutspendingkeyencoding

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

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

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

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

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

Notes:

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

5.7 BCTV14 zk-SNARK Parameters

#bctvparameters

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

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

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

6 Network Upgrades

#networkupgrades

Zcash launched with a protocol revision that we call **Sprout**.

This section summarizes the strategy for upgrading from **Sprout** to subsequent versions of the protocol (**Overwinter**, **Sapling**, **Blossom**, **Heartwood**, **Canopy**, and **NU5**), and for future upgrades.

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

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

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

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

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

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

7 Consensus Changes from Bitcoin

#consensusfrombitcoin

7.1 Transaction Encoding and Consensus

#txnencoding

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

Version*	Bytes	Name	Data Type	Description
1 .. 4	4	`header`	`uint32`	Contains: · `fOverwintered` flag (bit 31) · `version` (bits 30 .. 0) – *transaction version*.
3 .. 4	4	`nVersionGroupId`	`uint32`	Version group ID (nonzero).
1 .. 4	*Varies*	`tx_in_count`	`compactSize`	Number of *transparent inputs*.
1 .. 4	*Varies*	`tx_in`	`tx_in`	*Transparent* inputs, encoded as in **Bitcoin**.
1 .. 4	*Varies*	`tx_out_count`	`compactSize`	Number of *transparent outputs*.
1 .. 4	*Varies*	`tx_out`	`tx_out`	*Transparent* outputs, encoded as in **Bitcoin**.
1 .. 4	4	`lock_time`	`uint32`	Unix-epoch UTC time or *block height*, encoded as in **Bitcoin**.
3 .. 4	4	`nExpiryHeight`	`uint32`	A *block height* after which the *transaction* will expire, or 0 to disable expiry. [**ZIP-203**]
4	8	`valueBalanceSapling`	`int64`	The net value of **Sapling** spends minus outputs.
4	*Varies*	`nSpendsSapling`	`compactSize`	The number of *Spend descriptions* in `vSpendsSapling`.
4	384· `nSpendsSapling`	`vSpendsSapling`	`SpendDescriptionV4 [nSpendsSapling]`	A sequence of *Spend descriptions*, encoded per **§? '??'** on p. **??**.
4	*Varies*	`nOutputsSapling`	`compactSize`	The number of *Output descriptions* in `vOutputsSapling`.
4	948· `nOutputsSapling`	`vOutputsSapling`	`OutputDescriptionV4 [nOutputsSapling]`	A sequence of *Output descriptions*, encoded per **§? '??'** on p. **??**.
2 .. 4	*Varies*	`nJoinSplit`	`compactSize`	The number of *JoinSplit descriptions* in `vJoinSplit`.
2 .. 3	1802· `nJoinSplit`	`vJoinSplit`	`JSDescriptionBCTV14 [nJoinSplit]`	A sequence of *JoinSplit descriptions* using BCTV14 proofs, encoded per **§? '??'** on p. **??**.
4	1698· `nJoinSplit`	`vJoinSplit`	`JSDescriptionGroth16 [nJoinSplit]`	A sequence of *JoinSplit descriptions* using Groth16 proofs, encoded per **§? '??'** on p. **??**.
2 .. 4 †	32	`joinSplitPubKey`	`byte[32]`	An encoding of a JoinSplitSig public *validating key*.
2 .. 4 †	64	`joinSplitSig`	`byte[64]`	A signature on a prefix of the *transaction* encoding, validated using `joinSplitPubKey` as specified in **§? '??'** on p. **??**.
4 ‡	64	`bindingSigSapling`	`byte[64]`	A *Sapling binding signature* on the *SIGHASH transaction hash*, validated as specified in **§? '??'** on p. **??**.

* Version constraints apply to the effectiveVersion, which is equal to $\mathsf{min}(2, \texttt{version})$ when $\texttt{fOverwintered} = 0$ and to `version` otherwise. The consensus rules later in this section specify constraints on `nVersionGroupId` depending on effectiveVersion.

† The `joinSplitPubKey` and `joinSplitSig` fields are present if and only if effectiveVersion ≥ 2 and $\texttt{nJoinSplit} > 0$.

‡ `bindingSigSapling` is present if and only if effectiveVersion $= 4$ and $\texttt{nSpendsSapling} + \texttt{nOutputsSapling} > 0$.

7.1.1 Transaction Identifiers #txnidentifiers

The *transaction ID* of a or earlier *transaction* is the SHA-256d hash of the *transaction* encoding in the format described above.

7.1.2 Transaction Consensus Rules #txnconsensus

Consensus rules:

- The *transaction version number* **MUST** be greater than or equal to 1.
- The `fOverwintered` flag **MUST NOT** be set.
- The encoded size of the *transaction* **MUST** be less than or equal to 100000 bytes.
- If effectiveVersion $= 1$ or `nJoinSplit` $= 0$, then both `tx_in_count` and `tx_out_count` **MUST** be nonzero.
- A *transaction* with one or more *transparent inputs* from *coinbase transactions* **MUST** have no *transparent outputs* (i.e. `tx_out_count` **MUST** be 0). Inputs from *coinbase transactions* include *Founders' Reward* outputs.
- If effectiveVersion ≥ 2 and `nJoinSplit` > 0, then:
  - `joinSplitPubKey` **MUST** be a valid encoding (see **§? '??'** on p. **??**) of an Ed25519 *validating key*.
  - `joinSplitSig` **MUST** represent a valid signature under `joinSplitPubKey` of dataToBeSigned, as defined in **§? '??'** on p. **??**.
- The total value in *zatoshi* of *transparent outputs* from a *coinbase transaction* **MUST NOT** be greater than the value in *zatoshi* of *block subsidy* plus the *transaction fees* paid by *transactions* in this *block*.
- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions*.
-
-
- A *coinbase transaction* for a *block* at *block height* greater than 0 **MUST** have a script that, as its first item, encodes the *block height* height as follows. For height in the range $\{1 .. 16\}$, the encoding is a single byte of value 0x50 + height. Otherwise, let heightBytes be the signed little-endian representation of height, using the minimum nonzero number of bytes such that the most significant byte is $<$ 0x80. The length of heightBytes **MUST** be in the range $\{1 .. 5\}$. Then the encoding is the length of heightBytes encoded as one byte, followed by heightBytes itself. This matches the encoding used by **Bitcoin** in the implementation of [**BIP-34**] (but the description here is to be considered normative).
- A *coinbase transaction* script **MUST** have length in $\{2 .. 100\}$ bytes.
- A *transparent input* in a non-coinbase *transaction* **MUST NOT** have a null *prevout*.
- Every non-null *prevout* **MUST** point to a unique *UTXO* in either a preceding *block*, or a ***previous*** *transaction* in the same *block*.
- A *transaction* **MUST NOT** spend a *transparent output* of a *coinbase transaction* from a *block* less than 100 *blocks* prior to the spend. Note that *transparent outputs* of *coinbase transactions* include *Founders' Reward* outputs .
- A *transaction* **MUST NOT** spend an output of the *genesis block coinbase transaction*. (There is one such zero-valued output, on each of *Testnet* and *Mainnet*.)
- TODO: Other rules inherited from **Bitcoin**.

The types specified in **§? '??'** on p. **??** are part of the consensus rules.

Consensus rules associated with each *JoinSplit description* (**§? '??'** on p. **??**) **MUST** also be followed.

Notes:

- Previous versions of this specification defined what is now the `header` field as a signed `int32` field which was required to be positive. The consensus rule that the `fOverwintered` flag **MUST NOT** be set before **Overwinter** has activated, has the same effect.
- The semantics of *transactions* with *version number* not equal to 1, 2, is not currently defined.
- The exclusion of *transactions* with *transaction version number* ***greater than*** 2 is not a consensus rule before **Overwinter** activation. Such *transactions* may exist in the *block chain* and **MUST** be treated identically to version 2 *transactions*.
- The *transaction version number* `0x7FFFFFFF`, and the *version group ID* `0xFFFFFFFF`, are reserved for use in experimental extensions to *transaction* format or semantics on private testnets. They **MUST NOT** be used on the **Zcash** *Mainnet* or *Testnet*.
- Note that a future upgrade might use ***any*** *transaction version number*. It is likely that an upgrade that changes the *transaction version number* will also change the *transaction* format, and software that parses *transactions* **SHOULD** take this into account.
- A *transaction version number* of 2 does not have the same meaning as in **Bitcoin**, where it is associated with support for `OP_CHECKSEQUENCEVERIFY` as specified in [**BIP-68**]. **Zcash** was forked from Bitcoin Core v0.11.2 and does not currently support BIP 68.

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

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

7.2 JoinSplit Description Encoding and Consensus

#joinsplitencodingandconsensus

An abstract *JoinSplit description*, as described in **§? '??'** on p. **??**, is encoded in a *transaction* as an instance of a `JoinSplitDescription` type:

Bytes	Name	Data Type	Description
8	`vpub_old`	`uint64`	A value v_{pub}^{old} that the *JoinSplit transfer* removes from the *transparent transaction value pool*.
8	`vpub_new`	`uint64`	A value v_{pub}^{new} that the *JoinSplit transfer* inserts into the *transparent transaction value pool*.
32	`anchor`	`byte[32]`	A *root* $\mathsf{rt}^{\mathsf{Sprout}}$ of the **Sprout** *note commitment tree* at some *block height* in the past, or the *root* produced by a previous *JoinSplit transfer* in this *transaction*.
64	`nullifiers`	`byte[32]` [N^{old}]	A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{old}_{1..N^{old}}$.
64	`commitments`	`byte[32]` [N^{new}]	A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{new}_{1..N^{new}}$.
32	`ephemeralKey`	`byte[32]`	A Curve25519 *public key* epk.
32	`randomSeed`	`byte[32]`	A 256-bit seed that must be chosen independently at random for each *JoinSplit description*.
64	`vmacs`	`byte[32]` [N^{old}]	A sequence of message authentication tags $h_{1..N^{old}}$ binding h_{Sig} to each a_{sk} of the *JoinSplit description*, computed as described in **§? '??'** on p. **??**.
296 †	`zkproof`	`byte[296]`	An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see **§? '??'** on p. **??**).
192 ‡	`zkproof`	`byte[192]`	An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see **§? '??'** on p. **??**).
1202	`encCiphertexts`	`byte[601]` [N^{new}]	A sequence of ciphertext components for the encrypted output *notes*, $C^{enc}_{1..N^{new}}$.

† BCTV14 proofs are used when the *transaction version* is 2 or 3, i.e. before **Sapling** activation.

The `ephemeralKey` and `encCiphertexts` fields together form the *transmitted notes ciphertext*, which is computed as described in **§? '??'** on p. **??**.

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

7.3 Block Header Encoding and Consensus

#blockheader

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

Bytes	Name	Data Type	Description
4	`nVersion`	`int32`	The *block version number* indicates which set of *block* validation rules to follow. The current and only defined *block version number* for **Zcash** is 4.
32	`hashPrevBlock`	`byte[32]`	A SHA-256d hash in internal byte order of the previous *block*'s *header*. This ensures no previous *block* can be changed without also changing this *block*'s *header*.
32	`hashMerkleRoot`	`byte[32]`	A SHA-256d hash in internal byte order. The merkle root is derived from the hashes of all *transactions* included in this *block*, ensuring that none of those *transactions* can be modified without modifying the *header*.
32	`hashReserved` /	`byte[32]`	A reserved field, to be ignored.
4	`nTime`	`uint32`	The *block timestamp* is a Unix epoch time (UTC) when the miner started hashing the *header* (according to the miner).
4	`nBits`	`uint32`	An encoded version of the *target threshold* this *block*'s *header* hash must be less than or equal to, in the same nBits format used by **Bitcoin**. [**Bitcoin-nBits**]
32	`nNonce`	`byte[32]`	An arbitrary field that miners can change to modify the *header* hash in order to produce a hash less than or equal to the *target threshold*.
3	`solutionSize`	`compactSize`	The size of an *Equihash* solution in bytes (always 1344).
1344	`solution`	`byte[1344]`	The *Equihash* solution.

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

Let ThresholdBits be as defined in **§? '??'** on p. **??**, and let PoWMedianBlockSpan be the constant defined in **§? '??'** on p. **??**.

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

Consensus rules:

- The *block version number* **MUST** be greater than or equal to 4.
- For a *block* at *block height* height, `nBits` **MUST** be equal to ThresholdBits(height).
- The *block* **MUST** pass the difficulty filter defined in **§? '??'** on p. **??**.
- `solution` **MUST** represent a *valid Equihash solution* as defined in **§? '??'** on p. **??**.
- For each *block* other than the *genesis block*, `nTime` **MUST** be strictly greater than the *median-time-past* of that *block*.

- For each *block* at *block height* 2 or greater on *Mainnet*, or *block height* 653606 or greater on *Testnet*, `nTime` **MUST** be less than or equal to the *median-time-past* of that *block* plus $90 \cdot 60$ seconds.
- The size of a *block* **MUST** be less than or equal to 2000000 bytes.
- A *block* **MUST** have at least one *transaction*.
- The first *transaction* in a *block* **MUST** be a *coinbase transaction*, and subsequent *transactions* **MUST NOT** be *coinbase transactions*.
- TODO: Other rules inherited from **Bitcoin**.

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

Notes:

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

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

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

7.4 Proof of Work

#pow

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

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

- The `solution` field encodes a *valid Equihash solution* according to **§? '??'** on p. **??**.
- The *block header* satisfies the difficulty check according to **§? '??'** on p. **??**.

7.4.1 Equihash

#equihash

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

The *Equihash* parameters for *Mainnet* and *Testnet* are $n = 200, k = 9$.

Equihash is based on a variation of the Generalized Birthday Problem [**AR2017**]: given a sequence $X_{1\,..\,\mathrm{N}}$ of n-bit strings, find 2^k distinct X_{i_j} such that $\bigoplus_{j=1}^{2^k} X_{i_j} = 0$.

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

Let powheader :=

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

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

EquihashGen is instantiated in **§? '??'** on p. **??**.

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

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

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

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

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

7.4.2 Difficulty filter

#difficulty

Let ToTarget be as defined in § ?? '??' on p. ??.

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

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

7.4.3 Difficulty adjustment

#diffadjustment

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

PoWLimit, HalvingInterval, PoWAveragingWindow, PoWMaxAdjustDown, PoWMaxAdjustUp, PoWDampingFactor, and PoWTargetSpacing are specified in section § ?? '??' on p. ??.

Let ToCompact and ToTarget be as defined in § ?? '??' on p. ??.

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 § ?? '??' on p. ??.

Define:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Notes:

- The convention used for the height parameters to the functions MedianTime, MeanTarget, ActualTimespan, ActualTimespanDamped, ActualTimespanBounded, Threshold, and ThresholdBits is that these functions use only information from *blocks **preceding*** the given *block height*.
- When the median function is applied to a sequence of even length (which only happens in the definition of MedianTime during the first PoWAveragingWindow $- 1$ *blocks* of the *block chain*), the element that begins the second half of the sequence is taken. This corresponds to the zcashd implementation, but was not specified correctly in versions of this specification prior to v2019.0.0.

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

7.4.4 nBits conversion

#nbits

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

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

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

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

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

7.4.5 Definition of Work

#workdef

As explained in **§? '??'** on p. **??**, 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 **§? '??'** on p. **??**.

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

7.5 Calculation of Block Subsidy and Founders' Reward

#subsidies

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

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

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

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

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

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

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

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

7.6 Payment of Founders' Reward

#foundersreward

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

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

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

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

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

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

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

Let SlowStartShift and Halving be defined as in the previous section.

Define:

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

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

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

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

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

Notes:

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

7.7 Changes to the Script System

#scripts

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

7.8 Bitcoin Improvement Proposals

#bips

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

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

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

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

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

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

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

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

8 Differences from the Zerocash paper #differences

8.1 Transaction Structure #trstructure

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

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

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

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

8.2 Memo Fields #memodiffs

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 **§? '??'** on p. **??**.

