# Zcash Protocol Specification

Version v2021.1.19-6-g75a8a9 [NU5 proposal]

Daira Hopwood[†]

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

March 22, 2021

[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.

This draft specification defines the **Zcash** consensus protocol at launch; after each of the upgrades codenamed **Overwinter**, **Sapling**, **Blossom**, **Heartwood**, and **Canopy**; and proposed changes for **NU5**. It is a work in progress. Protocol differences from **Zerocash** and **Bitcoin** are also explained.

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

## Contents 1

[†] Electric Coin Company

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

# 1 Introduction

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

The most significant changes from the original **Zerocash** are explained in § 8 *'Differences from the **Zerocash** paper'* on p. 132.

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

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

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

Changes specific to the **Heartwood** upgrade following **Blossom** are highlighted in orange.

Changes specific to the **Canopy** upgrade following **Heartwood** are highlighted in purple.

Changes specific to the **NU5** proposal following **Canopy** are highlighted in slate blue.

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.

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

The key words **MUST**, **MUST NOT**, **SHOULD**, **SHOULD NOT**, **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.

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

**Zcash** security depends on consensus. Should a program interacting with the **Zcash** network diverge from consensus, its security will be weakened or destroyed. The cause of the divergence doesn't matter: it could be a bug in your program, it could be an error in this documentation which you implemented as described, or it could be that you do everything right but other software on the network behaves unexpectedly. The specific cause will not matter to the users of your software whose wealth is lost.

Having said that, a specification of ***intended*** behaviour is essential for security analysis, understanding of the protocol, and maintenance of **Zcash** and related software. If you find any mistake in this specification, please file an issue at `https://github.com/zcash/zips/issues` or contact `<security@z.cash>`.

## 1.2 High-level Overview

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

Value in **Zcash** is either *transparent* or *shielded*. Transfers of *transparent* value work essentially as in **Bitcoin** and have the same privacy properties. *Shielded* value is carried by *notes*[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*[2] unique to that *note*. Computing the *nullifier* requires the associated private *spending key* (or the *nullifier deriving key* for **Sapling** or **Orchard** *notes*). 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*, *Spend descriptions*, *Output descriptions* and *Action descriptions*. Together these describe *shielded transfers* which take in *shielded input notes*, and/or produce *shielded output notes*. (For **Sprout**, each *JoinSplit description* handles up to two *shielded inputs* and up to two *shielded outputs*. For **Sapling**, each *shielded input* or *shielded output* has its own description. For **Orchard**, each *Action description* handles up to one *shielded input* and up to one *shielded output*.) It is also possible for value to be transferred between the *transparent* and *shielded* domains.

The *nullifiers* of the input *notes* are revealed (preventing them from being spent again) and the commitments of the output *notes* are revealed (allowing them to be spent in future). A *transaction* also includes computationally sound *zk-SNARK* proofs and signatures, which prove that all of the following hold except with insignificant probability:

For each *shielded input*,

- [**Sapling** onward] there is a revealed *value commitment* to the same value as the input *note*;[3]
- 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*,

- [**Sapling** onward] there is a revealed *value commitment* to the same value as the output *note*;[3]
- 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. For **Sapling** and **Orchard**, the *value commitments* corresponding to the inputs and outputs are checked to balance (together with any net *transparent* input or output) outside the *zk-SNARK*.

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

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

[3] For **Orchard**, each Action reveals a single *value commitment* to the net value spent by the Action, rather than one *value commitment* for the input *note* and one for the output *note*.

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

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

In **Sapling** and **Orchard**, for each *spending key* there is a *full viewing key* that allows recognizing both incoming and outgoing *notes* without having spend authority. This is implemented by an additional ciphertext in each *Output description* or *Action description*.

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.[4] 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

$\mathbb{B}$ 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 $S \rightarrow T$. An argument to a function can determine other argument or result types.

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

Initial arguments to a function or randomized algorithm may be written as subscripts, e.g. if $x ⦂ X$, $y ⦂ Y$, and $f ⦂ X \times Y \rightarrow Z$, 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 $T \rightarrow U$ 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 $U$.

[4] 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].

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

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

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

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

$\mathsf{truncate}_k(S)$ 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. $[\mathtt{0x00}]^{\ell}$ means the sequence of $\ell$ zero bytes.

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

$[0]^{\ell}$ means the sequence of $\ell$ zero bits. $[1]^{\ell}$ means the sequence of $\ell$ 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 $S$ viewed as bit sequences. If the elements of $S$ are byte sequences, they are converted to bit sequences with the ***most significant*** bit of each byte first.

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

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

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

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

$a + b$ means the sum of $a$ and $b$. This may refer to addition of integers, rationals, finite field elements, or group elements (see § 4.1.9 ***'Represented Group'*** on p. 29) 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.

$\sum\limits_{i=1}^{\mathrm{N}} a_i$ means the sum of $a_{1..\mathrm{N}}$. $\prod\limits_{i=1}^{\mathrm{N}} a_i$ means the product of $a_{1..\mathrm{N}}$. $\bigoplus\limits_{i=1}^{\mathrm{N}} a_i$ 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 $\left\{0 \,..\, \frac{q-1}{2}\right\}$. 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 $\bot$ if no such square root exists.

$b \mathbin{?} x : y$ means $x$ when $b = 1$, or $y$ when $b = 0$.

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

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

The $[k]\,P$ notation for scalar multiplication in a group is defined in § 4.1.9 *‘Represented Group’* on p. 29.

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 $\ell$ such that $2^\ell > x$.

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

The following integer constants will be instantiated in § 5.3 *‘Constants’* on p. 67:

> $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, $\ell_{\mathsf{value}}$, $\ell_{\mathsf{hSig}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell_{\mathsf{PRFexpand}}$, $\ell_{\mathsf{PRFnfSapling}}$, $\ell_{\mathsf{rcm}}$, $\ell_{\mathsf{Seed}}$, $\ell_{\mathsf{a_{sk}}}$, $\ell^{\mathsf{Sprout}}_{\varphi}$, $\ell_{\mathsf{sk}}$, $\ell_{\mathsf{d}}$, $\ell_{\mathsf{dk}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}$, $\ell_{\mathsf{ovk}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}$, $\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}$, $\ell^{\mathsf{Orchard}}_{\mathsf{base}}$, $\mathsf{MAX\_MONEY}$, $\mathsf{BlossomActivationHeight}$, $\mathsf{CanopyActivationHeight}$, $\mathsf{ZIP212GracePeriod}$, $\mathsf{SlowStartInterval}$, $\mathsf{PreBlossomHalvingInterval}$, $\mathsf{MaxBlockSubsidy}$, $\mathsf{NumFounderAddresses}$, $\mathsf{PoWLimit}$, $\mathsf{PoWAveragingWindow}$, $\mathsf{PoWMedianBlockSpan}$, $\mathsf{PoWDampingFactor}$, $\mathsf{PreBlossomPoWTargetSpacing}$, and $\mathsf{PostBlossomPoWTargetSpacing}$.

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

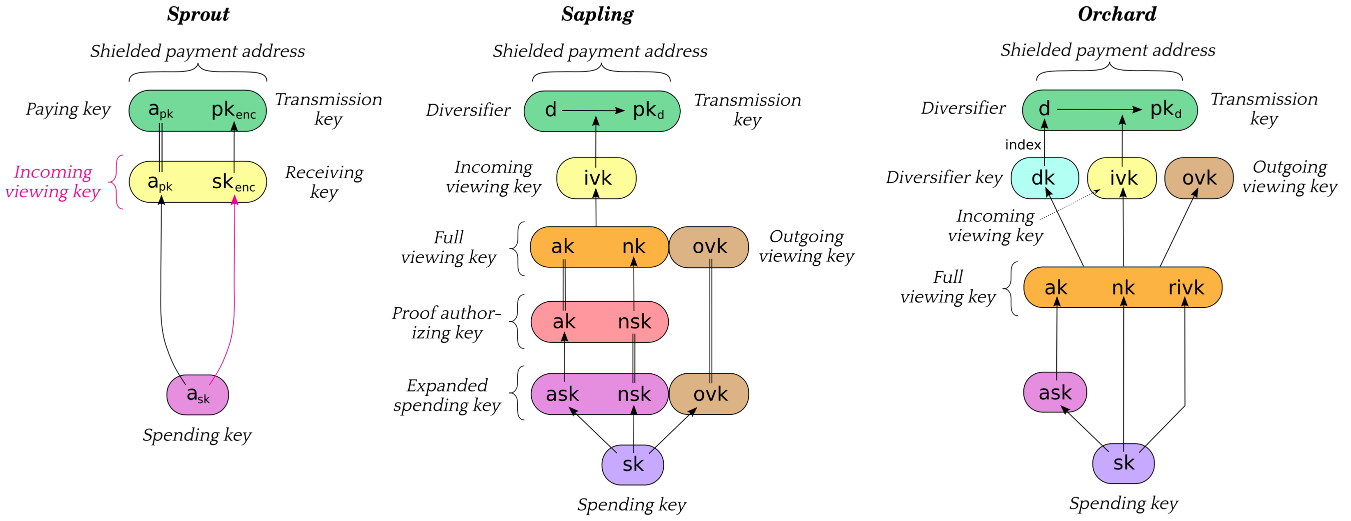
We use the abbreviation “*ctEdwards*” to refer to *complete twisted Edwards elliptic curves* and coordinates (see § 5.4.9.3 ‘Jubjub’ on p. 94).

# 3 Concepts

## 3.1 Payment Addresses and Keys

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** and **Sapling** and **Orchard**. 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* $\mathsf{sk_{enc}}$, *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* $\mathsf{a_{sk}}$, as described in § 4.2.1 *'**Sprout** Key Components'* on p. 32.

[**Sapling** onward] An *expanded spending key* is composed of a *Spend authorizing key* $\mathsf{ask}$, a *nullifier private key* $\mathsf{nsk}$, and an *outgoing viewing key* $\mathsf{ovk}$. From these components we can derive an *proof authorizing key* $(\mathsf{ak}, \mathsf{nsk})$, a *full viewing key* $(\mathsf{ak}, \mathsf{nk}, \mathsf{ovk})$, an *incoming viewing key* $\mathsf{ivk}$, and a set of *diversified payment addresses* $\mathsf{addr_d} = (\mathsf{d}, \mathsf{pk_d})$, as described in § 4.2.2 *'**Sapling** Key Components'* on p. 32.

The consensus protocol does not depend on how an *expanded spending key* is constructed. Two methods of doing so are defined:

1. Generate a *spending key* $\mathsf{sk}$ at random and derive the *expanded spending key* $(\mathsf{ask}, \mathsf{nsk}, \mathsf{ovk})$ from it, as shown in the diagram above and described in § 4.2.2 *'**Sapling** Key Components'* on p. 32.
2. Obtain an *extended spending key* as specified in [ZIP-32]; this includes a superset of the components of an *expanded spending key*. This method is used in the context of a *Hierarchical Deterministic Wallet*.

[**NU5** onward] An **Orchard** *spending key* $\mathsf{sk}$ is used to derive a *Spend authorizing key* $\mathsf{ask}$, and a *full viewing key* $(\mathsf{ak}, \mathsf{nk}, \mathsf{rivk})$. From the *full viewing key* we can also derive a *diversifier key* $\mathsf{dk}$, an *incoming viewing key* $\mathsf{ivk}$, an *outgoing viewing key* $\mathsf{ovk}$, and a set of *diversified payment addresses* $\mathsf{addr_d} = (\mathsf{d}, \mathsf{pk_d})$, as described in § 4.2.3 *'**Orchard** Key Components'* on p. 34.

**Non-normative note:** In zcashd, all **Sapling** and **Orchard** keys and addresses are derived according to [ZIP-32].

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

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

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

**Note:** It is conventional in cryptography to 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 ($\mathsf{pk_{enc}}$ or $\mathsf{pk_d}$) need not be publically distributed; it has the same distribution as the *shielded payment address* itself. As mentioned above, limiting the distribution of the *shielded payment address* is important for some use cases. This also helps to reduce reliance of the overall protocol on the security of the cryptosystem used for *note* encryption (see § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59 and § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60), since an adversary would have to know $\mathsf{pk_{enc}}$ or some $\mathsf{pk_d}$ in order to exploit a hypothetical weakness in that cryptosystem.

## 3.2 Notes

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

Let $\mathsf{MAX\_MONEY}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell_{\mathsf{PRFnfSapling}}$, $\ell_{\mathsf{d}}$, and $\ell_{\mathsf{value}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 5.4.8.1 ***'Sprout Note Commitments'*** on p. 88.

Let $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as defined in § 5.4.8.2 ***'Windowed Pedersen commitments'*** on p. 88.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in § 5.4.5.3 ***'Sapling Key Agreement'*** on p. 82.

Let $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ be as defined in § 5.4.1.6 *'*$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ ***and*** $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ ***Hash Functions'*** on p. 71.

Let $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ be as defined in § 5.4.8.4 ***'Sinsemilla commitments'*** on p. 90.

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as defined in § 5.4.5.5 ***'Orchard Key Agreement'*** on p. 82.

Let $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ be as defined in § 5.4.1.6 *'*$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ ***and*** $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ ***Hash Functions'*** on p. 71.

Let $\mathsf{PRF}^{\mathsf{nfOrchard}}$ be as defined in § 5.4.2 ***'Pseudo Random Functions'*** on p. 79.

Let $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 *'*$\mathsf{Pallas}$ ***and*** $\mathsf{Vesta}$*'* on p. 97.

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

- $\mathsf{a_{pk}} \mathrel{⦂} \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}$ is the *paying key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0 \mathrel{..} \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* ($1$ **ZEC** $= 10^8$ *zatoshi*);
- $\mathsf{\rho} \mathrel{⦂} \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}$ is used as input to $\mathsf{PRF}^{\mathsf{nfSprout}}_{\mathsf{a_{sk}}}$ to derive the *nullifier* of the *note*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as defined in § 4.1.8 ***'Commitment'*** on p. 27.

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

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

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

- $\mathsf{d} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the *diversifier* of the recipient's *shielded payment address*;
- $\mathsf{pk_d} \mathrel{⦂} \mathsf{KA^{Sapling}.PublicPrimeSubgroup}$ is the *diversified transmission key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0\,..\,\mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit^{Sapling}.Trapdoor}$ is a random *commitment trapdoor* as defined in § 4.1.8 *'Commitment'* on p. 27.

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

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

An **Orchard** *note* is a tuple $(\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$, where:

- $\mathsf{d} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the *diversifier* of the recipient's *shielded payment address*;
- $\mathsf{pk_d} \mathrel{⦂} \mathsf{KA^{Orchard}.Public}$ is the *diversified transmission key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}$ is an integer representing the value of the *note* in *zatoshi*;
- $\rho \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}}$ is used as input to $\mathsf{PRF^{nfOrchard}_{nk}}$ as part of deriving the *nullifier* of the *note*;
- $\psi \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}}$ is additional randomness used in deriving the *nullifier*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit^{Orchard}.Trapdoor}$ is a random *commitment trapdoor* as defined in § 4.1.8 *'Commitment'* on p. 27.

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

$$\mathsf{Note^{Orchard}} := \mathbb{B}^{[\ell_{\mathsf{d}}]} \times \mathsf{KA^{Orchard}.Public} \times \{0\,..\,2^{\ell_{\mathsf{value}}}-1\} \times \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{F}_{q_{\mathbb{P}}} \times \mathsf{NoteCommit^{Orchard}.Trapdoor}.$$

Creation of new *notes* is described in § 4.7 *'Sending Notes'* on p. 40. When *notes* are sent, only a commitment (see § 4.1.8 *'Commitment'* on p. 27) 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

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

where $\mathsf{NoteCommit^{Sprout}}$ is instantiated in § 5.4.8.1 *'**Sprout** Note Commitments'* on p. 88.

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

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

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

where $\mathsf{NoteCommit^{Sapling}}$ is instantiated in § 5.4.8.2 *'Windowed Pedersen commitments'* on p. 88.

Notice that the above definition of a **Sapling** *note* does not have a $\rho$ field. There is in fact a $\rho$ value associated with each **Sapling** *note*, but this can only be computed once its position in the *note commitment tree* is known (see § 3.4 *'Transactions and Treestates'* on p. 16 and § 3.8 *'Note Commitment Trees'* on p. 19). We refer to the combination of a *note* and its *note position* $\mathsf{pos}$, as a *positioned note*.

For a *positioned note*, we can compute the value $\rho$ as described in § 4.16 *'Note Commitments and Nullifiers'* on p. 53.

An **Orchard** *note commitment* on a *note* $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$ is computed as

$$\mathsf{g_d} := \mathsf{DiversifyHash}^{\mathsf{Orchard}}(\mathsf{d})$$
$$\mathsf{NoteCommitment}^{\mathsf{Orchard}}(\mathbf{n}) := \mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{P}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{P}}(\mathsf{pk_d}), \mathsf{v}, \rho, \psi)$$

where $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ is instantiated in § 5.4.8.4 ***'Sinsemilla commitments'*** on p. 90.

Unlike in **Sapling**, the definition of an **Orchard** *note* includes the $\rho$ field; the *note*'s position in the *note commitment tree* does not need to be known in order to compute this value.

The *nullifier* of a *note* is denoted $\mathsf{nf}$.

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

A *nullifier* for a **Sapling** *note* is derived from the $\rho$ value and the recipient's *nullifier deriving key* $\mathsf{nk}$.

A *nullifier* for an **Orchard** *note* is derived from the $\rho$ and $\psi$ values, the recipient's *nullifier deriving key* $\mathsf{nk}$, and the *note commitment*.

The *nullifier* computation uses a *Pseudo Random Function* (see § 4.1.2 ***'Pseudo Random Functions'*** on p. 21), as described in § 4.16 ***'Note Commitments and Nullifiers'*** on p. 53.

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

### 3.2.1 Note Plaintexts and Memo Fields

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

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

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

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

[**Sapling** onward] The *note plaintext* in each *Output description* or *Action description* is encrypted to the *diversified payment address* $(\mathsf{d}, \mathsf{pk_d})$.

Each **Sapling** or **Orchard** *note plaintext* (denoted $\mathbf{np}$) consists of

$$(\mathsf{leadByte} ⦂ \mathbb{BY}, \mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\}, \mathsf{rseed} ⦂ \mathbb{BY}^{[32]}, \mathsf{memo} ⦂ \mathbb{BY}^{[512]})$$

The fields $\mathsf{d}$ and $\mathsf{v}$ are as defined in § 3.2 ***'Notes'*** on p. 13.

The field $\mathsf{rseed}$ is described in § 4.7.2 ***'Sending Notes (Sapling)'*** on p. 41.

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

Encodings are given in § 5.5 ***'Encodings of Note Plaintexts and Memo Fields'*** on p. 104. The result of encryption forms part of a *transmitted note(s) ciphertext*. For further details, see § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59 and § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60.

## 3.3 The Block Chain

At a given point in time, each *full validator* is aware of a set of candidate *blocks*. These form a tree rooted at the *genesis block*, where each node in the tree refers to its parent via the `hashPrevBlock` *block header* field (see § 7.6 ***'Block Header Encoding and Consensus'*** on p. 122).

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

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

In order to choose the *best valid block chain* in its view of the overall *block* tree, a node sums the work, as defined in § 7.7.5 ***'Definition of Work'*** on p. 127, of all *blocks* in each *valid block chain*, and considers the *valid block chain* with greatest total work to be best. To break ties between leaf *blocks*, a node will prefer the *block* that it received first.

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

## 3.4 Transactions and Treestates

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

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

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

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

- a *note commitment tree* (§ 3.8 ***'Note Commitment Trees'*** on p. 19);
- a *nullifier set* (§ 3.9 ***'Nullifier Sets'*** on p. 19).

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* (either the **Sprout** tree or the **Sapling** tree or the **Orchard** tree). It uniquely identifies a *note commitment tree* state given the assumed security properties of the Merkle tree's *hash function*. Since the *nullifier set* is always updated together with the *note commitment tree*, this also identifies a particular state of the associated *nullifier set*.

In a given *block chain*, for each of **Sprout** and **Sapling** and **Orchard**, *treestates* are chained as follows:

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

*JoinSplit descriptions* also have interstitial input and output *treestates* for **Sprout**, explained in the following section. There is no equivalent of interstitial *treestates* for **Sapling** or for **Orchard**.

## 3.5 JoinSplit Transfers and Descriptions

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

A *JoinSplit transfer* spends $\mathrm{N}^{\mathsf{old}}$ *notes* $\mathbf{n}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$ and *transparent* input $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$, and creates $\mathrm{N}^{\mathsf{new}}$ *notes* $\mathbf{n}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$ and *transparent* output $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$. It is associated with a *JoinSplit statement* instance (§ 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54), for which it provides a *zk-SNARK proof*.

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

The total $\mathsf{v^{new}_{pub}}$ value adds to, and the total $\mathsf{v^{old}_{pub}}$ 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 § 3.9 ***'Nullifier Sets'*** on p. 19.

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

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

**Consensus rules:**

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

## 3.6 Spend Transfers, Output Transfers, and their Descriptions

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

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

A *Spend transfer* spends a *note* $\mathbf{n}^{\mathsf{old}}$. Its *Spend description* includes a *Pedersen value commitment* to the value of the *note*. It is associated with an instance of a *Spend statement* (§ 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55) for which it provides a *zk-SNARK proof*.

An *Output transfer* creates a *note* $\mathbf{n}^{\mathsf{new}}$. Similarly, its *Output description* includes a *Pedersen value commitment* to the *note* value. It is associated with an instance of an *Output statement* (§ 4.17.3 ***'Output Statement (Sapling)'*** on p. 56) for which it provides a *zk-SNARK proof*.

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

To ensure balance, we use a homomorphic property of *Pedersen commitments* that allows them to be added and subtracted, as elliptic curve points (§ 5.4.8.3 ***'Homomorphic Pedersen commitments (Sapling and Orchard)'*** on p. 89). The result of adding two *Pedersen value commitments*, committing to values $\mathsf{v}_1$ and $\mathsf{v}_2$, is a new *Pedersen value commitment* that commits to $\mathsf{v}_1 + \mathsf{v}_2$. Subtraction works similarly.

Therefore, balance can be enforced by adding all of the *value commitments* for *shielded inputs*, subtracting all of the *value commitments* for *shielded outputs*, and proving by use of a *Sapling binding signature* (as described in § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47) that the result commits to a value consistent with the net *transparent* value change. This approach allows all of the *zk-SNARK statements* to be independent of each other, potentially increasing opportunities for precomputation.

A *Spend description* specifies an *anchor*, which refers to the output **Sapling** *treestate* of a previous *block*. It also reveals a *nullifier*, which allows detection of double-spends as described in § 3.9 ***'Nullifier Sets'*** on p. 19.

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

**Consensus rules:**

- The *Spend transfers* and *Action transfers* of a *transaction* **MUST** be consistent with its $\mathsf{v^{balanceSapling}}$ value as specified in § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47.
- The *anchor* of each *Spend description* **MUST** refer to some earlier *block*'s final **Sapling** *treestate*. The *anchor* is encoded separately in each *Spend description* for v4 *transactions*, or encoded once and shared between all *Spend descriptions* in a v5 *transaction*.

## 3.7 Action Transfers and their Descriptions

**Orchard** introduces *Action transfers*, each of which can optionally perform a spend, and optionally perform an output.

*Action descriptions* are data included in a *transaction* that describe *Action transfers*.

An *Action transfer* spends a *note* $\mathbf{n}^{\mathsf{old}}$, and creates a *note* $\mathbf{n}^{\mathsf{new}}$. Its *Action description* includes a *Pedersen value commitment* to the net value, i.e. the value of the spent *note* minus the value of the created *note*. It is associated with an instance of an *Action statement* (§ 4.17.4 ***'Action Statement (Orchard)'*** on p. 57) for which it provides a *zk-SNARK proof* .

Each version 5 *transaction* has a sequence of *Action descriptions*. Version 4 *transactions* cannot contain *Action descriptions*.

As in **Sapling**, we use the homomorphic property of *Pedersen commitments* to enforce balance: we add all of the *value commitments* and prove by use of an *Orchard binding signature* that the result commits to a value consistent with the net *transparent* value change (as described in § 4.14 ***'Balance and Binding Signature (Orchard)'*** on p. 50). This approach allows all of the *zk-SNARK statements* to be independent of each other, potentially increasing opportunities for precomputation.

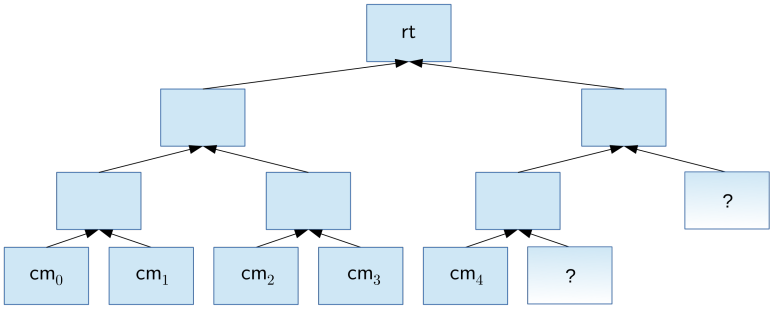
The fields of an *Action description* are essentially a merger of the fields of a *Spend description* and an *Output description*, but with only a single *value commitment*. Also, the *zk-SNARK proof* is encoded outside the *Action description*, in order to more easily take advantage of space and performance optimizations in the Halo 2 proof system (§ 5.4.10.3 'Halo 2' on p. 103) that apply when multiple proofs are aggregated. An *Action description* does not include an *anchor*, because that is encoded once in the `anchorOrchard` field of the *transaction*.

**Non-normative note:** As with **Sapling**, interstitial *treestates* are not necessary for **Orchard**, because an *Action transfer* in a given *transaction* cannot spend any of the *shielded outputs* of the same *transaction*.

**Consensus rules:**

- The *Action transfers* of a *transaction* **MUST** be consistent with its $\mathsf{v^{balanceOrchard}}$ value as specified in § 4.14 ***'Balance and Binding Signature (Orchard)'*** on p. 50.
- The `anchorOrchard` field of the *transaction*, whenever it exists (i.e. when there are any *Action descriptions*), **MUST** refer to some earlier *block*'s final **Orchard** *treestate*.

## 3.8 Note Commitment Trees



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

A *root* of a *note commitment tree* is associated with each *treestate* (§ 3.4 ***'Transactions and Treestates'*** on p. 16).

Each *node* in the *incremental Merkle tree* is associated with a *hash value* of size $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$ or $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$ or $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}$ 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 ($\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ or $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ or $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$) is called its *note position*.

## 3.9 Nullifier Sets

Each *full validator* maintains a *nullifier set* logically associated with each *treestate*. As valid *transactions* containing *JoinSplit transfers* or *Spend transfers* or *Action transfers* are processed, the *nullifiers* revealed in *JoinSplit descriptions* and *Spend descriptions* and *Action 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*. **Sprout** and **Sapling** and **Orchard** *nullifiers* are considered disjoint, even if they have the same bit pattern.

## 3.10 Block Subsidy, Funding Streams, and Founders' Reward

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

[Pre-**Canopy**] The *block subsidy* is composed of a *miner subsidy* and a *Founders' Reward*.

[**Canopy** onward] The *block subsidy* is composed of a *miner subsidy* and a series of *funding streams*.

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

The calculations of the *block subsidy*, *miner subsidy*, *Founders' Reward*, and *funding streams* depend on the *block height*, as defined in § 3.3 ***'The Block Chain'*** on p. 15.

The calculations are described in § 7.8 ***'Calculation of Block Subsidy, Funding Streams, and Founders' Reward'*** on p. 127.

## 3.11 Coinbase Transactions

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

[Pre-**Canopy**] As described in § 7.9 ***‘Payment of Founders’ Reward’*** on p. 128, the *coinbase transaction* **MUST** also pay the *Founders’ Reward*.