8.3 Unification of Mints and Pours #mintsandpours

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

- a "Mint" transaction takes value from *UTXOs (unspent transaction outputs)* as input and produces a new *shielded note* as output.
- a "Pour" transaction takes up to N^{old} *shielded notes* as input, and produces up to N^{new} *shielded notes* and a *UTXO* as output.

Only "Pour" transactions included a *zk-SNARK* proof.

In **Zcash**, the sequence of operations added to a *transaction* (see **§? '??'** on p. **??**) consists only of *JoinSplit transfers*. A *JoinSplit transfer* is a Pour operation generalized to take a *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 a *UTXO* can produce up to N^{new} output *notes*, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the *JoinSplit transfer*: an unused (zero-value) input is indistinguishable from an input that takes value from a *note*.

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

8.4 Faerie Gold attack and fix #faeriegold

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

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by Receive (as defined in [**BCGGMTV2014**]), but only one of which can be spent.

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

This attack does not violate the security definitions given in [**BCGGMTV2014**]. The issue could be framed as a problem either with the definition of Completeness, or the definition of Balance:

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

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

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

[**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 **§? '??'** on p. **??**). 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 **§? '??'** on p. **??**. ($\mathsf{h_{Sig}}$ 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 **§? '??'** on p. **??** 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 $\mathsf{PRF}^{\mathsf{nfSprout}}_{\mathsf{a_{sk}}}(\rho)$; this is only possible if the adversary finds a collision across both inputs on $\mathsf{PRF}^{\mathsf{nfSprout}}$, which is assumed to be infeasible — see **§? '??'** on p. **??**.

[**Sprout**] Crucially, "*nullifier* integrity" is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note* (**§? '??'** on p. **??**). 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 adversary sees a victim's *transaction*, and is able to publish another *transaction* that is mined first and blocks the victim's *transaction*. This attack would be possible if the public value(s) used to enforce uniqueness of ρ could be chosen arbitrarily by the *transaction* creator: the victim's *transaction*, rather than the adversary's, would be considered to be repeating these values. In the chosen solution that uses *nullifiers* for these public values, they are enforced to be dependent on *spending keys* controlled by the original *transaction* creator (whether or not each input *note* is a *dummy*), and so a roadblock attack cannot be performed by another party who does not know these keys.

8.5 Internal hash collision attack and fix

#internalh

The **Zerocash** security proof requires that the composition of $\mathsf{COMM}_{\mathsf{rcm}}$ and $\mathsf{COMM}_{\mathsf{s}}$ is a computationally *binding* commitment to its inputs a_{pk}, v, and ρ. However, the instantiation of $\mathsf{COMM}_{\mathsf{rcm}}$ and $\mathsf{COMM}_{\mathsf{s}}$ in section 5.1 of the paper did not meet the definition of a *binding* commitment at a 128-bit security level. Specifically, the internal hash of a_{pk} and ρ is truncated to 128 bits (motivated by providing statistical *hiding* security). This allows an attacker, with a work factor on the order of 2^{64}, to find distinct pairs $(\mathsf{a}_{\mathsf{pk}}, \rho)$ and $(\mathsf{a}_{\mathsf{pk}}', \rho')$ 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** *notes*. The motivation for the nested construction in **Zerocash** was to allow Mint transactions to be publically verified without requiring *zk-SNARK proofs* ([**BCGGMTV2014**]). Since **Zcash** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers* (for **Sprout**), and each transfer always uses a *zk-SNARK 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 SHA256Compress evaluations needed to compute each *note commitment* from three to two, saving a total of four SHA256Compress evaluations in the *JoinSplit statement*.

[**Sprout**] **Note:** The full SHA-256 algorithm is used for $\mathsf{NoteCommit}^{\mathsf{Sprout}}$, with randomness appended after the commitment input. The commitment input can be split into two blocks, call them x of length 64 bytes, and y of the remaining length (9 bytes). Let $\mathsf{COMM}'_r(z : \mathbb{B}^{\mathbb{Y}^{[41]}})$ be the *commitment scheme* that applies SHA256Compress with the first 32 bytes of z in the IV, and the rest of z (9 bytes), the randomness r (32 bytes), and padding up to 64 bytes in the SHA256Compress input block. Then we have $\mathsf{NoteCommit}^{\mathsf{Sprout}}_r(x \,||\, y) = \mathsf{COMM}'_r(\mathsf{SHA256Compress}(x) \,||\, y)$. Suppose we make the reasonable assumption that COMM' is a computationally *binding* and *hiding commitment scheme*. If SHA256Compress is *collision-resistant* with the standard IV[8], then $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is as secure for *binding* as COMM'. Also $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is as secure for *hiding* as COMM' (without any assumption on SHA256Compress). This effectively rules out potential concerns about the Merkle–Damgård structure [**Damgard1989**] of SHA-256 causing any security problem for $\mathsf{NoteCommit}^{\mathsf{Sprout}}$.

[**Sprout**] **Note:** **Sprout** *note commitments* are not statistically *hiding*, so for **Sprout** notes, **Zcash** does not support the "everlasting anonymity" property described in [**BCGGMTV2014**], even when used as described in that section. While it is possible to define a statistically *hiding*, computationally *binding commitment scheme* for this use at a 128-bit security level, the overhead of doing so within the *JoinSplit statement* was not considered to justify the benefits.

8.6 Changes to PRF inputs and truncation

#truncation

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

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

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

[8] If SHA256Compress is not *collision-resistant* with the standard IV, then SHA-256 is not *collision-resistant* for a 2-block input.

In more detail:

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

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

[**Sprout**] For resistance to Faerie Gold attacks as described in **§? '??'** on p. **??**, **Zcash** depends on *collision resistance* of hSigCRH and PRF^{ρ} (instantiated using BLAKE2b-256 and SHA256Compress respectively). *Collision resistance* of a truncated hash does not follow from *collision resistance* of the original hash, even if the truncation is only by one bit. This motivated avoiding truncation along any path from the inputs to the computation of $\mathsf{h_{Sig}}$ to the uses of ρ.

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

8.7 In-band secret distribution

#inbandrationale

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**) and the authenticated encryption algorithm AEAD_CHACHA20_POLY1305. This scheme is still loosely based on ECIES, and on the crypto_box_seal scheme defined in libsodium [**libsodium-Seal**].

The motivations for this change were as follows:

- The **Zerocash** paper did not specify the curve to be used. We believe that Curve25519 has significant side-channel resistance, performance, implementation complexity, and robustness advantages over most other available curve choices, as explained in [**Bernstein2006**].
- ECIES permits many options, which were not specified. There are at least –counting conservatively– 576 possible combinations of options and algorithms over the four standards (ANSI X9.63, IEEE Std 1363a-2004, ISO/IEC 18033-2, and SEC 1) that define ECIES variants [**MAEA2010**].
- Although the **Zerocash** paper states that ECIES satisfies *key privacy* (as defined in [**BBDP2001**]), it is not clear that this holds for all curve parameters and key distributions. For example, if a group of non-prime order is used, the distribution of ciphertexts could be distinguishable depending on the order of the points representing the ephemeral and recipient *public keys*. Public key validity is also a concern. Curve25519 key agreement is defined in a way that avoids these concerns due to the curve structure and the "clamping" of *private keys*.
- Unlike the DHAES/DHIES proposal on which it is based [**ABR1999**], ECIES does not require a representation of the sender's *ephemeral public key* to be included in the input to the KDF, which may impair the security properties of the scheme. (The Std 1363a-2004 version of ECIES [**IEEE2004**] has a "DHAES mode"

that allows this, but the representation of the key input is underspecified, leading to incompatible implementations.) The scheme we use for **Sprout** has both the ephemeral and recipient *public key* encodings –which are unambiguous for Curve25519– and also $\mathsf{h}_{\mathsf{Sig}}$ and a nonce as described below, as input to the KDF. Note that being able to break the Elliptic Curve Diffie–Hellman Problem on Curve25519 (without breaking AEAD_CHACHA20_POLY1305 as an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the *transmitted note(s) ciphertext* unless $\mathsf{pk}_{\mathsf{enc}}$ is known or guessed.

- [**Sprout**] The KDF also takes a public seed $\mathsf{h}_{\mathsf{Sig}}$ 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 $\mathsf{h}_{\mathsf{Sig}}$ is authenticated, by the *zk-SNARK proof*, as having been chosen with knowledge of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$, so an adversary cannot modify it in a ciphertext from someone else's transaction for use in a chosen-ciphertext attack without detection.
- [**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 [**Bernstein2005**] in the multi-user setting [**Zaverucha2012**]. This is especially necessary because the privacy of **Zcash** transactions may need to be maintained far into the future, and upgrading the encryption algorithm would not prevent a future adversary from attempting to decrypt ciphertexts encrypted before the upgrade. Other cryptovalues that could be attacked to break the privacy of transactions are also sufficiently long to resist parallel brute force in the multi-user setting: for **Sprout**, a_{sk} is 252 bits, and $\mathsf{sk}_{\mathsf{enc}}$ is no shorter than a_{sk}.

For all shielded protocols, the checking of *note commitments* makes *partitioning oracle attacks* [**LGR2021**] against the *transmitted note ciphertext* infeasible, at least in the absence of side-channel attacks.

8.8 Omission in Zerocash security proof

#crprf

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

Suppose that an adversary finds a collision on $\mathsf{PRF}^{\mathsf{addr}}$ such that $\mathsf{a}^1_{\mathsf{sk}}$ and $\mathsf{a}^2_{\mathsf{sk}}$ are distinct *spending keys* for the same a_{pk}. Because the *note commitment* is to a_{pk}, but the *nullifier* is computed from a_{sk} (and ρ), the adversary is able to double-spend the note, once with each 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 a_{sk} in the witness is ***some*** preimage of the a_{pk} used in the *note commitment*.

The error is in the proof of Balance in [**BCGGMTV2014**]. 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 $(\mathsf{v}^{\mathsf{old}}_i, \mathsf{COMM}_{\mathsf{s}}(\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}, \rho^{\mathsf{old}}_i))$.)

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 $\mathsf{PRF}^{\mathsf{addr}}$ in **Zerocash** and **Sprout**, which *are collision-resistant* assuming that SHA256Compress is.

The proof can be straightforwardly repaired. The intuition is that we can rely on *collision resistance* of $\mathsf{PRF}^{\mathsf{addr}}$ (on both its arguments) to argue that distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}$, together with constraint 1(b) of the *JoinSplit statement* (see **§? '??'** on p. **??**), 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

#miscdiffs

- The paper defines a *note* as $((\mathsf{a}_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}}), \mathsf{v}, \rho, \mathsf{rcm}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines a **Sprout** *note* as $(\mathsf{a}_{\mathsf{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$. The instantiation of $\mathsf{COMM}_{\mathsf{s}}$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ in **Sprout**. $\mathsf{pk}_{\mathsf{enc}}$ is also not needed as part of a *note*: it is not an input to $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ nor is it constrained by the **Zerocash** `POUR` *statement* or the **Zcash** *JoinSplit statement*. cm can be computed from the other fields.
- The length of proof encodings given in the paper is 288 bytes. [**Sprout**] This differs from the 296 bytes specified in **§? '??'** on p. **??**, 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 $\{0 \,..\, \text{MAX\_MONEY}\}$, while in **Zerocash** it is $\{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\}$. (The *JoinSplit statement* still only directly enforces that the sum of amounts in a given *JoinSplit transfer* is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

9 Acknowledgements

#acknowledgements

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

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

The authors would like to thank everyone with whom they have discussed the **Zerocash** and **Zcash** protocol designs; in addition to the preceding, this includes Mike Perry, isis agora lovecruft, Leif Ryge, Andrew Miller, Ben Blaxill, Samantha Hulsey, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, John Tromp, Paige Peterson, jl777, Alison Stevenson, Maureen Walsh, Filippo Valsorda, Zaki Manian, Kexin Hu, Brian Warner, Mary Maller, Michael Dixon, Andrew Poelstra, Benjamin Winston, Josh Cincinnati, Kobi Gurkan, Weikeng Chen, Henry de Valence, Deirdre Connolly, Chelsea Komlo, Zancas Wilcox, Jane Lusby, Teor, Izaak Meckler, Zac Williamson, Vitalik Buterin, Jakub Zalewski, Oana Ciobotaru, Andre Serrano, Brad Miller, Charlie O'Keefe, David Campbell, Elena Giralt, Francisco Gindre, Joseph Van Geffen, Josh Swihart, Kevin Gorham, Larry Ruane, Marshall Gaucher, Ryan Taylor, Sasha Meyer, and no doubt others. We would also like to thank the designers and developers of **Bitcoin** and Bitcoin Core.

Zcash has benefited from security audits performed by NCC Group, Coinspect, Least Authority, Mary Maller, Kudelski Security, QEDIT, and Trail of Bits. We also thank Mary Maller for her work on reviewing the security proofs for Halo 2 (any remaining errors are ours).

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 **§? '??'** on p. **??** were also found by Daira Hopwood.

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

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

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

The observation in ***§?? '??'*** on p. **??** that *diversified payment address* unlinkability can be proven in the same way as *key privacy* for ElGamal, is due to Mary Maller.

We thank Ariel Gabizon for teaching us the techniques of [**BFIJSV2010**] used in ***§?? '??'*** on p. **??**, by applying them to BCTV14.

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

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

We thank the organizers of the ZKProof standardization effort and workshops; and also Anna Rose, Fredrik Harrysson, Terun Chitra, James Prestwich, Josh Cincinnati, Tanya Karsou, Henrik Jose, Chris Ward, and others for their work on the Zero Knowledge Podcast, ZK Summits, and ZK Study Club. These efforts have enriched the zero knowledge community immeasurably.

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

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

Finally, we would like to thank the Internet Archive for their scan of Peter Newell's illustration of the Jubjub bird, from [**Carroll1902**].

10 Change History

#changehistory

2022.3.8 2022-09-15 #2022.3.8

- Correct Jurgen Bos' name.

2022.3.7 2022-09-10 #2022.3.7

- Specify in ***§?? '??'*** on p. **??** that **NU5** is the most recent settled *network upgrade*.

2022.3.6 2022-09-01 #2022.3.6

- Correct Kexin Hu's name.
- Correct cross-references for the definition of an *anchor*.
- Replace ResearchGate links for [**CDvdG1987**] with alternatives that do not cause false-positive link checker errors.
- In `protocol/README.rst`: update the build dependency documentation for Debian Bullseye, mention the "`make linkcheck`" target, and correct the description of "`make all`".
- Update the `Makefile` to build correctly with newer versions of `latexmk`.

2022.3.5 2022-08-02 #2022.3.5

2022.3.4 2022-06-22 #2022.3.4

- Update references for [**ECCZF2019**] and [**ZIP-302**].

2022.3.3 2022-06-21 #2022.3.3

- Rename ExcludedPointEncodings to PreCanopyExcludedPointEncodings.
- In **§? '??'** on p. **??**, remove the statement that future key representations might use the padding bits of **Sprout** spending keys.
- Give a full-text URL for [**Nakamoto2008**].

2022.3.2 2022-06-06 #2022.3.2

- Make **§? '??'** on p. **??** more precise about *chain value pools*.

2022.3.1 2022-04-28 #2022.3.1

- Correct "block chain branch" to "*consensus branch*" to match [**ZIP-200**].
- Add an acknowledgement to Mary Maller for reviewing the Halo 2 security proofs.
- Add an acknowledgement to Josh Cincinnati for discussions on the Zcash protocol.
- Add acknowledgements to more people associated with the ZK Podcast.

2022.3.0 2022-03-18 #2022.3.0

- In **§? '??'** on p. **??**, define what a *settled network upgrade* is, specify requirements for checkpointing, and allow nodes to impose a limitation on rollback depth.
- Document the consensus rule that coinbase script length **MUST** be $\{2\,..\,100\}$ bytes.
- **§? '??'** on p. **??** effectively defined a *coinbase transaction* as the first *transaction* in a *block*. This wording was copied from the Bitcoin Developer Reference [**Bitcoin-CbInput**], but it does not match the implementation in zcashd that was inherited from Bitcoin Core. Instead, a *coinbase transaction* should be, and now is, defined as a *transaction* with a single null *prevout*. The specifications of consensus rules have been clarified and adjusted (without any actual consensus change) to take this into account, as follows:
  - a *block* **MUST** have at least one *transaction*;
  - the first *transaction* in a *block* **MUST** be a *coinbase transaction*, and subsequent *transactions* **MUST NOT** be *coinbase transactions*;
  - a *transparent input* in a non-coinbase *transaction* **MUST NOT** have a null *prevout*;
  - every non-null *prevout* **MUST** point to a unique *UTXO* in either a preceding *block*, or a ***previous*** *transaction* in the same *block* (this rule was previously not given explicitly because it was assumed to be inherited from **Bitcoin**);

- the rule that "A *coinbase transaction* **MUST NOT** have any *transparent inputs* with non-null `prevout` fields" is removed as an explicit consensus rule because it is implied by the corrected definition of *coinbase transaction*.

2022.2.19 2022-01-19 #2022.2.19

- In **§? '??'** on p. **??**, clarify that balance for *JoinSplit transfers* is enforced by the *JoinSplit statement*, and that there is no consensus rule to check it directly.
- In **§? '??'** on p. **??**, add a security argument for why the SHA-256-based *commitment scheme* $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is *binding* and *hiding*, under reasonable assumptions about SHA256Compress.

2022.2.18 2022-01-03 #2022.2.18

- Refine the security argument about *partitioning oracle attacks* in **§? '??'** on p. **??**:
  - The argument for decryption with an *incoming viewing key* does not need to depend on the *Decisional Diffie–Hellman Problem*, since $\mathsf{g_d}$ is committed to by the *note commitment* as well as $\mathsf{pk_d}$.
  - It is necessary to say that the *note commitment* is always checked for a successful decryption.
  - Pedantically, it was not correct to conclude from the given security argument that *partitioning oracle attacks* against an *outgoing ciphertext* are necessarily prevented, according to the definition in [**LGR2021**]. Instead, the correct conclusions are that such attacks could not feasibly result in any equivocation of the decrypted data, or in recovery of ovk or ock.
- Correct the note about domain separators for $\mathsf{PRF}^{\mathsf{expand}}$ in **§? '??'** on p. **??**, and ensure that new domain separators for deriving internal keys from [**ZIP-32**] and [**ZIP-316**] are included.

2021.2.17 2021-12-01 #2021.2.17

- Add notes in that z_j may be sampled from $\{0 \,..\, 2^{128} - 1\}$ instead of $\{1 \,..\, 2^{128} - 1\}$.
- Add note in **§? '??'** on p. **??** about resistance of *note* encryption to *partitioning oracle attacks* [**LGR2021**].
- Add acknowledgement to Mihir Bellare for contributions to the science of zero-knowledge proofs.
- Add acknowledgement to Sasha Meyer.

2021.2.16 2021-09-30 #2021.2.16

- Correct the consensus rule about the maximum value of outputs in a *coinbase transaction*: it should reference the *block subsidy* rather than the *miner subsidy*.

2021.2.15 2021-09-01 #2021.2.15

- Fix a reference to nonexistent version 2019.0-beta-40 of this specification (in **§? '??'** on p. **??**) that should be v2019.0.0.
- Fix URL links to [**BBDP2001**] and [**BDJR2000**].
- Improve `protocol/links_and_dests.py` to eliminate false positives when checking DOI links.

2021.2.14 2021-08-12 #2021.2.14

2021.2.13 2021-07-29 #2021.2.13

- Add consensus rules in **§? '??'** on p. **??** that a *block* **MUST NOT** add *note commitments* that exceed the capacity of any of the **Sprout** *note commitment trees*.

2021.2.12 2021-07-29 #2021.2.12

- No changes before **NU5**.

2021.2.11 2021-07-20 #2021.2.11

- No changes before **NU5**.

2021.2.10 2021-07-13 #2021.2.10

- Remove a spurious reference to rseed in ***§?? '??'*** on p. **??**. There were no changes for **Sprout** in [**ZIP-212**].

2021.2.9 2021-07-01 #2021.2.9

- Correct an erroneous statement in ***§?? '??'*** on p. **??** that claimed *transaction IDs* are not part of the consensus protocol.

2021.2.8 2021-06-29 #2021.2.8

- Describe *transaction IDs* in ***§?? '??'*** on p. **??**.
- Add a section ***§?? '??'*** on p. **??** on how to compute *transaction IDs*.
- Split the *transaction*-related consensus rules into their own subsection ***§?? '??'*** on p. **??**, for more precise cross-referencing.

2021.2.7 2021-06-28 #2021.2.7

- No changes before **NU5**.

2021.2.6 2021-06-26 #2021.2.6

- Give cross-references to ***§?? '??'*** on p. **??** where $\sqrt[2]{\cdot}$ and $\sqrt[+]{\cdot}$ are used.

2021.2.5 2021-06-19 #2021.2.5

- No changes before **NU5**.

2021.2.4 2021-06-08 #2021.2.4

- No changes before **NU5**.

2021.2.3 2021-06-06 #2021.2.3

- Move the section on abstraction (previously section 5.1) to ***§?? '??'*** on p. **??**. Section 5.2 has been split into two (***§?? '??'*** on p. **??** and ***§?? '??'*** on p. **??**) to avoid renumbering later subsections.
- Correct an error in the encoding of height-in-coinbase for *blocks* at heights $1 .. 16$.
- Clarify, in ***§?? '??'*** on p. **??**, requirements on the range of *block heights* that should be supported.
- Delete the sentence "All conversions between Ed25519 points, byte sequences, and integers used in this section are as specified in [**BDLSY2012**]." from ***§?? '??'*** on p. **??**. This sentence was misleading given that the conversions in [**BDLSY2012**] are not sufficiently well-specified for a consensus protocol; it should have been deleted earlier when explicit definitions for $\mathsf{reprBytes}_{\mathsf{Ed25519}}$ and $\mathsf{abstBytes}_{\mathsf{Ed25519}}$ were added.
- Make the **NU5** specification the default.

2021.2.2 2021-05-20 #2021.2.2

- No changes before **NU5**.

2021.2.1 2021-05-20 #2021.2.1

2021.2.0 2021-05-07 #2021.2.0

- Clarify notation by changing ℓ_{rcm} to $\ell_{\mathsf{rcm}}^{\mathsf{Sprout}}$.

2021.1.24 2021-04-23 #2021.1.24

- Explicitly say that *coinbase transactions* **MUST NOT** have *transparent inputs* (this is a consensus rule inherited from **Bitcoin** which has been present since launch).

2021.1.23 2021-04-19 #2021.1.23

- Fix some URLs in references.

2021.1.22 2021-04-05 #2021.1.22

- Make sure that Change History entries are URL destinations.

2021.1.21 2021-04-01 #2021.1.21

- Correct the set of inputs to $\mathsf{PRF}^{\mathsf{expand}}$ used for [**ZIP-32**] in **§? '??'** on p. **??**.
- Write the caution about linkage between the abstract and concrete protocols in **§? '??'** on p. **??**.
- Update the **Sprout** key component diagram in **§? '??'** on p. **??** to remove magenta highlighting.

2021.1.20 2021-03-25 #2021.1.20

- Credit Eirik Ogilvie-Wigley as a designer of the Zcash protocol. Add Andre Serrano, Brad Miller, Charlie O'Keefe, David Campbell, Elena Giralt, Francisco Gindre, Joseph Van Geffen, Josh Swihart, Kevin Gorham, Larry Ruane, Marshall Gaucher, and Ryan Taylor to the acknowledgements.
- Move the definition of $\bot$ to before its first use.
- Delete a confusing part of the definition of $\mathsf{concat}_{\mathbb{B}}$ that we don't rely on.
- Add a definition for the § symbol in **§? '??'** on p. **??**, before its first use.
- Remove specification of *memo field* contents, which will be in [**ZIP-302**].
- Remove support for building the **Sprout**-only specification (`sprout.pdf`).
- Remove magenta highlighting of differences from **Zerocash**.

2021.1.19 2021-03-17 #2021.1.19

- No changes before **NU5**.

2021.1.18 2021-03-17 #2021.1.18

2021.1.17 2021-03-15 #2021.1.17

- The definition of an abstraction function in **§? '??'** on p. **??** incorrectly required canonicity, i.e. that $\mathsf{abst}_{\mathbb{G}}$ does not accept inputs outside the range of $\mathsf{repr}_{\mathbb{G}}$. While this was originally intended, it is not true of $\mathsf{abst}_{\mathbb{J}}$. (It is also not true of $\mathsf{abstBytes}_{\mathsf{Ed25519}}$, but Ed25519 is not strictly defined as a *represented group* in this specification.)
- Rename `char` to `byte` in field type declarations.

2021.1.16 2021-01-11 #2021.1.16

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

2020.1.15 2020-11-06 #2020.1.15

- Add a missing consensus rule that has always been implemented in zcashd: there must be at least one *transparent output*, *Output description*, or *JoinSplit description* in a *transaction*.
- Add a consensus rule that the (zero-valued) *coinbase transaction* output of the *genesis block* cannot be spent.
- Define ***Sprout*** *chain value pool balance* and include consensus rules from [**ZIP-209**].
- Reserve *transaction version number* `0x7FFFFFFF` and *version group ID* `0xFFFFFFFF` for experimental use.
- Remove a statement that the language consisting of key and address encoding possibilities is prefix-free. (The human-readable forms are prefix-free but the raw encodings are not; for example, the *raw encoding* of a **Sapling** *spending key* can be a prefix of several of the other encodings.)
- Use "let mutable" to introduce mutable variables in algorithms.
- Include a reference to [**BFIJSV2010**] for batch pairing verification techniques.
- Acknowledge Jack Gavigan as a co-designer of **Sapling** and of the **Zcash** protocol.
- Acknowledge Izaak Meckler, Zac Williamson, Vitalik Buterin, and Jakub Zalewski.
- Acknowledge Alexandra Elbakyan.

2020.1.14 2020-08-19 #2020.1.14

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

2020.1.13 2020-08-11 #2020.1.13

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

2020.1.12 2020-08-03 #2020.1.12

- Include SHA-512 in **§? '??'** on p. **??**.
- Add a reference to [**BCCGLRT2014**] in **§? '??'** on p. **??**.

2020.1.11 2020-07-13 #2020.1.11

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

2020.1.10 2020-07-05 #2020.1.10

- Corrections to a note in **§? '??'** on p. **??**.

2020.1.9 2020-07-05 #2020.1.9

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

2020.1.8 2020-07-04 #2020.1.8

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

2020.1.7 2020-06-26 #2020.1.7

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

2020.1.6 2020-06-17 #2020.1.6

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

2020.1.5 2020-06-02 #2020.1.5

- Mark more index entries as definitions.

2020.1.4 2020-05-27 #2020.1.4

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

2020.1.3 2020-04-22 #2020.1.3

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

2020.1.2 2020-03-20 #2020.1.2

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

2020.1.1 2020-02-13 #2020.1.1

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

2020.1.0 2020-02-06 #2020.1.0

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

2019.0.9 2019-12-27 #2019.0.9

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

2019.0.8 2019-09-24 #2019.0.8

2019.0.7 2019-09-24 #2019.0.7

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

2019.0.6 2019-08-23 #2019.0.6

- No changes to **Sprout** or **Sapling**.

2019.0.5 2019-08-23 #2019.0.5

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

2019.0.4 2019-07-23 #2019.0.4

- Clicking on a section heading now shows section labels.
- Add a **List of Theorems and Lemmata**.
- Changes needed to support TeXLive 2019.

2019.0.3 2019-07-08 #2019.0.3

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

2019.0.2 2019-06-18 #2019.0.2

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

2019.0.1 2019-05-20 #2019.0.1

- No changes to **Sprout** or **Sapling**.

2019.0.0 2019-05-01 #2019.0.0

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

2019.0-beta-39 2019-04-18 #2019.0-beta-39

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

2019.0-beta-38 2019-04-18 #2019.0-beta-38

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

2019.0-beta-37 2019-02-22 #2019.0-beta-37

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

2019.0-beta-36 2019-02-09 #2019.0-beta-36

- Correct isis agora lovecruft's name.

2019.0-beta-35 2019-02-08 #2019.0-beta-35

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

2019.0-beta-34 2019-02-05 #2019.0-beta-34

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

2018.0-beta-33 2018-11-14 #2018.0-beta-33

2018.0-beta-32 2018-10-24 #2018.0-beta-32

2018.0-beta-31 2018-09-30 #2018.0-beta-31

- Add the QED-it report to the acknowledgements.

2018.0-beta-30 2018-09-02 #2018.0-beta-30

- Add dates to Change History entries. (These are the dates of the git tags in local, i.e. UK, time.)

2018.0-beta-29 2018-08-15 #2018.0-beta-29

2018.0-beta-28 2018-08-14 #2018.0-beta-28

2018.0-beta-27 2018-08-12 #2018.0-beta-27

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

2018.0-beta-26 2018-08-05 #2018.0-beta-26

2018.0-beta-25 2018-08-05 #2018.0-beta-25

- `Makefile` changes: name the PDF file for the **Sprout** version of the specification as `sprout.pdf`, and make `protocol.pdf` link to the **Sapling** version.

2018.0-beta-24 2018-07-31 #2018.0-beta-24

2018.0-beta-23 2018-07-27 #2018.0-beta-23

2018.0-beta-22 2018-07-18 #2018.0-beta-22

2018.0-beta-21 2018-06-22 #2018.0-beta-21

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

2018.0-beta-20 2018-05-22 #2018.0-beta-20

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

2018.0-beta-19 2018-04-23 #2018.0-beta-19

2018.0-beta-18 2018-04-23 #2018.0-beta-18

2018.0-beta-17 2018-04-21 #2018.0-beta-17

2018.0-beta-16 2018-04-21 #2018.0-beta-16

- Explicitly note that outputs from *coinbase transactions* include *Founders' Reward* outputs.
- The point represented by $\underline{R}$ in an Ed25519 signature is checked to not be of small order; this is not the same as checking that it is of prime order ℓ.
- Specify support for [**BIP-111**] (the `NODE_BLOOM` service bit) in peer-to-peer network protocol version 170004.
- Give references [**Vercauter2009**] and [**AKLGL2010**] for the optimal ate pairing.
- Give references for BLS [**BLS2002**] and BN [**BN2005**] curves.

- Define $\mathsf{KA}^{\mathsf{Sprout}}$.DerivePublic for Curve25519.
- Caveat the claim about *note traceability set* in **§? '??'** on p. **??** and link to [**Peterson2017**] and [**Quesnelle2017**].
- Do not require a generator as part of the specification of a *represented group*; instead, define it in the *represented pairing* or scheme using the group.
- Refactor the abstract definition of a *signature scheme* to allow derivation of *validating keys* independent of key pair generation.
- Add acknowledgements for Brian Warner, Mary Maller, and the Least Authority audit.
- `Makefile` improvements.

2018.0-beta-15 2018-03-19 #2018.0-beta-15

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

2018.0-beta-14 2018-03-11 #2018.0-beta-14

- Only cosmetic changes to **Sprout**.

2018.0-beta-13 2018-03-11 #2018.0-beta-13

- Only cosmetic changes to **Sprout**.