[**Canopy** onward] As described in § 7.10 ***‘Payment of Funding Streams’*** on p. 130, the *coinbase transaction* **MUST** also pay the *funding streams*.

## 3.12 Mainnet and Testnet

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

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

*Mainnet genesis block*: `00040fe8ec8471911baa1db1266ea15dd06b4a8a5c453883c000b031973dce08`

*Mainnet* **Canopy** *activation block*: `0000000000d2038016f976744c369dce7419fca30e7171dfac703af5e5f7ad1d4`

There is also a public test *network* called *Testnet*. It supports a **TAZ** token which is intended to have no monetary value. By convention, *Testnet* activates *network upgrades* (as described in § 6 ***‘Network Upgrades’*** on p. 112) before *Mainnet*, in order to allow for errors or ambiguities in their specification and implementation to be discovered. The *Testnet block chain* is subject to being rolled back to a prior *block* at any time.

*Testnet genesis block*: `05a60a92d99d85997cce3b87616c089f6124d7342af37106edc76126334a2c38`

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

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

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

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

# 4 Abstract Protocol

## 4.1 Abstract Cryptographic Schemes

### 4.1.1 Hash Functions

Let $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$, $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$, $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}$, $\ell_{\mathsf{d}}$, $\ell_{\mathsf{Seed}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell_{\mathsf{hSig}}$, and $\mathrm{N}^{\mathsf{old}}$ be as defined in § 5.3 ***‘Constants’*** on p. 67.

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, $r_{\mathbb{J}}$, and $\ell_{\mathbb{J}}$ be as defined in § 5.4.9.3 ‘Jubjub’ on p. 94.

Let $\mathbb{P}^*$ be as defined in § 5.4.9.6 ‘Pallas *and* Vesta’ on p. 97.

The following *hash functions* are used in § 4.9 ***'Merkle Path Validity'*** on p. 45:

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

$\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ is *collision-resistant* except on its first argument. $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ and $\mathsf{MerkleCRH}^{\mathsf{Orchard}}$ are *collision-resistant* on all their arguments.

These functions are instantiated in § 5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 69.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a *collision-resistant hash function* used in § 4.3 ***'JoinSplit Descriptions'*** on p. 36. It is instantiated in § 5.4.1.4 '$\mathsf{h_{Sig}}$ ***Hash Function'*** on p. 70.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{N}^+ \to \mathbb{B}^{[n]}$ is another *hash function*, used in § 7.7.1 ***'Equihash'*** on p. 124 to generate input to the *Equihash* solver. The first two arguments, representing the *Equihash* parameters $n$ and $k$, are written subscripted. It is instantiated in § 5.4.1.11 ***'Equihash Generator'*** on p. 78.

$\mathsf{CRH}^{\mathsf{ivk}} : \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} \to \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}}-1\}$ is a *collision-resistant hash function* used in § 4.2.2 ***'Sapling Key Components'*** on p. 32 to derive an *incoming viewing key* for a **Sapling** *shielded payment address*. It is also used in the *Spend statement* (§ 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55) to confirm use of the correct keys for the *note* being spent. It is instantiated in § 5.4.1.5 '$\mathsf{CRH}^{\mathsf{ivk}}$ ***Hash Function'*** on p. 71.

$\mathsf{MixingPedersenHash} : \mathbb{J} \times \{0\,..\,r_{\mathbb{J}}-1\} \to \mathbb{J}$ is a *hash function* used in § 4.16 ***'Note Commitments and Nullifiers'*** on p. 53 to derive the unique $\mathsf{\rho}$ value for a **Sapling** *note*. It is also used in the *Spend statement* to confirm use of the correct $\mathsf{\rho}$ value as an input to *nullifier* derivation. It is instantiated in § 5.4.1.8 ***'Mixing Pedersen Hash Function'*** on p. 74.

$\mathsf{DiversifyHash}^{\mathsf{Sapling}} : \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{J}^{(r)*} \cup \{\bot\}$ and $\mathsf{DiversifyHash}^{\mathsf{Orchard}} : \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{P}^*$ are *hash functions* instantiated in § 5.4.1.6 '$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ ***Hash Functions'*** on p. 71, satisfying the Unlinkability security property described in that section. They are used to derive a *diversified base* from a *diversifier*, which is specified in § 4.2.2 ***'Sapling Key Components'*** on p. 32 and in § 4.2.3 ***'Orchard Key Components'*** on p. 34.

### 4.1.2 Pseudo Random Functions

$\mathsf{PRF}_x$ denotes a *Pseudo Random Function* keyed by $x$.

Let $\ell_{\mathsf{a_{sk}}}$, $\ell_{\mathsf{hSig}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{\varphi}}$, $\ell_{\mathsf{sk}}$, $\ell_{\mathsf{ovk}}$, $\ell_{\mathsf{PRFexpand}}$, $\ell_{\mathsf{PRFnfSapling}}$, $\mathrm{N}^{\mathsf{old}}$, and $\mathrm{N}^{\mathsf{new}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{Sym}$ be as defined in § 5.4.3 ***'Symmetric Encryption'*** on p. 81.

Let $\ell_{\mathbb{J}}$ and $\mathbb{J}^{(r)}_{\star}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\ell_{\mathbb{P}}$ and $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

For **Sprout**, four ***independent*** $\mathsf{PRF}_x$ are needed:

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

These are used in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in § 4.2.1 ***'Sprout Key Components'*** on p. 32.

For **Sapling**, three additional $\mathsf{PRF}_x$ are needed:

$$\begin{array}{llllll}
\mathsf{PRF}^{\mathsf{expand}} & ⦂\ \mathbb{B}^{[\ell_{\mathsf{sk}}]} & \times\, \mathbb{BY}^{[\mathbb{N}]} & & & \to \mathbb{BY}^{[\ell_{\mathsf{PRFexpand}}/8]} \\
\mathsf{PRF}^{\mathsf{ockSapling}} & ⦂\ \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{J}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{J}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{J}}/8]} & \to \mathsf{Sym}.\mathbf{K} \\
\mathsf{PRF}^{\mathsf{nfSapling}} & ⦂\ \mathbb{J}_{\star}^{(r)} & \times\, \mathbb{B}^{[\ell_{\mathbb{J}}]} & & & \to \mathbb{BY}^{[\ell_{\mathsf{PRFnfSapling}}/8]}
\end{array}$$

For **Orchard**, we need $\mathsf{PRF}^{\mathsf{expand}}$, and also:

$$\begin{array}{llllll}
\mathsf{PRF}^{\mathsf{ockOrchard}} & ⦂\ \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{P}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{P}}/8]} & \times\, \mathbb{BY}^{[\ell_{\mathbb{P}}/8]} & \to \mathsf{Sym}.\mathbf{K} \\
\mathsf{PRF}^{\mathsf{nfOrchard}} & ⦂\ \mathbb{F}_{q_{\mathbb{P}}} & \times\, \mathbb{F}_{q_{\mathbb{P}}} & & & \to \mathbb{F}_{q_{\mathbb{P}}}
\end{array}$$

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

- § 4.2.2 *‘**Sapling** Key Components’* on p. 32, with inputs $[0]$, $[1]$, $[2]$, and $[3, i ⦂ \mathbb{BY}]$;
- [**NU5** onward] in § 4.2.3 *‘**Orchard** Key Components’* on p. 34, with inputs $[6]$, $[7]$, $[8]$, and $[\mathtt{0x81}]$ (the last of these is also specified in [ZIP-32]);
- in the processes of sending (§ 4.7.2 *‘Sending Notes (**Sapling**)’* on p. 41 and § 4.7.3 *‘Sending Notes (**Orchard**)’* on p. 42) and of receiving (§ 4.19 *‘In-band secret distribution (**Sapling** and **Orchard**)’* on p. 60) *notes*, with inputs $[4]$ and $[5]$, and for **Orchard** also $[9]$;
- in [ZIP-32], with inputs $[0]$, $[1]$, $[2]$ (intentionally matching § 4.2.2 on p. 32), $[t ⦂ \{16 \,..\, 22\}]$, and $[\mathtt{0x80}]$.

$\mathsf{PRF}^{\mathsf{ockSapling}}$ and $\mathsf{PRF}^{\mathsf{ockOrchard}}$ are used in § 4.19 *‘In-band secret distribution (**Sapling** and **Orchard**)’* on p. 60.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used in § 4.17.2 *‘Spend Statement (**Sapling**)’* on p. 55.

$\mathsf{PRF}^{\mathsf{nfOrchard}}$ is used in § 4.17.4 *‘Action Statement (**Orchard**)’* on p. 57.

All of these *Pseudo Random Functions* are instantiated in § 5.4.2 *‘Pseudo Random Functions’* on p. 79.

**Security requirements:**

- Security definitions for *Pseudo Random Functions* are given in [BDJR2000, section 4].
- In addition to being *Pseudo Random Functions*, it is required that $\mathsf{PRF}^{\mathsf{addr}}_x$, $\mathsf{PRF}^{\rho}_x$, $\mathsf{PRF}^{\mathsf{nfSprout}}_x$, $\mathsf{PRF}^{\mathsf{nfSapling}}_x$ and $\mathsf{PRF}^{\mathsf{nfOrchard}}_x$ 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 $\mathsf{PRF}^{\rho}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{nfSapling}}$, and $\mathsf{PRF}^{\mathsf{nfOrchard}}$.

**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 Pseudo Random Permutations

$\mathsf{PRP}_x$ denotes a *Pseudo Random Permutation* keyed by $x$.

Let $\ell_{\mathsf{dk}}$ and $\ell_{\mathsf{d}}$ be as defined in § 5.3 *‘Constants’* on p. 67.

One *Pseudo Random Permutation* is used for **Orchard**, to generate *diversifiers* from a *diversifier key* and index (an identical construction is also used for **Sapling** in [ZIP-32]):

$$\mathsf{PRP}^{\mathsf{d}} ⦂ \mathbb{BY}^{[\ell_{\mathsf{dk}}/8]} \times \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{B}^{[\ell_{\mathsf{d}}]}.$$

It is instantiated in § 5.4.4 *‘Pseudo Random Permutations’* on p. 81.

**Security requirement:** $\mathsf{PRP}^{\mathsf{d}}$ is a keyed *Pseudo Random Permutation* as defined in [BKR2001].

### 4.1.4 Symmetric Encryption

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

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

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

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

### 4.1.5 Key Agreement

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

A *key agreement scheme* KA defines a type of *public keys* KA.Public, a type of *private keys* KA.Private, and a type of shared secrets KA.SharedSecret. Optionally, it also defines a type $\mathsf{KA.PublicPrimeSubgroup} \subseteq \mathsf{KA.Public}$.

Optional: 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.

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

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

**Security requirements:**

- KA.FormatPrivate must preserve sufficient entropy from its input to be used as a secure KA *private key*.
- The key agreement and the KDF defined in the next section must together satisfy a suitable adaptive security assumption along the lines of [Bernstein2006, section 3] or [ABR1999, Definition 3].

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

### 4.1.6 Key Derivation

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

The inputs to the *Key Derivation Function* differ between the **Sprout** and **Sapling** and **Orchard** KDFs:

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

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

$\mathsf{KDF}^{\mathsf{Sapling}}$ takes as input the shared Diffie-Hellman secret $\mathsf{sharedSecret}$ and the *ephemeral public key* $\mathsf{epk}$. (It does not have inputs taking the place of the output index, $\mathsf{h_{Sig}}$, or $\mathsf{pk_{enc}}$.) It is suitable for use with $\mathsf{KA}^{\mathsf{Sapling}}$ and derives keys for $\mathsf{Sym.Encrypt}$.

$$\mathsf{KDF}^{\mathsf{Sapling}} ⦂ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret} \times \mathbb{BY}^{[\ell_{\mathbb{J}}/8]} \to \mathsf{Sym}.\mathbf{K}$$

As in **Sapling**, $\mathsf{KDF}^{\mathsf{Orchard}}$ takes as input the shared Diffie-Hellman secret $\mathsf{sharedSecret}$ and the *ephemeral public key* $\mathsf{epk}$. It is suitable for use with $\mathsf{KA}^{\mathsf{Orchard}}$ and derives keys for $\mathsf{Sym.Encrypt}$.

$$\mathsf{KDF}^{\mathsf{Orchard}} ⦂ \mathsf{KA}^{\mathsf{Orchard}}.\mathsf{SharedSecret} \times \mathbb{BY}^{[\ell_{\mathbb{P}}/8]} \to \mathsf{Sym}.\mathbf{K}$$

**Security requirements:**

- The asymmetric encryption scheme in § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59, constructed from $\mathsf{KA}^{\mathsf{Sprout}}$, $\mathsf{KDF}^{\mathsf{Sprout}}$ and $\mathsf{Sym}$, is required to be IND-CCA2-secure and *key-private*.
- The asymmetric encryption scheme in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60, constructed from $\mathsf{KA}^{\mathsf{Sapling}}$, $\mathsf{KDF}^{\mathsf{Sapling}}$ and $\mathsf{Sym}$ or from $\mathsf{KA}^{\mathsf{Orchard}}$, $\mathsf{KDF}^{\mathsf{Orchard}}$ and $\mathsf{Sym}$, is required to be IND-CCA2-secure and *key-private*.

*Key privacy* is defined in [BBDP2001].

### 4.1.7 Signature

A *signature scheme* $\mathsf{Sig}$ defines:

- a type of *signing keys* $\mathsf{Sig.Private}$;
- a type of *validating keys* $\mathsf{Sig.Public}$;
- a type of messages $\mathsf{Sig.Message}$;
- a type of signatures $\mathsf{Sig.Signature}$;
- a randomized *signing key* generation algorithm $\mathsf{Sig.GenPrivate} ⦂ () \overset{\mathrm{R}}{\to} \mathsf{Sig.Private}$;
- an injective *validating key* derivation algorithm $\mathsf{Sig.DerivePublic} ⦂ \mathsf{Sig.Private} \to \mathsf{Sig.Public}$;
- a randomized signing algorithm $\mathsf{Sig.Sign} ⦂ \mathsf{Sig.Private} \times \mathsf{Sig.Message} \overset{\mathrm{R}}{\to} \mathsf{Sig.Signature}$;
- 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* $\mathsf{sk} \overset{\mathrm{R}}{\leftarrow} \mathsf{Sig.GenPrivate}()$ and corresponding *validating key* $\mathsf{vk} = \mathsf{Sig.DerivePublic}(\mathsf{sk})$, and any $m ⦂ \mathsf{Sig.Message}$ and $s ⦂ \mathsf{Sig.Signature} \overset{\mathrm{R}}{\leftarrow} \mathsf{Sig.Sign_{sk}}(m)$, $\mathsf{Sig.Validate_{vk}}(m, s) = 1$.

**Zcash** uses four *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 $\mathsf{JoinSplitSig}$ which is used to sign *transactions* that contain at least one *JoinSplit description* (instantiated in § 5.4.6 'Ed25519' on p. 83);
- [**Sapling** onward] one called $\mathsf{SpendAuthSig}$ which is used to sign authorizations of *Spend transfers* (instantiated in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87);
- [**Sapling** onward] one called $\mathsf{BindingSig}$. A *Sapling binding signature* is used to enforce balance of *Spend transfers* and *Output transfers*, and to prevent their replay across *transactions*. Similarly, an *Orchard binding signature* is used to enforce balance of *Action transfers* and to prevent their replay. $\mathsf{BindingSig}$ is instantiated for both **Sapling** and **Orchard** in § 5.4.7.2 ***'Binding Signature (Sapling and Orchard)'*** on p. 88.

The signature scheme used in script operations is instantiated by ECDSA on the secp256k1 curve. $\mathsf{JoinSplitSig}$ is instantiated by $\mathsf{Ed25519}$. $\mathsf{SpendAuthSig}$ and $\mathsf{BindingSig}$ are instantiated by $\mathsf{RedDSA}$; on the Jubjub curve in **Sapling**, and on the Pallas curve in **Orchard**.

The following security property is needed for $\mathsf{JoinSplitSig}$ and $\mathsf{BindingSig}$. Security requirements for $\mathsf{SpendAuthSig}$ are defined in the next section, § 4.1.7.1 ***'Signature with Re-Randomizable Keys'*** on p. 25. An additional requirement for $\mathsf{BindingSig}$ is defined in § 4.1.7.2 ***'Signature with Signing Key to Validating Key Monomorphism'*** on p. 26.

**Security requirement:** JoinSplitSig and each instantiation of BindingSig must be Strongly Unforgeable under (non-adaptive) Chosen Message Attack (SU-CMA), as defined for example in [BDEHR2011, Definition 6].[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 $\mathsf{Sig.Gen} ⦂ () \xrightarrow{\mathsf{R}} \mathsf{Sig.Private} \times \mathsf{Sig.Public}$, to support the key derivation in § 4.2.2 ***'Sapling Key Components'*** on p. 32 and in § 4.2.3 ***'Orchard Key Components'*** on p. 34.

  The definitions of schemes with additional features in § 4.1.7.1 ***'Signature with Re-Randomizable Keys'*** on p. 25 and in § 4.1.7.2 ***'Signature with Signing Key to Validating Key Monomorphism'*** on p. 26 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 § 4.11 ***'Non-malleability (Sprout)'*** on p. 46), 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.
- [**Sapling** onward] The same remarks as above apply to BindingSig, except that the key is derived from the randomness of *value commitments*. This results in the same distribution as of freshly generated key pairs, for each *transaction* containing *Spend descriptions* or *Output descriptions* or *Action descriptions*.
- 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* and *Sapling binding signatures* and *Orchard binding 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*.

### 4.1.7.1 Signature with Re-Randomizable Keys

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

- a type of *randomizers* $\mathsf{Sig.Random}$;
- a *randomizer* generator $\mathsf{Sig.GenRandom} ⦂ () \xrightarrow{\mathsf{R}} \mathsf{Sig.Random}$;
- a *signing key* randomization algorithm $\mathsf{Sig.RandomizePrivate} ⦂ \mathsf{Sig.Random} \times \mathsf{Sig.Private} \to \mathsf{Sig.Private}$;
- a *validating key* randomization algorithm $\mathsf{Sig.RandomizePublic} ⦂ \mathsf{Sig.Random} \times \mathsf{Sig.Public} \to \mathsf{Sig.Public}$;
- a distinguished "identity" *randomizer* $\mathcal{O}_{\mathsf{Sig.Random}} ⦂ \mathsf{Sig.Random}$

such that:

- for any $\alpha ⦂ \mathsf{Sig.Random}$, $\mathsf{Sig.RandomizePrivate}_{\alpha} ⦂ \mathsf{Sig.Private} \to \mathsf{Sig.Private}$ is injective and easily invertible;
- $\mathsf{Sig.RandomizePrivate}_{\mathcal{O}_{\mathsf{Sig.Random}}}$ is the identity function on $\mathsf{Sig.Private}$.
- for any $\mathsf{sk} ⦂ \mathsf{Sig.Private}$,

  $$\mathsf{Sig.RandomizePrivate}(\alpha, \mathsf{sk}) : \alpha \xleftarrow{\mathsf{R}} \mathsf{Sig.GenRandom}()$$

  is identically distributed to $\mathsf{Sig.GenPrivate}()$.
- for any $\mathsf{sk} ⦂ \mathsf{Sig.Private}$ and $\alpha ⦂ \mathsf{Sig.Random}$,

  $$\mathsf{Sig.RandomizePublic}(\alpha, \mathsf{Sig.DerivePublic}(\mathsf{sk})) = \mathsf{Sig.DerivePublic}(\mathsf{Sig.RandomizePrivate}(\alpha, \mathsf{sk})).$$

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

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

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

For any $\mathsf{sk} ⦂ \mathsf{Sig.Private}$, let

$$\mathsf{O_{sk}} ⦂ \mathsf{Sig.Message} \times \mathsf{Sig.Random} \to \mathsf{Sig.Signature}$$

be a signing oracle with state $Q ⦂ \mathscr{P}(\mathsf{Sig.Message} \times \mathsf{Sig.Signature})$ initialized to $\{\}$ that records queried messages and corresponding signatures.

```
$\mathsf{O_{sk}} :=$ let mutable $Q \leftarrow \{\}$ in $(m ⦂ \mathsf{Sig.Message}, \alpha ⦂ \mathsf{Sig.Random}) \mapsto$
    let $\sigma = \mathsf{Sig.Sign}_{\mathsf{Sig.RandomizePrivate}(\alpha,\mathsf{sk})}(m)$
    set $Q \leftarrow Q \cup \{(m, \sigma)\}$
    return $\sigma ⦂ \mathsf{Sig.Signature}$.
```

For random $\mathsf{sk} \overset{\mathrm{R}}{\leftarrow} \mathsf{Sig.GenPrivate}()$ and $\mathsf{vk} = \mathsf{Sig.DerivePublic}(\mathsf{sk})$, it must be infeasible for an adversary given $\mathsf{vk}$ and a new instance of $\mathsf{O_{sk}}$ to find $(m', \sigma', \alpha')$ such that $\mathsf{Sig.Validate}_{\mathsf{Sig.RandomizePublic}(\alpha',\mathsf{vk})}(m', \sigma') = 1$ and $(m', \sigma') \notin \mathsf{O_{sk}}.Q$.

**Non-normative notes:**

- The *randomizer* and key arguments to $\mathsf{Sig.RandomizePrivate}$ and $\mathsf{Sig.RandomizePublic}$ are swapped relative to [FKMSSS2016, section 3].
- The requirement for the identity *randomizer* $\mathcal{O}_{\mathsf{Sig.Random}}$ simplifies the definition of SURK-CMA by removing the need for two oracles (because the oracle for original keys, called $\mathsf{O}_1$ in [FKMSSS2016], is a special case of the oracle for randomized keys).
- Since $\mathsf{Sig.RandomizePrivate}(\alpha, \mathsf{sk}) : \alpha \overset{\mathrm{R}}{\leftarrow} \mathsf{Sig.Random}$ has an identical distribution to $\mathsf{Sig.GenPrivate}()$, and since $\mathsf{Sig.DerivePublic}$ is a deterministic function, the combination of a re-randomized *validating key* and signature(s) under that key do not reveal the key from which it was re-randomized.
- Since $\mathsf{Sig.RandomizePrivate}_\alpha$ is injective and easily invertible, knowledge of $\mathsf{Sig.RandomizePrivate}(\alpha, \mathsf{sk})$ *and* $\alpha$ implies knowledge of $\mathsf{sk}$.

### 4.1.7.2 Signature with Signing Key to Validating Key Monomorphism

A *signature scheme with key monomorphism* $\mathsf{Sig}$ is a *signature scheme* that additionally defines:

- an abelian group on *signing keys*, with operation $\boxplus ⦂ \mathsf{Sig.Private} \times \mathsf{Sig.Private} \to \mathsf{Sig.Private}$ and identity $\mathcal{O}_{\boxplus}$;
- an abelian group on *validating keys*, with operation $\diamondplus ⦂ \mathsf{Sig.Public} \times \mathsf{Sig.Public} \to \mathsf{Sig.Public}$ and identity $\mathcal{O}_{\diamondplus}$.

such that for any $\mathsf{sk}_{1..2} ⦂ \mathsf{Sig.Private}$, $\mathsf{Sig.DerivePublic}(\mathsf{sk}_1 \boxplus \mathsf{sk}_2) = \mathsf{Sig.DerivePublic}(\mathsf{sk}_1) \diamondplus \mathsf{Sig.DerivePublic}(\mathsf{sk}_2)$.

In other words, $\mathsf{Sig.DerivePublic}$ is a *monomorphism* (that is, an injective homomorphism) from the *signing key* group to the *validating key* group.

For $\mathrm{N} : \mathbb{N}^+$,

- $\bigboxplus_{i=1}^{\mathrm{N}} \mathsf{sk}_i$ means $\mathsf{sk}_1 \boxplus \mathsf{sk}_2 \boxplus \cdots \boxplus \mathsf{sk}_\mathrm{N}$;
- $\bigdiamondplus_{i=1}^{\mathrm{N}} \mathsf{vk}_i$ means $\mathsf{vk}_1 \diamondplus \mathsf{vk}_2 \diamondplus \cdots \diamondplus \mathsf{vk}_\mathrm{N}$.

When $\mathrm{N} = 0$ these yield the appropriate group identity, i.e. $\bigboxplus_{i=1}^{0} \mathsf{sk}_i = \mathcal{O}_{\boxplus}$ and $\bigdiamondplus_{i=1}^{0} \mathsf{vk}_i = \mathcal{O}_{\diamondplus}$.

$\boxminus \mathsf{sk}$ means the *signing key* such that $(\boxminus \mathsf{sk}) \boxplus \mathsf{sk} = \mathcal{O}_{\boxplus}$, and $\mathsf{sk}_1 \boxminus \mathsf{sk}_2$ means $\mathsf{sk}_1 \boxplus (\boxminus \mathsf{sk}_2)$.

$\diamondminus \mathsf{vk}$ means the *validating key* such that $(\diamondminus \mathsf{vk}) \diamondplus \mathsf{vk} = \mathcal{O}_{\diamondplus}$, and $\mathsf{vk}_1 \diamondminus \mathsf{vk}_2$ means $\mathsf{vk}_1 \diamondplus (\diamondminus \mathsf{vk}_2)$.

With a change of notation from $\mu$ to $\mathsf{Sig.DerivePublic}$, $+$ to $\boxplus$, and $\cdot$ to $\diamondplus$, this is similar to the definition of a "*Signature with Secret Key to Public Key Homomorphism*" in [DS2016, Definition 13], except for an additional requirement for the homomorphism to be injective.

**Security requirement:** For any $\mathsf{sk}_1 : \mathsf{Sig.Private}$, and an unknown $\mathsf{sk}_2 \overset{\mathsf{R}}{\leftarrow} \mathsf{Sig.GenPrivate}()$ chosen independently of $\mathsf{sk}_1$, the distribution of $\mathsf{sk}_1 \boxplus \mathsf{sk}_2$ is computationally indistinguishable from that of $\mathsf{Sig.GenPrivate}()$. (Since $\boxplus$ is an abelian group operation, this implies that for $n : \mathbb{N}^+$, $\bigboxplus_{i=1}^{n} \mathsf{sk}_i$ is computationally indistinguishable from $\mathsf{Sig.GenPrivate}()$ when at least one of $\mathsf{sk}_{1..n}$ is unknown.)

### 4.1.8 Commitment

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

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

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

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

**Security requirements:**

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

**Notes:**

- $\mathsf{COMM.GenTrapdoor}$ need not produce the uniform distribution on $\mathsf{COMM.Trapdoor}$. In that case, it is incorrect to choose a *trapdoor* from the latter distribution.
- If it were only feasible to find $x : \mathsf{COMM.Input}$ and $r, r' : \mathsf{COMM.Trapdoor}$ such that $r \neq r'$ and $\mathsf{COMM}_r(x) = \mathsf{COMM}_{r'}(x)$, this would not contradict the computational binding security requirement. (In fact, this is feasible for $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ and $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ because *trapdoors* are equivalent modulo $r_{\mathbb{J}}$, and the range of a *trapdoor* for those algorithms is $\{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} - 1\}$ where $2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}} > r_{\mathbb{J}}$.)

Let $\ell_{\mathsf{rcm}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, and $\ell_{\mathsf{value}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

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

**Sprout** uses a *note commitment scheme*

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

instantiated in § 5.4.8.1 ***Sprout** Note Commitments*' on p. 88.

Let $\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}$ be as defined in § 5.3 '*Constants*' on p. 67.

Let $\mathbb{J}^{(r)}$, $\ell_{\mathbb{J}}$, and $r_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Define:

$\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} := \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}}-1\}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Output} := \mathbb{J}$;

$\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} := \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}}-1\}$ and $\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output} := \mathbb{J}$.