2018.0-beta-12 2018-03-06 #2018.0-beta-12

2018.0-beta-11 2018-02-26 #2018.0-beta-11

2018.0-beta-10 2018-02-26 #2018.0-beta-10

- Split the descriptions of SHA-256 and SHA256Compress into their own sections. Specify SHA256Compress more precisely.
- Add Kexin Hu to acknowledgements.
- Move bit/byte/integer conversion primitives into **§? '??'** on p. **??**.

2018.0-beta-9 2018-02-10 #2018.0-beta-9

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

2018.0-beta-8 2018-02-08 #2018.0-beta-8

2018.0-beta-7 2018-02-07 #2018.0-beta-7

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

2018.0-beta-6 2018-01-31 #2018.0-beta-6

2018.0-beta-5 2018-01-30 #2018.0-beta-5

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

2018.0-beta-4 2018-01-25 #2018.0-beta-4

2018.0-beta-3 2018-01-22 #2018.0-beta-3

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

2017.0-beta-2.9 2017-12-17 #2017.0-beta-2.9

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

2017.0-beta-2.8 2017-12-02 #2017.0-beta-2.8

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

2017.0-beta-2.7 2017-07-10 #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 2017-05-09 #2017.0-beta-2.6

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

2017.0-beta-2.5 2017-03-07 #2017.0-beta-2.5

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

2017.0-beta-2.4 2017-02-25 #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 $(\mathsf{a}_{\mathsf{pk}}, \rho)$ rather than just on ρ.
- Rename $\mathsf{enforce}_i$ to $\mathsf{enforceMerklePath}_i$.

2017.0-beta-2.3 2017-02-12 #2017.0-beta-2.3

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

2017.0-beta-2.2 2017-02-11 #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 [**BGG2017**].

2017.0-beta-2.1 2017-02-06 #2017.0-beta-2.1

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

2017.0-beta-2 2017-02-04 #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 valid block chain*.

2016.0-beta-1.13 2017-01-20 #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 **§? '??'** on p. **??**.

2016.0-beta-1.12 2017-01-09 #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 2016-12-19 #2016.0-beta-1.11

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

2016.0-beta-1.10 2016-10-30 #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 2016-10-28 #2016.0-beta-1.9

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

2016.0-beta-1.8 2016-10-04 #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 *SIGHASH transaction hashes*.
- Change the representation type of `vpub_old` and `vpub_new` to `uint64`. (This is not a consensus change because the type of $v_{\mathsf{pub}}^{\mathsf{old}}$ and $v_{\mathsf{pub}}^{\mathsf{new}}$ was already specified to be $\{0 \,..\, \mathsf{MAX\_MONEY}\}$; it just better reflects the implementation.)
- Correct the representation type of the *block* `nVersion` field to `uint32`.

2016.0-beta-1.7 2016-10-02 #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 2016-09-26 #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 $\mathsf{PRF}^{\mathsf{nfSprout}}$ corresponding to $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash**.
- Add a paragraph about key length in **§? '??'** on p. **??**.
- Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

2016.0-beta-1.5 2016-09-22 #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 2016-09-19 #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 2016-09-16 #2016.0-beta-1.3

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

2016.0-beta-1.2 2016-09-11 #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 2016-09-05 #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 2016-09-04 #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 $\mathsf{h}_{\mathsf{Sig}}$.
- Fix the lead bytes in *shielded payment address* and *spending key* encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ and $\mathsf{v}_{\mathsf{pub}}^{\mathsf{new}}$.
- Clarify cryptographic security requirements and added definitions relating to the in-band secret distribution.

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

2016.0-alpha-3.1 2016-05-20 #2016.0-alpha-3.1

- Change main font to Quattrocento.

2016.0-alpha-3 2016-05-09 #2016.0-alpha-3

- Change version numbering convention (no other changes).

2.0-alpha-3 2016-05-06 #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 2016-04-21 #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 **§? '??'** on p. **??**).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in **§? '??'** on p. **??** about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

2.0-alpha-1 2016-03-30 #2.0-alpha-1

- First version intended for public review.

11 References

#references

Appendices

#appendices

A Circuit Design

#circuitdesign

A.1 Quadratic Constraint Programs

#constraintprograms

Sapling defines two circuits, Spend and Output, each implementing an abstract *statement* described in **§? '??'** on p. **??** and **§? '??'** on p. **??** respectively. It also adds a Groth16 circuit for the *JoinSplit statement* described in **§? '??'** on p. **??**.

At the next lower level, each circuit is defined in terms of a *quadratic constraint program* (specifying a *Rank 1 Constraint System*), as detailed in this section. In the BCTV14 or Groth16 proving systems, this program is translated to a *Quadratic Arithmetic Program* [**BCTV2014a**] [**WCBTV2015**]. The circuit descriptions given here are necessary to compute witness elements for each circuit, as well as the proving and verifying keys.

Let $\mathbb{F}_{r_\mathbb{S}}$ be the finite field over which Jubjub is defined, as given in **§? '??'** on p. **??**.

A *quadratic constraint 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*. $\times$ should not be confused with $\times$ which is defined as cartesian product in **§? '??'** on p. **??**.

A.2 Elliptic curve background

#ecbackground

The **Sapling** circuits make use of a *complete twisted Edwards elliptic curve* ("*ctEdwards curve*") Jubjub, defined in **§? '??'** on p. **??**, and also a *Montgomery elliptic curve* $\mathbb{M}$ that is birationally equivalent to Jubjub. Following the notation in [**BL2017**] we use (u, v) for affine coordinates on the *ctEdwards curve*, and (x, y) for affine coordinates on the *Montgomery curve*.

A point P is normally represented by two $\mathbb{F}_{r_\mathbb{S}}$ variables, which we name as (P^u, P^v) for an *affine-ctEdwards* point, for instance.

The implementations of scalar multiplication require the scalar to be represented as a bit sequence. We therefore allow the notation $[k\star]\,P$ meaning $[\mathsf{LEBS2IP}_{\mathsf{length}(k\star)}(k\star)]\,P$. There will be no ambiguity because variables representing bit sequences are named with a $\star$ suffix.

The *Montgomery curve* $\mathbb{M}$ 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-ctEdwards* and *affine-Montgomery* coordinates respectively:

$$\mathsf{AffineCtEdwardsJubjub} := (u :: \mathbb{F}_{r_\mathbb{S}}) \times (v :: \mathbb{F}_{r_\mathbb{S}}) : a_\mathbb{J} \cdot u^2 + v^2 = 1 + d_\mathbb{J} \cdot u^2 \cdot v^2$$

$$\mathsf{AffineMontJubjub} := (x :: \mathbb{F}_{r_\mathbb{S}}) \times (y :: \mathbb{F}_{r_\mathbb{S}}) : B_\mathbb{M} \cdot y^2 = x^3 + A_\mathbb{M} \cdot x^2 + x$$

We also define a type representing compressed, ***not necessarily valid***, ctEdwards coordinates:

$$\mathsf{CompressedCtEdwardsJubjub} := (\tilde{u} : \mathbb{B}) \times (v : \mathbb{F}_{r_{\mathbb{S}}})$$

See **§? '??'** on p. **??** 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-ctEdwards* 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 $\mathbb{F}_{r_{\mathbb{S}}}$.

Theorem A.2.1. *$(0, 0)$ is the only point with $y = 0$ on certain Montgomery curves.* #thmmontynotzero

Let $P = (x, y)$ be a point other than $(0, 0)$ on a Montgomery curve $E_{\mathsf{Mont}(A,B)}$ over $\mathbb{F}_r$, 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

#cctcomponents

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

#cctbitops

A.3.1.1 Boolean constraints

#cctboolean

A boolean constraint $b \in \mathbb{B}$ can be implemented as:

$$(1 - b) \ \mathsf{X} \ (b) \ = \ (0)$$

A.3.1.2 Conditional equality

#cctcondeq

The constraint "either $a = 0$ or $b = c$" can be implemented as:

$$(a) \ \mathsf{X} \ (b - c) \ = \ (0)$$

A.3.1.3 Selection constraints

#cctselection

A selection constraint $(b \ ? \ x : y) = z$, where $b : \mathbb{B}$ has been boolean-constrained, can be implemented as:

$$(b) \ \mathsf{X} \ (y - x) \ = \ (y - z)$$

A.3.1.4 Nonzero constraints

#cctnonzero

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, $a_{\mathsf{inv}} = a^{-1} \pmod{r_\mathbb{S}}$:

$$(a_{\mathsf{inv}}) \ \mathsf{X} \ (a) \ = \ (1)$$

This technique comes from [**SVPBABW2012**].

Non-normative note: 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. However, the **Sapling** circuit does not use this optimization.

A.3.1.5 Exclusive-or constraints

#cctxor

An exclusive-or operation $a \oplus b = c$, where $a, b : \mathbb{B}$ are already boolean-constrained, can be implemented in one constraint as:

$$(2 \cdot a) \ \mathsf{X} \ (b) \ = \ (a + b - c)$$

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

#cctmultibitops

A.3.2.1 [Un]packing modulo $r_\mathbb{S}$

#cctmodpack

Let $n : \mathbb{N}^+$ be a constant. The operation of converting a field element, $a : \mathbb{F}_{r_\mathbb{S}}$, to a sequence of boolean variables $b_{0\,..\,n-1} : \mathbb{B}^{[n]}$ 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 constraint 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 $a \bmod r_\mathbb{S} = \left(\sum_{i=0}^{n-1} b_i \cdot 2^i \right) \bmod r_\mathbb{S} = \left(\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_\mathbb{S}) \right) \bmod r_\mathbb{S}$.

This can be implemented in one constraint:

$$\left(\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_{\mathbb{S}})\right) \mathsf{X}\ (1)\ =\ (a)$$

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.
- 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

#cctrange

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.

Define $\Pi_m = \prod_{i=m}^{n-1}(c_i = 0 \vee a_i = 1)$ for $m \in \{0\,..\,n-1\}$. Notice that for any $m < n-1$ such that $c_m = 0$, we have $\Pi_m = \Pi_{m+1}$, and so it is only necessary to allocate separate variables for the Π_m such that $m < n-1$ and $c_m = 1$. Furthermore if $c_{n-2\,..\,0}$ has $t > 0$ trailing 1 bits, then we do not need to allocate variables for $\Pi_{0\,..\,t-1}$ because those variables will not be used below.

More explicitly:

Let $\Pi_{n-1} = a_{n-1}$.

For i from $n-2$ down to t,

- if $c_i = 0$, then let $\Pi_i = \Pi_{i+1}$;
- if $c_i = 1$, then constrain $(\Pi_{i+1})\ \mathsf{X}\ (a_i)\ =\ (\Pi_i)$.

Then we constrain the a_i as follows:

For i from $n-1$ down to 0,

- if $c_i = 0$, constrain $(1 - \Pi_{i+1} - a_i)\ \mathsf{X}\ (a_i)\ =\ (0)$;
- if $c_i = 1$, boolean-constrain a_i as in §**??** '**??**' on p. **??**.

Note that the constraints corresponding to zero bits of c are ***in place of*** boolean constraints on bits of a_i.

This costs $n + k$ constraints, where k is the number of non-trailing 1 bits in $c_{n-2\,..\,0}$.

Theorem A.3.1. *Correctness of a constraint system for range checks.* #thmrangeconstraints

Assume $c_{0\,..\,n-1} : \mathbb{B}^{[n]}$ *and* $c_{n-1} = 1$. *Define* $A_m := \sum_{i=m}^{n-1} a_i \cdot 2^i$ *and* $C_m := \sum_{i=m}^{n-1} c_i \cdot 2^i$. *For any* $m \in \{0\,..\,n-1\}$, $A_m \leq C_m$ *if and only if the restriction of the above constraint system to* $i \in \{m\,..\,n-1\}$ *is satisfied. Furthermore the system at least boolean-constrains* $a_{0\,..\,n-1}$.

Proof. For $i \in \{0\,..\,n-1\}$ such that $c_i = 1$, the corresponding a_i are unconditionally boolean-constrained. This implies that the system constrains $\Pi_i \in \mathbb{B}$ for all $i \in \{0\,..\,n-1\}$. For $i \in \{0\,..\,n-1\}$ such that $c_i = 0$, the constraint $\left(1 - \Pi_{i+1} - a_i\right) \mathbin{\mathsf{X}} \left(a_i\right) = \left(0\right)$ constrains a_i to be 0 if $\Pi_{i+1} = 1$, otherwise it constrains $a_i \in \mathbb{B}$. So all of $a_{0\,..\,n-1}$ are at least boolean-constrained.

To prove the rest of the theorem we proceed by induction on decreasing m, i.e. taking successively longer prefixes of the big-endian binary representations of a and c.

Base case $m = n - 1$: since $c_{n-1} = 1$, the constraint system has just one boolean constraint on a_{n-1}, which fulfils the theorem since $A_{n-1} \leq C_{n-1}$ is always satisfied.

Inductive case $m < n - 1$:

- If $A_{m+1} > C_{m+1}$, then by the inductive hypothesis the constraint system must fail, which fulfils the theorem regardless of the value of a_m.
- If $A_{m+1} \leq C_{m+1}$, then by the inductive hypothesis the constraint system restricted to $i \in \{m+1\,..\,n-1\}$ succeeds. We have $\Pi_{m+1} = \prod_{i=m+1}^{n-1} (c_i = 0 \vee a_i = 1) = \prod_{i=m+1}^{n-1} (a_i \geq c_i)$.
  - If $A_{m+1} = C_{m+1}$, then $a_i = c_i$ for all $i \in \{m+1\,..\,n-1\}$ and so $\Pi_{m+1} = 1$. Also $A_m \leq C_m$ if and only if $a_m \leq c_m$.
    When $c_m = 1$, only a boolean constraint is added for a_m which fulfils the theorem.
    When $c_m = 0$, a_m is constrained to be 0 which fulfils the theorem.
  - If $A_{m+1} < C_{m+1}$, then it cannot be the case that $a_i \geq c_i$ for all $i \in \{m+1\,..\,n-1\}$, so $\Pi_{m+1} = 0$.
    This implies that the constraint on a_m is always equivalent to a boolean constraint, which fulfils the theorem because $A_m \leq C_m$ must be true regardless of the value of a_m.

This covers all cases. □

Correctness of the full constraint system follows by taking $m = 0$ in the above theorem.

The algorithm in **§? '??'** on p. **??** uses range checks with $c = r_{\mathbb{S}} - 1$ to validate *ctEdwards compressed encodings*. In that case $n = 255$ and $k = 132$, so the cost of each such range check is 387 constraints.

Non-normative note: It is possible to optimize the computation of $\Pi_{t\,..\,n-2}$ further. Notice that Π_m is only used when m is the index of the last bit of a run of 1 bits in c. So for each such run of 1 bits $c_{m\,..\,m+N-2}$ of length $N-1$, it is sufficient to compute an N-ary AND of $a_{m\,..\,m+N-2}$ and Π_{m+N-1}: $R = \prod_{i=0}^{N-1} X_i$. This can be computed in 3 constraints for any N; boolean-constrain the output R, and then add constraints

$$\left(N - \sum_{i=0}^{N-1} X_i\right) \mathbin{\mathsf{X}} \left(\mathsf{inv}\right) = \left(1 - R\right) \text{ to enforce that } \sum_{i=0}^{N-1} X_i \neq N \text{ when } R = 0;$$

$$\left(N - \sum_{i=0}^{N-1} X_i\right) \mathbin{\mathsf{X}} \left(R\right) = \left(0\right) \qquad \text{to enforce that } \sum_{i=0}^{N-1} X_i = N \text{ when } R = 1.$$

where inv is witnessed as $\left(N - \sum_{i=0}^{N-1} X_i\right)^{-1}$ if $R = 0$ or is unconstrained otherwise. (Since $N < r_{\mathbb{S}}$, the sums cannot overflow.)