**Sapling** uses two additional commitment schemes:

$$\begin{aligned}\mathsf{NoteCommit}^{\mathsf{Sapling}} &⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \{0\,..\,2^{\ell_{\mathsf{value}}}-1\} \to \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Output}\\ \mathsf{ValueCommit}^{\mathsf{Sapling}} &⦂ \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} \times \left\{-\tfrac{r_{\mathbb{J}}-1}{2}\,..\,\tfrac{r_{\mathbb{J}}-1}{2}\right\} \to \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}\end{aligned}$$

$\mathsf{NoteCommit}^{\mathsf{Sapling}}$ is instantiated in § 5.4.8.2 '*Windowed Pedersen commitments*' on p. 88, and $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ is instantiated in § 5.4.8.3 '*Homomorphic Pedersen commitments (**Sapling** and **Orchard**)*' on p. 89.

**Non-normative note:** $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ and $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ always return points in the subgroup $\mathbb{J}^{(r)}$. However, we declare the type of these commitment outputs to be $\mathbb{J}$ because they are not directly checked to be in the subgroup when $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ outputs appear in *Spend descriptions* and *Output descriptions*, or when the `cmu` field derived from a $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ appears in an *Output description*.

Let $\ell_{\mathsf{scalar}}^{\mathsf{Orchard}}$ be as defined in § 5.3 '*Constants*' on p. 67.

Let $\mathbb{P}$, $\mathbb{P}_x$, $\ell_{\mathbb{P}}$, $q_{\mathbb{P}}$, and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Define:

$\mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor} := \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Orchard}}}-1\}$ and $\mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{Output} := \mathbb{P}$;

$\mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor} := \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Orchard}}}-1\}$ and $\mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output} := \mathbb{P}$.

$\mathsf{Commit}^{\mathsf{ivk}}.\mathsf{Trapdoor} := \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Orchard}}}-1\}$ and $\mathsf{Commit}^{\mathsf{ivk}}.\mathsf{Output} := \{0\,..\,q_{\mathbb{P}}-1\}$.

**Orchard** uses three additional commitment schemes:

$$\begin{aligned}\mathsf{NoteCommit}^{\mathsf{Orchard}} &⦂ \mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor} \times \mathbb{B}^{[\ell_{\mathbb{P}}]} \times \mathbb{B}^{[\ell_{\mathbb{P}}]} \times \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}\\ &\qquad \times \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{F}_{q_{\mathbb{P}}} \to \mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{Output}\\ \mathsf{ValueCommit}^{\mathsf{Orchard}} &⦂ \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor} \times \left\{-\tfrac{r_{\mathbb{P}}-1}{2}\,..\,\tfrac{r_{\mathbb{P}}-1}{2}\right\} \to \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output}\\ \mathsf{Commit}^{\mathsf{ivk}} &⦂ \mathsf{Commit}^{\mathsf{ivk}}.\mathsf{Trapdoor} \times \mathbb{P}_x \times \mathbb{F}_{q_{\mathbb{P}}} \to \mathsf{Commit}^{\mathsf{ivk}}.\mathsf{Output}\end{aligned}$$

$\mathsf{NoteCommit}^{\mathsf{Orchard}}$ and $\mathsf{Commit}^{\mathsf{ivk}}$ are instantiated in § 5.4.8.4 '*Sinsemilla commitments*' on p. 90. $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ is instantiated in § 5.4.8.3 '*Homomorphic Pedersen commitments (**Sapling** and **Orchard**)*' on p. 89.

### 4.1.9 Represented Group

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

- a subgroup order parameter $r_{\mathbb{G}} \mathrel{⦂} \mathbb{N}^+$, which must be prime;
- a cofactor parameter $h_{\mathbb{G}} \mathrel{⦂} \mathbb{N}^+$;
- a group $\mathbb{G}$ of order $h_{\mathbb{G}} \cdot r_{\mathbb{G}}$, written additively with operation $+ \mathrel{⦂} \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, and additive identity $\mathcal{O}_{\mathbb{G}}$;
- a bit-length parameter $\ell_{\mathbb{G}} \mathrel{⦂} \mathbb{N}$;
- a representation function $\mathsf{repr}_{\mathbb{G}} \mathrel{⦂} \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]}$ and an abstraction function $\mathsf{abst}_{\mathbb{G}} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\}$, 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 $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{abst}_{\mathbb{G}}(S) = \bot$. This may not be true in all cases, i.e. there can be *non-canonical* encodings $P\star$ such that $\mathsf{repr}_{\mathbb{G}}\big(\mathsf{abst}_{\mathbb{G}}(P\star)\big) \neq P\star$.

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

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

For $G \mathrel{⦂} \mathbb{G}$ 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 $G \mathrel{⦂} \mathbb{G}$ and $k \mathrel{⦂} \mathbb{Z}$ 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 $G \mathrel{⦂} \mathbb{G}$ and $a \mathrel{⦂} \mathbb{F}_{r_{\mathbb{G}}}$, 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 $\mathbb{F}_{r_{\mathbb{G}}}$.)

### 4.1.10 Coordinate Extractor

A *coordinate extractor* for a *represented group* $\mathbb{G}$ is a function $\mathsf{Extract}_{\mathbb{G}^{(r)}} \mathrel{⦂} \mathbb{G}^{(r)} \to T$ for some type $T$.

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

### 4.1.11 Group Hash

Given a *represented subgroup* $\mathbb{G}^{(r)}$, a *family of group hashes into the subgroup*, denoted $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}$, consists of:

- a type $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}.\mathsf{URSType}$ of *Uniform Random Strings*;
- a type $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}.\mathsf{Input}$ of inputs;
- a function $\mathsf{GroupHash}^{\mathbb{G}^{(r)}} \mathrel{⦂} \mathsf{GroupHash}^{\mathbb{G}^{(r)}}.\mathsf{URSType} \times \mathsf{GroupHash}^{\mathbb{G}^{(r)}}.\mathsf{Input} \to \mathbb{G}^{(r)}$.

In § 5.4.9.5 ‘*Group Hash into* Jubjub’ on p. 96, we instantiate a family of *group hashes* into the Jubjub curve defined by § 5.4.9.3 ‘Jubjub’ on p. 94.

**Security requirement:** For a randomly selected URS ⦂ $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}$.URSType, it must be reasonble to model $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}_{\mathsf{URS}}$ (restricted to inputs for which it does not return $\bot$) as a *random oracle*.

In § 5.4.9.8 *'Group Hash into* Pallas *and* Vesta*'* on p. 98, we instantiate *group hashes* into the Pallas and Vesta curves. These are not strictly speaking *families of group hashes*, because they have a trivial URS, and so the above security definition does not apply. Nevertheless, they can be heuristically modelled as *random oracles*.

**Non-normative notes:**

- $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ is used to obtain generators of the Jubjub curve for various purposes: the bases $\mathcal{G}^{\mathsf{Sapling}}$ and $\mathcal{H}^{\mathsf{Sapling}}$ used in **Sapling** key generation, the *Pedersen hash* defined in § 5.4.1.7 *'Pedersen Hash Function'* on p. 72, and the commitment schemes defined in § 5.4.8.2 *'Windowed Pedersen commitments'* on p. 88 and in § 5.4.8.3 *'Homomorphic Pedersen commitments (**Sapling** and **Orchard**)'* on p. 89.

  The security property needed for these uses can alternatively be defined in the standard model as follows:

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

- Under the Discrete Logarithm assumption on $\mathbb{G}^{(r)}$, a *random oracle* almost surely satisfies Discrete Logarithm Independence. Discrete Logarithm Independence implies *collision resistance*, since a collision $(m_1, m_2)$ for $\mathsf{GroupHash}^{\mathbb{G}^{(r)}}_{\mathsf{URS}}$ trivially gives a discrete logarithm relation with $x_1 = 1$ and $x_2 = -1$.

- $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ is used in § 5.4.1.6 *'*$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ *Hash Functions'* on p. 71 to instantiate $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$. We do not know how to prove the Unlinkability property defined in that section in the standard model, but in a model where $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ (restricted to inputs for which it does not return $\bot$) is taken as a *random oracle*, it is implied by the Decisional Diffie–Hellman assumption on $\mathbb{J}^{(r)}$, and similarly for $\mathsf{GroupHash}^{\mathbb{P}}$.

- URS is a *Uniform Random String*; we chose it verifiably at random (see § 5.9 *'Randomness Beacon'* on p. 112), after fixing the concrete group hash algorithm to be used. This mitigates the possibility that the group hash algorithm could have been backdoored. For **Orchard**, we considered a URS to be unnecessary, because we follow [ID-hashtocurve] which does not use one.

### 4.1.12 Represented Pairing

A *represented pairing* $\mathbb{P}\mathrm{AIR}$ consists of:

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

### 4.1.13 Zero-Knowledge Proving System

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

A *preprocessing zk-SNARK* instance ZK defines:

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

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

**Security requirements:**

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

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

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

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

**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*.

**Zcash** uses three *proving systems*:

- BCTV14 (§ 5.4.10.1 'BCTV14' on p. 102) is used with the BN-254 pairing (§ 5.4.9.1 'BN-254' on p. 91), to prove and verify the **Sprout** *JoinSplit statement* (§ 4.17.1 *'JoinSplit Statement (Sprout)'* on p. 54) before **Sapling** activation.
- Groth16 (§ 5.4.10.2 'Groth16' on p. 103) is used with the BLS12-381 pairing (§ 5.4.9.2 'BLS12-381' on p. 93), to prove and verify the **Sapling** *Spend statement* (§ 4.17.2 *'Spend Statement (Sapling)'* on p. 55) and *Output statement* (§ 4.17.3 *'Output Statement (Sapling)'* on p. 56). It is also used to prove and verify the *JoinSplit statement* after **Sapling** activation.
- [**NU5** onward] Halo 2 (§ 5.4.10.3 'Halo 2' on p. 103) is used with the Vesta curve (§ 5.4.9.6 'Pallas *and* Vesta' on p. 97) to prove and verify the **Orchard** *Action statement* (§ 4.17.4 *'Action Statement (Orchard)'* on p. 57).

These specializations are:

- ZKJoinSplit for the **Sprout** *JoinSplit statement* (with BCTV14 and BN-254, or Groth16 and BLS12-381);
- ZKSpend for the **Sapling** *Spend statement*;
- ZKOutput for the **Sapling** *Output statement*;
- [**NU5** onward] ZKAction for the **Orchard** *Action statement*.

We omit key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be either the BCTV14 *proving key* and *verifying key* defined in § 5.7 'BCTV14 *zk-SNARK Parameters*' on p. 111, or the `sprout-groth16.params` Groth16 *proving key* and *verifying key* defined in § 5.8 'Groth16 *zk-SNARK Parameters*' on p. 112, according to whether the proof appears in a *block* before or after **Sapling** activation.

We omit subscripts on ZKSpend.Prove, ZKSpend.Verify, ZKOutput.Prove, and ZKOutput.Verify, taking them to be the relevant Groth16 *proving keys* and *verifying keys* defined in § 5.8 'Groth16 *zk-SNARK Parameters*' on p. 112.

We also omit subscripts on ZKAction.Prove and ZKAction.Verify. For Halo 2, parameters for a given circuit implementation are generated on the fly by the halo2 library, and do not require parameter files.

## 4.2 Key Components

### 4.2.1 Sprout Key Components

Let $\ell_{\mathsf{a_{sk}}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{PRF^{addr}}$ be a *Pseudo Random Function*, instantiated in § 5.4.2 *'Pseudo Random Functions'* on p. 79.

Let $\mathsf{KA^{Sprout}}$ be a *key agreement scheme*, instantiated in § 5.4.5.1 *'Sprout Key Agreement'* on p. 81.

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:

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

### 4.2.2 Sapling Key Components

Let $\ell_{\mathsf{PRFexpand}}$, $\ell_{\mathsf{sk}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}$, $\ell_{\mathsf{ovk}}$, and $\ell_{\mathsf{d}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, $\mathbb{J}^{(r)}_{\star}$, $\mathsf{repr}_{\mathbb{J}}$, and $r_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94, and let $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ be as defined in § 5.4.9.5 *'Group Hash into* Jubjub' on p. 96.

Let $\mathsf{PRF}^{\mathsf{expand}}$ and $\mathsf{PRF}^{\mathsf{ockSapling}}$, instantiated in § 5.4.2 ‘*Pseudo Random Functions*’ on p. 79, be *Pseudo Random Functions*.

Let $\mathsf{KA}^{\mathsf{Sapling}}$, instantiated in § 5.4.5.3 ‘***Sapling** Key Agreement*’ on p. 82, be a *key agreement scheme*.

Let $\mathsf{CRH}^{\mathsf{ivk}}$, instantiated in § 5.4.1.5 ‘$\mathsf{CRH}^{\mathsf{ivk}}$ *Hash Function*’ on p. 71, be a *hash function*.

Let $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$, instantiated in § 5.4.1.6 ‘$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ *Hash Functions*’ on p. 71, be a *hash function*.

Let $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$, instantiated in § 5.4.7.1 ‘*Spend Authorization Signature (**Sapling** and **Orchard**)*’ on p. 87, be a *signature scheme with re-randomizable keys*.

Let $\mathsf{LEBS2OSP} \colon (\ell \colon \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ and $\mathsf{LEOS2IP} \colon (\ell \colon \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\ell/8]} \to \{0\,..\,2^{\ell}-1\}$ be as defined in § 5.2 ‘*Integers, Bit Sequences, and Endianness*’ on p. 66.

Define $\mathcal{H}^{\mathsf{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\texttt{Zcash\_H\_}\text{”}, \text{“”})$.

Define $\mathsf{ToScalar}^{\mathsf{Sapling}}(x \colon \mathbb{BY}^{[\ell_{\mathsf{PRFexpand}}/8]}) := \mathsf{LEOS2IP}_{\ell_{\mathsf{PRFexpand}}}(x) \pmod{r_{\mathbb{J}}}$.

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

From this *spending key*, the *Spend authorizing key* $\mathsf{ask} \colon \mathbb{F}^{*}_{r_{\mathbb{J}}}$, the *proof authorizing key* $\mathsf{nsk} \colon \mathbb{F}_{r_{\mathbb{J}}}$, and the *outgoing viewing key* $\mathsf{ovk} \colon \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]}$ are derived as follows:

$$
\begin{aligned}
\mathsf{ask} &:= \mathsf{ToScalar}^{\mathsf{Sapling}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([0])) \\
\mathsf{nsk} &:= \mathsf{ToScalar}^{\mathsf{Sapling}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([1])) \\
\mathsf{ovk} &:= \mathsf{truncate}_{(\ell_{\mathsf{ovk}}/8)}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([2]))
\end{aligned}
$$

If $\mathsf{ask} = 0$, discard this key and repeat with a new $\mathsf{sk}$.

$\mathsf{ak} \colon \mathbb{J}^{(r)*}$, $\mathsf{nk} \colon \mathbb{J}^{(r)}$, and the *incoming viewing key* $\mathsf{ivk} \colon \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}}-1\}$ are then derived as:

$$
\begin{aligned}
\mathsf{ak} &:= \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{ask}) \\
\mathsf{nk} &:= [\mathsf{nsk}]\, \mathcal{H}^{\mathsf{Sapling}} \\
\mathsf{ivk} &:= \mathsf{CRH}^{\mathsf{ivk}}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak}), \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big).
\end{aligned}
$$

If $\mathsf{ivk} = 0$, discard this key and repeat with a new $\mathsf{sk}$.

As explained in § 3.1 ‘*Payment Addresses and Keys*’ on p. 12, **Sapling** allows the efficient creation of multiple *diversified payment addresses* with the same spending authority. A group of such addresses shares the same *full viewing key* and *incoming viewing key*.

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

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

The resulting *diversified payment address* is $(\mathsf{d} \colon \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{pk_d} \colon \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup})$.

For each *spending key*, there is also a *default diversified payment address* with a “random-looking” *diversifier*. This allows an implementation that does not expose diversified addresses as a user-visible feature, to use a default address that cannot be distinguished (without knowledge of the *spending key*) from one with a random *diversifier* as above.

Let $\mathsf{first} ⦂ (\mathbb{B}^{\mathbb{Y}} \to T \cup \{\bot\}) \to T \cup \{\bot\}$ be as defined in § 5.4.9.5 ***'Group Hash into* Jubjub'** on p. 96. Define:

$$\mathsf{CheckDiversifier}(\mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}) := \begin{cases} \bot, & \text{if } \mathsf{DiversifyHash}^{\mathsf{Sapling}}(\mathsf{d}) = \bot \\ \mathsf{d}, & \text{otherwise} \end{cases}$$

$$\mathsf{DefaultDiversifier}(\mathsf{sk} ⦂ \mathbb{B}^{[\ell_{\mathsf{sk}}]}) := \mathsf{first}\big(i ⦂ \mathbb{B}^{\mathbb{Y}} \mapsto \mathsf{CheckDiversifier}(\mathsf{truncate}_{(\ell_{\mathsf{d}}/8)}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([3, i]))) ⦂ \mathbb{J}^{(r)*} \cup \{\bot\}\big).$$

For a random *spending key*, $\mathsf{DefaultDiversifier}$ returns $\bot$ with probability approximately $2^{-256}$; if this happens, discard the key and repeat with a different $\mathsf{sk}$.

**Notes:**

- The protocol does not prevent using the *diversifier* $\mathsf{d}$ to produce *"vanity"* addresses that start with a meaningful string when encoded in *Bech32* (see § 5.6.3.1 ***'Sapling Payment Addresses'*** on p. 107). Users and writers of software that generates addresses should be aware that this provides weaker privacy properties than a randomly chosen *diversifier*, since a vanity address can obviously be distinguished, and might leak more information than intended as to who created it.
- Similarly, address generators **MAY** encode information in the *diversifier* that can be recovered by the recipient of a payment to determine which *diversified payment address* was used. It is **RECOMMENDED** that such *diversifiers* be randomly chosen unique values used to index into a database, rather than directly encoding the needed data.

**Non-normative notes:**

- Assume that $\mathsf{PRF}^{\mathsf{expand}}$ is a *PRF* with output range $\mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{PRFexpand}}/8]}$, where $2^{\ell_{\mathsf{PRFexpand}}}$ is large compared to $r_{\mathbb{J}}$.

  Define $f ⦂ \mathbb{B}^{[\ell_{\mathsf{sk}}]} \times \mathbb{B}^{\mathbb{Y}[\mathbb{N}]} \to \mathbb{F}_{r_{\mathbb{J}}}$ by $f_{\mathsf{sk}}(t) := \mathsf{ToScalar}^{\mathsf{Sapling}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(t))$.

  $f$ is also a *PRF* since $\mathsf{LEOS2IP}_{\ell_{\mathsf{PRFexpand}}} ⦂ \mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{PRFexpand}}/8]} \to \{0 .. 2^{\ell_{\mathsf{PRFexpand}}}-1\}$ is injective; the bias introduced by reduction modulo $r_{\mathbb{J}}$ is small because § 5.3 ***'Constants'*** on p. 67 defines $\ell_{\mathsf{PRFexpand}}$ as 512, while $r_{\mathbb{J}}$ has length 252 bits. It follows that the distribution of $\mathsf{ask}$, i.e. $\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([0]) : \mathsf{sk} \xleftarrow{\mathrm{R}} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$, is computationally indistinguishable from $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{GenPrivate}()$ defined in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.
- The distribution of $\mathsf{nsk}$, i.e. $\mathsf{ToScalar}^{\mathsf{Sapling}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([1])) : \mathsf{sk} \xleftarrow{\mathrm{R}} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$, is computationally indistinguishable from the uniform distribution on $\mathbb{F}_{r_{\mathbb{J}}}$. Since $\mathsf{nsk} ⦂ \mathbb{F}_{r_{\mathbb{J}}} \mapsto \mathsf{repr}_{\mathbb{J}}\big([\mathsf{nsk}]\, \mathcal{H}^{\mathsf{Sapling}} ⦂ \mathbb{J}^{(r)}_{\star}\big)$ is bijective, the distribution of $\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$ will be computationally indistinguishable from uniform on $\mathbb{J}^{(r)}_{\star}$ (the keyspace of $\mathsf{PRF}^{\mathsf{nfSapling}}$).
- The zcashd wallet picks *diversifiers* as in [ZIP-32], rather than using the default *diversifier* specified above.

### 4.2.3 Orchard Key Components

Let $\ell_{\mathsf{PRFexpand}}$, $\ell_{\mathsf{sk}}$, $\ell_{\mathsf{ovk}}$, $\ell_{\mathsf{d}}$, and $\ell_{\mathsf{dk}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathbb{P}$, $\mathbb{P}_x$, $\mathsf{repr}_{\mathbb{P}}$, $\ell_{\mathbb{P}}$, $q_{\mathbb{P}}$, and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let $\mathsf{Extract}_{\mathbb{P}}$ be as defined in § 5.4.9.7 *'Coordinate Extractor for* Pallas' on p. 98.

Let $\mathsf{GroupHash}^{\mathbb{P}}$ be as defined in § 5.4.9.8 ***'Group Hash into*** Pallas ***and*** Vesta' on p. 98.

Let $\mathsf{PRF}^{\mathsf{expand}}$ and $\mathsf{PRF}^{\mathsf{ockOrchard}}$, instantiated in § 5.4.2 *'Pseudo Random Functions'* on p. 79, be *Pseudo Random Functions*.

Let $\mathsf{PRP}^{\mathsf{d}} ⦂ \mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{dk}}/8]} \times \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{B}^{[\ell_{\mathsf{d}}]}$ be as defined in § 5.4.4 *'Pseudo Random Permutations'* on p. 81.

Let $\mathsf{KA}^{\mathsf{Orchard}}$, instantiated in § 5.4.5.5 ***'Orchard Key Agreement'*** on p. 82, be a *key agreement scheme*.

Let $\mathsf{Commit}^{\mathsf{ivk}}$, instantiated in § 5.4.8.4 *'Sinsemilla commitments'* on p. 90, be a *commitment scheme*.

Let $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$, instantiated in § 5.4.1.6 '$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ *Hash Functions*' on p. 71, be a *hash function*.

Let $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ instantiated in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87 be a *signature scheme with re-randomizable keys*.

Let $\mathsf{I2LEBSP}$, $\mathsf{I2LEOSP}$, and $\mathsf{LEOS2IP}$ be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Define $\mathsf{ToBase}^{\mathsf{Orchard}}(x : \mathbb{BY}^{[\ell_{\mathsf{PRFexpand}}/8]}) := \mathsf{LEOS2IP}_{\ell_{\mathsf{PRFexpand}}}(x) \pmod{q_{\mathbb{P}}}$.

Define $\mathsf{ToScalar}^{\mathsf{Orchard}}(x : \mathbb{BY}^{[\ell_{\mathsf{PRFexpand}}/8]}) := \mathsf{LEOS2IP}_{\ell_{\mathsf{PRFexpand}}}(x) \pmod{r_{\mathbb{P}}}$.

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

From this *spending key*, the *Spend authorizing key* $\mathsf{ask} : \mathbb{F}^*_{r_{\mathbb{P}}}$, the *Spend validating key* $\mathsf{ak} : \mathbb{P}_x$, the *nullifier deriving key* $\mathsf{nk} : \mathbb{F}_{q_{\mathbb{P}}}$, the $\mathsf{Commit}^{\mathsf{ivk}}$ randomness $\mathsf{rivk} : \mathbb{F}_{r_{\mathbb{P}}}$, the *incoming viewing key* $\mathsf{ivk} : \{0 \,..\, q_{\mathbb{P}} - 1\}$, and the *outgoing viewing key* $\mathsf{ovk} : \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]}$ are derived as follows:

```
let mutable $\mathsf{ask} \leftarrow \mathsf{ToScalar}^{\mathsf{Orchard}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([6]))$
let $\mathsf{nk} = \mathsf{ToBase}^{\mathsf{Orchard}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([7]))$
let $\mathsf{rivk} = \mathsf{ToScalar}^{\mathsf{Orchard}}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([8]))$
if $\mathsf{ask} = 0$, discard this key and repeat with a new $\mathsf{sk}$.
let $\mathsf{ak}^{\mathbb{P}} = \mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{DerivePublic}(\mathsf{ask})$
if the last bit (that is, the $\tilde{y}$ bit) of $\mathsf{repr}_{\mathbb{P}}(\mathsf{ak}^{\mathbb{P}})$ is $1$:
    set $\mathsf{ask} \leftarrow -\mathsf{ask}$
let $\mathsf{ak} = \mathsf{Extract}_{\mathbb{P}}(\mathsf{ak}^{\mathbb{P}})$
let $\mathsf{ivk} = \mathsf{Commit}^{\mathsf{ivk}}_{\mathsf{rivk}}(\mathsf{ak}, \mathsf{nk})$
let $K = \mathsf{I2LEBSP}_{\ell_{\mathsf{sk}}}(\mathsf{rivk})$
let $R = \mathsf{PRF}^{\mathsf{expand}}_{K}([\mathtt{0x82}] \,||\, \mathsf{I2LEOSP}_{256}(\mathsf{ak}) \,||\, \mathsf{I2LEOSP}_{256}(\mathsf{nk}))$
let $\mathsf{dk}$ be the first $\ell_{\mathsf{dk}}/8$ bytes of $R$ and let $\mathsf{ovk}$ be the remaining $\ell_{\mathsf{ovk}}/8$ bytes of $R$.
```

As explained in § 3.1 ***'Payment Addresses and Keys'*** on p. 12, **Orchard** allows the efficient creation of multiple *diversified payment addresses* with the same spending authority. A group of such addresses shares the same *full viewing key*, *incoming viewing key*, and *outgoing viewing key*.

To create a new *diversified payment address* given an *incoming viewing key* $\mathsf{ivk}$, pick a *diversifier index* $\mathsf{index}$ uniformly at random from $\mathbb{B}^{[\ell_{\mathsf{d}}]}$. Then calculate the *diversified transmission key* $\mathsf{pk_d}$:

$$\mathsf{d} := \mathsf{PRP}^{\mathsf{d}}_{\mathsf{dk}}(\mathsf{index})$$

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

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

The resulting *diversified payment address* is $(\mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{pk_d} : \mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public})$.

The *diversified payment address* with *diversifier index* $0$ is called the *default diversified payment address*.

**Notes:**

- The protocol does not prevent using the *diversifier* $\mathsf{d}$ to produce *"vanity"* addresses that start with a meaningful string when encoded in *Bech32* (see § 5.6.4.2 ***'Orchard Raw Payment Addresses'*** on p. 110). Users and writers of software that generates addresses should be aware that this provides weaker privacy properties than a randomly chosen *diversifier*, since a vanity address can obviously be distinguished, and might leak more information than intended as to who created it.
- Similarly, address generators **MAY** encode information in the *diversifier index* that can be recovered by the recipient of a payment, given the *diversifier key*.

TODO: Security analysis of the uses of $\mathsf{ToScalar}^{\mathsf{Orchard}}$ and $\mathsf{ToBase}^{\mathsf{Orchard}}$.

## 4.3 JoinSplit Descriptions

A *JoinSplit transfer*, as specified in § 3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 16, is encoded in *transactions* as a *JoinSplit description*.

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

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\ell_{\mathsf{Seed}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, and MAX_MONEY be as defined in § 5.3 ***'Constants'*** on p. 67.

Let hSigCRH be as defined in § 4.1.1 ***'Hash Functions'*** on p. 20.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 4.1.5 ***'Key Agreement'*** on p. 23.

Let Sym be as defined in § 4.1.4 ***'Symmetric Encryption'*** on p. 23.

Let ZKJoinSplit be as defined in § 4.1.13 ***'Zero-Knowledge Proving System'*** on p. 31.