In fact the last constraint is not needed in this context because it is sufficient to compute an upper bound on each Π_m (i.e. it does not benefit a malicious prover to witness $R = 1$ when the result of the AND should be 0). So the cost of computing Π variables for an arbitrarily long run of 1 bits can be reduced to 2 constraints. For example, for $c = r_{\mathbb{S}} - 1$ the overall cost would be reduced to $255 + 68 = 323$ constraints.

These optimizations are not used in **Sapling**.

A.3.3 Elliptic curve operations

#cctelliptic

A.3.3.1 Checking that Affine-ctEdwards coordinates are on the curve

#cctedvalidate

To check that (u, v) is a point on the *ctEdwards curve*, the **Sapling** circuit uses 4 constraints:

$$(u) \ \mathsf{X} \ (u) \ = \ (uu)$$
$$(v) \ \mathsf{X} \ (v) \ = \ (vv)$$
$$(uu) \ \mathsf{X} \ (vv) \ = \ (uuvv)$$
$$(a_{\mathbb{J}} \cdot uu + vv) \ \mathsf{X} \ (1) \ = \ (1 + d_{\mathbb{J}} \cdot uuvv)$$

Non-normative note: The last two constraints can be combined into $(d_{\mathbb{J}} \cdot uu) \ \mathsf{X} \ (vv) \ = \ (a_{\mathbb{J}} \cdot uu + vv - 1)$. The **Sapling** circuit does not use this optimization.

A.3.3.2 ctEdwards [de]compression and validation

#ccteddecompressvalidate

Define DecompressValidate : CompressedCtEdwardsJubjub $\to$ AffineCtEdwardsJubjub as follows:

DecompressValidate$(\tilde{u}, v)$:

// Prover supplies the u-coordinate.
Let $u : \mathbb{F}_{r_{\mathbb{S}}}$.

// **§? '??'** on p. **??**.
Check that (u, v) is a point on the *ctEdwards curve*.

// **§? '??'** on p. **??**.
Unpack u to $\sum_{i=0}^{254} u_i \cdot 2^i$, equating $\tilde{u}$ with u_0.

// **§? '??'** on p. **??**.
Check that $\sum_{i=0}^{254} u_i \cdot 2^i \leq r_{\mathbb{S}} - 1$.

Return (u, v).

This costs 4 constraints for the curve equation check, 1 constraint for the unpacking, and 387 constraints for the range check (as computed in **§? '??'** on p. **??**) for a total of 392 constraints. The cost of the range check includes boolean-constraining $u_{0\,..\,254}$.

The same *quadratic constraint program* is used for compression and decompression.

Non-normative 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 consistency check on the elliptic curve arithmetic.

A.3.3.3 ctEdwards $\leftrightarrow$ Montgomery conversion

#cctconversion

Define the notation $\sqrt[+]{\bullet}$ as in **§? '??'** on p. **??**.

Define CtEdwardsToMont : AffineCtEdwardsJubjub $\to$ AffineMontJubjub as follows:

$$\mathsf{CtEdwardsToMont}(u, v) = \left(\frac{1+v}{1-v}, \ \sqrt[+]{-40964} \cdot \frac{1+v}{(1-v) \cdot u}\right) \qquad [1 - v \neq 0 \ \text{ and } \ u \neq 0]$$

Define MontToCtEdwards : AffineMontJubjub $\to$ AffineCtEdwardsJubjub as follows:

$$\mathsf{MontToCtEdwards}(x, y) = \left(\sqrt[+]{-40964} \cdot \frac{x}{y}, \ \frac{x-1}{x+1}\right) \qquad [x + 1 \neq 0 \ \text{ and } \ y \neq 0]$$

Either of these conversions can be implemented by the same *quadratic constraint program*:

$$(y) \;\mathsf{X}\; (u) \;=\; \left(\sqrt[+]{-40964} \cdot x\right)$$
$$(x+1) \;\mathsf{X}\; (v) \;=\; (x-1)$$

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.3.2. *Exceptional points (ctEdwards → Montgomery).* #thmconversiontomontnoexcept

Let (u, v) be an affine point on a ctEdwards curve $E_{\mathsf{ctEdwards}(a,d)}$. Then the only points with $u = 0$ or $1 - v = 0$ are $(0, 1) = \mathcal{O}_{\mathbb{J}}$, and $(0, -1)$ of order 2.

Proof. The curve equation is $a \cdot u^2 + v^2 = 1 + d \cdot u^2 \cdot v^2$ with $a \neq d$ (see [**BBJLP2008**]). By substituting $u = 0$ we obtain $v = \pm 1$, and by substituting $v = 1$ and using $a \neq d$ we obtain $u = 0$. □

Theorem A.3.3. *Exceptional points (Montgomery → ctEdwards).* #thmconversiontoedwardsnoexcept

Let (x, y) be an affine point on a Montgomery curve $E_{\mathsf{Mont}(A,B)}$ over $\mathbb{F}_r$ with parameters A and B such that $A^2 - 4$ is a nonsquare in $\mathbb{F}_r$, that is birationally equivalent to a ctEdwards 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 **??** on p. **??**.

If $x + 1 = 0$, then subtituting $x = -1$ into the *Montgomery curve* equation gives $B \cdot y^2 = x^3 + A \cdot x^2 + x = A - 2$. So in that case $y^2 = (A - 2)/B$. The right-hand-side is equal to the parameter d of a particular *ctEdwards curve* birationally equivalent to the *Montgomery curve* (see [**BL2017**]). For all *ctEdwards curves*, d is nonsquare, so this equation has no solutions for y, hence $x + 1 \neq 0$. □

(When the theorem is applied with $E_{\mathsf{Mont}(A,B)} = \mathbb{M}$ defined in **§? '??'** on p. **??**, the *ctEdwards curve* referred to in the proof is an isomorphic rescaling of the Jubjub curve.)

A.3.3.4 Affine-Montgomery arithmetic

#cctmontarithmetic

The incomplete *affine-Montgomery* addition formulae given in [**BL2017**] are:

$$x_3 = B_{\mathbb{M}} \cdot \lambda^2 - A_{\mathbb{M}} - x_1 - x_2$$
$$y_3 = (x_1 - x_3) \cdot \lambda - y_1$$
$$\text{where } \lambda = \begin{cases} \frac{3 \cdot x_1^2 + 2 \cdot A_{\mathbb{M}} \cdot x_1 + 1}{2 \cdot B_{\mathbb{M}} \cdot y_1}, & \text{if } x_1 = x_2 \\ \frac{y_2 - y_1}{x_2 - x_1}, & \text{otherwise.} \end{cases}$$

The following theorem helps to determine when these incomplete addition formulae can be safely used:

Theorem A.3.4. *Distinct-x theorem.* #thmdistinctx

Let Q be a point of odd-prime order s on a Montgomery curve $\mathbb{M} = E_{\mathsf{Mont}(A_{\mathbb{M}}, B_{\mathbb{M}})}$ over $\mathbb{F}_{r_{\mathbb{S}}}$. 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 $k_2 \neq \pm k_1$. Then the non-unified addition constraints

$$(x_2 - x_1) \;\mathsf{X}\; (\lambda) \;=\; (y_2 - y_1)$$
$$(B_{\mathbb{M}} \cdot \lambda) \;\mathsf{X}\; (\lambda) \;=\; (A_{\mathbb{M}} + x_1 + x_2 + x_3)$$
$$(x_1 - x_3) \;\mathsf{X}\; (\lambda) \;=\; (y_3 + y_1)$$

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 that $x_1 \neq x_2$. (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 $k : \left\{-\frac{s-1}{2} \,..\, \frac{s-1}{2}\right\} \mapsto [k]\,Q : \mathbb{M}$ is injective and $k_{1\,..\,2}$ are in $\left\{-\frac{s-1}{2} \,..\, \frac{s-1}{2}\right\}$, 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 $\left\{1 \,..\, \frac{s-1}{2}\right\}$ then it is sufficient to require $k_2 \neq k_1$, since that implies $k_2 \neq \pm k_1$.

Affine-Montgomery doubling can be implemented as:

$$
\begin{aligned}
&(x) \;\mathsf{X}\; (x) \;=\; (xx)\\
&(2{\cdot}B_{\mathbb{M}}{\cdot}y) \;\mathsf{X}\; (\lambda) \;=\; (3{\cdot}xx + 2{\cdot}A_{\mathbb{M}}{\cdot}x + 1)\\
&(B_{\mathbb{M}}{\cdot}\lambda) \;\mathsf{X}\; (\lambda) \;=\; (A_{\mathbb{M}} + 2{\cdot}x + x_3)\\
&(x - x_3) \;\mathsf{X}\; (\lambda) \;=\; (y_3 + y)
\end{aligned}
$$

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 **??** on p. **??**.

A.3.3.5 Affine-ctEdwards arithmetic

#cctedarithmetic

Formulae for *affine-ctEdwards* addition are given in [**BBJLP2008**]. 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**]:

$$
\begin{aligned}
&(u_1 + v_1) \;\mathsf{X}\; (v_2 - a_{\mathbb{J}}{\cdot}u_2) \;=\; (T)\\
&(u_1) \;\mathsf{X}\; (v_2) \;=\; (A)\\
&(v_1) \;\mathsf{X}\; (u_2) \;=\; (B)\\
&(d_{\mathbb{J}}{\cdot}A) \;\mathsf{X}\; (B) \;=\; (C)\\
&(1 + C) \;\mathsf{X}\; (u_3) \;=\; (A + B)\\
&(1 - C) \;\mathsf{X}\; (v_3) \;=\; (T - A + a_{\mathbb{J}}{\cdot}B)
\end{aligned}
$$

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-ctEdwards* doubling $[2]\,(u, v) = (u_3, v_3)$ can also be implemented slightly more efficiently as:

$$(u + v)\ \mathsf{X}\ (v - a_{\mathbb{J}} \cdot u)\ =\ (T)$$
$$(u)\ \mathsf{X}\ (v)\ =\ (A)$$
$$(d_{\mathbb{J}} \cdot A)\ \mathsf{X}\ (A)\ =\ (C)$$
$$(1 + C)\ \mathsf{X}\ (u_3)\ =\ (2 \cdot A)$$
$$(1 - C)\ \mathsf{X}\ (v_3)\ =\ (T + (a_{\mathbb{J}} - 1) \cdot A)$$

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-ctEdwards nonsmall-order check

#cctednonsmallorder

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 (**§? '??'** on p. **??**), 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, the **Sapling** circuit doubles three times (as in **§? '??'** on p. **??**) and checks that the resulting u-coordinate is not 0 (as in **§? '??'** on p. **??**).

On a *ctEdwards curve*, only the zero point $\mathcal{O}_{\mathbb{J}}$, and the unique point of order 2 at $(0, -1)$ have zero u-coordinate. The point of order 2 cannot occur as the result of three doublings. So this u-coordinate check rejects only $\mathcal{O}_{\mathbb{J}}$.

The total cost, including the curve check, is $4 + 3 \cdot 5 + 1 = 20$ constraints.

Note: This ***does not*** ensure that the point is in the prime-order subgroup.

Non-normative notes:

- It would have been sufficient to do two doublings rather than three, because the check that the u-coordinate is nonzero would reject both $\mathcal{O}_{\mathbb{J}}$ and the point of order 2.
- It is possible to reduce the cost to 8 constraints by eliminating the redundant constraint in the curve point check (as mentioned in **§? '??'** on p. **??**); merging the first doubling with the curve point check; and then optimizing the second doubling based on the fact that we only need to check whether the resulting u-coordinate is zero. The **Sapling** circuit does not use these optimizations.

A.3.3.7 Fixed-base Affine-ctEdwards scalar multiplication

#cctfixedscalarmult

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:

$$(s_1)\ \mathsf{X}\ (s_2)\ =\ (s_{\&})$$

$$\begin{aligned}(s_0)\ \mathsf{X}\ \big(&-u_0{\cdot}s_{\&} + u_0{\cdot}s_2 + u_0{\cdot}s_1 - u_0 + u_2{\cdot}s_{\&} - u_2{\cdot}s_1 + u_4{\cdot}s_{\&} - u_4{\cdot}s_2 - u_6{\cdot}s_{\&}\\ &+ u_1{\cdot}s_{\&} - u_1{\cdot}s_2 - u_1{\cdot}s_1 + u_1 - u_3{\cdot}s_{\&} + u_3{\cdot}s_1 - u_5{\cdot}s_{\&} + u_5{\cdot}s_2 + u_7{\cdot}s_{\&}\big) =\\ \big(&u_s - u_0{\cdot}s_{\&} + u_0{\cdot}s_2 + u_0{\cdot}s_1 - u_0 + u_2{\cdot}s_{\&} - u_2{\cdot}s_1 + u_4{\cdot}s_{\&} - u_4{\cdot}s_2 - u_6{\cdot}s_{\&}\big)\end{aligned}$$

$$\begin{aligned}(s_0)\ \mathsf{X}\ \big(&-v_0{\cdot}s_{\&} + v_0{\cdot}s_2 + v_0{\cdot}s_1 - v_0 + v_2{\cdot}s_{\&} - v_2{\cdot}s_1 + v_4{\cdot}s_{\&} - v_4{\cdot}s_2 - v_6{\cdot}s_{\&}\\ &+ v_1{\cdot}s_{\&} - v_1{\cdot}s_2 - v_1{\cdot}s_1 + v_1 - v_3{\cdot}s_{\&} + v_3{\cdot}s_1 - v_5{\cdot}s_{\&} + v_5{\cdot}s_2 + v_7{\cdot}s_{\&}\big) =\\ \big(&v_s - v_0{\cdot}s_{\&} + v_0{\cdot}s_2 + v_0{\cdot}s_1 - v_0 + v_2{\cdot}s_{\&} - v_2{\cdot}s_1 + v_4{\cdot}s_{\&} - v_4{\cdot}s_2 - v_6{\cdot}s_{\&}\big)\end{aligned}$$

For a full-length (252-bit) scalar this costs 3 constraints for each of 84 window lookups, plus 6 constraints for each of 83 ctEdwards additions (as in ***§? '??'*** on p. **??**), for a total of 750 constraints.

Fixed-base scalar multiplication is also used in two places with shorter scalars:

- ***§? '??'*** on p. **??** uses 64 bits for the v input to $\mathsf{ValueCommit}^{\mathsf{Sapling}}$, requiring 22 windows at a cost of $3{\cdot}22 - 1 + 6{\cdot}21 = 191$ constraints;
- ***§? '??'*** on p. **??** uses 32 bits for the pos input to MixingPedersenHash, requiring 11 windows at a cost of $3{\cdot}11 - 1 + 6{\cdot}10 = 92$ constraints.

None of these costs include the cost of boolean-constraining the scalar.

Non-normative notes:

- It would be more efficient to use arithmetic on the *Montgomery curve*, as in ***§? '??'*** on p. **??**. However since there are only three instances of fixed-base scalar multiplication in the *Spend circuit* and two in the *Output circuit*[9], the additional complexity was not considered justified for **Sapling**.
- For the multiplications with 64-bit and 32-bit scalars, the scalar is padded to a multiple of 3 bits with zeros. This causes the computation of $s_{\&}$ in the lookup for the most significant window to be optimized out, which is where the "-1" comes from in the above cost calculations. No further optimization is done for this lookup.

A.3.3.8 Variable-base Affine-ctEdwards scalar multiplication

#cctvarscalarmult

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 $\mathsf{Base}_0 = B$
let $\mathsf{Acc}_0^u = k_0\ ?\ \mathsf{Base}_0^u : 0$
let $\mathsf{Acc}_0^v = k_0\ ?\ \mathsf{Base}_0^v : 1$

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
  let $\mathsf{Addend}_i^u = k_i\ ?\ \mathsf{Base}_i^u : 0$
  let $\mathsf{Addend}_i^v = k_i\ ?\ \mathsf{Base}_i^v : 1$
  let $\mathsf{Acc}_i = \mathsf{Acc}_{i-1} + \mathsf{Addend}_i$

let $R = \mathsf{Acc}_{250}$.