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

- $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent transaction value pool*;
- $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} ⦂ \{0 .. \mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent transaction value pool*;
- $\mathsf{rt}^{\mathsf{Sprout}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in § 3.3 ***'The Block Chain'*** on p. 15, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is the sequence of *nullifiers* for the input *notes*;
- $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} ⦂ \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} ⦂ \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ is a key agreement *public key*, used to derive the key for encryption of the *transmitted notes ciphertext* (§ 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59);
- $\mathsf{randomSeed} ⦂ \mathbb{B}^{[\ell_{\mathsf{Seed}}]}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}][\mathrm{N}^{\mathsf{old}}]}$ is a sequence of tags that bind $\mathsf{h}_{\mathsf{Sig}}$ to each $\mathsf{a}_{\mathsf{sk}}$ of the input *notes*;
- $\pi_{\mathsf{ZKJoinSplit}} ⦂ \mathsf{ZKJoinSplit.Proof}$ is a *zk proof* with *primary input* $(\mathsf{rt}^{\mathsf{Sprout}}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}})$ for the *JoinSplit statement* defined in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54 (this is a BCTV14 proof before **Sapling** activation, and a Groth16 proof after **Sapling** activation);
- $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}} ⦂ \mathsf{Sym}.\mathbf{C}^{[\mathrm{N}^{\mathsf{new}}]}$ 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.
- [**Canopy** onward] $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ **MUST** be zero.

## 4.4 Spend Descriptions

A *Spend transfer*, as specified in § 3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 17, is encoded in *transactions* as a *Spend description*.

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

Each *Spend description* is authorized by a signature, called the *spend authorization signature*.

Let $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$ and $\ell_{\mathsf{PRFnfSapling}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathcal{O}_{\mathbb{J}}$, $\mathsf{abst}_{\mathbb{J}}$, $\mathsf{repr}_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ be as defined in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ be as defined in § 4.15 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 52.

Let $\mathsf{ZKSpend}$ be as defined in § 4.1.13 ***'Zero-Knowledge Proving System'*** on p. 31.

A *Spend description* comprises $(\mathsf{cv}, \mathsf{rt}^{\mathsf{Sapling}}, \mathsf{nf}, \mathsf{rk}, \pi_{\mathsf{ZKSpend}}, \mathsf{spendAuthSig})$ where

- $\mathsf{cv} ⦂ \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ is the *value commitment* to the value of the input *note*;
- $\mathsf{rt}^{\mathsf{Sapling}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in § 3.3 ***'The Block Chain'*** on p. 15, for the output *treestate* of a previous *block*;
- $\mathsf{nf} ⦂ \mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{PRFnfSapling}}/8]}$ is the *nullifier* for the input *note*;
- $\mathsf{rk} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$ is a randomized *validating key* that should be used to validate $\mathsf{spendAuthSig}$;
- $\pi_{\mathsf{ZKSpend}} ⦂ \mathsf{ZKSpend}.\mathsf{Proof}$ is a *zk-SNARK proof* with *primary input* $(\mathsf{cv}, \mathsf{rt}^{\mathsf{Sapling}}, \mathsf{nf}, \mathsf{rk})$ for the *Spend statement* defined in § 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55;
- $\mathsf{spendAuthSig} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Signature}$ is a *spend authorization signature*, validated as specified in § 4.15 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 52.

**Consensus rules:**

- Elements of a *Spend description* **MUST** be valid encodings of the types given above.
- $\mathsf{cv}$ and $\mathsf{rk}$ **MUST NOT** be of small order, i.e. $[h_{\mathbb{J}}]\,\mathsf{cv}$ **MUST NOT** be $\mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\,\mathsf{rk}$ **MUST NOT** be $\mathcal{O}_{\mathbb{J}}$.
- The proof $\pi_{\mathsf{ZKSpend}}$ **MUST** be valid given a *primary input* formed from the other fields except $\mathsf{spendAuthSig}$ — i.e. $\mathsf{ZKSpend}.\mathsf{Verify}\big((\mathsf{cv}, \mathsf{rt}^{\mathsf{Sapling}}, \mathsf{nf}, \mathsf{rk}), \pi_{\mathsf{ZKSpend}}\big) = 1$.
- Let $\mathsf{SigHash}$ be the *SIGHASH transaction hash* of this *transaction*, not associated with an input, as defined in § 4.10 ***'SIGHASH Transaction Hashing'*** on p. 45 using $\mathsf{SIGHASH\_ALL}$.

  The *spend authorization signature* **MUST** be a valid $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ signature over $\mathsf{SigHash}$ using $\mathsf{rk}$ as the *validating key* — i.e. $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Validate}_{\mathsf{rk}}(\mathsf{SigHash}, \mathsf{spendAuthSig}) = 1$.

  [**NU5** onward] As specified in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85, the validation of the $\underline{R}$ component of the signature changes to prohibit *non-canonical* encodings.

**Non-normative notes:**

- The check that $\mathsf{rk}$ is not of small order is technically redundant with a check in the *Spend circuit*, but it is simple and cheap to also check this outside the circuit.
- The rule that $\mathsf{cv}$ and $\mathsf{rk}$ **MUST** not be small-order has the effect of also preventing non-canonical encodings of these fields, as required by [ZIP-216]. That is, it is necessarily the case that $\mathsf{repr}_{\mathbb{J}}\big(\mathsf{abst}_{\mathbb{J}}(\mathsf{cv})\big) = \mathsf{cv}$ and $\mathsf{repr}_{\mathbb{J}}\big(\mathsf{abst}_{\mathbb{J}}(\mathsf{rk})\big) = \mathsf{rk}$.

## 4.5 Output Descriptions

An *Output transfer*, as specified in § 3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 17, is encoded in *transactions* as an *Output description*.

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

Let $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathcal{O}_{\mathbb{J}}$, $\mathsf{abst}_{\mathbb{J}}$, $\mathsf{repr}_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ be as defined in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in § 4.1.5 ***'Key Agreement'*** on p. 23.

Let $\mathsf{Sym}$ be as defined in § 4.1.4 ***'Symmetric Encryption'*** on p. 23.

Let $\mathsf{ZKOutput}$ be as defined in § 4.1.13 ***'Zero-Knowledge Proving System'*** on p. 31.

An *Output description* comprises $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi_{\mathsf{ZKOutput}})$ where

- $\mathsf{cv} ⦂ \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ is the *value commitment* to the value of the output *note*;
- $\mathsf{cm}_u ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}$ is the result of applying $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ (defined in § 5.4.9.4 ***'Coordinate Extractor for*** Jubjub' on p. 96) to the *note commitment* for the output *note*;
- $\mathsf{epk} ⦂ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public}$ is a key agreement *public key*, used to derive the key for encryption of the *transmitted note ciphertext* (§ 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60);
- $\mathsf{C}^{\mathsf{enc}} ⦂ \mathsf{Sym}.\mathbf{C}$ is a ciphertext component for the encrypted output *note*;
- $\mathsf{C}^{\mathsf{out}} ⦂ \mathsf{Sym}.\mathbf{C}$ is a ciphertext component that allows the holder of a *full viewing key* to recover the recipient *diversified transmission key* $\mathsf{pk_d}$ and the *ephemeral private key* $\mathsf{esk}$ (and therefore the entire *note plaintext*);
- $\pi_{\mathsf{ZKOutput}} ⦂ \mathsf{ZKOutput}.\mathsf{Proof}$ is a *zk-SNARK proof* with *primary input* $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk})$ for the *Output statement* defined in § 4.17.3 ***'Output Statement (Sapling)'*** on p. 56.

**Consensus rules:**

- Elements of an *Output description* **MUST** be valid encodings of the types given above.
- $\mathsf{cv}$ and $\mathsf{epk}$ **MUST NOT** be of small order, i.e. $[h_{\mathbb{J}}]\,\mathsf{cv}$ **MUST NOT** be $\mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\,\mathsf{epk}$ **MUST NOT** be $\mathcal{O}_{\mathbb{J}}$.
- The proof $\pi_{\mathsf{ZKOutput}}$ **MUST** be valid given a *primary input* formed from the other fields except $\mathsf{C}^{\mathsf{enc}}$ and $\mathsf{C}^{\mathsf{out}}$ — i.e. $\mathsf{ZKSpend}.\mathsf{Verify}\big((\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk}), \pi_{\mathsf{ZKOutput}}\big) = 1$.

**Non-normative note:** The rule that $\mathsf{cv}$ and $\mathsf{epk}$ **MUST** not be small-order, has the effect of also preventing non-canonical encodings of these fields, as required by [ZIP-216]. That is, it is necessarily the case that $\mathsf{repr}_{\mathbb{J}}\big(\mathsf{abst}_{\mathbb{J}}(\mathsf{cv})\big) = \mathsf{cv}$ and $\mathsf{repr}_{\mathbb{J}}\big(\mathsf{abst}_{\mathbb{J}}(\mathsf{epk})\big) = \mathsf{epk}$.

## 4.6 Action Descriptions

An *Action transfer*, as specified in § 3.7 ***'Action Transfers and their Descriptions'*** on p. 18, is encoded in *transactions* as an *Action description*.

Each version 5 *transaction* includes a sequence of zero or more *Action descriptions*. (Version 4 *transactions* cannot contain *Action descriptions*.)

Each *Action description* is authorized by a signature, called the *spend authorization signature*.

Let $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathbb{P}_x$ and $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let $\mathsf{Extract}_{\mathbb{P}}$ be as defined in § 5.4.9.7 *‘Coordinate Extractor for* Pallas*’* on p. 98.

Let $\mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output}$ be as defined in § 4.1.8 *‘Commitment’* on p. 27.

Let $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ be as defined in § 4.15 *‘Spend Authorization Signature (**Sapling** and **Orchard**)’* on p. 52.

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as defined in § 4.1.5 *‘Key Agreement’* on p. 23.

Let Sym be as defined in § 4.1.4 *‘Symmetric Encryption’* on p. 23.

Let ZKAction be as defined in § 4.1.13 *‘Zero-Knowledge Proving System’* on p. 31.

An *Action description* comprises $(\mathsf{cv}^{\mathsf{net}}, \mathsf{rt}^{\mathsf{Orchard}}, \mathsf{nf}, \mathsf{rk}, \mathsf{spendAuthSig}, \mathsf{cm}_x, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \mathsf{enableSpend}, \mathsf{enableOutput}, \pi)$ where

- $\mathsf{cv}^{\mathsf{net}} ⦂ \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output}$ is the *value commitment* to the value of the input *note* minus the value of the output *note*;
- $\mathsf{rt}^{\mathsf{Orchard}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}]}$ is an *anchor*, as defined in § 3.3 *‘The Block Chain’* on p. 15, for the output *treestate* of a previous *block*;
- $\mathsf{nf} ⦂ \mathbb{F}_{q_{\mathbb{P}}}$ is the *nullifier* for the input *note*;
- $\mathsf{rk} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{Public}$ is a randomized *validating key* that should be used to validate spendAuthSig;
- $\mathsf{spendAuthSig} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{Signature}$ is a *spend authorization signature*, validated as specified in § 4.15 *‘Spend Authorization Signature (**Sapling** and **Orchard**)’* on p. 52;
- $\mathsf{cm}_x ⦂ \mathbb{F}_{q_{\mathbb{P}}}$ is the result of applying $\mathsf{Extract}_{\mathbb{P}}$ to the *note commitment* for the output *note*;
- $\mathsf{epk} ⦂ \mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public}$ is a key agreement *public key*, used to derive the key for encryption of the *transmitted note ciphertext* (§ 4.19 *‘In-band secret distribution (**Sapling** and **Orchard**)’* on p. 60);
- $\mathsf{C}^{\mathsf{enc}} ⦂ \mathsf{Sym}.\mathbf{C}$ is a ciphertext component for the encrypted output *note*;
- $\mathsf{C}^{\mathsf{out}} ⦂ \mathsf{Sym}.\mathbf{C}$ is a ciphertext component that allows the holder of a *full viewing key* to recover the recipient *diversified transmission key* $\mathsf{pk}_{\mathsf{d}}$ and the *ephemeral private key* esk (and therefore the entire *note plaintext*);
- $\mathsf{enableSpend} ⦂ \mathbb{B}$ is a flag that is set in order to enable non-zero-valued spends in this Action;
- $\mathsf{enableOutput} ⦂ \mathbb{B}$ is a flag that is set in order to enable non-zero-valued outputs in this Action;
- $\pi ⦂ \mathsf{ZKAction}.\mathsf{Proof}$ is a *zk-SNARK proof* with *primary input* $(\mathsf{cv}, \mathsf{rt}^{\mathsf{Orchard}}, \mathsf{nf}, \mathsf{rk}, \mathsf{cm}_x, \mathsf{enableSpend}, \mathsf{enableOutput})$ for the *Action statement* defined in § 4.17.4 *‘Action Statement (**Orchard**)’* on p. 57.

**Note:** The $\mathsf{rt}^{\mathsf{Orchard}}$, enableSpend, and enableOutput components are the same for all *Action transfers* in a *transaction*. They are encoded once in the *transaction* body (specified in § 7.1 *‘Transaction Encoding and Consensus’* on p. 114), not in the `ActionDescription` structure. $\pi$ is aggregated with other Action proofs and encoded in the `proofsOrchard` field of a *transaction*.

**Consensus rules:**

- Elements of an *Action description* **MUST** be canonical encodings of the types given above.
- Let SigHash be the *SIGHASH transaction hash* of this *transaction*, not associated with an input, as defined in § 4.10 *‘SIGHASH Transaction Hashing’* on p. 45 using SIGHASH_ALL.

  The *spend authorization signature* **MUST** be a valid $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ signature over SigHash using rk as the *validating key* — i.e. $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{Validate}_{\mathsf{rk}}(\mathsf{SigHash}, \mathsf{spendAuthSig}) = 1$. As specified in § 5.4.7 ‘RedDSA, RedJubjub, *and* RedPallas’ on p. 85, validation of the $\underline{R}$ component of the signature prohibits *non-canonical* encodings.
- The proof $\pi_{\mathsf{ZKAction}}$ **MUST** be valid given a *primary input* $(\mathsf{cv}, \mathsf{rt}^{\mathsf{Orchard}}, \mathsf{nf}, \mathsf{rk}, \mathsf{cm}_x, \mathsf{enableSpend}, \mathsf{enableOutput})$ — i.e. $\mathsf{ZKAction}.\mathsf{Verify}\big((\mathsf{cv}, \mathsf{rt}^{\mathsf{Orchard}}, \mathsf{nf}, \mathsf{rk}, \mathsf{cm}_x, \mathsf{enableSpend}, \mathsf{enableOutput}), \pi_{\mathsf{ZKAction}}\big) = 1$.

**Non-normative notes:**

- $\mathsf{cv}$, $\mathsf{rk}$, and $\mathsf{epk}$ can be the zero point $\mathcal{O}_{\mathbb{P}}$.
- Despite the return type of $\mathsf{Extract}_{\mathbb{P}}$ being $\mathbb{P}_x$, $\mathsf{nf}$ and $\mathsf{cm}_x$ are ***not*** checked to be in $\mathbb{P}_x$; they are only checked to be canonical encodings of $\mathbb{F}_{q_{\mathbb{P}}}$ elements.

## 4.7 Sending Notes

### 4.7.1 Sending Notes (Sprout)

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

Let JoinSplitSig be as specified in § 4.1.7 ***'Signature'*** on p. 24.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as specified in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\ell_{\mathsf{Seed}}$ and $\ell_{\varphi}^{\mathsf{Sprout}}$ be as specified in § 5.3 ***'Constants'*** on p. 67.

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

$$\mathtt{joinSplitPrivKey} \xleftarrow{\mathsf{R}} \mathsf{JoinSplitSig.GenPrivate}()$$
$$\mathtt{joinSplitPubKey} := \mathsf{JoinSplitSig.DerivePublic}(\mathtt{joinSplitPrivKey}).$$

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

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

$\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ are then encrypted to the recipient *transmission keys* $\mathsf{pk}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, giving the *transmitted notes ciphertext* $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$, as described in § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59.

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

After generating all of the *JoinSplit descriptions*, the sender obtains $\mathsf{dataToBeSigned} : \mathbb{BY}^{[\mathbb{N}]}$ as described in § 4.11 ***'Non-malleability (Sprout)'*** on p. 46, and signs it with the private *JoinSplit signing key*:

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

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

[**Canopy** onward] **Note:** [ZIP-211] specifies that nodes and wallets **MUST** disable any facilities to send to **Sprout** addresses. This **SHOULD** be made clear in user interfaces and API documentation.

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

### 4.7.2 Sending Notes (Sapling)

In order to send **Sapling** *shielded* value, the sender constructs a *transaction* with one or more *Output descriptions*.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as specified in § 4.1.5 ***'Key Agreement'*** on p. 23.

Let $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ be as specified in § 4.1.1 ***'Hash Functions'*** on p. 20.

Let $\mathsf{ToScalar}^{\mathsf{Sapling}}$ be as specified in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

Let $\mathsf{repr}_{\mathbb{J}}$ and $r_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let I2LEOSP be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Let $\mathsf{ovk}$ be a **Sapling** *outgoing viewing key* that is intended to be able to decrypt this payment. This may be one of:

- the *outgoing viewing key* for the address (or one of the addresses) from which the payment was sent;
- the *outgoing viewing key* for all payments associated with an *"account"*, to be defined in [ZIP-32];
- $\bot$, if the sender should not be able to decrypt the payment once it has deleted its own copy.

**Note:** Choosing $\mathsf{ovk} = \bot$ is useful if the sender prefers to obtain forward secrecy of the payment information with respect to compromise of its own secrets.

Let CanopyActivationHeight be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{leadByte}$ be the *note plaintext lead byte*. This **MUST** be `0x01` if for the next *block*, $\mathsf{height} < \mathsf{CanopyActivationHeight}$, or `0x02` if $\mathsf{height} \geq \mathsf{CanopyActivationHeight}$.

For each *Output description*, the sender selects a value $\mathsf{v} : \{0\,..\,\mathsf{MAX\_MONEY}\}$ and a destination **Sapling** *shielded payment address* $(\mathsf{d}, \mathsf{pk_d})$, and then performs the following steps:

```
Check that $\mathsf{pk_d}$ is of type $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup}$, i.e. it MUST be a valid ctEdwards curve point on the Jubjub curve (as defined in § 5.4.9.3 'Jubjub' on p. 94), and $[r_{\mathbb{J}}]\,\mathsf{pk_d} = \mathcal{O}_{\mathbb{J}}$.
Calculate $\mathsf{g_d} = \mathsf{DiversifyHash}^{\mathsf{Sapling}}(\mathsf{d})$ and check that $\mathsf{g_d} \neq \bot$.
Choose a uniformly random commitment trapdoor $\mathsf{rcv} \xleftarrow{R} \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$.
If $\mathsf{leadByte} = \mathtt{0x01}$:
    Choose a uniformly random ephemeral private key $\mathsf{esk} \xleftarrow{R} \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private} \setminus \{0\}$.
    Choose a uniformly random commitment trapdoor $\mathsf{rcm} \xleftarrow{R} \mathsf{NoteCommit}.\mathsf{GenTrapdoor}()$.
    Set $\mathsf{rseed} := \mathsf{I2LEOSP}_{256}(\mathsf{rcm})$.
else:
    Choose uniformly random $\mathsf{rseed} \xleftarrow{R} \mathbb{BY}^{[32]}$.
    Derive $\mathsf{esk} = \mathsf{ToScalar}^{\mathsf{Sapling}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([4])\big)$.
    Derive $\mathsf{rcm} = \mathsf{ToScalar}^{\mathsf{Sapling}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big)$.
Let $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v})$.
Let $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v})$.
Let $\mathbf{np} = (\mathsf{leadByte}, \mathsf{d}, \mathsf{v}, \mathsf{rseed}, \mathsf{memo})$.
Encrypt $\mathbf{np}$ to the recipient diversified transmission key $\mathsf{pk_d}$ with diversified base $\mathsf{g_d}$, and to the outgoing viewing key $\mathsf{ovk}$, giving the transmitted note ciphertext $(\mathsf{epk}, \mathsf{C^{enc}}, \mathsf{C^{out}})$. This procedure is described in § 4.19.1 'Encryption (Sapling and Orchard)' on p. 61; it also uses cv and cmu to derive ock, and takes esk as input.
Generate a proof $\pi_{\mathsf{ZKOutput}}$ for the Output statement in § 4.17.3 'Output Statement (Sapling)' on p. 56.
Return $(\mathsf{cv}, \mathsf{cm}, \mathsf{epk}, \mathsf{C^{enc}}, \mathsf{C^{out}}, \pi_{\mathsf{ZKOutput}})$.
```

In order to minimize information leakage, the sender **SHOULD** randomize the order of *Output descriptions* in a *transaction*. Other considerations relating to information leakage from the structure of *transactions* are beyond the scope of this specification. The encoded *transaction* is submitted to the peer-to-peer network.

### 4.7.3 Sending Notes (Orchard)

In order to send **Orchard** *shielded* value, the sender constructs a *transaction* with one or more *Action descriptions*. This section describes how to produce the output-related fields of an *Action description*.

Let $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ and $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ be as specified in § 4.1.8 *'Commitment'* on p. 27.

Let $\mathsf{PRF}^{\mathsf{expand}}_{Orchard}$ be as specified in **§?** *'??'* on p. **??**.

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as specified in § 4.1.5 *'Key Agreement'* on p. 23.

Let $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ be as specified in § 4.1.1 *'Hash Functions'* on p. 20.

Let $\mathsf{ToScalar}^{\mathsf{Orchard}}$ and $\mathsf{ToBase}^{\mathsf{Orchard}}$ be as specified in § 4.2.3 *'**Orchard** Key Components'* on p. 34.

Let $\mathsf{repr}_{\mathbb{P}}$ and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let I2LEOSP be as defined in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

Let $\mathsf{ovk}$ be an **Orchard** *outgoing viewing key* that is intended to be able to decrypt this payment. The considerations for choosing *outgoing viewing keys* are as described for **Sapling** in § 4.7.2 *'Sending Notes (**Sapling**)'* on p. 41.

Let $\mathsf{leadByte}$ be the *note plaintext lead byte*, which **MUST** be `0x02`.

For each *Action description*, the sender selects a value $\mathsf{v} \mathrel{⦂} \{0 \,..\, \mathsf{MAX\_MONEY}\}$ and a destination **Orchard** *shielded payment address* $(\mathsf{d}, \mathsf{pk_d})$, and then performs the following steps:

Check that $\mathsf{pk_d}$ is of type $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public}$, i.e. it **MUST** be a valid *short Weierstrass curve* point on the Pallas curve (as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97).

Calculate $\mathsf{g_d} = \mathsf{DiversifyHash}^{\mathsf{Orchard}}(\mathsf{d})$.

Choose a uniformly random *commitment trapdoor* $\mathsf{rcv} \xleftarrow{\mathrm{R}} \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{GenTrapdoor}()$.

Choose uniformly random $\mathsf{rseed} \xleftarrow{\mathrm{R}} \mathbb{B}^{\mathbb{Y}[32]}$.

Derive $\mathsf{esk} = \mathsf{ToScalar}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([4])\big)$.

Derive $\mathsf{rcm} = \mathsf{ToScalar}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big)$.

Derive $\psi = \mathsf{ToBase}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([9])\big)$.

Let $\rho$ be equal to $\mathsf{nf}^{\mathsf{old}}$ from the same *Action description*.

Let $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v})$.

Let $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{P}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{P}}(\mathsf{pk_d}), \mathsf{v}, \rho, \psi)$.

Let $\mathbf{np} = (\mathsf{leadByte}, \mathsf{d}, \mathsf{v}, \mathsf{rseed}, \mathsf{memo})$.

Encrypt $\mathbf{np}$ to the recipient *diversified transmission key* $\mathsf{pk_d}$ with *diversified base* $\mathsf{g_d}$, and to the *outgoing viewing key* $\mathsf{ovk}$, giving the *transmitted note ciphertext* $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$. This procedure is described in § 4.19.1 *'Encryption (**Sapling** and **Orchard**)'* on p. 61; it also uses $\mathsf{cv}$ and `cmx` to derive $\mathsf{ock}$, and takes $\mathsf{esk}$ as input.

For an **Orchard** *note*, generate a proof $\pi$ for the *Action statement* in § 4.17.4 *'Action Statement (**Orchard**)'* on p. 57.

Return $(\mathsf{cv}, \mathsf{cm}, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi)$.

In order to minimize information leakage, the sender **SHOULD** randomize the order of *Action descriptions* in a *transaction*. Other considerations relating to information leakage from the structure of *transactions* are beyond the scope of this specification. The encoded *transaction* is submitted to the peer-to-peer network.

**Non-normative note:** The inputs [4] and [5] are used as inputs to $\mathsf{PRF}^{\mathsf{expand}}$ in both **Sapling** and **Orchard** shielded protocols. Since a fresh $\mathsf{rseed}$ is generated for each *note*, this should have no negative effect on security.

## 4.8 Dummy Notes

### 4.8.1 Dummy Notes (Sprout)

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

Let $\ell_{\mathsf{a_{sk}}}$ and $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 21.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27.

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

- Generate a new uniformly random *spending key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i} \overset{\mathrm{R}}{\leftarrow} \mathbb{B}^{[\ell_{\mathsf{a_{sk}}}]}$ and derive its *paying key* $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},i}$.
- Set $\mathsf{v}^{\mathsf{old}}_i = 0$.
- Choose uniformly random $\rho^{\mathsf{old}}_i \overset{\mathrm{R}}{\leftarrow} \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}$ and $\mathsf{rcm}^{\mathsf{old}}_i \overset{\mathrm{R}}{\leftarrow} \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{GenTrapdoor}()$.
- Compute $\mathsf{nf}^{\mathsf{old}}_i = \mathsf{PRF}^{\mathsf{nfSprout}}_{\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},i}}(\rho^{\mathsf{old}}_i)$.
- Let $\mathsf{path}_i$ 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*.

### 4.8.2 Dummy Notes (Sapling)

In **Sapling** there is no need to use *dummy notes* simply in order to fill otherwise unused inputs as in the case of a *JoinSplit description*; nevertheless it may be useful for privacy to obscure the number of real *shielded inputs* from **Sapling** *notes*.

Let $\ell_{\mathsf{sk}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27.

Let $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ be as specified in § 4.1.1 *'Hash Functions'* on p. 20.

Let $\mathsf{ToScalar}^{\mathsf{Sapling}}$ be as specified in § 4.2.2 *'**Sapling** Key Components'* on p. 32.

Let $\mathsf{repr}_{\mathbb{J}}$ and $r_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{PRF}^{\mathsf{nfSapling}}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 21.

Let $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27.

A *Spend description* for a *dummy* **Sapling** input *note* is constructed as follows:

- Choose uniformly random $\mathsf{sk} \overset{\mathrm{R}}{\leftarrow} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$.
- Generate a *full viewing key* $(\mathsf{ak}, \mathsf{nk})$ and a *diversified payment address* $(\mathsf{d}, \mathsf{pk_d})$ for $\mathsf{sk}$ as described in § 4.2.2 *'**Sapling** Key Components'* on p. 32.
- Let $\mathsf{v} = 0$ and $\mathsf{pos} = 0$.
- Choose uniformly random $\mathsf{rcv} \overset{\mathrm{R}}{\leftarrow} \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$.
- Choose uniformly random $\mathsf{rseed} \overset{\mathrm{R}}{\leftarrow} \mathbb{BY}^{[32]}$.
- Derive $\mathsf{rcm} = \mathsf{ToScalar}^{\mathsf{Sapling}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big)$.

- Let $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v})$.
- Let $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}\big)$.
- Let $\rho\star = \mathsf{repr}_{\mathbb{J}}\big(\mathsf{MixingPedersenHash}(\mathsf{cm}, \mathsf{pos})\big)$.
- Let $\mathsf{nk}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$.
- Let $\mathsf{nf} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$.
- Construct a *dummy Merkle path* $\mathsf{path}$ for use in the *auxiliary input* to the *Spend statement* (this will not be checked, because $\mathsf{v} = 0$).

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

### 4.8.3 Dummy Notes (Orchard)

As for **Sapling**, it may be useful for privacy to obscure the number of real *shielded inputs* from **Orchard** *notes*.

Let $\ell_{\mathsf{sk}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ and $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27.

Let $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ be as specified in § 4.1.1 *'Hash Functions'* on p. 20.

Let $\mathsf{ToScalar}^{\mathsf{Orchard}}$ and $\mathsf{ToBase}^{\mathsf{Orchard}}$ be as specified in § 4.2.3 *'Orchard Key Components'* on p. 34.

Let $\mathsf{repr}_{\mathbb{P}}$ and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let DeriveNullifier be as defined in § 4.16 *'Note Commitments and Nullifiers'* on p. 53.

Let $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27.

The spend-related fields of an *Action description* for a *dummy* **Orchard** input *note* are constructed as follows:

- Choose uniformly random $\mathsf{sk} \xleftarrow{\mathrm{R}} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$.
- Generate a *full viewing key* $(\mathsf{ak}, \mathsf{nk}, \mathsf{rivk})$ and a *diversified payment address* $(\mathsf{d}, \mathsf{pk_d})$ for $\mathsf{sk}$ as described in § 4.2.3 *'Orchard Key Components'* on p. 34.
- Let $\mathsf{v} = 0$ and $\mathsf{pos} = 0$.
- Choose uniformly random $\mathsf{rcv} \xleftarrow{\mathrm{R}} \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{GenTrapdoor}()$.
- Choose uniformly random $\mathsf{rseed} \xleftarrow{\mathrm{R}} \mathbb{BY}^{[32]}$.
- Derive $\mathsf{rcm} = \mathsf{ToScalar}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big)$.
- Derive $\psi = \mathsf{ToBase}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([9])\big)$.
- Let $\rho$ be equal to $\mathsf{nf}^{\mathsf{old}}$ from the same *Action description*.
- Let $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v})$.
- Let $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}}\big(\mathsf{repr}_{\mathbb{P}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{P}}(\mathsf{pk_d}), \mathsf{v}, \rho, \psi\big)$.
- Let $\mathsf{nf} = \mathsf{DeriveNullifier}_{\mathsf{nk}}(\rho, \psi, \mathsf{cm})$.
- Construct a *dummy Merkle path* $\mathsf{path}$ for use in the *auxiliary input* to the *Spend statement* (this will not be checked, because $\mathsf{v} = 0$).

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

## 4.9 Merkle Path Validity

Let $\mathsf{MerkleDepth}$ be $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ for the **Sprout** *note commitment tree*, or $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ for the **Sapling** *note commitment tree*, or $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$ for the **Orchard** *note commitment tree*. These constants are defined in § 5.3 *'Constants'* on p. 67.

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

The following discussion applies independently to the **Sprout** and **Sapling** and **Orchard** *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* $\mathsf{MerkleDepth}$ are called *leaf nodes*. When a *note commitment* is added to the tree, it occupies the *leaf node hash value* $\mathsf{M}^{\mathsf{MerkleDepth}}_i$ for the next available $i$.

As-yet unused *leaf nodes* are associated with a distinguished *hash value* $\mathsf{Uncommitted}^{\mathsf{Sprout}}$ or $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ or $\mathsf{Uncommitted}^{\mathsf{Orchard}}$. It is assumed to be infeasible to find a preimage *note* $\mathbf{n}$ such that $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}) = \mathsf{Uncommitted}^{\mathsf{Sprout}}$. (No similar assumption is needed for **Sapling** or **Orchard** because we use a representation for $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ that cannot occur as an output of $\mathsf{NoteCommitment}^{\mathsf{Sapling}}$, and similarly for **Orchard**.)

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

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

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

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

where

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

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

## 4.10 SIGHASH Transaction Hashing

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

In the case of **Zcash**, the BCTV14 and Groth16 and Halo 2 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 § 4.11 *'Non-malleability (**Sprout**)'* on p. 46, § 4.13 *'Balance and Binding Signature (**Sapling**)'* on p. 47, and § 4.15 *'Spend Authorization Signature (**Sapling** and **Orchard**)'* on p. 52.

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*, *spend authorization signatures*, *Sapling binding signatures*, and *Orchard binding 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 § 7.1 *'Transaction Encoding and Consensus'* on p. 114. The hash ***does not*** cover the field `joinSplitSig`. After **Overwinter** activation, all *SIGHASH types* are also extended to cover *transaction* fields introduced in that upgrade, and similarly after **Sapling** activation and after **NU5** activation.

The original *SIGHASH algorithm* defined by **Bitcoin** suffered from some deficiencies as described in [ZIP-143]; in **Zcash** these were addressed by changing this algorithm as part of the **Overwinter** upgrade.

**Orchard** and the **NU5** *network upgrade* introduce *transaction* version 5, which **MUST** be used if any *Action transfers* are present. This version also provides nonmalleable *transaction* identifiers, and **MAY** be used for that reason whether or not *Action transfers* are present.

[Pre-**Overwinter**] 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).

[**Overwinter** only, pre-**Sapling**] The *SIGHASH algorithm* used after **Overwinter** activation and before **Sapling** activation, i.e. for version 3 *transactions*, is defined in [ZIP-143].

[**Sapling** onward] The *SIGHASH algorithm* used after **Sapling** activation, i.e. for version 4 *transactions*, is defined in [ZIP-243].

[**Blossom** onward] The *SIGHASH algorithm* used after **Blossom** activation is the same as for **Sapling**, but using the **Blossom** *consensus branch ID* `0x2BB40E60` as defined in [ZIP-206].

[**Heartwood** onward] The *SIGHASH algorithm* used after **Heartwood** activation is the same as for **Sapling**, but using the **Heartwood** *consensus branch ID* `0xF5B9230B` as defined in [ZIP-250].

[**Canopy** onward] The *SIGHASH algorithm* used after **Canopy** activation is the same as for **Sapling**, but using the **Canopy** *consensus branch ID* `0xE9FF75A6` as defined in [ZIP-251].

[**NU5** onward] The *SIGHASH algorithm* used after activation of the **NU5** *network upgrade*, for both version 4 and version 5 *transactions*, is defined in [ZIP-244] as modified by [ZIP-225]. It will use a new *consensus branch ID* `0xF919A198` as defined in [ZIP-252].

## 4.11 Non-malleability (Sprout)

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

$\mathsf{JoinSplitSig}$ is instantiated in § 5.4.6 'Ed25519' on p. 83.

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 $\mathsf{h_{Sig}}$ be computed as specified in § 4.3 *'JoinSplit Descriptions'* on p. 36.

Let $\mathsf{PRF^{pk}}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 21.

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

The correctness of $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ is enforced by the *JoinSplit statement* given in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54. This ensures that a holder of all of the $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$ for every *JoinSplit description* in the *transaction* has authorized the use of the private *signing key* corresponding to `joinSplitPubKey` to sign this *transaction*.

## 4.12 Balance (Sprout)

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

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

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

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

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

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

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

As stated in § 4.3 ***'JoinSplit Descriptions'*** on p. 36, either $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ or $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ **MUST** be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

## 4.13 Balance and Binding Signature (Sapling)

**Sapling** adds *Spend transfers* and *Output transfers* to the transparent and *JoinSplit transfers* present in **Sprout**. The net value of *Spend transfers* minus *Output transfers* in a *transaction* is called the *Sapling balancing value*, measured in *zatoshi* as a signed integer $\mathsf{v}^{\mathsf{balanceSapling}}$.

$\mathsf{v}^{\mathsf{balanceSapling}}$ is encoded in a *transaction* as the field `valueBalanceSapling`. For a v4 *transaction*, $\mathsf{v}^{\mathsf{balanceSapling}}$ is always explicitly encoded. For a v5 *transaction*, $\mathsf{v}^{\mathsf{balanceSapling}}$ is implicitly zero if the *transaction* has no *Spend descriptions* or *Output descriptions*. Transaction fields are described in § 7.1 ***'Transaction Encoding and Consensus'*** on p. 114.

A positive *Sapling balancing value* takes value from the ***Sapling*** *transaction value pool* and adds it to the *transparent transaction value pool*. A negative *Sapling balancing value* does the reverse. As a result, positive $\mathsf{v}^{\mathsf{balanceSapling}}$ is treated like an ***input*** to the *transparent transaction value pool*, whereas negative $\mathsf{v}^{\mathsf{balanceSapling}}$ is treated like an ***output*** from that pool.

As defined in [ZIP-209], the ***Sapling*** *chain value pool balance* for a given *block chain* is the negation of the sum of all `valueBalanceSapling` field values for *transactions* in the *block chain*.

**Consensus rule:** If the ***Sapling*** *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.

Consistency of $\mathsf{v}^{\mathsf{balanceSapling}}$ with the *value commitments* in *Spend descriptions* and *Output descriptions* is enforced by the *Sapling binding signature*. This signature has a dual rôle in the **Sapling** protocol:

- To prove that the total value spent by *Spend transfers*, minus that produced by *Output transfers*, is consistent with the $\mathsf{v}^{\mathsf{balanceSapling}}$ field of the *transaction*;
- To prove that the signer knew the randomness used for the Spend and Output *value commitments*, in order to prevent *Output descriptions* from being replayed by an adversary in a different *transaction*. (A *Spend description* already cannot be replayed due to its *spend authorization signature*.)

Instead of generating a key pair at random, we generate it as a function of the *value commitments* in the *Spend descriptions* and *Output descriptions* of the *transaction*, and the *Sapling balancing value*.

Let $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $r_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

§ 5.4.8.3 ***'Homomorphic Pedersen commitments (Sapling and Orchard)'*** on p. 89 instantiates:

$\mathsf{ValueCommit}^{\mathsf{Sapling}} ⦂ \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} \times \left\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\right\} \to \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$;
$\mathcal{V}^{\mathsf{Sapling}} ⦂ \mathbb{J}^{(r)*}$, the value base in $\mathsf{ValueCommit}^{\mathsf{Sapling}}$;
$\mathcal{R}^{\mathsf{Sapling}} ⦂ \mathbb{J}^{(r)*}$, the randomness base in $\mathsf{ValueCommit}^{\mathsf{Sapling}}$.

$\mathsf{BindingSig}^{\mathsf{Sapling}}$, $\oplus$, and $\boxplus$ are instantiated in § 5.4.7.2 ***'Binding Signature (Sapling and Orchard)'*** on p. 88.

§ 4.1.7.2 ***'Signature with Signing Key to Validating Key Monomorphism'*** on p. 26 specifies these operations and the derived notation $\ominus$, $\bigoplus_{i=1}^{\mathrm{N}}$, $\boxminus$, and $\boxplus_{i=1}^{\mathrm{N}}$, which in this section are to be interpreted as operating on the prime-order subgroup of the Jubjub curve and its scalar field.

Suppose that the *transaction* has:

- $n$ *Spend descriptions* with *value commitments* $\mathsf{cv}^{\mathsf{old}}_{1..n}$, committing to values $\mathsf{v}^{\mathsf{old}}_{1..n}$ with randomness $\mathsf{rcv}^{\mathsf{old}}_{1..n}$;
- $m$ *Output descriptions* with *value commitments* $\mathsf{cv}^{\mathsf{new}}_{1..m}$, committing to values $\mathsf{v}^{\mathsf{new}}_{1..m}$ with randomness $\mathsf{rcv}^{\mathsf{new}}_{1..m}$;
- *Sapling balancing value* $\mathsf{v}^{\mathsf{balanceSapling}}$.

In a correctly constructed *transaction*, $\mathsf{v}^{\mathsf{balanceSapling}} = \sum_{i=1}^{n} \mathsf{v}^{\mathsf{old}}_i - \sum_{j=1}^{m} \mathsf{v}^{\mathsf{new}}_j$, but validators cannot check this directly because the values are hidden by the commitments.

Instead, validators calculate the *transaction binding validating key* as:

$$\mathsf{bvk}^{\mathsf{Sapling}} := \left(\bigoplus_{i=1}^{n} \mathsf{cv}^{\mathsf{old}}_i\right) \ominus \left(\bigoplus_{j=1}^{m} \mathsf{cv}^{\mathsf{new}}_j\right) \ominus \mathsf{ValueCommit}^{\mathsf{Sapling}}_0\big(\mathsf{v}^{\mathsf{balanceSapling}}\big).$$

(This key is not encoded explicitly in the *transaction* and must be recalculated.)

The signer knows $\mathsf{rcv}^{\mathsf{old}}_{1..n}$ and $\mathsf{rcv}^{\mathsf{new}}_{1..m}$, and so can calculate the corresponding *signing key* as:

$$\mathsf{bsk}^{\mathsf{Sapling}} := \left(\boxplus_{i=1}^{n} \mathsf{rcv}^{\mathsf{old}}_i\right) \boxminus \left(\boxplus_{j=1}^{m} \mathsf{rcv}^{\mathsf{new}}_j\right).$$

In order to check for implementation faults, the signer **SHOULD** also check that

$$\mathsf{bvk}^{\mathsf{Sapling}} = \mathsf{BindingSig}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{bsk}^{\mathsf{Sapling}}).$$

Let $\mathsf{SigHash}$ be the *SIGHASH transaction hash* as defined in [ZIP-243] for a version 4 *transaction* or [ZIP-244] as modified by [ZIP-225] for a version 5 *transaction*, not associated with an input, using the *SIGHASH type* SIGHASH_ALL.

A validator checks balance by validating that $\mathsf{BindingSig}^{\mathsf{Sapling}}.\mathsf{Validate}_{\mathsf{bvk}^{\mathsf{Sapling}}}(\mathsf{SigHash}, \mathtt{bindingSigSapling}) = 1$.

We now explain why this works.

A *Sapling binding signature* proves knowledge of the discrete logarithm $\mathsf{bsk}^{\mathsf{Sapling}}$ of $\mathsf{bvk}^{\mathsf{Sapling}}$ with respect to $\mathcal{R}^{\mathsf{Sapling}}$. That is, $\mathsf{bvk}^{\mathsf{Sapling}} = [\mathsf{bsk}^{\mathsf{Sapling}}]\, \mathcal{R}^{\mathsf{Sapling}}$. So the value $0$ and randomness $\mathsf{bsk}^{\mathsf{Sapling}}$ is an opening of the *Pedersen commitment* $\mathsf{bvk}^{\mathsf{Sapling}} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{bsk}^{\mathsf{Sapling}}}(0)$. By the binding property of the *Pedersen commitment*, it is infeasible to find another opening of this commitment to a different value.

Similarly, the binding property of the *value commitments* in the *Spend descriptions* and *Output descriptions* ensures that an adversary cannot find an opening to more than one value for any of those commitments, i.e. we may assume that $\mathsf{v}^{\mathsf{old}}_{1..n}$ are determined by $\mathsf{cv}^{\mathsf{old}}_{1..n}$, and that $\mathsf{v}^{\mathsf{new}}_{1..m}$ are determined by $\mathsf{cv}^{\mathsf{new}}_{1..m}$. We may also assume, from Knowledge Soundness of Groth16, that the Spend proofs could not have been generated without knowing $\mathsf{rcv}^{\mathsf{old}}_{1..n} \pmod{r_{\mathbb{J}}}$, and the Output proofs could not have been generated without knowing $\mathsf{rcv}^{\mathsf{new}}_{1..m} \pmod{r_{\mathbb{J}}}$.

Using the fact that $\mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v}) = [\mathsf{v}]\, \mathcal{V}^{\mathsf{Sapling}} \oplus [\mathsf{rcv}]\, \mathcal{R}^{\mathsf{Sapling}}$, the expression for $\mathsf{bvk}^{\mathsf{Sapling}}$ above is equivalent to:

$$\begin{aligned}\mathsf{bvk}^{\mathsf{Sapling}} &= \left[\left(\mathop{\boxplus}_{i=1}^{n} \mathsf{v}^{\mathsf{old}}_i\right) \boxminus \left(\mathop{\boxplus}_{j=1}^{m} \mathsf{v}^{\mathsf{new}}_j\right) \boxminus \mathsf{v}^{\mathsf{balanceSapling}}\right] \mathcal{V}^{\mathsf{Sapling}} \oplus \left[\left(\mathop{\boxplus}_{i=1}^{n} \mathsf{rcv}^{\mathsf{old}}_i\right) \boxminus \left(\mathop{\boxplus}_{j=1}^{m} \mathsf{rcv}^{\mathsf{new}}_j\right)\right] \mathcal{R}^{\mathsf{Sapling}} \\ &= \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{bsk}^{\mathsf{Sapling}}}\left(\sum_{i=1}^{n} \mathsf{v}^{\mathsf{old}}_i - \sum_{j=1}^{m} \mathsf{v}^{\mathsf{new}}_j - \mathsf{v}^{\mathsf{balanceSapling}}\right).\end{aligned}$$

Let $\mathsf{v}^* = \sum_{i=1}^{n} \mathsf{v}^{\mathsf{old}}_i - \sum_{j=1}^{m} \mathsf{v}^{\mathsf{new}}_j - \mathsf{v}^{\mathsf{balanceSapling}}$.

Suppose that $\mathsf{v}^* = \mathsf{v}^{\mathsf{bad}} \neq 0 \pmod{r_{\mathbb{J}}}$. Then $\mathsf{bvk}^{\mathsf{Sapling}} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{bsk}^{\mathsf{Sapling}}}(\mathsf{v}^{\mathsf{bad}})$. If the adversary were able to find the discrete logarithm of this $\mathsf{bvk}^{\mathsf{Sapling}}$ with respect to $\mathcal{R}^{\mathsf{Sapling}}$, say $\mathsf{bsk}'$ (as needed to create a valid *Sapling binding signature*), then $(\mathsf{v}^{\mathsf{bad}}, \mathsf{bsk}^{\mathsf{Sapling}})$ and $(0, \mathsf{bsk}')$ would be distinct openings of $\mathsf{bvk}^{\mathsf{Sapling}}$ to different values, breaking the binding property of the *value commitment scheme*.

The above argument shows only that $\mathsf{v}^* = 0 \pmod{r_{\mathbb{J}}}$; in order to show that $\mathsf{v}^* = 0$, we will also demonstrate that it does not overflow $\left\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\right\}$.

The *Spend statements* (§ 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55) prove that all of $\mathsf{v}^{\mathsf{old}}_{1..n}$ are in $\{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\}$. Similarly the *Output statements* (§ 4.17.3 ***'Output Statement (Sapling)'*** on p. 56) prove that all of $\mathsf{v}^{\mathsf{new}}_{1..m}$ are in $\{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\}$. $\mathsf{v}^{\mathsf{balanceSapling}}$ is encoded in the *transaction* as a signed two's complement 64-bit integer in the range $\{-2^{63} \,..\, 2^{63}-1\}$. $\ell_{\mathsf{value}}$ is defined as 64, so $\mathsf{v}^*$ is in the range $\{-m \cdot (2^{64}-1) - 2^{63} + 1 \,..\, n \cdot (2^{64}-1) + 2^{63}\}$. The maximum *transaction* size is 2 MB, and the minimum contributions of a *Spend description* and an *Output description* to *transaction* size are (in a v5 *transaction*) 352 bytes and 948 bytes respectively, limiting $n$ to at most $\mathsf{floor}\left(\frac{2000000}{352}\right) = 5681$ and $m$ to at most $\mathsf{floor}\left(\frac{2000000}{948}\right) = 2109$.

This ensures that $\mathsf{v}^* \in \{-38913406623490299131842 \,..\, 104805176454780817500623\}$, a subrange of $\left\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\right\}$.

Thus checking the *Sapling binding signature* ensures that the *Spend transfers* and *Output transfers* in the *transaction* balance, without their individual values being revealed.

In addition this proves that the signer, knowing the $\boxplus$-sum of the **Sapling** *value commitment* randomnesses, authorized a *transaction* with the given *SIGHASH transaction hash* by signing $\mathsf{SigHash}$.

**Note:** The spender **MAY** reveal any strict subset of the **Sapling** *value commitment* randomnesses to other parties that are cooperating to create the *transaction*. If all of the *value commitment* randomnesses are revealed, that could allow replaying the *Output descriptions* of the *transaction*.

**Non-normative note:** The technique of checking signatures using a *validating key* derived from a sum of *Pedersen commitments* is also used in the **Mimblewimble** protocol [Jedusor2016]. The *private key* $\mathsf{bsk}^{\mathsf{Sapling}}$ acts as a "*synthetic blinding factor*", in the sense that it is synthesized from the other blinding factors (*trapdoors*) $\mathsf{rcv}^{\mathsf{old}}_{1..n}$ and $\mathsf{rcv}^{\mathsf{new}}_{1..m}$; this technique is also used in **Bulletproofs** [Dalek-notes].

## 4.14 Balance and Binding Signature (Orchard)

**Orchard** introduces *Action transfers*, each of which can optionally perform a spend, and optionally perform an output. Similarly to **Sapling**, the net value of **Orchard** spends minus outputs in a *transaction* is called the *Orchard balancing value*, measured in *zatoshi* as a signed integer $\mathsf{v}^{\mathsf{balanceOrchard}}$.

$\mathsf{v}^{\mathsf{balanceOrchard}}$ is encoded in a *transaction* as the field `valueBalanceOrchard`. If a *transaction* has no *Action descriptions*, $\mathsf{v}^{\mathsf{balanceOrchard}}$ is implicitly zero. Transaction fields are described in § 7.1 *'Transaction Encoding and Consensus'* on p. 114.

A positive *Orchard balancing value* takes value from the ***Orchard*** *transaction value pool* and adds it to the *transparent transaction value pool*. A negative *Orchard balancing value* does the reverse. As a result, positive $\mathsf{v}^{\mathsf{balanceOrchard}}$ is treated like an ***input*** to the *transparent transaction value pool*, whereas negative $\mathsf{v}^{\mathsf{balanceOrchard}}$ is treated like an ***output*** from that pool.

Similarly to the ***Sapling*** *chain value pool balance* defined in [ZIP-209], the ***Orchard*** *chain value pool balance* for a given *block chain* is the negation of the sum of all `valueBalanceOrchard` field values for *transactions* in the *block chain*.

**Consensus rule:** If the ***Orchard*** *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.

Consistency of $\mathsf{v}^{\mathsf{balanceOrchard}}$ with the *value commitments* in *Action descriptions* is enforced by the *Orchard binding signature*. The rôle of this signature in the **Orchard** protocol is to prove that the net value spent (i.e. the total value spent minus the total value produced) by *Action transfers* is consistent with the $\mathsf{v}^{\mathsf{balanceOrchard}}$ field of the *transaction*.

**Non-normative note:** The other rôle of *Sapling binding signatures*, to prove that the signer knew the randomness used for commitments in order to prevent them from being replayed, is less important in **Orchard** because all *Action descriptions* have a *spend authorization signature*. Still, an *Orchard binding signature* does prove that the signer knew this commitment randomness; this provides defence in depth and reduces the differences of **Orchard** from **Sapling**, which may simplify security analysis.

Instead of generating a key pair at random, we generate it as a function of the *value commitments* in the *Action descriptions* of the *transaction*, and the *Orchard balancing value*.

Let $\mathbb{P}$, $\mathbb{P}^*$, and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

§ 5.4.8.3 *'Homomorphic Pedersen commitments (**Sapling** and **Orchard**)'* on p. 89 instantiates:

$\mathsf{ValueCommit}^{\mathsf{Orchard}} ⦂ \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor} \times \left\{-\frac{r_{\mathbb{P}}-1}{2} \,..\, \frac{r_{\mathbb{P}}-1}{2}\right\} \to \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output}$;

$\mathcal{V}^{\mathsf{Orchard}} ⦂ \mathbb{P}^*$, the value base in $\mathsf{ValueCommit}^{\mathsf{Orchard}}$;

$\mathcal{R}^{\mathsf{Orchard}} ⦂ \mathbb{P}^*$, the randomness base in $\mathsf{ValueCommit}^{\mathsf{Orchard}}$.

$\mathsf{BindingSig}^{\mathsf{Orchard}}$, $\diamond\!\!\!+$, and $\boxplus$ are instantiated in § 5.4.7.2 *'Binding Signature (**Sapling** and **Orchard**)'* on p. 88.

§ 4.1.7.2 *'Signature with Signing Key to Validating Key Monomorphism'* on p. 26 specifies these operations and the derived notation $\diamond\!\!\!-$, $\bigoplus_{i=1}^{\mathrm{N}}$, $\boxminus$, and $\bigboxplus_{i=1}^{\mathrm{N}}$, which in this section are to be interpreted as operating on the Pallas curve and its scalar field.

Suppose that the *transaction* has:

- $n$ *Action descriptions* with *value commitments* $\mathsf{cv}^{\mathsf{net}}1..n$, committing to values $\mathsf{v}^{\mathsf{net}}_{1..n}$ with randomness $\mathsf{rcv}^{\mathsf{net}}_{1..n}$;
- *Orchard balancing value* $\mathsf{v}^{\mathsf{balanceOrchard}}$.

In a correctly constructed *transaction*, $\mathsf{v}^{\mathsf{balanceOrchard}} = \sum_{i=1}^{n} \mathsf{v}^{\mathsf{net}}_i$, but validators cannot check this directly because the values are hidden by the commitments.

Instead, validators calculate the *transaction binding validating key* as:

$$\mathsf{bvk}^{\mathsf{Orchard}} := \left( \bigoplus_{i=1}^{n} \mathsf{cv}^{\mathsf{net}} i \right) \ominus \mathsf{ValueCommit}_0^{\mathsf{Orchard}}\big(\mathsf{v}^{\mathsf{balanceOrchard}}\big).$$

(This key is not encoded explicitly in the *transaction* and must be recalculated.)

The signer knows $\mathsf{rcv}^{\mathsf{net}}_{1..n}$, and so can calculate the corresponding *signing key* as:

$$\mathsf{bsk}^{\mathsf{Orchard}} := \boxplus_{i=1}^{n} \mathsf{rcv}^{\mathsf{net}}_i.$$

In order to check for implementation faults, the signer **SHOULD** also check that

$$\mathsf{bvk}^{\mathsf{Orchard}} = \mathsf{BindingSig}^{\mathsf{Orchard}}.\mathsf{DerivePublic}(\mathsf{bsk}^{\mathsf{Orchard}}).$$

A *transaction* containing *Action descriptions* is necessarily a version 5 *transaction*. Let SigHash be the *SIGHASH transaction hash* for a version 5 *transaction* as defined in [ZIP-244] as modified by [ZIP-225], not associated with an input, using the *SIGHASH type* SIGHASH_ALL.

A validator checks balance by validating that $\mathsf{BindingSig}^{\mathsf{Orchard}}.\mathsf{Validate}_{\mathsf{bvk}^{\mathsf{Orchard}}}(\mathsf{SigHash}, \mathtt{bindingSigOrchard}) = 1$.

The security argument is very similar to that for *Sapling binding signatures*, but for completeness we spell it out, since there are minor differences due to the net value commitments, and a different bound on the net value sum $\mathsf{v}^*$.

An *Orchard binding signature* proves knowledge of the discrete logarithm $\mathsf{bsk}^{\mathsf{Orchard}}$ of $\mathsf{bvk}^{\mathsf{Orchard}}$ with respect to $\mathcal{R}^{\mathsf{Orchard}}$. That is, $\mathsf{bvk}^{\mathsf{Orchard}} = [\mathsf{bsk}^{\mathsf{Orchard}}]\,\mathcal{R}^{\mathsf{Orchard}}$. So the value $0$ and randomness $\mathsf{bsk}^{\mathsf{Orchard}}$ is an opening of the *Pedersen commitment* $\mathsf{bvk}^{\mathsf{Orchard}} = \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{bsk}^{\mathsf{Orchard}}}(0)$. By the binding property of the *Pedersen commitment*, it is infeasible to find another opening of this commitment to a different value.

Similarly, the binding property of the *value commitments* in the *Action descriptions* ensures that an adversary cannot find an opening to more than one value for any of those commitments, i.e. we may assume that $\mathsf{v}^{\mathsf{net}}_{1..n}$ are determined by $\mathsf{cv}^{\mathsf{net}} 1..n$. We may also assume, from Knowledge Soundness of Halo 2, that the Action proofs could not have been generated without knowing $\mathsf{rcv}^{\mathsf{net}}_{1..n} \pmod{r_{\mathbb{P}}}$.