This costs 5 constraints for each of 250 ctEdwards doublings, 6 constraints for each of 250 ctEdwards additions, and 2 constraints for each of 251 point selections, for a total of 3252 constraints.

[9] A *Pedersen commitment* uses fixed-base scalar multiplication as a subcomponent.

Non-normative 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 **§? '??'** on p. **??**. 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

#cctpedersenhash

The specification of the *Pedersen hashes* used in **Sapling** is given in **§? '??'** on p. **??**. It is based on the scheme from [**CvHP1991**] –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 *note 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 a ctEdwards 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 **§? '??'** on p. **??** 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 *ctEdwards curve* and summing the intermediate results using ctEdwards addition.

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 $\langle \bullet \rangle$ and $\mathcal{I}(D, j)$ are defined as in **§? '??'** on p. **??**.

We have to prove that:

- the Montgomery-to-ctEdwards conversions can be implemented without exceptional cases;
- the *distinct-x criterion* is met for all Montgomery additions within a segment.

The proof of Theorem **??** on p. **??** showed that all indices of addition inputs are in the range $\left\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\right\} \setminus \{0\}$.

Because the $\mathcal{I}(D, j)$ (which are outputs of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$) 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 ctEdwards form are of prime order. From Theorem **??** on p. **??**, 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.3.5. *Concerning addition inputs in the Pedersen circuit.* #thmpedersendistinctabsindices

For all disjoint nonempty subsets S and S' of $\{1\,..\,c\}$, all $m \in \mathbb{B}^{[3][c]}$, and all $\Theta \in \{-1, 1\}$:

$$\sum_{j \in S} \mathsf{enc}(m_j) \cdot 2^{4\cdot(j-1)} \neq \Theta \cdot \sum_{j' \in S'} \mathsf{enc}(m_{j'}) \cdot 2^{4\cdot(j'-1)}.$$

Proof. Suppose for a contradiction that S, S', m, Θ is a counterexample. Taking the multiplication by Θ on the right hand side inside the summation, we have:

$$\sum_{j \in S} \mathsf{enc}(m_j) \cdot 2^{4\cdot(j-1)} = \sum_{j' \in S'} \Theta \cdot \mathsf{enc}(m_{j'}) \cdot 2^{4\cdot(j'-1)}.$$

Define $\mathsf{enc}' : \{-1, 1\} \times \mathbb{B}^{[3]} \to \{0\,..\,8\} \setminus \{4\}$ as $\mathsf{enc}'_\theta(m_i) := 4 + \theta \cdot \mathsf{enc}(m_i)$.

Let $\Delta = 4 \cdot \sum_{i=1}^{c} 2^{4\cdot(i-1)}$ as in the proof of Theorem **??** on p. **??**. By adding Δ to both sides, we get

$$\sum_{j \in S} \mathsf{enc}'_1(m_j) \cdot 2^{4\cdot(j-1)} + \sum_{j \in \{1\,..\,c\} \setminus S} 4 \cdot 2^{4\cdot(j-1)} = \sum_{j' \in S'} \mathsf{enc}'_\Theta(m_{j'}) \cdot 2^{4\cdot(j'-1)} + \sum_{j' \in \{1\,..\,c\} \setminus S'} 4 \cdot 2^{4\cdot(j'-1)}$$

where all of the $\mathsf{enc}'_1(m_j)$ and $\mathsf{enc}'_\Theta(m_{j'})$ are in $\{0\,..\,8\} \setminus \{4\}$.

Each term on the left and on the right affects the single hex digit indexed by j and j' respectively. Since S and S' are disjoint subsets of $\{1\,..\,c\}$ and S is nonempty, $S \cap (\{1\,..\,c\} \setminus S')$ is nonempty. Therefore the left hand side has at least one hex digit not equal to 4 such that the corresponding right hand side digit is 4; contradiction. □

This implies that the terms in the Montgomery addition –as well as any intermediate results formed from adding a distinct subset of terms– have distinct indices disregarding sign, hence distinct x-coordinates by Theorem **??** on p. **??**. (We make no assumption about the order of additions.)

We now describe the subcircuit used to process each chunk, which contributes most of the constraint cost of the hash. This subcircuit is used to perform a lookup of a Montgomery point in a 2-bit window table, conditionally negate the result, and add it to an accumulator holding another Montgomery point.

Suppose that the bits of the chunk, $[s_0, s_1, s_2]$, are already boolean-constrained.

We aim to compute $C = A + [(1 - 2 \cdot s_2) \cdot (1 + s_0 + 2 \cdot s_1)]\, P$ for some fixed base point P and accumulated sum A.

We first compute $s_\& = s_0 \,\&\, s_1$:

$$(s_0) \;\mathsf{X}\; (s_1) \;=\; (s_\&)$$

Let $(x_k, y_k) = [k]\, P$ for $k \in \{1\,..\,4\}$. Define each coordinate of $(x_S, y_R) = [1 + s_0 + 2 \cdot s_1]\, P$ as a linear combination of s_0, s_1, and $s_\&$:

let $x_S = x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_\&$
let $y_R = y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\&$

We implement the conditional negation as $(2 \cdot y_R) \;\mathsf{X}\; (s_2) \;=\; (y_R - y_S)$. After substitution of y_R this becomes:

$$\begin{aligned}&\big(2 \cdot (y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\&)\big) \;\mathsf{X}\; (s_2) = \\ &\quad \big(y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\& - y_S\big)\end{aligned}$$

Then we substitute x_S into the Montgomery addition constraints from **§? '??'** on p. **??**, as follows:

$$\begin{aligned}
&\big(x_1 + (x_2 - x_1)\cdot s_0 + (x_3 - x_1)\cdot s_1 + (x_4 + x_1 - x_2 - x_3)\cdot s_{\&} - x_A\big) \;\mathsf{X}\; (\lambda) \;=\; \big(y_S - y_A\big)\\
&\big(B_{\mathbb{M}}\cdot\lambda\big) \;\mathsf{X}\; (\lambda) \;=\; \big(A_{\mathbb{M}} + x_A + x_1 + (x_2 - x_1)\cdot s_0 + (x_3 - x_1)\cdot s_1 + (x_4 + x_1 - x_2 - x_3)\cdot s_{\&} + x_C\big)\\
&\big(x_A - x_C\big) \;\mathsf{X}\; (\lambda) \;=\; \big(y_C + y_A\big)
\end{aligned}$$

(In the sapling-crypto implementation, linear combinations are first-class values, so these substitutions do not need to be done "by hand".)

For the first addition in each segment, both sides are looked up and substituted into the Montgomery addition, so the first lookup takes only 2 constraints.

When these hashes are used in the circuit, the first 6 bits of the input are fixed. For example, in the Merkle tree hashes they represent the layer number. This would allow a precomputation for the first two windows, but that optimization is not done in **Sapling**.

The cost of a Pedersen hash over ℓ bits (where ℓ includes the fixed bits) is as follows. The number of chunks is $c = \mathsf{ceiling}\left(\frac{\ell}{3}\right)$ and the number of segments is $n = \mathsf{ceiling}\left(\frac{\ell}{3\cdot 63}\right)$.

The cost is then:

- $2{\cdot}c$ constraints for the lookups;
- $3{\cdot}(c - n)$ constraints for incomplete additions on the *Montgomery curve*;
- $2{\cdot}n$ constraints for Montgomery-to-ctEdwards conversions;
- $6{\cdot}(n - 1)$ constraints for ctEdwards additions;

for a total of $5{\cdot}c + 5{\cdot}n - 6$ constraints. This does not include the cost of boolean-constraining inputs.

In particular,

- for the Merkle tree hashes $\ell = 516$, so $c = 172$, $n = 3$, and the cost is 869 constraints;
- when a Pedersen hash is used to implement part of a Pedersen commitment for $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ (**§? '??'** on p. **??**), $\ell = 6 + \ell_{\mathsf{value}} + 2{\cdot}\ell_{\mathbb{J}} = 582$, $c = 194$, and $n = 4$, so the cost of the hash alone is 984 constraints.

A.3.3.10 Mixing Pedersen hash

#cctmixinghash

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

Let $\mathcal{J}^{\mathsf{Sapling}}$ be as defined in **§? '??'** on p. **??**.

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

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

This costs 92 constraints for a scalar multiplication (**§? '??'** on p. **??**), and 6 constraints for a ctEdwards addition (**§? '??'** on p. **??**), for a total of 98 constraints.

A.3.4 Merkle path check

#cctmerklepath

Checking each layer of a Merkle authentication path, as described in **§? '??'** on p. **??**, 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 left and right hash inputs to two sequences of 255 bits;
- compute the Merkle hash for this node.

The unpacking need not be canonical in the sense discussed in **§? '??'** on p. **??**; that is, it is ***not*** necessary to ensure that the left or right inputs to the hash 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.

For each layer, the cost is $1 + 2{\cdot}255$ boolean constraints, 2 constraints for the conditional swap (implemented as two selection constraints), and 869 constraints for the Merkle hash (**§? '??'** on p. **??**), for a total of 1380 constraints.

Non-normative 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

#cctwindowedcommit

We construct *windowed Pedersen commitments* by reusing the Pedersen hash implementation described in **§? '??'** on p. **??**, and adding a randomized point:

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

This can be implemented in:

- $5{\cdot}c + 5{\cdot}n - 6$ constraints for the Pedersen hash applied to $\ell = 6 + \mathsf{length}(s)$ bits, where $c = \mathsf{ceiling}\left(\frac{\ell}{3}\right)$ and $n = \mathsf{ceiling}\left(\frac{\ell}{3 \cdot 63}\right)$;
- 750 constraints for the fixed-base scalar multiplication;
- 6 constraints for the final ctEdwards addition.

When $\mathsf{WindowedPedersenCommit}$ is used to instantiate $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, the cost of the Pedersen hash is 984 constraints as calculated in **§? '??'** on p. **??**, and so the total cost in that case is 1740 constraints. This does not include the cost of boolean-constraining the input s or the randomness r.

A.3.6 Homomorphic Pedersen Commitment

#ccthomomorphiccommit

The *windowed Pedersen commitments* defined in the preceding section are highly efficient, but they do not support the homomorphic property we need when instantiating $\mathsf{ValueCommit}$.

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

$$\mathsf{HomomorphicPedersenCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(D, \mathsf{v}) = [\mathsf{v}]\,\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \texttt{“v”}) + [\mathsf{rcv}]\,\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \texttt{“r”})$$

In the case that we need for $\mathsf{ValueCommit}$, v has 64 bits[10]. This value is given as a bit representation, which does not need to be constrained equal to an integer.

[10] It would be sufficient to use 51 bits, which accomodates the range $\{0 \,..\, \mathsf{MAX\_MONEY}\}$, but the **Sapling** circuit uses 64.

ValueCommit can be implemented in:

- 750 constraints for the 252-bit fixed-base multiplication by rcv;
- 191 constraints for the 64-bit fixed-base multiplication by v;
- 6 constraints for the ctEdwards addition

for a total cost of 947 constraints. This does not include the cost to boolean-constrain the input v or randomness rcv.

A.3.7 BLAKE2s hashes

#cctblake2s

BLAKE2s is defined in [**ANWW2013**]. Its main subcomponent is a "G function", defined as follows:

$G : \{0..9\} \times \{0..2^{32}-1\}^{[4]} \to \{0..2^{32}-1\}^{[4]}$

$G(a, b, c, d, x, y) = (a'', b'', c'', d'')$ where

$$\begin{aligned}
a' &= (a + b + x) \bmod 2^{32} \\
d' &= (d \oplus a') \ggg 16 \\
c' &= (c + d') \bmod 2^{32} \\
b' &= (b \oplus c') \ggg 12 \\
a'' &= (a' + b' + y) \bmod 2^{32} \\
d'' &= (d' \oplus a'') \ggg 8 \\
c'' &= (c' + d'') \bmod 2^{32} \\
b'' &= (b' \oplus c'') \ggg 7
\end{aligned}$$

The following table is used to determine which message words the x and y arguments to G are selected from:

$$\begin{aligned}
\sigma_0 &= [\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15\ ] \\
\sigma_1 &= [\ 14,\ 10,\ 4,\ 8,\ 9,\ 15,\ 13,\ 6,\ 1,\ 12,\ 0,\ 2,\ 11,\ 7,\ 5,\ 3\ ] \\
\sigma_2 &= [\ 11,\ 8,\ 12,\ 0,\ 5,\ 2,\ 15,\ 13,\ 10,\ 14,\ 3,\ 6,\ 7,\ 1,\ 9,\ 4\ ] \\
\sigma_3 &= [\ 7,\ 9,\ 3,\ 1,\ 13,\ 12,\ 11,\ 14,\ 2,\ 6,\ 5,\ 10,\ 4,\ 0,\ 15,\ 8\ ] \\
\sigma_4 &= [\ 9,\ 0,\ 5,\ 7,\ 2,\ 4,\ 10,\ 15,\ 14,\ 1,\ 11,\ 12,\ 6,\ 8,\ 3,\ 13\ ] \\
\sigma_5 &= [\ 2,\ 12,\ 6,\ 10,\ 0,\ 11,\ 8,\ 3,\ 4,\ 13,\ 7,\ 5,\ 15,\ 14,\ 1,\ 9\ ] \\
\sigma_6 &= [\ 12,\ 5,\ 1,\ 15,\ 14,\ 13,\ 4,\ 10,\ 0,\ 7,\ 6,\ 3,\ 9,\ 2,\ 8,\ 11\ ] \\
\sigma_7 &= [\ 13,\ 11,\ 7,\ 14,\ 12,\ 1,\ 3,\ 9,\ 5,\ 0,\ 15,\ 4,\ 8,\ 6,\ 2,\ 10\ ] \\
\sigma_8 &= [\ 6,\ 15,\ 14,\ 9,\ 11,\ 3,\ 0,\ 8,\ 12,\ 2,\ 13,\ 7,\ 1,\ 4,\ 10,\ 5\ ] \\
\sigma_9 &= [\ 10,\ 2,\ 8,\ 4,\ 7,\ 6,\ 1,\ 5,\ 15,\ 11,\ 9,\ 14,\ 3,\ 12,\ 13,\ 0\ ]
\end{aligned}$$

The Initialization Vector is defined as:

$$\begin{aligned}
\mathsf{IV} : \{0..2^{32}-1\}^{[8]} := [\ &\texttt{0x6A09E667},\ \texttt{0xBB67AE85},\ \texttt{0x3C6EF372},\ \texttt{0xA54FF53A} \\
&\texttt{0x510E527F},\ \texttt{0x9B05688C},\ \texttt{0x1F83D9AB},\ \texttt{0x5BE0CD19}\ ]
\end{aligned}$$

The full hash function applied to an 8-byte personalization string and a single 64-byte block, in sequential mode with 32-byte output, can be expressed as follows.