Using the fact that $\mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v}) = [\mathsf{v}]\,\mathcal{V}^{\mathsf{Orchard}} \oplus [\mathsf{rcv}]\,\mathcal{R}^{\mathsf{Orchard}}$, the expression for $\mathsf{bvk}^{\mathsf{Orchard}}$ above is equivalent to:

$$\begin{aligned}\mathsf{bvk}^{\mathsf{Orchard}} &= \left[ \left( \boxplus_{i=1}^{n} \mathsf{v}^{\mathsf{net}}_i \right) \boxminus \mathsf{v}^{\mathsf{balanceOrchard}} \right] \mathcal{V}^{\mathsf{Orchard}} \oplus \left[ \left( \boxplus_{i=1}^{n} \mathsf{rcv}^{\mathsf{net}}_i \right) \right] \mathcal{R}^{\mathsf{Orchard}} \\ &= \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{bsk}^{\mathsf{Orchard}}} \left( \sum_{i=1}^{n} \mathsf{v}^{\mathsf{net}}_i - \mathsf{v}^{\mathsf{balanceOrchard}} \right).\end{aligned}$$

Let $\mathsf{v}^* = \sum_{i=1}^{n} \mathsf{v}^{\mathsf{net}}_i - \mathsf{v}^{\mathsf{balanceOrchard}}$.

Suppose that $\mathsf{v}^* = \mathsf{v}^{\mathsf{bad}} \neq 0 \pmod{r_{\mathbb{J}}}$. Then $\mathsf{bvk}^{\mathsf{Orchard}} = \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{bsk}^{\mathsf{Orchard}}}(\mathsf{v}^{\mathsf{bad}})$. If the adversary were able to find the discrete logarithm of this $\mathsf{bvk}^{\mathsf{Orchard}}$ with respect to $\mathcal{R}^{\mathsf{Orchard}}$, say $\mathsf{bsk}'$ (as needed to create a valid *Orchard binding signature*), then $(\mathsf{v}^{\mathsf{bad}}, \mathsf{bsk}^{\mathsf{Orchard}})$ and $(0, \mathsf{bsk}')$ would be distinct openings of $\mathsf{bvk}^{\mathsf{Orchard}}$ to different values, breaking the binding property of the *value commitment scheme*.

The above argument shows only that $\mathsf{v}^* = 0 \pmod{r_{\mathbb{P}}}$; in order to show that $\mathsf{v}^* = 0$, we will also demonstrate that it does not overflow $\left\{-\frac{r_{\mathbb{P}}-1}{2} \,..\, \frac{r_{\mathbb{P}}-1}{2}\right\}$.

The *Action statements* (§ 4.17.4 ***'Action Statement (Orchard)'*** on p. 57) prove that all $\mathsf{v}^{\mathsf{net}}_{1..n}$ are in $\{-2^{64}+1 \,..\, 2^{64}-1\}$. $\mathsf{v}^{\mathsf{balanceOrchard}}$ is encoded in the *transaction* as a signed two's complement 64-bit integer in the range $\{-2^{63} \,..\, 2^{63}-1\}$. Therefore, $\mathsf{v}^*$ is is in the range $\{-n \cdot (2^{64}-1) - 2^{63} + 1 \,..\, n \cdot (2^{64}-1) + 2^{63}\}$. $n$ is limited by consensus rule to at most $2^{16}-1$ (this rule is technically redundant due to the 2 MB *transaction* size limit, but it suffices here).

This ensures that $\mathsf{v}^* \in \{-1208916596242592319864832 \,..\, 1208916596242592319864833\}$, a subrange of $\left\{-\frac{r_{\mathbb{P}}-1}{2} \,..\, \frac{r_{\mathbb{P}}-1}{2}\right\}$.

Thus checking the *Orchard binding signature* ensures that the *Action transfers* in the *transaction* balance, without their individual net values being revealed.

In addition this proves that the signer, knowing the $\boxplus$-sum of the **Orchard** *value commitment* randomnesses, authorized a *transaction* with the given *SIGHASH transaction hash* by signing $\mathsf{SigHash}$.

**Note:** The spender **MAY** reveal any strict subset of the **Orchard** *value commitment* randomnesses to other parties that are cooperating to create the *transaction*.

## 4.15 Spend Authorization Signature (Sapling and Orchard)

$\mathsf{SpendAuthSig}$ is used in **Sapling** and **Orchard** to prove knowledge of the *spending key* authorizing spending of an input *note*. It is instantiated in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.

We use $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ to refer to the *spend authorization signature scheme* for **Sapling**, which is instantiated on the Jubjub curve. We use $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ to refer to the *spend authorization signature scheme* for **Orchard**, which is instantiated on the Pallas curve. The following discussion applies to both.

Knowledge of the *spending key* could have been proven directly in the *Spend statement* or *Action statement*, similar to the check in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54 that is part of the *JoinSplit statement*. The motivation for a separate signature is to allow devices that are limited in memory and computational capacity, such as hardware wallets, to authorize a **Sapling** shielded Spend. Typically such devices cannot create, and may not be able to verify, *zk-SNARK proofs* for a *statement* of the size needed using the $\mathsf{BCTV14}$ or $\mathsf{Groth16}$ proving systems.

The *validating key* of the signature must be revealed in the *Spend description* so that the signature can be checked by validators. To ensure that the *validating key* cannot be linked to the *shielded payment address* or *spending key* from which the *note* was spent, we use a *signature scheme with re-randomizable keys*. The *Spend statement* proves that this *validating key* is a re-randomization of the *spend authorization address key* $\mathsf{ak}$ with a *randomizer* known to the signer. The *spend authorization signature* is over the *SIGHASH transaction hash*, so that it cannot be replayed in other *transactions*.

Let $\mathsf{SigHash}$ be the *SIGHASH transaction hash* as defined in [ZIP-243], not associated with an input, using the *SIGHASH type* $\mathsf{SIGHASH\_ALL}$.

Let $\mathsf{ask}$ be the *spend authorization private key* as defined in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

Let $\mathsf{SpendAuthSig}$ be $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ or $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ as applicable.

For each *Spend description*, the signer chooses a fresh *spend authorization randomizer* $\alpha$:

1. Choose $\alpha \xleftarrow{\mathsf{R}} \mathsf{SpendAuthSig.GenRandom}()$.
2. Let $\mathsf{rsk} = \mathsf{SpendAuthSig.RandomizePrivate}(\alpha, \mathsf{ask})$.
3. Let $\mathsf{rk} = \mathsf{SpendAuthSig.DerivePublic}(\mathsf{rsk})$.
4. Generate a proof $\pi$ of the *Spend statement* (§ 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55) or *Action statement* (§ 4.17.4 ***'Action Statement (Orchard)'*** on p. 57), with $\alpha$ in the *auxiliary input* and $\mathsf{rk}$ in the *primary input*.
5. Let $\mathsf{spendAuthSig} = \mathsf{SpendAuthSig.Sign}_{\mathsf{rsk}}(\mathsf{SigHash})$.

The resulting $\mathsf{spendAuthSig}$ and $\pi$ are included in the *Spend description*, or in the `vSpendAuthSigsSapling` or `vSpendAuthSigsOrchard` field of a version 5 *transaction*.

**Note:** If the spender is computationally or memory-limited, step 4 (and only step 4) **MAY** be delegated to a different party that is capable of performing the *zk-SNARK proof*. In this case privacy will be lost to that party since it needs $\mathsf{ak}$ and the *proof authorizing key* $\mathsf{nsk}$; this allows also deriving the $\mathsf{nk}$ component of the *full viewing key*. (In **Orchard**, that party needs the $\mathsf{nk}$ directly to make the *zk-SNARK proof*.) Together $\mathsf{ak}$ and $\mathsf{nk}$ are sufficient to recognize spent *notes* and to recognize and decrypt incoming *notes*. However, the other party will not obtain spending authority for other *transactions*, since it is not able to create a *spend authorization signature* by itself.

## 4.16 Note Commitments and Nullifiers

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

All of the constituent *nullifiers* are also entered into the *nullifier set* of the associated *treestate*. A *transaction* is not valid if it would have added a *nullifier* to the *nullifier set* that already exists in the set (see § 3.9 *'Nullifier Sets'* on p. 19).

In **Sprout**, each *note* has a $\rho$ component.

In **Sapling**, each *positioned note* has an associated $\rho$ value which is computed from its *note commitment* $\mathsf{cm}$ and *note position* $\mathsf{pos}$ as follows:

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

$\mathsf{MixingPedersenHash}$ is defined in § 5.4.1.8 *'Mixing Pedersen Hash Function'* on p. 74.

Let $\mathsf{PRF}^{\mathsf{nfSprout}}$ and $\mathsf{PRF}^{\mathsf{nfSapling}}$ and $\mathsf{PRF}^{\mathsf{nfOrchard}}$ be as instantiated in § 5.4.2 *'Pseudo Random Functions'* on p. 79.

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

For a **Sapling** *note*, the *nullifier* is derived as $\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$, where $\mathsf{nk}\star$ is a representation of the *nullifier deriving key* associated with the *note* and $\rho\star = \mathsf{repr}_{\mathbb{J}}(\rho)$.

The derivation of *nullifiers* for **Orchard** *notes* is a little more complicated.

Let $\mathbb{P}$ and $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let $\mathsf{Extract}_{\mathbb{P}}$ be as defined in § 5.4.9.7 *'Coordinate Extractor for* Pallas' on p. 98.

Let $\mathsf{GroupHash}^{\mathbb{P}}$ be as defined in § 5.4.9.8 *'Group Hash into* Pallas *and* Vesta' on p. 98.

Define $\mathcal{K}^{\mathsf{Orchard}} := \mathsf{GroupHash}^{\mathbb{P}}(\texttt{"z.cash:Orchard"}, \texttt{"K"})$.

To avoid repetition, we define a function $\mathsf{DeriveNullifier} : \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{P} \to \mathbb{F}_{q_{\mathbb{P}}}$ as follows:

$$\mathsf{DeriveNullifier}_{\mathsf{nk}}(\rho, \psi, \mathsf{cm}) = \mathsf{Extract}_{\mathbb{P}}\big(\big[(\mathsf{PRF}^{\mathsf{nfOrchard}}_{\mathsf{nk}}(\rho) + \psi) \bmod q_{\mathbb{P}}\big]\, \mathcal{K}^{\mathsf{Orchard}} + \mathsf{cm}\big).$$

where $\mathsf{nk}$ is the *nullifier deriving key* associated with the *note*; $\rho$ and $\psi$ are part of the *note*; and $\mathsf{cm}$ is the *note commitment*.

**Note:** The addition of $\mathsf{PRF}^{\mathsf{nfOrchard}}_{\mathsf{nk}}(\rho)$ and $\psi$ is intentionally done modulo $q_{\mathbb{P}}$, even though the scalar multiplication is on the Pallas curve which has scalar field $\mathbb{F}_{r_{\mathbb{P}}}$.

**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*, and possibly its position, in a way that can be checked in the corresponding statement that controls spends (i.e. the *JoinSplit statement*, *Spend statement*, or *Action statement*).
- Under the assumption that $\rho$ values are unique, it must not be possible to generate two *notes* with distinct *note commitments* but the same *nullifier*. (See § 8.4 *'Faerie Gold attack and fix'* on p. 133 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*.

## 4.17 Zk-SNARK Statements

### 4.17.1 JoinSplit Statement (Sprout)

Let $\ell^{\mathsf{Sprout}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$, $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell_{\mathsf{value}}$, $\ell_{\mathsf{a_{sk}}}$, $\ell^{\mathsf{Sprout}}_{\varphi}$, $\ell_{\mathsf{hSig}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and $\mathsf{PRF}^{\rho}$ be as defined in § 4.1.2 *'Pseudo Random Functions'* on p. 21.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in § 4.1.8 *'Commitment'* on p. 27, and let $\mathsf{Note}^{\mathsf{Sprout}}$ and $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$ be as defined in § 3.2 *'Notes'* on p. 13.

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

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

the prover knows an *auxiliary input*:

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

where:

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

such that the following conditions hold:

**Merkle path validity** for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforceMerklePath}_i = 1$: $(\mathsf{path}_i, \mathsf{pos}_i)$ is a valid *Merkle path* (see § 4.9 *'Merkle Path Validity'* on p. 45) of depth $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ from $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}^{\mathsf{old}}_i)$ to the *anchor* $\mathsf{rt}^{\mathsf{Sprout}}$.

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

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

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

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

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

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

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

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

For details of the form and encoding of proofs, see § 5.4.10.1 'BCTV14' on p. 102.

### 4.17.2 Spend Statement (Sapling)

Let $\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}$, $\ell_{\mathsf{PRFnfSapling}}$, $\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}$, and $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in § 4.1.8 *'Commitment'* on p. 27.

Let $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ be as defined in § 5.4.7.1 *'Spend Authorization Signature (**Sapling** and **Orchard**)'* on p. 87.

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathsf{repr}_{\mathbb{J}}$, $q_{\mathbb{J}}$, $r_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{Extract}_{\mathbb{J}^{(r)}} ⦂ \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}$ be as defined in § 5.4.9.4 *'Coordinate Extractor for* Jubjub' on p. 96.

Let $\mathcal{H}^{\mathsf{Sapling}}$ be as defined in § 4.2.2 *'**Sapling** Key Components'* on p. 32.

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

$\big(\mathsf{rt}^{\mathsf{Sapling}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}$,
$\mathsf{cv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$,
$\mathsf{nf}^{\mathsf{old}} ⦂ \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{PRFnfSapling}}/8]}}$,
$\mathsf{rk} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}\big)$,

the prover knows an *auxiliary input*:

$\big(\mathsf{path} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sapling}}]}$,
$\mathsf{pos} ⦂ \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}-1\}$,
$\mathsf{g_d} ⦂ \mathbb{J}$,
$\mathsf{pk_d} ⦂ \mathbb{J}$,
$\mathsf{v}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}$,
$\mathsf{rcv}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$,
$\mathsf{cm}^{\mathsf{old}} ⦂ \mathbb{J}$,
$\mathsf{rcm}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$,
$\alpha ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$,
$\mathsf{ak} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$,
$\mathsf{nsk} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}\big)$

such that the following conditions hold:

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

**Merkle path validity** Either $\mathsf{v}^{\mathsf{old}} = 0$; or $(\mathsf{path}, \mathsf{pos})$ is a valid *Merkle path* of depth $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, as defined in § 4.9 *'Merkle Path Validity'* on p. 45, from $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm}^{\mathsf{old}})$ to the *anchor* $\mathsf{rt}^{\mathsf{Sapling}}$.

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

**Small order checks** $\mathsf{g_d}$ and $\mathsf{ak}$ are not of small order, i.e. $[h_{\mathbb{J}}]\,\mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\,\mathsf{ak} \neq \mathcal{O}_{\mathbb{J}}$.

**Nullifier integrity** $\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$ where
$\mathsf{nk}\star = \mathsf{repr}_{\mathbb{J}}\big([\mathsf{nsk}]\,\mathcal{H}^{\mathsf{Sapling}}\big)$
$\rho\star = \mathsf{repr}_{\mathbb{J}}\big(\mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})\big)$.

**Spend authority** $\mathsf{rk} = \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{RandomizePublic}(\alpha, \mathsf{ak})$.

**Diversified address integrity** $\mathsf{pk_d} = [\mathsf{ivk}]\,\mathsf{g_d}$ where
$\mathsf{ivk} = \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}\star, \mathsf{nk}\star)$
$\mathsf{ak}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{ak})$.

For details of the form and encoding of *Spend statement* proofs, see § 5.4.10.2 'Groth16' on p. 103.

**Notes:**

- *Primary* and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see § A.3.3.2 ***'ctEdwards [de]compression and validation'*** on p. 178, for required validity checks on compressed representations of Jubjub curve points.

  The $\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ and $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$ types also represent points, i.e. $\mathbb{J}$.

- In the Merkle path validity check, each *layer* does ***not*** check that its input bit sequence is a canonical encoding (in $\{0\,..\,q_{\mathbb{J}}-1\}$) of the integer from the previous *layer*.

- It is ***not*** checked in the *Spend statement* that $\mathsf{rk}$ is not of small order. However, this ***is*** checked outside the *Spend statement*, as specified in § 4.4 ***'Spend Descriptions'*** on p. 37.

- It is ***not*** checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.

- $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{RandomizePublic}(\alpha, \mathsf{ak}) = \mathsf{ak} + [\alpha]\, \mathcal{G}^{\mathsf{Sapling}}$.

  ($\mathcal{G}^{\mathsf{Sapling}}$ is as defined in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.)

### 4.17.3 Output Statement (Sapling)

Let $\ell_{\mathsf{Merkle}}^{\mathsf{Sapling}}$ and $\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in § 4.1.8 ***'Commitment'*** on p. 27.

Let $\mathbb{J}$, $\mathsf{repr}_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

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

$$\begin{aligned}\big(&\mathsf{cv}^{\mathsf{new}} ⦂ \mathsf{ValueCommit}'.\mathsf{Output}\\ &\mathsf{cm}_u ⦂ \mathbb{B}^{[\ell_{\mathsf{Merkle}}^{\mathsf{Sapling}}]},\\ &\mathsf{epk} ⦂ \mathbb{J}\big),\end{aligned}$$

the prover knows an *auxiliary input*:

$$\begin{aligned}(&\mathsf{g_d} ⦂ \mathbb{J},\\ &\mathsf{pk{\star}_d} ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]},\\ &\mathsf{v}^{\mathsf{new}} ⦂ \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\ &\mathsf{rcv}^{\mathsf{new}} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}}-1\},\\ &\mathsf{rcm}^{\mathsf{new}} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}}-1\},\\ &\mathsf{esk} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}^{\mathsf{Sapling}}}-1\})\end{aligned}$$

such that the following conditions hold:

**Note commitment integrity** $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}\big(\mathsf{NoteCommit}_{\mathsf{rcm}^{\mathsf{new}}}^{\mathsf{Sapling}}(\mathsf{g{\star}_d}, \mathsf{pk{\star}_d}, \mathsf{v}^{\mathsf{new}})\big)$, where $\mathsf{g{\star}_d} = \mathsf{repr}_{\mathbb{J}}(\mathsf{g_d})$.

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

**Small order check** $\mathsf{g_d}$ is not of small order, i.e. $[h_{\mathbb{J}}]\, \mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$.

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

For details of the form and encoding of *Output statement* proofs, see § 5.4.10.2 'Groth16' on p. 103.

**Notes:**

- *Primary* and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see § A.3.3.2 *'ctEdwards [de]compression and validation'* on p. 178, for required validity checks on compressed representations of Jubjub curve points. The $\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ type also represents points, i.e. $\mathbb{J}$.
- The validity of $\mathsf{pk}\star_{\mathsf{d}}$ is *not* checked in this circuit.
- It is *not* checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.

### 4.17.4 Action Statement (Orchard)

Let $\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}$, $\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}$, and $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$ be as defined in § 5.3 *'Constants'* on p. 67.

Let $\mathsf{ValueCommit}^{\mathsf{Orchard}}$, $\mathsf{NoteCommit}^{\mathsf{Orchard}}$, and $\mathsf{Commit}^{\mathsf{ivk}}$ be as specified in § 4.1.8 *'Commitment'* on p. 27.

Let $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ be as defined in § 5.4.7.1 *'Spend Authorization Signature (**Sapling** and **Orchard**)'* on p. 87.

Let $\mathbb{P}$, $\mathbb{P}^*$, $\mathbb{P}_x$, $\mathsf{repr}_{\mathbb{P}}$, $q_{\mathbb{P}}$, and $r_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let $\mathsf{Extract}_{\mathbb{P}}$ and $\mathsf{Extract}^{\perp}_{\mathbb{P}}$ be as defined in § 5.4.9.7 *'Coordinate Extractor for* Pallas' on p. 98.

Let $\mathsf{DeriveNullifier}$ be as defined in § 4.16 *'Note Commitments and Nullifiers'* on p. 53.

A valid instance of a *Action statement*, $\pi$, assures that given a *primary input*:

$$\begin{aligned}
\big(&\mathsf{rt}^{\mathsf{Orchard}} \mathrel{⦂} \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}]},\\
&\mathsf{cv}^{\mathsf{net}} \mathrel{⦂} \mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output},\\
&\mathsf{nf}^{\mathsf{old}} \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\mathsf{rk} \mathrel{⦂} \mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{Public},\\
&\mathsf{cm}_x \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\mathsf{enableSpend} \mathrel{⦂} \mathbb{B},\\
&\mathsf{enableOutput} \mathrel{⦂} \mathbb{B}\big),
\end{aligned}$$

the prover knows an *auxiliary input*:

$$\begin{aligned}
\big(&\mathsf{path} \mathrel{⦂} \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Orchard}}]},\\
&\mathsf{pos} \mathrel{⦂} \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Orchard}}}-1\},\\
&\mathsf{g}^{\mathsf{old}}_{\mathsf{d}} \mathrel{⦂} \mathbb{P}^*,\\
&\mathsf{pk}^{\mathsf{old}}_{\mathsf{d}} \mathrel{⦂} \mathbb{P},\\
&\mathsf{v}^{\mathsf{old}} \mathrel{⦂} \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\rho^{\mathsf{old}} \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\psi^{\mathsf{old}} \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\mathsf{rcm}^{\mathsf{old}} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}}-1\},\\
&\mathsf{cm}^{\mathsf{old}} \mathrel{⦂} \mathbb{P},\\
&\alpha \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}}-1\},\\
&\mathsf{ak}^{\mathbb{P}} \mathrel{⦂} \mathbb{P},\\
&\mathsf{nk} \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\mathsf{rivk} \mathrel{⦂} \mathsf{Commit}^{\mathsf{ivk}}.\mathsf{Trapdoor},\\
&\mathsf{g}\star^{\mathsf{new}}_{\mathsf{d}} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathbb{P}}]},\\
&\mathsf{pk}\star^{\mathsf{new}}_{\mathsf{d}} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathbb{P}}]},\\
&\mathsf{v}^{\mathsf{new}} \mathrel{⦂} \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\psi^{\mathsf{new}} \mathrel{⦂} \mathbb{F}_{q_{\mathbb{P}}},\\
&\mathsf{rcm}^{\mathsf{new}} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}}-1\},\\
&\mathsf{rcv} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Orchard}}_{\mathsf{scalar}}}-1\}\big)
\end{aligned}$$

such that the following conditions hold:

**Old note commitment integrity** $\mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}^{\mathsf{old}}}(\mathsf{repr}_{\mathbb{P}}(\mathsf{g}^{\mathsf{old}}_{\mathsf{d}}), \mathsf{repr}_{\mathbb{P}}(\mathsf{pk}^{\mathsf{old}}_{\mathsf{d}}), \mathsf{v}^{\mathsf{old}}, \rho^{\mathsf{old}}, \psi^{\mathsf{old}}) \in \{\mathsf{cm}^{\mathsf{old}}, \bot\}$.

**Merkle path validity** Either $\mathsf{v}^{\mathsf{old}} = 0$; or $(\mathsf{path}, \mathsf{pos})$ is a valid *Merkle path* of depth $\mathsf{MerkleDepth}^{\mathsf{Orchard}}$, as defined in § 4.9 *'Merkle Path Validity'* on p. 45, from $\mathsf{cm}^{\mathsf{old}}$ to the *anchor* $\mathsf{rt}^{\mathsf{Orchard}}$.

**Value commitment integrity** $\mathsf{cv}^{\mathsf{net}} = \mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}} - \mathsf{v}^{\mathsf{new}})$.

**Nullifier integrity** $\mathsf{nf}^{\mathsf{old}} = \mathsf{DeriveNullifier}_{\mathsf{nk}}(\rho^{\mathsf{old}}, \psi^{\mathsf{old}}, \mathsf{cm}^{\mathsf{old}})$.

**Spend authority** $\mathsf{rk} = \mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{RandomizePublic}(\alpha, \mathsf{ak}^{\mathbb{P}})$.

**Diversified address integrity** $\mathsf{pk}^{\mathsf{old}}_{\mathsf{d}} = [\mathsf{ivk}]\, \mathsf{g}^{\mathsf{old}}_{\mathsf{d}}$ where $\mathsf{ivk} = \mathsf{Commit}^{\mathsf{ivk}}_{\mathsf{rivk}}(\mathsf{Extract}_{\mathbb{P}}(\mathsf{ak}^{\mathbb{P}}), \mathsf{nk})$.

**New note commitment integrity** $\mathsf{Extract}^{\bot}_{\mathbb{P}}(\mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}^{\mathsf{new}}}(\mathsf{g}\star^{\mathsf{new}}_{\mathsf{d}}, \mathsf{pk}\star^{\mathsf{new}}_{\mathsf{d}}, \mathsf{v}^{\mathsf{new}}, \rho^{\mathsf{new}}, \psi^{\mathsf{new}})) \in \{\mathsf{cm}_x, \bot\}$, where $\rho^{\mathsf{new}} = \mathsf{nf}^{\mathsf{old}}$.

**Enable spend flag** $\mathsf{v}^{\mathsf{old}} = 0$ or $\mathsf{enableSpend} = 1$.

**Enable output flag** $\mathsf{v}^{\mathsf{new}} = 0$ or $\mathsf{enableOutput} = 1$.

For details of the form and encoding of *Action statement* proofs, see § 5.4.10.3 'Halo 2' on p. 103.

**Notes:**

- *Primary* and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, $\mathsf{g}^{\mathsf{old}}_{\mathsf{d}}$ cannot be $\mathcal{O}_{\mathbb{P}}$. The $\mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{Output}$ and $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{Public}$ types represent Pallas curve points, i.e. $\mathbb{P}$.
- The scalar multiplication used in $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ must operate correctly on the range $\{-2^{64}+1 \,..\, 2^{64}-1\}$, which is different to the range $\{-2^{63} \,..\, 2^{63}-1\}$ of $\mathsf{v}^{\mathsf{balanceOrchard}}$.
- In the Merkle path validity check, each *layer* does *not* check that its input bit sequence is a canonical encoding (in $\{0 \,..\, q_{\mathbb{P}}-1\}$) of the integer from the previous *layer*.
- It is *not* checked that $\mathsf{rcv} < r_{\mathbb{P}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{P}}$ or that $\mathsf{rcm}^{\mathsf{new}} < r_{\mathbb{P}}$.
- $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}.\mathsf{RandomizePublic}(\alpha, \mathsf{ak}^{\mathbb{P}}) = \mathsf{ak}^{\mathbb{P}} + [\alpha]\, \mathcal{G}^{\mathsf{Orchard}}$.

  ($\mathcal{G}^{\mathsf{Orchard}}$ is as defined in § 5.4.7.1 *'Spend Authorization Signature (**Sapling** and **Orchard**)'* on p. 87.)
- The validity of $\mathsf{g}\star_{\mathsf{d}}$ and $\mathsf{pk}\star_{\mathsf{d}}$ are *not* checked in this circuit. Also, $\mathsf{nf}^{\mathsf{old}}$ and $\mathsf{cm}_x$ are *not* checked to be in $\mathbb{P}_x$.

**Non-normative notes:**

- The procedure in § 4.2.3 *'**Orchard** Key Components'* on p. 34 will always produce a *spend authorization address key* that effectively has the compressed $y$-coordinate, $\tilde{y}$, set to 0. The *Action statement*, on the other hand, allows the prover to witness $\mathsf{ak}^{\mathbb{P}}$ with $\tilde{y}$ set to 0 or 1. This is harmless because if the prover and signer(s) of the *spend authorization signature* collectively know $\mathsf{rsk}$ and $\alpha$, we can conclude that they collectively know $\mathsf{ask}$ up to sign, which is sufficient for spend authorization.
- There is intentionally no equivalent to the **Ephemeral public key integrity** check from the **Sapling** *Output statement*. It is unnecessary for the sender of an **Orchard** *note* to prove knowledge of $\mathsf{esk}$, because the potential attack this originally addressed for **Sapling** is prevented by checks added at **Canopy** activation in [ZIP-212] (which are required after the end of the ZIP 212 grace period).
- If $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ returns $\bot$ for the old or new *note*, then the corresponding note commitment integrity check is satisfied. This models the fact that the implemented circuit uses incomplete addition to compute $\mathsf{SinsemillaHashToPoint}$. If an exceptional case were to occur, the prover could arbitrarily choose the intermediate $\lambda$ value in an addition, which must be assumed to allow them to control the output. (The formal output of $\mathsf{SinsemillaHashToPoint}$ is $\bot$ in such a case, while the output computed by the circuit would be nondeterministic.) But as proven in Theorem 5.4.4 on p. 77, these exceptional cases allow immediately finding a nontrivial discrete logarithm, which is infeasible by assumption, and so finding such a case is infeasible.

## 4.18 In-band secret distribution (Sprout)

In **Sprout**, the secrets that need to be transmitted to a recipient of funds in order for them to later spend, are $\mathsf{v}$, $\rho$, and $\mathsf{rcm}$. (After **Canopy** activation, $\mathsf{rcm}$ is replaced by $\mathsf{rseed}$.) A *memo field* (§ 3.2.1 ***'Note Plaintexts and Memo Fields'*** on p. 15) 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* $\mathsf{ivk}$ 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 $\mathsf{Sym}$ be the scheme instantiated in § 5.4.3 ***'Symmetric Encryption'*** on p. 81;
- let $\mathsf{KDF}^{\mathsf{Sprout}}$ be the *Key Derivation Function* instantiated in § 5.4.5.2 ***'Sprout Key Derivation'*** on p. 82;
- let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81;
- let $\mathsf{h_{Sig}}$ be the value computed for this *JoinSplit description* in § 4.3 ***'JoinSplit Descriptions'*** on p. 36.

### 4.18.1 Encryption (Sprout)

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81.

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 § 5.5 ***'Encodings of Note Plaintexts and Memo Fields'*** on p. 104.

Then to encrypt:

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

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

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

### 4.18.2 Decryption (Sprout)

Let $\mathsf{ivk} = (\mathsf{a_{pk}}, \mathsf{sk_{enc}})$ be the recipient's *incoming viewing key*, and let $\mathsf{pk_{enc}}$ be the corresponding *transmission key* derived from $\mathsf{sk_{enc}}$ as specified in § 4.2.1 ***'Sprout Key Components'*** on p. 32.

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 $\mathtt{DecryptNoteSprout}(\mathsf{K}_i^{\mathsf{enc}}, \mathsf{C}_i^{\mathsf{enc}}, \mathsf{cm}_i, \mathsf{a_{pk}})$.
```

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

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

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

**Notes:**

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

See § 8.7 ***'In-band secret distribution'*** on p. 136 for further discussion of the security and engineering rationale behind this encryption scheme.

## 4.19 In-band secret distribution (Sapling and Orchard)

In **Sapling** and **Orchard**, the secrets that need to be transmitted to a recipient of funds in order for them to later spend, are $\mathsf{d}$, $\mathsf{v}$, and $\mathsf{rcm}$. A *memo field* (§ 3.2.1 ***'Note Plaintexts and Memo Fields'*** on p. 15) is also transmitted.

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

Unlike in a **Sprout** *JoinSplit description*, each **Sapling** or **Orchard** *shielded output* is encrypted by a fresh *ephemeral public key*.

For both encryption and decryption,

- let $\ell_{\mathsf{ovk}}$ be as defined in § 5.3 ***'Constants'*** on p. 67;
- let Sym be the encryption scheme instantiated in § 5.4.3 ***'Symmetric Encryption'*** on p. 81;
- let KA be the *key agreement scheme* $\mathsf{KA}^{\mathsf{Sapling}}$ or $\mathsf{KA}^{\mathsf{Orchard}}$ instantiated in § 5.4.5.3 ***'Sapling Key Agreement'*** on p. 82 or § 5.4.5.5 ***'Orchard Key Agreement'*** on p. 82;
- let KDF be the *Key Derivation Function* $\mathsf{KDF}^{\mathsf{Sapling}}$ or $\mathsf{KDF}^{\mathsf{Orchard}}$ instantiated in § 5.4.5.4 ***'Sapling Key Derivation'*** on p. 82 or § 5.4.5.6 ***'Orchard Key Derivation'*** on p. 83;
- let $\mathbb{G}$, $\ell_{\mathbb{G}}$, and $\mathsf{repr}_{\mathbb{G}}$ be instantiated as $\mathbb{J}$, $\ell_{\mathbb{J}}$, and $\mathsf{repr}_{\mathbb{J}}$ defined in § 5.4.9.3 'Jubjub' on p. 94, or $\mathbb{P}$, $\ell_{\mathbb{P}}$, and $\mathsf{repr}_{\mathbb{P}}$ defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97;

- let $\mathsf{Extract}_{\mathbb{G}^{(r)}}$ be $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ as defined in § 5.4.9.4 *‘Coordinate Extractor for* Jubjub*’* on p. 96 or $\mathsf{Extract}_{\mathbb{P}}$ as defined in § 5.4.9.7 *‘Coordinate Extractor for* Pallas*’* on p. 98;
- let $\mathsf{PRF}^{\mathsf{ock}}$ be $\mathsf{PRF}^{\mathsf{ockSapling}}$ or $\mathsf{PRF}^{\mathsf{ockOrchard}}$ instantiated in § 5.4.2 *‘Pseudo Random Functions’* on p. 79;
- let DiversifyHash be $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ in § 5.4.1.6 ‘$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ *Hash Functions’* on p. 71, or $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ in the same section;
- let NoteCommitment be $\mathsf{NoteCommitment}^{\mathsf{Sapling}}$ or $\mathsf{NoteCommitment}^{\mathsf{Orchard}}$ instantiated in § 3.2 *‘Notes’* on p. 13;
- let ToScalar be $\mathsf{ToScalar}^{\mathsf{Sapling}}$ defined in § 4.2.2 ***‘Sapling*** *Key Components’* on p. 32 or $\mathsf{ToScalar}^{\mathsf{Orchard}}$ defined in § 4.2.3 ***‘Orchard*** *Key Components’* on p. 34.

### 4.19.1 Encryption (Sapling and Orchard)

Let $\mathsf{pk_d} ⦂ \mathsf{KA.PublicPrimeSubgroup}$ be the *diversified transmission key* for the intended recipient address of a new **Sapling** or **Orchard** *note*, and let $\mathsf{g_d} ⦂ \mathsf{KA.PublicPrimeSubgroup}$ be the corresponding *diversified base* computed as $\mathsf{DiversifyHash}(\mathsf{d})$.

Since **Sapling** *note* encryption is used only in the context of § 4.7.2 *‘Sending Notes (**Sapling**)’* on p. 41, and similarly **Orchard** *note* encryption is used only in the context of § 4.7.3 *‘Sending Notes (**Orchard**)’* on p. 42, we may assume that $\mathsf{g_d}$ has already been calculated and is not $\bot$. Also, the *ephemeral private key* esk has been chosen.

Let $\mathsf{ovk} ⦂ \mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{ovk}}/8]} \cup \{\bot\}$ be as described in § 4.7.2 on p. 41 or § 4.7.3 on p. 42, i.e. the *outgoing viewing key* of the *shielded payment address* from which the *note* is being spent, or an *outgoing viewing key* associated with a [ZIP-32] account, or $\bot$.

Let $\mathbf{np} = (\mathsf{leadByte}, \mathsf{d}, \mathsf{v}, \mathsf{rseed}, \mathsf{memo})$ be the **Sapling** or **Orchard** *note plaintext*.

$\mathbf{np}$ is encoded as defined in § 5.5 *‘Encodings of Note Plaintexts and Memo Fields’* on p. 104.

Let cv be the *value commitment* for the new *note*, and let cm be the *note commitment*. (These are needed to derive the *outgoing cipher key* ock in order to produce the *Output ciphertext* $\mathsf{C^{out}}$.)

Then to encrypt:

```
let $\mathsf{P^{enc}}$ be the raw encoding of $\mathbf{np}$
let $\mathsf{epk} = \mathsf{KA.DerivePublic}(\mathsf{esk}, \mathsf{g_d})$
let $\mathtt{ephemeralKey} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{epk})\big)$
let $\mathsf{sharedSecret} = \mathsf{KA.Agree}(\mathsf{esk}, \mathsf{pk_d})$
let $\mathsf{K^{enc}} = \mathsf{KDF}(\mathsf{sharedSecret}, \mathtt{ephemeralKey})$
let $\mathsf{C^{enc}} = \mathsf{Sym.Encrypt}_{\mathsf{K^{enc}}}(\mathsf{P^{enc}})$
if $\mathsf{ovk} = \bot$:
    choose random $\mathsf{ock} \xleftarrow{R} \mathsf{Sym}.\mathbf{K}$ and $\mathbf{op} \xleftarrow{R} \mathbb{B}^{\mathbb{Y}[(\ell_{\mathbb{G}}+256)/8]}$
else:
    let $\mathtt{cv} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{cv})\big)$
    let $\mathtt{cm*} = \mathsf{LEBS2OSP}_{256}\big(\mathsf{Extract}_{\mathbb{G}^{(r)}}(\mathsf{cm})\big)$
    let $\mathsf{ock} = \mathsf{PRF}^{\mathsf{ock}}_{\mathsf{ovk}}(\mathtt{cv}, \mathtt{cm*}, \mathtt{ephemeralKey})$
    let $\mathbf{op} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}+256}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{pk_d})\ \|\ \mathsf{I2LEBSP}_{256}(\mathsf{esk})\big)$
let $\mathsf{C^{out}} = \mathsf{Sym.Encrypt}_{\mathsf{ock}}(\mathbf{op})$
```

The resulting *transmitted note ciphertext* is $(\mathtt{ephemeralKey}, \mathsf{C^{enc}}, \mathsf{C^{out}})$.

**Note:** It is technically possible to replace $\mathsf{C^{enc}}$ 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 *Output descriptions*. This mode of operation raises further security considerations, for example of how to validate a **Sapling** or **Orchard** *note* received out-of-band, which are not addressed in this document.

### 4.19.2 Decryption using an Incoming Viewing Key (Sapling and Orchard)

Let $\mathsf{ivk} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}}-1\}$ (in **Sapling**) or $\{0\,..\,q_{\mathbb{P}}-1\}$ (in **Orchard**) be the recipient's *incoming viewing key*, specified in § 4.2.2 ***'Sapling Key Components'*** on p. 32 or § 4.2.3 ***'Orchard Key Components'*** on p. 34.

Let $(\mathtt{ephemeralKey}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$ be the *transmitted note ciphertext* from the *Output description*. Let $\mathtt{cm}*$ be the $\mathtt{cmu}$ or $\mathtt{cmx}$ field of the *Output description* or *Action description* respectively. (This encodes the $u$-coordinate or $x$-coordinate of the *note commitment*, i.e. $\mathsf{Extract}_{\mathbb{G}^{(r)}}(\mathsf{cm})$.)

Let the constant $\mathsf{CanopyActivationHeight}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{height}$ be the *block height* of the *block* containing this *transaction*.

The recipient will attempt to decrypt the $\mathtt{ephemeralKey}$ and $\mathsf{C}^{\mathsf{enc}}$ components of the *transmitted note ciphertext*:

```
let $\mathsf{epk} = \mathsf{abst}_{\mathbb{G}}(\mathtt{ephemeralKey})$
if $\mathsf{epk} = \bot$, return $\bot$
let $\mathsf{sharedSecret} = \mathsf{KA.Agree}(\mathsf{ivk}, \mathsf{epk})$
let $\mathsf{K}^{\mathsf{enc}} = \mathsf{KDF}(\mathsf{sharedSecret}, \mathtt{ephemeralKey})$
let $\mathsf{P}^{\mathsf{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K}^{\mathsf{enc}}}(\mathsf{C}^{\mathsf{enc}})$
if $\mathsf{P}^{\mathsf{enc}} = \bot$, return $\bot$
extract $\mathbf{np} = (\mathsf{leadByte} ⦂ \mathbb{BY}, \mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}, \mathsf{rseed} ⦂ \mathbb{BY}^{[32]}, \mathsf{memo} ⦂ \mathbb{BY}^{[512]})$ from $\mathsf{P}^{\mathsf{enc}}$
[Pre-Canopy] if $\mathsf{leadByte} \neq \mathtt{0x01}$, return $\bot$
[Pre-Canopy] let $\underline{\mathsf{rcm}} = \mathsf{rseed}$
[Canopy onward] if $\mathsf{height} < \mathsf{CanopyActivationHeight} + \mathsf{ZIP212GracePeriod}$ and $\mathsf{leadByte} \notin \{\mathtt{0x01}, \mathtt{0x02}\}$, return $\bot$
[Canopy onward] if $\mathsf{height} \geq \mathsf{CanopyActivationHeight} + \mathsf{ZIP212GracePeriod}$ and $\mathsf{leadByte} \neq \mathtt{0x02}$, return $\bot$
[Canopy onward] let $\underline{\mathsf{rcm}} = \begin{cases} \mathsf{rseed}, & \text{if } \mathsf{leadByte} = \mathtt{0x01} \\ \mathsf{ToScalar}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big), & \text{otherwise} \end{cases}$
let $\mathsf{rcm} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{rcm}})$ and $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$
if $\mathsf{rcm} \geq r_{\mathbb{G}}$ or (for Sapling) $\mathsf{g_d} = \bot$, return $\bot$
[Canopy onward] if $\mathsf{leadByte} \neq \mathtt{0x01}$:
    $\mathsf{esk} = \mathsf{ToScalar}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([4])\big)$
    if $\mathsf{repr}_{\mathbb{G}}\big(\mathsf{KA.DerivePublic}(\mathsf{esk}, \mathsf{g_d})\big) \neq \mathtt{ephemeralKey}$, return $\bot$
let $\mathsf{pk_d} = \mathsf{KA.DerivePublic}(\mathsf{ivk}, \mathsf{g_d})$
for Sapling, let $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm})$
for Orchard:
    let $\psi = \mathsf{ToBase}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([9])\big)$
    let $\rho$ be equal to $\mathsf{nf}^{\mathsf{old}}$ from the same Action description.
    let $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$
if $\mathsf{LEBS2OSP}_{256}\big(\mathsf{Extract}_{\mathbb{G}^{(r)}}\big(\mathsf{NoteCommitment}(\mathbf{n})\big)\big) \neq \mathtt{cm}*$, return $\bot$
return $(\mathbf{n}, \mathsf{memo})$.
```

**Notes:**

- $\mathsf{g_d}$ has already been computed when applying $\mathsf{NoteCommitment}$, and need not be computed again.
- For **Sapling**, as explained in the note in § 5.4.9.3 'Jubjub' on p. 94, $\mathsf{abst}_{\mathbb{J}}$ accepts *non-canonical* compressed encodings of Jubjub curve points. Therefore, an implementation **MUST** use the original $\mathtt{ephemeralKey}$ field as encoded in the *transaction* as input to $\mathsf{KDF}^{\mathsf{Sapling}}$, and (if **Canopy** is active and $\mathsf{leadByte} \neq \mathtt{0x01}$) in the comparison against $\mathsf{repr}_{\mathbb{G}}\big(\mathsf{KA.DerivePublic}(\mathsf{esk}, \mathsf{g_d})\big)$. For consistency this is also what is specified for **Orchard**.

- Normally only *transmitted note ciphertexts* of *transactions* in *blocks* need to be decrypted. In that case, any received **Sapling** *note* is necessarily a *positioned note*, so its $\rho$ value can immediately be calculated as in § 4.16 *'Note Commitments and Nullifiers'* on p. 53. To test whether a **Sapling** or **Orchard** *note* is unspent in a particular *block chain* also requires the *nullifier deriving key* $\mathsf{nk}$; the coin is unspent if and only if the *nullifier* computed as described in § 4.16 *'Note Commitments and Nullifiers'* on p. 53 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.
- A client **MAY** attempt to decrypt a *transmitted note ciphertext* of a *transaction* in the *mempool*, using the next *block height* for $\mathsf{height}$. However, in that case it **MUST NOT** assume that the *transaction* will be mined and **MUST** treat the decrypted information as provisional, and private.

### 4.19.3 Decryption using a Full Viewing Key (Sapling and Orchard)

Let $\mathsf{ovk} \;⦂\; \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]}$ be the *outgoing viewing key*, as specified in § 4.2.2 *'Sapling Key Components'* on p. 32 or § 4.2.3 *'Orchard Key Components'* on p. 34, that is to be used for decryption. (If $\mathsf{ovk} = \bot$ was used for encryption, the payment is not decryptable by this method.)

Let $(\mathtt{ephemeralKey}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$ be the *transmitted note ciphertext*.

For a **Sapling** *transmitted note ciphertext*, let $\mathtt{cv}$ and $\mathtt{cm}*$ be the $\mathtt{cv}$ and $\mathtt{cmu}$ fields of the *Output description*.

For an **Orchard** *transmitted note ciphertext*, let $\mathtt{cv}$ and $\mathtt{cm}*$ be the $\mathtt{cv}$ and $\mathtt{cmx}$ fields of the *Action description*.

The *outgoing viewing key* holder will attempt to decrypt the *transmitted note ciphertext* as follows:

```
let $\mathsf{ock} = \mathsf{PRF}^{\mathsf{ock}}_{\mathsf{ovk}}(\mathtt{cv}, \mathtt{cm}*, \mathtt{ephemeralKey})$
let $\mathbf{op} = \mathsf{Sym.Decrypt}_{\mathsf{ock}}(\mathsf{C}^{\mathsf{out}})$
if $\mathbf{op} = \bot$, return $\bot$
extract $(\mathsf{pk}\star_{\mathsf{d}} ⦂ \mathbb{B}^{[\ell_{\mathbb{G}}]}, \underline{\mathsf{esk}} ⦂ \mathbb{BY}^{[32]})$ from $\mathbf{op}$
let $\mathsf{esk} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{esk}})$ and $\mathsf{pk_d} = \mathsf{abst}_{\mathbb{G}}(\mathsf{pk}\star_{\mathsf{d}})$
if $\mathsf{esk} \geq r_{\mathbb{G}}$ or $\mathsf{pk_d} = \bot$, return $\bot$
[NU5 onward] if $\mathsf{repr}_{\mathbb{P}}(\mathsf{pk_d}) \neq \mathsf{pk}\star_{\mathsf{d}}$, return $\bot$
let $\mathsf{sharedSecret} = \mathsf{KA.Agree}(\mathsf{esk}, \mathsf{pk_d})$
let $\mathsf{K}^{\mathsf{enc}} = \mathsf{KDF}(\mathsf{sharedSecret}, \mathtt{ephemeralKey})$
let $\mathsf{P}^{\mathsf{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K}^{\mathsf{enc}}}(\mathsf{C}^{\mathsf{enc}})$
if $\mathsf{P}^{\mathsf{enc}} = \bot$, return $\bot$
extract $\mathbf{np} = (\mathsf{leadByte} ⦂ \mathbb{BY}, \mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0\,..\,2^{\ell_{\mathsf{value}}}-1\}, \mathsf{rseed} ⦂ \mathbb{BY}^{[32]}, \mathsf{memo} ⦂ \mathbb{BY}^{[512]})$ from $\mathsf{P}^{\mathsf{enc}}$
[Pre-Canopy] if $\mathsf{leadByte} \neq \mathtt{0x01}$, return $\bot$
[Pre-Canopy] let $\underline{\mathsf{rcm}} = \mathsf{rseed}$
[Canopy onward] if $\mathsf{height} < \mathsf{CanopyActivationHeight} + \mathsf{ZIP212GracePeriod}$ and $\mathsf{leadByte} \notin \{\mathtt{0x01}, \mathtt{0x02}\}$, return $\bot$
[Canopy onward] if $\mathsf{height} \geq \mathsf{CanopyActivationHeight} + \mathsf{ZIP212GracePeriod}$ and $\mathsf{leadByte} \neq \mathtt{0x02}$, return $\bot$
[Canopy onward] if $\mathsf{leadByte} \neq \mathtt{0x01}$ and $\mathsf{ToScalar}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([4])\big) \neq \mathsf{esk}$, return $\bot$
[Canopy onward] let $\underline{\mathsf{rcm}} = \begin{cases} \mathsf{rseed}, & \text{if } \mathsf{leadByte} = \mathtt{0x01} \\ \mathsf{ToScalar}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5])\big), & \text{otherwise} \end{cases}$
let $\mathsf{rcm} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{rcm}})$ and $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$
if $\mathsf{rcm} \geq r_{\mathbb{G}}$ or (for Sapling) $\mathsf{g_d} = \bot$ or $\mathsf{pk_d} \notin \mathbb{J}^{(r)}$, return $\bot$
for Sapling, let $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm})$
for Orchard:
        let $\psi = \mathsf{ToBase}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([9])\big)$
        let $\rho$ be equal to $\mathsf{nf}^{\mathsf{old}}$ from the same Action description.
        let $\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$
    if $\mathsf{LEBS2OSP}_{256}\big(\mathsf{Extract}_{\mathbb{G}^{(r)}}\big(\mathsf{NoteCommitment}(\mathbf{n})\big)\big) \neq \mathtt{cm*}$, return $\bot$
    if $\mathsf{repr}_{\mathbb{G}}\big(\mathsf{KA.DerivePublic}(\mathsf{esk}, \mathsf{g_d})\big) \neq \mathtt{ephemeralKey}$, return $\bot$
    return $(\mathbf{n}, \mathsf{memo})$.
```

**Notes:**

- $\mathsf{g_d}$ has already been computed when applying NoteCommitment, and need not be computed again.
- A previous version of this specification did not have the requirement for the decoded point $\mathsf{pk_d}$ of a **Sapling** *note* to be in the subgroup $\mathbb{J}^{(r)}$ (i.e. "if ... $\mathsf{pk_d} \not\in \mathbb{J}^{(r)}$, return $\bot$"). That did not match the implementation in zcashd, which ***does*** require $\mathsf{pk_d}$ to be in the subgroup. The specification has been changed to match zcashd.
- As explained in the note in § 5.4.9.3 'Jubjub' on p. 94, $\mathsf{abst}_{\mathbb{J}}$ accepts *non-canonical* compressed encodings of Jubjub curve points. Therefore, an implementation **MUST** use the original `ephemeralKey` field as encoded in the *transaction* as input to $\mathsf{PRF}^{\mathsf{ock}}$ and $\mathsf{KDF}^{\mathsf{Sapling}}$, and in the comparison against $\mathsf{repr}_{\mathbb{J}}\big(\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}, \mathsf{g_d})\big)$. For consistency this is also what is specified for **Orchard**.
- [Pre-**NU5**] $\mathsf{pk}\star_{\mathsf{d}}$ can also be *non-canonical*. Since $\bot$ is returned if $\mathsf{g_d} \not\in \mathbb{J}^{(r)}$, the only accepted *non-canonical* encoding for $\mathsf{pk}\star_{\mathsf{d}}$ of a **Sapling** *note* is $\mathsf{I2LEBSP}_{256}\big(2^{255}+1\big)$.
- [**NU5** onward] This procedure returns $\bot$ if $\mathsf{pk}\star_{\mathsf{d}}$ is *non-canonical* (which can only occur for **Sapling** *notes*), as specified in [ZIP-216].
- The comments in § 4.19.2 ***'Decryption using an Incoming Viewing Key (Sapling and Orchard)'*** on p. 62 concerning calculation of ρ, detection of spent *notes*, and decryption of *transmitted note ciphertexts* for *transactions* in the *mempool* also apply to *notes* decrypted by this procedure.

**Non-normative note:** Implementors should pay close attention to similarities and differences between this procedure and § 4.19.2 ***'Decryption using an Incoming Viewing Key (Sapling and Orchard)'*** on p. 62. In particular:

- in this procedure, the ephemeral *private key* $\mathsf{esk}'$ derived from rseed is checked to be identical to that obtained from $\mathbf{op}$ (when $\mathsf{leadByte} \neq \mathtt{0x01}$);
- in this procedure, $\mathsf{pk_d}$ is obtained from $\mathbf{op}$ rather than being derived as $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{ivk}, \mathsf{g_d})$;
- in this procedure, the check that $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}, \mathsf{g_d}) = \mathsf{epk}$ is unconditional rather than being dependent on $\mathsf{leadByte} \neq \mathtt{0x01}$, and it uses the esk obtained from $\mathbf{op}$.

## 4.20 Block Chain Scanning (Sprout)

Let $\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathsf{Note}^{\mathsf{Sprout}}$ be as defined in § 3.2 ***'Notes'*** on p. 13.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81.

Let $\mathsf{ivk} = (\mathsf{a_{pk}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sprout}}_{\mathsf{PRF}}]}, \mathsf{sk_{enc}} ⦂ \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private})$ be the *incoming viewing key* corresponding to $\mathsf{a_{sk}}$, and let $\mathsf{pk_{enc}}$ be the associated *transmission key*, as specified in § 4.2.1 ***'Sprout Key Components'*** on p. 32.

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 ReceivedSet ⦂ $\mathscr{P}(\mathsf{Note}^{\mathsf{Sprout}} \times \mathbb{B}^{\mathbb{Y}[512]}) \leftarrow \{\}$
let mutable SpentSet ⦂ $\mathscr{P}(\mathsf{Note}^{\mathsf{Sprout}}) \leftarrow \{\}$
let mutable 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 § 4.18.2 ‘Decryption (Sprout)’ on p. 59. 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 § 3.2 ‘Notes’ on p. 13.
                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 $\mathsf{nf}_i$ is present in NullifierMap, add $\mathsf{NullifierMap}(\mathsf{nf}_i)$ to SpentSet
return (ReceivedSet, SpentSet).
```

## 4.21 Block Chain Scanning (Sapling and Orchard)

In **Sapling** and **Orchard**, *block chain* scanning requires only the nk and ivk key components, rather than a *spending key* as in **Sprout**.

Typically, these components are derived from a *full viewing key* as described in § 4.2.2 ***‘Sapling Key Components’*** on p. 32 or § 4.2.3 ***‘Orchard Key Components’*** on p. 34.

Let $\ell_{\mathsf{PRFnfSapling}}$ be as defined in § 5.3 ***‘Constants’*** on p. 67.

Let $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 ‘Pallas *and* Vesta’ on p. 97.

Let Note be $\mathsf{Note}^{\mathsf{Sapling}}$ or $\mathsf{Note}^{\mathsf{Orchard}}$ as defined in § 3.2 ***‘Notes’*** on p. 13.

Let KA be either $\mathsf{KA}^{\mathsf{Sapling}}$ as defined in § 5.4.5.3 ***‘Sapling Key Agreement’*** on p. 82, or $\mathsf{KA}^{\mathsf{Orchard}}$ as defined in § 5.4.5.5 ***‘Orchard Key Agreement’*** on p. 82.

Let NullifierType be $\mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{PRFnfSapling}}/8]}$ for **Sapling**, or $\mathbb{F}_{q_{\mathbb{P}}}$ for **Orchard**.