Define BLAKE2s-256 $: (p : \mathbb{B}^{\mathbb{Y}[8]}) \times (x : \mathbb{B}^{\mathbb{Y}[64]}) \to \mathbb{B}^{\mathbb{Y}[32]}$ as:

let $\mathsf{PB} : \mathbb{B}^{\mathbb{Y}[32]} = [32, 0, 1, 1] \,||\, [\texttt{0x00}]^{20} \,||\, p$

let $[\, t_0, t_1, f_0, f_1 \,] : \{0 \,..\, 2^{32}{-}1\}^{[4]} = [\, 0, 0, 0, \texttt{0xFFFFFFFF}, 0 \,]$

let $h : \{0 \,..\, 2^{32}{-}1\}^{[8]} = [\![\, \mathsf{LEOS2IP}_{32}(\mathsf{PB}_{4\cdot i \,..\, 4\cdot i+3}) \oplus \mathsf{IV}_i \text{ for } i \text{ from } 0 \text{ up to } 7 \,]\!]$

let $m : \{0 \,..\, 2^{32}{-}1\}^{[16]} = [\![\, \mathsf{LEOS2IP}_{32}(x_{4\cdot i \,..\, 4\cdot i+3}) \text{ for } i \text{ from } 0 \text{ up to } 15 \,]\!]$

let mutable $v : \{0 \,..\, 2^{32}{-}1\}^{[16]} \leftarrow h \,||\, [\, \mathsf{IV}_0, \mathsf{IV}_1, \mathsf{IV}_2, \mathsf{IV}_3, t_0 \oplus \mathsf{IV}_4, t_1 \oplus \mathsf{IV}_5, f_0 \oplus \mathsf{IV}_6, f_1 \oplus \mathsf{IV}_7 \,]$

for r from 0 up to 9:

set $(v_0, v_4, v_8, \ v_{12}) \leftarrow G(v_0, v_4, v_8, \ v_{12}, m_{\sigma_{r,0}}, \ m_{\sigma_{r,1}} \,)$

set $(v_1, v_5, v_9, \ v_{13}) \leftarrow G(v_1, v_5, v_9, \ v_{13}, m_{\sigma_{r,2}}, \ m_{\sigma_{r,3}} \,)$

set $(v_2, v_6, v_{10}, v_{14}) \leftarrow G(v_2, v_6, v_{10}, v_{14}, m_{\sigma_{r,4}}, \ m_{\sigma_{r,5}} \,)$

set $(v_3, v_7, v_{11}, v_{15}) \leftarrow G(v_3, v_7, v_{11}, v_{15}, m_{\sigma_{r,6}}, \ m_{\sigma_{r,7}} \,)$

set $(v_0, v_5, v_{10}, v_{15}) \leftarrow G(v_0, v_5, v_{10}, v_{15}, m_{\sigma_{r,8}}, \ m_{\sigma_{r,9}} \,)$

set $(v_1, v_6, v_{11}, v_{12}) \leftarrow G(v_1, v_6, v_{11}, v_{12}, m_{\sigma_{r,10}}, m_{\sigma_{r,11}})$

set $(v_2, v_7, v_8, \ v_{13}) \leftarrow G(v_2, v_7, v_8, \ v_{13}, m_{\sigma_{r,12}}, m_{\sigma_{r,13}})$

set $(v_3, v_4, v_9, \ v_{14}) \leftarrow G(v_3, v_4, v_9, \ v_{14}, m_{\sigma_{r,14}}, m_{\sigma_{r,15}})$

return $\mathsf{LEBS2OSP}_{256}(\mathsf{concat}_{\mathbb{B}}([\![\, \mathsf{I2LEBSP}_{32}(h_i \oplus v_i \oplus v_{i+8}) \text{ for } i \text{ from } 0 \text{ up to } 7 \,]\!]))$

In practice the message and output will be expressed as bit sequences. In the **Sapling** circuit, the personalization string will be constant for each use.

Each 32-bit exclusive-or is implemented in 32 constraints, one for each bit position $a \oplus b = c$ as in **§? '??'** on p. **??**.

Additions not involving a message word, i.e. $(a + b) \bmod 2^{32} = c$, are implemented using 33 constraints and a 33-bit equality check: constrain 33 boolean variables $c_{0\,..\,32}$, and then check $\sum_{i=0}^{i=31} (a_i + b_i) \cdot 2^i = \sum_{i=0}^{i=32} c_i \cdot 2^i$.

Additions involving a message word, i.e. $(a + b + m) \bmod 2^{32} = c$, are implemented using 34 constraints and a 34-bit equality check: constrain 34 boolean variables $c_{0\,..\,33}$, and then check $\sum_{i=0}^{i=31} (a_i + b_i + m_i) \cdot 2^i = \sum_{i=0}^{i=33} c_i \cdot 2^i$.

For each addition, only $c_{0\,..\,31}$ are used subsequently.

The equality checks are batched; as many sets of 33 or 34 boolean variables as will fit in a $\mathbb{F}_{r_{\mathbb{S}}}$ field element are equated together using one constraint. This allows 7 such checks per constraint.

Each G evaluation requires 262 constraints:

- $4 \cdot 32 = 128$ constraints for $\oplus$ operations;
- $2 \cdot 33 = 66$ constraints for 32-bit additions not involving message words (excluding equality checks);
- $2 \cdot 34 = 68$ constraints for 32-bit additions involving message words (excluding equality checks).

The overall cost is 21006 constraints:

- $10 \cdot 8 \cdot 262 - 4 \cdot 2 \cdot 32 = 20704$ constraints for 80 G evaluations, excluding equality checks (the deduction of $4 \cdot 2 \cdot 32$ is because v is constant at the start of the first round, so in the first four calls to G, the parameters b and d are constant, eliminating the constraints for the first two XORs in those four calls to G);
- ceiling $\left(\frac{10 \cdot 8 \cdot 4}{7}\right) = 46$ constraints for equality checks;
- $8 \cdot 32 = 256$ constraints for final $v_i \oplus v_{i+8}$ operations (the h_i words are constants so no additional constraints are required to exclusive-or with them).

This cost includes boolean-constraining the hash output bits (done implicitly by the final $\oplus$ operations), but not the message bits.

Non-normative notes:

- The equality checks could be eliminated entirely by substituting each check into a boolean constraint for c_0, for instance, but this optimization is not done in **Sapling**.
- 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 Sapling Spend circuit

#cctsaplingspend

The **Sapling** Spend *statement* is defined in **§? '??'** on p. **??**.

The primary input is

$$\begin{aligned}&\big(\mathsf{rt}^{\mathsf{Sapling}} : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]},\\ &\ \mathsf{cv}^{\mathsf{old}} : \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output},\\ &\ \mathsf{nf}^{\mathsf{old}} : \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{PRFnfSapling}}/8]}},\\ &\ \mathsf{rk} : \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}\big),\end{aligned}$$

which is encoded as $8\,\mathbb{F}_{r_{\mathbb{S}}}$ elements (starting with the fixed element 1 required by Groth16):

$$\big[1, \mathcal{U}(\mathsf{rk}), \mathcal{V}(\mathsf{rk}), \mathcal{U}(\mathsf{cv}^{\mathsf{old}}), \mathcal{V}(\mathsf{cv}^{\mathsf{old}}), \mathsf{LEBS2IP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}\big(\mathsf{rt}^{\mathsf{Sapling}}\big), \mathsf{LEBS2IP}_{254}\big(\mathsf{nf}^{\mathsf{old}}_{\star\ 0\,..\,253}\big), \mathsf{LEBS2IP}_{2}\big(\mathsf{nf}^{\mathsf{old}}_{\star\ 254\,..\,255}\big)\big]$$

where $\mathsf{nf}^{\mathsf{old}}_{\star} = \mathsf{LEOS2BSP}_{\ell_{\mathsf{PRFnfSapling}}}\big(\mathsf{nf}^{\mathsf{old}}\big)$.

The auxiliary input is

$$\begin{aligned}&\big(\mathsf{path} : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sapling}}]},\\ &\ \mathsf{pos} : \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}-1\},\\ &\ \mathsf{g_d} : \mathbb{J},\\ &\ \mathsf{pk_d} : \mathbb{J},\\ &\ \mathsf{v}^{\mathsf{old}} : \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\ &\ \mathsf{rcv}^{\mathsf{old}} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{cm}^{\mathsf{old}} : \mathbb{J},\\ &\ \mathsf{rcm}^{\mathsf{old}} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \alpha : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{ak} : \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public},\\ &\ \mathsf{nsk} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}\big).\end{aligned}$$

$\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ and $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$ are of type $\mathbb{J}$, so we have $\mathsf{cv}^{\mathsf{old}}$, $\mathsf{cm}^{\mathsf{old}}$, rk, $\mathsf{g_d}$, $\mathsf{pk_d}$, and ak that represent Jubjub curve points. However,

- $\mathsf{cv}^{\mathsf{old}}$ will be constrained to an output of $\mathsf{ValueCommit}^{\mathsf{Sapling}}$;
- $\mathsf{cm}^{\mathsf{old}}$ will be constrained to an output of $\mathsf{NoteCommit}^{\mathsf{Sapling}}$;
- rk will be constrained to $[\alpha]\,\mathcal{G}^{\mathsf{Sapling}} + \mathsf{ak}$;
- $\mathsf{pk_d}$ will be constrained to $[\mathsf{ivk}]\,\mathsf{g_d}$

so $\mathsf{cv}^{\mathsf{old}}$, $\mathsf{cm}^{\mathsf{old}}$, rk, and $\mathsf{pk_d}$ do not need to be explicitly checked to be on the curve.

In addition, $\mathsf{nk}\star$ and $\rho\star$ used in **Nullifier integrity** are compressed representations of Jubjub curve points. TODO: explain why these are implemented as **§? '??'** on p. **??** even though the statement spec doesn't explicitly say to do validation.

Therefore we have $\mathsf{g_d}$, ak, nk, and ρ that need to be constrained to valid Jubjub curve points as described in **§? '??'** on p. **??**.

In order to aid in comparing the implementation with the specification, we present the checks needed in the order in which they are implemented in the sapling-crypto code:

Check	Implements	Cost	Reference
ak is on the curve TODO: FIXME also decompressed below	ak $:$ SpendAuthSig$^{\mathsf{Sapling}}$.Public	4	**§?** on p. **??**
ak is not small order	**Small order checks**	16	**§?** on p. **??**
$\alpha\star :\mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	$\alpha : \{0 \mathinner{..} 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} - 1\}$	252	**§?** on p. **??**
$\alpha' = [\alpha\star]\,\mathcal{G}^{\mathsf{Sapling}}$	**Spend authority**	750	**§?** on p. **??**
$\mathsf{rk} = \alpha' + \mathsf{ak}$		6	**§?** on p. **??**
inputize rk TODO: not ccteddecompressvalidate => wrong count	rk $:$ SpendAuthSig$^{\mathsf{Sapling}}$.Public	392?	**§?** on p. **??**
$\mathsf{nsk}\star :\mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	nsk $: \{0 \mathinner{..} 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} - 1\}$	252	**§?** on p. **??**
$\mathsf{nk} = [\mathsf{nsk}\star]\,\mathcal{H}^{\mathsf{Sapling}}$	**Nullifier integrity**	750	**§?** on p. **??**
$\mathsf{ak}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{ak} : \mathbb{J})$	**Diversified address integrity**	392	**§?** on p. **??**
$\mathsf{nk}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$ TODO: spec doesn't say to validate nk since it's calculated	**Nullifier integrity**	392	**§?** on p. **??**
$\mathsf{ivk}\star = \mathsf{I2LEBSP}_{251}\left(\mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}, \mathsf{nk})\right)$ †	**Diversified address integrity**	21006	**§?** on p. **??**
$\mathsf{g_d}$ is on the curve	$\mathsf{g_d} : \mathbb{J}$	4	**§?** on p. **??**
$\mathsf{g_d}$ is not small order	**Small order checks**	16	**§?** on p. **??**
$\mathsf{pk_d} = [\mathsf{ivk}\star]\,\mathsf{g_d}$	**Diversified address integrity**	3252	**§?** on p. **??**
$\mathsf{v}^{\mathsf{old}}_{\star} :\mathbb{B}^{[64]}$	$\mathsf{v}^{\mathsf{old}} : \{0 \mathinner{..} 2^{64} - 1\}$	64	**§?** on p. **??**
$\mathsf{rcv}\star :\mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	rcv $: \{0 \mathinner{..} 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} - 1\}$	252	**§?** on p. **??**
$\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{rcv}}_{(}\mathsf{v}^{\mathsf{old}})$	**Value commitment integrity**	947	**§?** on p. **??**
inputize cv		?	
$\mathsf{rcm}\star :\mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	rcm $: \{0 \mathinner{..} 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} - 1\}$	252	**§?** on p. **??**
$\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g_d}, \mathsf{pk_d}, \mathsf{v}^{\mathsf{old}})$	**Note commitment integrity**	1740	**§?** on p. **??**
$\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm})$	**Merkle path validity**	0	
rt′ is the root of a Merkle tree with leaf cm_u, and authentication path $(\mathsf{path}, \mathsf{pos}\star)$		$32 \cdot 1380$	**§?** on p. **??**
$\mathsf{pos}\star = \mathsf{I2LEBSP}_{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}(\mathsf{pos})$		1	**§?** on p. **??**
if $\mathsf{v}^{\mathsf{old}} \neq 0$ then $\mathsf{rt}' = \mathsf{rt}^{\mathsf{Sapling}}$		1	**§?** on p. **??**
inputize $\mathsf{rt}^{\mathsf{Sapling}}$		?	
$\rho = \mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})$	**Nullifier integrity**	98	**§?** on p. **??**
$\rho\star = \mathsf{repr}_{\mathbb{J}}(\rho)$ TODO: spec doesn't say to validate ρ since it's calculated		392	**§?** on p. **??**
$\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$		21006	**§?** on p. **??**
pack $\mathsf{nf}^{\mathsf{old}}_{0 \mathinner{..} 253}$ and $\mathsf{nf}^{\mathsf{old}}_{254 \mathinner{..} 255}$ into two $\mathbb{F}_{r_{\mathbb{S}}}$ inputs	input encoding	2	**§?** on p. **??**

† This is implemented by taking the output of BLAKE2s-256 as a bit sequence and dropping the most significant 5 bits, not by converting to an integer and back to a bit sequence as literally specified.

Note: The implementation represents $\alpha\star$, $\mathsf{nsk}\star$, $\mathsf{ivk}\star$, $\mathsf{rcm}\star$, $\mathsf{rcv}\star$, and $\mathsf{v}_\star^{\mathsf{old}}$ as bit sequences rather than integers. It represents nf as a bit sequence rather than a byte sequence.

A.5 The Sapling Output circuit

#cctsaplingoutput

The **Sapling** Output *statement* is defined in **§? '??'** on p. **??**.

The primary input is

$$\begin{aligned}&\big(\mathsf{cv}^{\mathsf{new}} \colon \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output},\\ &\ \mathsf{cm}_u \colon \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]},\\ &\ \mathsf{epk} \colon \mathbb{J}\big),\end{aligned}$$

which is encoded as $6\ \mathbb{F}_{r_{\mathbb{S}}}$ elements (starting with the fixed element 1 required by Groth16):

$$\big[1, \mathcal{U}(\mathsf{cv}^{\mathsf{new}}), \mathcal{V}(\mathsf{cv}^{\mathsf{new}}), \mathcal{U}(\mathsf{epk}), \mathcal{V}(\mathsf{epk}), \mathsf{LEBS2IP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}(\mathsf{cm}_u)\ \big]$$

The auxiliary input is

$$\begin{aligned}&(\mathsf{g_d} \colon \mathbb{J},\\ &\ \mathsf{pk}\star_{\mathsf{d}} \colon \mathbb{B}^{[\ell_{\mathbb{J}}]},\\ &\ \mathsf{v}^{\mathsf{new}} \colon \{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\},\\ &\ \mathsf{rcv}^{\mathsf{new}} \colon \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{rcm}^{\mathsf{new}} \colon \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{esk} \colon \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\})\end{aligned}$$

$\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ is of type $\mathbb{J}$, so we have $\mathsf{cv}^{\mathsf{new}}$, epk, and $\mathsf{g_d}$ that represent Jubjub curve points. However,

- $\mathsf{cv}^{\mathsf{new}}$ will be constrained to an output of $\mathsf{ValueCommit}^{\mathsf{Sapling}}$;
- epk will be constrained to $[\mathsf{esk}]\ \mathsf{g_d}$

so $\mathsf{cv}^{\mathsf{new}}$ and epk do not need to be explicitly checked to be on the curve.

Therefore we have only $\mathsf{g_d}$ that needs to be constrained to a valid Jubjub curve point as described in **§? '??'** on p. **??**.

Note: $\mathsf{pk}\star_{\mathsf{d}}$ is *not* checked to be a valid compressed representation of a Jubjub curve point.

In order to aid in comparing the implementation with the specification, we present the checks needed in the order in which they are implemented in the sapling-crypto code:

Check	Implements	Cost	Reference
$\mathsf{v}^{\mathsf{old}}_{\star} : \mathbb{B}^{[64]}$	$\mathsf{v}^{\mathsf{old}} : \{0 \,..\, 2^{64}-1\}$	64	**§? ‘??’** on p. **??**
$\mathsf{rcv}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	$\mathsf{rcv} : \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$	252	**§? ‘??’** on p. **??**
$\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}})$	**Value commitment integrity**	947	**§? ‘??’** on p. **??**
inputize cv		?	
$\mathsf{g}\star_{\mathsf{d}} = \mathsf{repr}_{\mathbb{J}}(\mathsf{g}_{\mathsf{d}} : \mathbb{J})$	**Note commitment integrity**	392	**§? ‘??’** on p. **??**
g_{d} is not small order	**Small order checks**	16	**§? ‘??’** on p. **??**
$\mathsf{esk}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	$\mathsf{esk} : \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$	252	**§? ‘??’** on p. **??**
$\mathsf{epk} = [\mathsf{esk}\star]\,\mathsf{g}_{\mathsf{d}}$	**Ephemeral public key integrity**	3252	**§? ‘??’** on p. **??**
inputize epk		?	
$\mathsf{pk}\star_{\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$	$\mathsf{pk}\star_{\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$	256	**§? ‘??’** on p. **??**
$\mathsf{rcm}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$	$\mathsf{rcm} : \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$	252	**§? ‘??’** on p. **??**
$\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g}_{\mathsf{d}}, \mathsf{pk}_{\mathsf{d}}, \mathsf{v}^{\mathsf{old}})$	**Note commitment integrity**	1740	**§? ‘??’** on p. **??**
pack inputs		?	

Note: The implementation represents $\mathsf{esk}\star$, $\mathsf{pk}\star_{\mathsf{d}}$, $\mathsf{rcm}\star$, $\mathsf{rcv}\star$, and $\mathsf{v}^{\mathsf{old}}_{\star}$ as bit sequences rather than integers.

B Batching Optimizations

#batching

B.1 RedDSA batch validation

#reddsabatchvalidate

The reference validation algorithm for RedDSA signatures is defined in **§? ‘??’** on p. **??**.

Let the RedDSA parameters $\mathbb{G}$ (defining a subgroup $\mathbb{G}^{(r)}$ of order $r_{\mathbb{G}}$, a cofactor $h_{\mathbb{G}}$, a group operation $+$, an additive identity $\mathcal{O}_{\mathbb{G}}$, a bit-length $\ell_{\mathbb{G}}$, a representation function $\mathsf{repr}_{\mathbb{G}}$, and an abstraction function $\mathsf{abst}_{\mathbb{G}}$); $\mathcal{P}_{\mathbb{G}} : \mathbb{G}$; $\ell_{\mathsf{H}} : \mathbb{N}$; $\mathsf{H} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{H}}/8]}}$; and the derived hash function $\mathsf{H}^{\circledast} : \mathbb{B}^{\mathbb{Y}^{[\mathbb{N}]}} \to \mathbb{F}_{r_{\mathbb{G}}}$ be as defined in that section.

Implementations **MAY** alternatively use the optimized procedure described in this section to perform faster validation of a batch of signatures, i.e. to determine whether all signatures in a batch are valid. Its input is a sequence of N *signature batch entries*, each of which is a (*validating key*, message, signature) triple.

Let LEOS2BSP, LEOS2IP, and LEBS2OSP be as defined in **§? ‘??’** on p. **??**.

Define RedDSA.BatchEntry := RedDSA.Public × RedDSA.Message × RedDSA.Signature.

Define $\mathsf{RedDSA.BatchValidate} : (\mathsf{entry}_{0\,..\,N-1} : \mathsf{RedDSA.BatchEntry}^{[N]}) \to \mathbb{B}$ as:

For each $j \in \{0\,..\,N-1\}$:

Let $(\mathsf{vk}_j, M_j, \sigma_j) = \mathsf{entry}_j$.

Let $\underline{R_j}$ be the first $\mathsf{ceiling}\left(\ell_{\mathbb{G}}/8\right)$ bytes of σ_j, and let $\underline{S_j}$ be the remaining $\mathsf{ceiling}\left(\mathsf{bitlength}(r_{\mathbb{G}})/8\right)$ bytes.

Let $R_j = \mathsf{abst}_{\mathbb{G}}\left(\mathsf{LEOS2BSP}_{\ell_{\mathbb{G}}}(\underline{R_j})\right)$, and let $S_j = \mathsf{LEOS2IP}_{8 \cdot \mathsf{length}(\underline{S_j})}(\underline{S_j})$.

Let $\underline{\mathsf{vk}_j} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\left(\mathsf{repr}_{\mathbb{G}}(\mathsf{vk}_j)\right)$.

Let $c_j = \mathsf{H}^{\circledast}(\underline{R_j} \,||\, \underline{\mathsf{vk}_j} \,||\, M_j)$.

Choose random $z_j : \mathbb{F}^*_{r_{\mathbb{G}}} \xleftarrow{R} \{1\,..\,2^{128}-1\}$.

Return 1 if

- for all $j \in \{0\,..\,N-1\}$, $R_j \neq \bot$ and $S_j < r_{\mathbb{G}}$; and
- $[h_{\mathbb{G}}]\left(-\left[\sum_{j=0}^{N-1}(z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\right]\mathcal{P}_{\mathbb{G}} + \sum_{j=0}^{N-1}[z_j]\,R_j + \sum_{j=0}^{N-1}[z_j \cdot c_j \pmod{r_{\mathbb{G}}}]\,\mathsf{vk}_j\right) = \mathcal{O}_{\mathbb{G}}$,

otherwise 0.

The z_j values **MUST** be chosen independently of the *signature batch entries*.

Non-normative note: It is also acceptable to sample each z_j from $\{0\,..\,2^{128}-1\}$, since the probability of obtaining zero for any z_j is negligible.

The performance benefit of this approach arises partly from replacing the per-signature scalar multiplication of the base $\mathcal{P}_{\mathbb{G}}$ with one such multiplication per batch, and partly from using an efficient algorithm for multiscalar multiplication such as Pippinger's method [**Bernstein2001**] or the Bos–Coster method [**deRooij1995**], as explained in [**BDLSY2012**].

Note: Spend authorization signatures (**§? '??'** on p. **??**) and binding signatures (**§? '??'** on p. **??**) use different bases $\mathcal{P}_{\mathbb{G}}$. It is straightforward to adapt the above procedure to handle multiple bases; there will be one $-\left[\sum_j (z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\right]\mathcal{P}$ term for each base $\mathcal{P}$. The benefit of this relative to using separate batches is that the multiscalar multiplication can be extended across a larger batch.

B.2 Groth16 batch verification

#grothbatchverify

The reference verification algorithm for Groth16 proofs is defined in **§? '??'** on p. **??**. The batch verification algorithm in this section applies techniques from [**BFIJSV2010**].

Let $q_{\mathbb{S}}$, $r_{\mathbb{S}}$, $\mathbb{S}^{(r)}_{1,2,T}$, $\mathbb{S}^{(r)*}_{1,2,T}$, $\mathcal{P}_{\mathbb{S}_{1,2,T}}$, $\mathbf{1}_{\mathbb{S}}$, and $\hat{e}_{\mathbb{S}}$ be as defined in **§? '??'** on p. **??**.

Define $\mathsf{MillerLoop}_{\mathbb{S}} : \mathbb{S}^{(r)}_1 \times \mathbb{S}^{(r)}_2 \to \mathbb{S}^{(r)}_T$ and $\mathsf{FinalExp}_{\mathbb{S}} : \mathbb{S}^{(r)}_T \to \mathbb{S}^{(r)}_T$ to be the Miller loop and final exponentiation respectively of the $\hat{e}_{\mathbb{S}}$ pairing computation, so that:

$$\hat{e}_{\mathbb{S}}(P, Q) = \mathsf{FinalExp}_{\mathbb{S}}(\mathsf{MillerLoop}_{\mathbb{S}}(P, Q))$$

where $\mathsf{FinalExp}_{\mathbb{S}}(R) = R^t$ for some fixed t.

Define $\mathsf{Groth16}_{\mathbb{S}}.\mathsf{Proof} := \mathbb{S}^{(r)*}_1 \times \mathbb{S}^{(r)*}_2 \times \mathbb{S}^{(r)*}_1$.

A $\mathsf{Groth16}_{\mathbb{S}}$ proof comprises a tuple $(\pi_A, \pi_B, \pi_C) : \mathsf{Groth16}_{\mathbb{S}}.\mathsf{Proof}$.

Verification of a single $\mathsf{Groth16}_{\mathbb{S}}$ proof against an instance encoded as $a_{0\,..\,\ell} : \mathbb{F}_{r_{\mathbb{S}}}^{[\ell+1]}$ requires checking the equation

$$\hat{e}_{\mathbb{S}}(\pi_A, \pi_B) = \hat{e}_{\mathbb{S}}(\pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\Big(\sum_{i=0}^{\ell} [a_i]\, \Psi_i, \Gamma\Big) \cdot Y$$

where $\Delta = [\delta]\, \mathcal{P}_{\mathbb{S}_2}$, $\Gamma = [\gamma]\, \mathcal{P}_{\mathbb{S}_2}$, $Y = [\alpha \cdot \beta]\, \mathcal{P}_{\mathbb{S}_T}$, and $\Psi_i = \Big[\frac{\beta \cdot u_i(x) + \alpha \cdot v_i(x) + w_i(x)}{\gamma}\Big] \mathcal{P}_{\mathbb{S}_1}$ for $i \in \{0\,..\,\ell\}$ are elements of the verification key, as described (with slightly different notation) in [**Groth2016**].

This can be written as:

$$\hat{e}_{\mathbb{S}}(\pi_A, -\pi_B) \cdot \hat{e}_{\mathbb{S}}(\pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\Big(\sum_{i=0}^{\ell} [a_i]\, \Psi_i, \Gamma\Big) \cdot Y = \mathbf{1}_{\mathbb{S}}.$$

Raising to the power of random $z \neq 0$ gives:

$$\hat{e}_{\mathbb{S}}([z]\, \pi_A, -\pi_B) \cdot \hat{e}_{\mathbb{S}}([z]\, \pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\Big(\sum_{i=0}^{\ell} [z \cdot a_i]\, \Psi_i, \Gamma\Big) \cdot Y^z = \mathbf{1}_{\mathbb{S}}.$$

This justifies the following optimized procedure for performing faster verification of a batch of $\mathsf{Groth16}_{\mathbb{S}}$ proofs. Implementations **MAY** use this procedure to determine whether all proofs in a batch are valid.

Define a type $\mathsf{Groth16}_{\mathbb{S}}.\mathsf{BatchEntry} := \mathsf{Groth16}_{\mathbb{S}}.\mathsf{Proof} \times \mathsf{Groth16}_{\mathbb{S}}.\mathsf{PrimaryInput}$ representing *proof batch entries*.

Define $\mathsf{Groth16}_{\mathbb{S}}.\mathsf{BatchVerify} : (\mathsf{entry}_{0\,..\,N-1} : \mathsf{Groth16}_{\mathbb{S}}.\mathsf{BatchEntry}^{[N]}) \to \mathbb{B}$ as:

For each $j \in \{0\,..\,N-1\}$:

Let $((\pi_{j,A},\, \pi_{j,B},\, \pi_{j,C}),\ a_{j,\,0\,..\,\ell}) = \mathsf{entry}_j$.

Choose random $z_j : \mathbb{F}_{r_{\mathbb{S}}}^{*} \xleftarrow{R} \{1\,..\,2^{128} - 1\}$.

Let $\mathsf{Accum}_{AB} = \prod_{j=0}^{N-1} \mathsf{MillerLoop}_{\mathbb{S}}\big([z_j]\, \pi_{j,A}, -\pi_{j,B}\big)$.

Let $\mathsf{Accum}_{\Delta} = \sum_{j=0}^{N-1} [z_j]\, \pi_{j,C}$.

Let $\mathsf{Accum}_{\Gamma,i} = \sum_{j=0}^{N-1} (z_j \cdot a_{j,i}) \pmod{r_{\mathbb{S}}}$ for $i \in \{0\,..\,\ell\}$.

Let $\mathsf{Accum}_{Y} = \sum_{j=0}^{N-1} z_j \pmod{r_{\mathbb{S}}}$.

Return 1 if

$$\mathsf{FinalExp}_{\mathbb{S}}\Big(\mathsf{Accum}_{AB} \cdot \mathsf{MillerLoop}_{\mathbb{S}}\big(\mathsf{Accum}_{\Delta}, \Delta\big) \cdot \mathsf{MillerLoop}_{\mathbb{S}}\Big(\sum_{i=0}^{\ell} [\mathsf{Accum}_{\Gamma,i}]\, \Psi_i, \Gamma\Big)\Big) \cdot Y^{\mathsf{Accum}_Y} = \mathbf{1}_{\mathbb{S}},$$

otherwise 0.

The z_j values **MUST** be chosen independently of the *proof batch entries*.

Non-normative note: It is also acceptable to sample each z_j from $\{0\,..\,2^{128} - 1\}$, since the probability of obtaining zero for any z_j is negligible.

The performance benefit of this approach arises from computing two of the three Miller loops, and the final exponentation, per batch instead of per proof. For the multiplications by z_j, an efficient algorithm for multiscalar multiplication such as Pippinger's method [**Bernstein2001**] or the Bos–Coster method [**deRooij1995**] may be used.

Note: Spend proofs (of the *statement* in **§? '??'** on p. **??**) and output proofs (of the *statement* in **§? '??'** on p. **??**) use different verification keys, with different parameters Δ, Γ, Y, and $\Psi_{0\,..\,\ell}$. It is straightforward to adapt the above procedure to handle multiple verification keys; the accumulator variables Accum_{Δ}, $\mathsf{Accum}_{\Gamma,i}$, and Accum_{Y} are duplicated, with one term in the verification equation for each variable, while Accum_{AB} is shared.

Neglecting multiplications in $\mathbb{S}_T^{(r)}$ and $\mathbb{F}_{r_{\mathbb{S}}}$, and other trivial operations, the cost of batched verification is therefore

- for each proof: the cost of decoding the proof representation to the form $\mathsf{Groth16}_{\mathbb{S}}.\mathsf{Proof}$, which requires three point decompressions and three subgroup checks (two for $\mathbb{S}_1^{(r)*}$ and one for $\mathbb{S}_2^{(r)*}$);
- for each successfully decoded proof: a Miller loop; and a 128-bit scalar multiplication by z_j in $\mathbb{S}_1^{(r)}$;
- for each verification key: two Miller loops; an exponentiation in $\mathbb{S}_T^{(r)}$; a multiscalar multiplication in $\mathbb{S}_1^{(r)}$ with N 128-bit scalars to compute Accum_{Δ}; and a multiscalar multiplication in $\mathbb{S}_1^{(r)}$ with $\ell + 1$ 255-bit scalars to compute $\sum_{i=0}^{\ell} [\mathsf{Accum}_{\Gamma,i}]\, \Psi_i$;
- one final exponentiation.

List of Theorems and Lemmata

#theorems