The following algorithm can be used, given the *block chain* and (nk, ivk), to obtain each *note* sent to the corresponding *shielded payment address*, its *memo field*, and its final status (spent or unspent).

```
let mutable ReceivedSet ⦂ $\mathscr{P}(\mathsf{Note} \times \mathbb{B}^{\mathbb{Y}[512]}) \leftarrow \{\}$
let mutable SpentSet ⦂ $\mathscr{P}(\mathsf{Note}) \leftarrow \{\}$
let mutable NullifierMap ⦂ $(\mathsf{NullifierType} \to \mathsf{Note}) \leftarrow$ the empty mapping
for each transaction tx:
    for each Output description or Action description in tx:
        Attempt to decrypt the transmitted note ciphertext components epk and $\mathsf{C}^{\mathsf{enc}}$ using ivk with the algorithm
        § 4.19.2 ‘Decryption using an Incoming Viewing Key (Sapling and Orchard)’ on p. 62. If this succeeds
        with $(\mathbf{n}, \mathsf{memo})$:
            Add $(\mathbf{n}, \mathsf{memo})$ to $\mathsf{ReceivedSet}$.
            Calculate the nullifier $\mathsf{nf}$ of $\mathbf{n}$ using $\mathsf{nk}$ as described in § 3.2 ‘Notes’ on p. 13. (This also requires $\mathsf{pos}$ from the Output description for Sapling notes.)
            Add the mapping $\mathsf{nf} \to \mathbf{n}$ to $\mathsf{NullifierMap}$.
    for each nullifier $\mathsf{nf}$ of a Spend description or Action description in $\mathsf{tx}$:
        if $\mathsf{nf}$ is present in $\mathsf{NullifierMap}$, add $\mathsf{NullifierMap}(\mathsf{nf})$ to $\mathsf{SpentSet}$
return $(\mathsf{ReceivedSet}, \mathsf{SpentSet})$.
```

**Non-normative notes:**

- The above algorithm does not use the ovk key component, or the $\mathsf{C^{out}}$ *transmitted note ciphertext* component. When scanning the whole *block chain*, these are indeed not necessary. The advantage of supporting decryption using ovk as described in § 4.19.3 ***‘Decryption using a Full Viewing Key (Sapling and Orchard)’*** on p. 63, is that it allows recovering information about the *note plaintexts* sent in a *transaction* from that *transaction* alone.
- When scanning only part of a *block chain*, it may be useful to augment the above algorithm with decryption of $\mathsf{C^{out}}$ components for each *transaction*, in order to obtain information about *notes* that were spent in the scanned period but received outside it.
- The above algorithm does not detect *notes* that were sent “out-of-band” or with incorrect *transmitted note ciphertexts*. It is possible to detect whether such *notes* were spent only if their *nullifiers* are known.

# 5 Concrete Protocol

## 5.1 Caution

TODO: Explain the kind of things that can go wrong with linkage between abstract and concrete protocol. E.g. § 8.5 ***‘Internal hash collision attack and fix’*** on p. 135

## 5.2 Integers, Bit Sequences, and Endianness

All integers in **Zcash**-specific encodings are unsigned, have a fixed bit length, and are encoded in little-endian byte order ***unless otherwise specified***.

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

- $\mathsf{I2LEBSP} \;⦂\; (\ell \;⦂\; \mathbb{N}) \times \{0 \,..\, 2^{\ell}-1\} \to \mathbb{B}^{[\ell]}$, such that $\mathsf{I2LEBSP}_{\ell}(x)$ is the sequence of $\ell$ bits representing $x$ in little-endian order;
- $\mathsf{I2LEOSP} \;⦂\; (\ell \;⦂\; \mathbb{N}) \times \{0 \,..\, 2^{\ell}-1\} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$, 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 $\ell$ 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 $\ell$.
- $\mathsf{LEOS2IP} \;⦂\; (\ell \;⦂\; \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\ell/8]} \to \{0 \,..\, 2^{\ell}-1\}$ such that $\mathsf{LEOS2IP}_{\ell}(S)$ is the integer represented in little-endian order by the byte sequence $S$ of length $\ell/8$.

- $\mathsf{BEOS2IP} ⦂ (\ell ⦂ \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\ell/8]} \to \{0\,..\,2^{\ell}-1\}$ such that $\mathsf{BEOS2IP}_{\ell}(S)$ is the integer represented in big-endian order by the byte sequence $S$ of length $\ell/8$.
- $\mathsf{LEBS2OSP} ⦂ (\ell ⦂ \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ defined as follows: pad the input on the right with $8 \cdot \mathsf{ceiling}(\ell/8) - \ell$ zero bits so that its length is a multiple of 8 bits. Then convert each group of 8 bits to a byte value with the ***least*** significant bit first, and concatenate the resulting bytes in the same order as the groups.
- $\mathsf{LEOS2BSP} ⦂ (\ell ⦂ \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]} \to \mathbb{B}^{[\ell]}$ 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.

In bit layout diagrams, each box of the diagram represents a sequence of bits. Diagrams are read from left-to-right, with lines read from top-to-bottom; the breaking of boxes across lines has no significance. The bit length $\ell$ 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 $\ell$ 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. (This convention is used only in descriptions of the **Sprout** design; in the **Sapling** and **Orchard** additions, bit/byte sequence conversions are always specified explicitly.) Where bit fields are used, the text will clarify their position in each case.

## 5.3 Constants

Define:

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

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

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

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

$\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}} ⦂ \mathbb{N} := 255$

$\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}} ⦂ \mathbb{N} := 255$

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

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

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

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

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

$\ell_{\mathsf{PRFexpand}} ⦂ \mathbb{N} := 512$

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

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

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

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

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

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

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

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

$\ell^{\mathsf{Sapling}}_{\mathsf{ivk}} ⦂ \mathbb{N} := 251$

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

$\ell^{\mathsf{Sapling}}_{\mathsf{scalar}} : \mathbb{N} := 252$

$\ell^{\mathsf{Orchard}}_{\mathsf{scalar}} : \mathbb{N} := 255$

$\ell^{\mathsf{Orchard}}_{\mathsf{base}} : \mathbb{N} := 255$

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

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

$\mathsf{Uncommitted}^{\mathsf{Orchard}} : \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}]} := \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(2)$

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

$\mathsf{BlossomActivationHeight} : \mathbb{N} := \begin{cases} 653600, & \text{for } \textit{Mainnet} \\ 584000, & \text{for } \textit{Testnet} \end{cases}$

$\mathsf{CanopyActivationHeight} : \mathbb{N} := \begin{cases} 1046400, & \text{for } \textit{Mainnet} \\ 1028500, & \text{for } \textit{Testnet} \end{cases}$

$\mathsf{ZIP212GracePeriod} : \mathbb{N} := 32256$

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

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

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

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

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

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

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

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

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

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

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

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

$\mathsf{PostBlossomPoWTargetSpacing} : \mathbb{N} := 75$ (seconds).

## 5.4 Concrete Cryptographic Schemes

### 5.4.1 Hash Functions

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

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

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

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

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

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

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

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

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

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

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

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

Ed25519 uses SHA-512:

$$\mathsf{SHA\text{-}512} : \mathbb{BY}^{[\mathbb{N}]} \rightarrow \mathbb{BY}^{[64]}$$

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

### 5.4.1.2 BLAKE2 Hash Functions

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

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

BLAKE2b is used to instantiate hSigCRH, EquihashGen, and $\mathsf{KDF}^{\mathsf{Sprout}}$. From **Overwinter** onward, it is used to compute *SIGHASH transaction hashes* as specified in [ZIP-143], or as in [ZIP-243] after **Sapling** activation. For **Sapling**, it is also used to instantiate $\mathsf{PRF}^{\mathsf{expand}}$, $\mathsf{PRF}^{\mathsf{ockSapling}}$, $\mathsf{KDF}^{\mathsf{Sapling}}$, and in the RedJubjub *signature scheme* which instantiates $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ and $\mathsf{BindingSig}^{\mathsf{Sapling}}$.

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

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

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

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

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

### 5.4.1.3 Merkle Tree Hash Function

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

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

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

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

SHA256Compress is defined in § 5.4.1.1 *'SHA-256, SHA-256d, SHA256Compress, and SHA-512 Hash Functions'* on p. 68.

**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.

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

Let PedersenHash be as specified in § 5.4.1.7 *'Pedersen Hash Function'* on p. 72.

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

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

**Security requirement:** PedersenHash must be *collision-resistant*.

**Note:** The prefix $l$ provides domain separation between inputs at different layers of the *note commitment tree*. $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, like PedersenHash, is defined in terms of PedersenHashToPoint, but using a prefix that cannot collide with a layer prefix, as noted in § 5.4.8.2 *'Windowed Pedersen commitments'* on p. 88.

### $\mathsf{MerkleCRH}^{\mathsf{Orchard}}$ Hash Function

Let SinsemillaHash be as specified in § 5.4.1.9 *'Sinsemilla Hash Function'* on p. 74.

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

$$\mathsf{MerkleCRH}^{\mathsf{Orchard}}(\mathsf{layer}, \mathsf{left}, \mathsf{right}) := \mathsf{SinsemillaHash}(\text{“}\mathtt{z.cash:Orchard\text{-}MerkleCRH}\text{”}, l \,\|\, \mathsf{left} \,\|\, \mathsf{right})$$
$$\text{where } l = \mathsf{I2LEBSP}_{10}\big(\mathsf{MerkleDepth}^{\mathsf{Orchard}} - 1 - \mathsf{layer}\big).$$

**Security requirement:** SinsemillaHash must be *collision-resistant*.

**Note:** The prefix $l$ provides domain separation between inputs at different layers of the *note commitment tree*.

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

hSigCRH is used to compute the value $\mathsf{h_{Sig}}$ in § 4.3 *'JoinSplit Descriptions'* on p. 36.

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

where

$$\mathsf{hSigInput} := \begin{array}{|c|c|c|c|}\hline \text{256-bit randomSeed} & \text{256-bit } \mathsf{nf}^{\mathsf{old}}_1 \;\ldots & \text{256-bit } \mathsf{nf}^{\mathsf{old}}_{\mathrm{N}^{\mathsf{old}}} & \text{256-bit } \mathtt{joinSplitPubKey} \\ \hline\end{array}\,.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 *'BLAKE2 Hash Functions'* on p. 69.

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

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

$\mathsf{CRH^{ivk}}$ is used to derive the *incoming viewing key* $\mathsf{ivk}$ for a **Sapling** *shielded payment address*. For its use when generating an address see § 4.2.2 ***'Sapling Key Components'*** on p. 32, and for its use in the *Spend statement* see § 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55.

It is defined as follows:

$$\mathsf{CRH^{ivk}}(\mathsf{ak}\star, \mathsf{nk}\star) := \mathsf{LEOS2IP}_{256}(\mathsf{BLAKE2s\text{-}256}(\texttt{"Zcashivk"},\ \mathsf{crhInput})) \bmod 2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}}$$

where

$$\mathsf{crhInput} := \boxed{\quad \mathsf{LEBS2OSP}_{256}(\mathsf{ak}\star) \quad}\boxed{\quad \mathsf{LEBS2OSP}_{256}(\mathsf{nk}\star) \quad}$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69.

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

**Non-normative note:** BLAKE2s has a variable output digest length feature, but it does not support arbitrary bit lengths, otherwise it would have been used rather than external truncation. However, the protocol-specific personalization string together with truncation achieve essentially the same effect as using that feature.

### 5.4.1.6 $\mathsf{DiversifyHash^{Sapling}}$ and $\mathsf{DiversifyHash^{Orchard}}$ Hash Functions

$\mathsf{DiversifyHash^{Sapling}} \colon \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{J}^{(r)*} \cup \{\bot\}$ is used to derive a *diversified base* in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

Let $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ and $U$ be as defined in § 5.4.9.5 ***'Group Hash into** Jubjub'* on p. 96.

Define

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

$\mathsf{DiversifyHash^{Orchard}} \colon \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{P}^*$ is used to derive a *diversified base* in § 4.2.3 ***'Orchard Key Components'*** on p. 34.

Let $\mathsf{GroupHash}^{\mathbb{P}}$ be as defined in § 5.4.9.8 ***'Group Hash into** Pallas **and** Vesta'* on p. 98.

Define

$$\mathsf{DiversifyHash^{Orchard}}(\mathsf{d}) := \begin{cases} \mathsf{GroupHash}^{\mathbb{P}}(\texttt{"z.cash:Orchard-gd"}, \texttt{""}), & \text{if } P = \mathcal{O}_{\mathbb{P}} \\ P, & \text{otherwise} \end{cases}$$

where $P = \mathsf{GroupHash}^{\mathbb{P}}(\texttt{"z.cash:Orchard-gd"}, \mathsf{LEBS2OSP}_{\ell_{\mathsf{d}}}(\mathsf{d}))$.

The following security property and notes apply to both **Sapling** and **Orchard**.

**Security requirement:** **Unlinkability:** Given two randomly selected *shielded payment addresses* from different spend authorities, and a third *shielded payment address* which could be derived from either of those authorities, such that the three addresses use different *diversifiers*, it is not possible to tell which authority the third address was derived from.

**Non-normative notes:**

- Suppose that $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ (restricted to inputs for which it does not return $\bot$) is modelled as a *random oracle* from *diversifiers* to points of order $r_{\mathbb{J}}$ on the Jubjub curve. In this model, Unlinkability of $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ holds under the Decisional Diffie–Hellman assumption on the prime-order subgroup of the Jubjub curve.

  To prove this, consider the ElGamal encryption scheme [ElGamal1985] on this prime-order subgroup, restricted to encrypting plaintexts encoded as the group identity $\mathcal{O}_{\mathbb{J}}$. (ElGamal was originally defined for $\mathbb{F}_p^*$ but works in any prime-order group.) ElGamal *public keys* then have the same form as *diversified payment addresses*. If we make the assumption above on $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$, then generating a new *diversified payment address* from a given address $\mathsf{pk}$, gives the same distribution of $(\mathsf{g_d}', \mathsf{pk_d}')$ pairs as the distribution of ElGamal ciphertexts obtained by encrypting $\mathcal{O}_{\mathbb{J}}$ under $\mathsf{pk}$. TODO: check whether this is justified. Then, the definition of *key privacy* (IK-CPA as defined in [BBDP2001, Definition 1]) for ElGamal corresponds to the definition of Unlinkability for $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$. (IK-CCA corresponds to the potentially stronger requirement that $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ remains Unlinkable when given Diffie–Hellman key agreement oracles for each of the candidate *diversified payment addresses*.) So if ElGamal is *key-private*, then $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ is Unlinkable under the same conditions. [BBDP2001, Appendix A] gives a security proof for *key privacy* (both IK-CPA and IK-CCA) of ElGamal under the Decisional Diffie–Hellman assumption on the relevant group. (In fact the proof needed is the "small modification" described in the last paragraph in which the generator is chosen at random for each key.)

- It is assumed (also for the security of other uses of the group hash, such as Pedersen hashes and commitments) that the discrete logarithm of the output group element with respect to any other generator is unknown. This assumption is justified if the group hash acts as a *random oracle*. Essentially, *diversifiers* act as handles to unknown random numbers. (The group hash inputs used with different personalizations are in different "namespaces".)

- Informally, the random self-reducibility property of DDH implies that an adversary would gain no advantage from being able to query an oracle for additional $(\mathsf{g_d}, \mathsf{pk_d})$ pairs with the same spend authority as an existing *shielded payment address*, since they could also create such pairs on their own. This justifies only considering two *shielded payment addresses* in the security definition.

  TODO: FIXME This is not correct, because additional pairs don't quite follow the same distribution as an address with a valid diversifier. The security definition may need to be more complex to model this properly.

- An $88$-bit diversifier cannot be considered cryptographically unguessable at a $128$-bit security level; also, randomly chosen diversifiers are likely to suffer birthday collisions when the number of choices approaches $2^{44}$.

  If most users are choosing diversifiers randomly (as recommended in § 4.2.2 ***'Sapling Key Components'*** on p. 32), then the fact that they may accidentally choose diversifiers that collide (and therefore reveal the fact that they are not derived from the same *incoming viewing key*) does not appreciably reduce the anonymity set.

  In [ZIP-32] and § 4.2.3 ***'Orchard Key Components'*** on p. 34 an $88$-bit *Pseudo Random Permutation*, keyed differently for each node of the derivation tree, is used to select new *diversifiers*. This resolves the potential problem, provided that the input to the *Pseudo Random Permutation* does not repeat for a given node.

- If the holder of an *incoming viewing key* permits an adversary to ask for a new address for that *incoming viewing key* with a given *diversifier*, then it can trivially break Unlinkability for the other *diversified payment addresses* associated with the *incoming viewing key* (this does not compromise other privacy properties). Implementations **SHOULD** avoid providing such a "chosen *diversifier*" oracle.

#### 5.4.1.7 Pedersen Hash Function

$\mathsf{PedersenHash}$ is an algebraic *hash function* with *collision resistance* (for fixed input length) derived from assumed hardness of the Discrete Logarithm Problem on the Jubjub curve. It is based on the work of David Chaum, Ivan

Damgård, Jeroen van de Graaf, Jurjen Bos, George Purdy, Eugène van Heijst and Birgit Pfitzmann in [CDvdG1987], [BCP1988] and [CvHP1991], and of Mihir Bellare, Oded Goldreich, and Shafi Goldwasser in [BGG1995], with optimizations for efficient instantiation in *zk-SNARK circuits* by Sean Bowe and Daira Hopwood.

PedersenHash is used in the definitions of *Pedersen commitments* (§ 5.4.8.2 ***'Windowed Pedersen commitments'*** on p. 88), and of the *Pedersen hash* for the **Sapling** *incremental Merkle tree* (§ 5.4.1.3 'MerkleCRH$^{\mathsf{Sapling}}$ ***Hash Function'*** on p. 70).

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathcal{O}_{\mathbb{J}}$, $q_{\mathbb{J}}$, $r_{\mathbb{J}}$, $a_{\mathbb{J}}$, and $d_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{Extract}_{\mathbb{J}^{(r)}} : \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell_{\mathsf{Merkle}}^{\mathsf{Sapling}}]}$ be as defined in § 5.4.9.4 ***'Coordinate Extractor for*** Jubjub' on p. 96.

Let $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ be as defined in § 5.4.9.5 ***'Group Hash into*** Jubjub' on p. 96.

Let $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $c$ be the largest integer such that $4 \cdot \frac{2^{4 \cdot c} - 1}{15} \leq \frac{r_{\mathbb{J}} - 1}{2}$, i.e. $c := 63$.

Define $\mathcal{I}^{\mathsf{Sapling}} : \mathbb{BY}^{[8]} \times \mathbb{N} \to \mathbb{J}^{(r)*}$ by:

$$\mathcal{I}_i^{\mathsf{D}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}\left(D, \boxed{\text{32-bit } i - 1}\right).$$

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

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

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

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

Return $\sum_{i=1}^{n} [\langle M_i \rangle]\, \mathcal{I}_i^{\mathsf{D}} : \mathbb{J}^{(r)}$.

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

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

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

Write each $m_j$ as $[s_0^j, s_1^j, s_2^j]$, and let $\mathsf{enc}(m_j) = (1 - 2 \cdot s_2^j) \cdot (1 + s_0^j + 2 \cdot s_1^j) : \mathbb{Z}$.

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

Finally, define $\mathsf{PedersenHash} : \mathbb{BY}^{[8]} \times \mathbb{B}^{[\mathbb{N}^+]} \to \mathbb{B}^{[\ell_{\mathsf{Merkle}}^{\mathsf{Sapling}}]}$ by:

$$\mathsf{PedersenHash}(D, M) := \mathsf{Extract}_{\mathbb{J}^{(r)}}\big(\mathsf{PedersenHashToPoint}(D, M)\big).$$

See § A.3.3.9 ***'Pedersen hash'*** on p. 183 for rationale and efficient circuit implementation of these functions.

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

**Non-normative note:** These *hash functions* are *not collision-resistant* for variable-length inputs.

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

*Proof.* We first check that the range of $\sum_{j=1}^{k_i} \mathsf{enc}(m_j) \cdot 2^{4\cdot(j-1)}$ is a subset of the allowable range $\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$. The range of this expression is a subset of $\{-\Delta \,..\, \Delta\} \setminus \{0\}$ where $\Delta = 4 \cdot \sum_{i=1}^{c} 2^{4\cdot(i-1)} = 4 \cdot \frac{2^{4\cdot c}-1}{15}$.

When $c = 63$, we have

$$4 \cdot \frac{2^{4\cdot c}-1}{15} = \mathtt{0x444444444444444444444444444444444444444444444444444444444444444}$$

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

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

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

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

#### 5.4.1.8 Mixing Pedersen Hash Function

A mixing *Pedersen hash* is used to compute ρ from cm and pos in § 4.16 *'Note Commitments and Nullifiers'* on p. 53. It takes as input a *Pedersen commitment* $P$, and hashes it with another input $x$.

Define $\mathcal{J}^{\mathsf{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\mathtt{Zcash\_J\_}\text{”}, \text{“”})$.

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

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

**Security requirement:** The function

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

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

See § A.3.3.10 *'Mixing Pedersen hash'* on p. 185 for efficient circuit implementation of this function.

#### 5.4.1.9 Sinsemilla Hash Function

SinsemillaHash is an algebraic *hash function* with *collision resistance* (for fixed input length) derived from assumed hardness of the Discrete Logarithm Problem. It is designed by Sean Bowe and Daira Hopwood. The motivation for introducing a new discrete-log-based hash function (rather than using PedersenHash) is to make efficient use of the lookups available in recent proof systems including Halo 2.

SinsemillaHash is used in the definition of SinsemillaCommit (§ 5.4.8.4 ‘***Sinsemilla commitments***’ on p. 90), and for the **Orchard** *incremental Merkle tree* (§ 5.4.1.3 ‘$\mathsf{MerkleCRH}^{\mathsf{Orchard}}$ ***Hash Function***’ on p. 70).

Let $\mathbb{P}$, $\mathcal{O}_{\mathbb{P}}$, $q_{\mathbb{P}}$, $r_{\mathbb{P}}$, and $b_{\mathbb{P}}$ be as defined in § 5.4.9.6 ‘Pallas ***and*** Vesta’ on p. 97.

Let $\mathsf{Extract}_{\mathbb{P}} : \mathbb{P} \to \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}]}$ be as defined in § 5.4.9.7 ‘***Coordinate Extractor for*** Pallas’ on p. 98.

Let $\mathsf{GroupHash}^{\mathbb{P}}$ be as defined in § 5.4.9.8 ‘***Group Hash into*** Pallas ***and*** Vesta’ on p. 98.

Let $\mathsf{Uncommitted}^{\mathsf{Orchard}}$ be as defined in § 5.3 ‘***Constants***’ on p. 67.

Let $\mathsf{I2LEOSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^{\ell}-1\} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ and $\mathsf{LEOS2IP} : (\ell : \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\ell/8]} \to \{0\,..\,2^{\ell}-1\}$ be as defined in § 5.2 ‘***Integers, Bit Sequences, and Endianness***’ on p. 66.

Let $k := 10$.

Let $c$ be the largest integer such that $2^n \leq \frac{r_{\mathbb{P}} - 1}{2}$, i.e. $c := 253$.

Define $\mathcal{Q} : \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{P}^*$ and $\mathcal{S} : \{0\,..\,2^k-1\} \to \mathbb{P}^*$ by:

$$\begin{aligned}\mathcal{Q}(D) &:= \mathsf{GroupHash}^{\mathbb{P}}\big(\text{“}\texttt{z.cash:SinsemillaQ}\text{”}, D\big)\\ \mathcal{S}(j) &:= \mathsf{GroupHash}^{\mathbb{P}}\big(\text{“}\texttt{z.cash:SinsemillaS}\text{”}, \mathsf{I2LEOSP}_{32}(j)\,\big).\end{aligned}$$

Define $\mathbin{⁙} : \mathbb{P} \times \mathbb{P} \to \mathbb{P} \cup \{\bot\}$ as incomplete addition on the Pallas curve:

$$\begin{aligned}\mathcal{O}_{\mathbb{P}} \mathbin{⁙} \mathcal{O}_{\mathbb{P}} &= \bot\\ \mathcal{O}_{\mathbb{P}} \mathbin{⁙} (x', y') &= \bot\\ (x, y) \mathbin{⁙} \mathcal{O}_{\mathbb{P}} &= \bot\\ (x, y) \mathbin{⁙} (x', y') &= \begin{cases}\bot, & \text{if } x = x'\\ (x, y) + (x', y'), & \text{otherwise.}\end{cases}\end{aligned}$$

Define $\mathsf{SinsemillaHashToPoint}(D : \mathbb{BY}^{[\mathbb{N}]}, M : \mathbb{B}^{[\{0\,..\,k \cdot c\}]}) \to \mathbb{P}$ as follows:

```
    pad $M$ to a multiple of $k$ bits by appending zero bits, giving $M'$.
    let $n : \{0\,..\,c\} = \mathsf{ceiling}\left(\frac{\mathsf{length}(M')}{k}\right)$
    split $M'$ into $n$ pieces $M_{1\,..\,n}$, each of length $k$ bits, so that $M' = \mathsf{concat}_{\mathbb{B}}(M_{1\,..\,n})$.
    let mutable $\mathsf{Acc} \leftarrow \mathcal{Q}(D)$
    for $i$ from 1 up to $n$:
        set $\mathsf{Acc} \leftarrow \Big(\mathsf{Acc} \mathbin{⁙} \mathcal{S}\big(\mathsf{LEBS2IP}_k(M_i)\big)\Big) \mathbin{⁙} \mathsf{Acc}$
    return Acc.
```

Finally, define $\mathsf{SinsemillaHash} : \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{B}^{[\{0\,..\,k \cdot c\}]} \to \mathbb{B}^{[\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}]} \cup \{\bot\}$ by:

$$\mathsf{SinsemillaHash}(D, M) := \mathsf{Extract}_{\mathbb{P}}\big(\mathsf{SinsemillaHashToPoint}(D, M)\big).$$

See [Zcash-Orchard, section TODO “Sinsemilla”] for rationale and efficient circuit implementation of these functions.

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

**Non-normative notes:**

- These *hash functions* are ***not** collision-resistant* across variable-length inputs for the same $D$ (that is, it is assumed that a single input length will be used for any given $D$).
- The intermediate value $[2]\, \mathsf{GroupHash}^{\mathbb{P}}(\text{“}\mathtt{z.cash:SinsemillaQ}\text{”}, D)$ for the first iteration of the loop can be precomputed, if $D$ is known in advance.

**Theorem 5.4.2.** *Collision resistance of generalized* SinsemillaHash.

*Consider ... We show that ...*

*Proof.* We show a correspondence between Sinsemilla and a vector Pedersen hash, which allows using the security argument from [BGG1995] to show that collision-resistance can be tightly reduced to the discrete log problem in $\mathbb{P}$.

We collect the scalars by which each generator $\mathcal{S}(j)$ is multiplied in the algorithm for SinsemillaHashToPoint. For a given $M$ define $m_i$ as in that algorithm.

For $j \in \{0 \,..\, 2^k{-}1\}$ define

$$x_j = \sum \left\{ 2^{n-1-i} \text{ for } i \in \{0..n-1\} \text{ if } m_i = j \right\}.$$

**Lemma 5.4.3.** *There is a* $1 : 1$ *mapping from* $M$ *to* $x_{0\,..\,2^k-1} \pmod q$.

*Proof.* There is a $1 : 1$ mapping from $M$ to the matrix of bits with $2^k$ columns and $n$ rows, such that the bit at column $j$ and row $i$ is set iff $m_i = j$. Then the binary representations of $x_{0\,..\,2^k-1}$ are given by the columns of this matrix, and they do not overflow due to the requirement $2^n \leq \frac{q-1}{2}$. The claim follows. □

Then we have

$$\mathsf{SinsemillaHashToPoint}(M) = [2^n]\, Q + \sum_{j=0}^{2^k-1} [x_j]\, \mathcal{S}(j).$$

which is a Pedersen vector hash of the $x_i$'s, with a fixed offset $[2^n]\, Q$. The fixed offset does not affect collision resistance in this context. (See below for why it cannot be eliminated for SinsemillaHash or SinsemillaCommit, or when using incomplete addition.) It follows that the collision resistance of SinsemillaHash can be tightly reduced, via the proof in [BGG1995, Appendix A], to the discrete log problem over $\mathbb{P}$.

Note that [BGG1995] requires for their main scheme that the scalars are non-zero, which is not necessarily the case in our context. However, their proof in Appendix A does not depend on this, given that $n$ is fixed. The restriction that scalars are non-zero appears to have been motivated by wanting to support variable-length messages and incremental hashing, which we do not.

Now we consider SinsemillaHash. We want to prove that if we can find two messages $M$ and $M'$ such that $\mathsf{Extract}_{\mathbb{P}}^{\perp}\big(\mathsf{SinsemillaHash}(M)\big) = \mathsf{Extract}_{\mathbb{P}}^{\perp}\big(\mathsf{SinsemillaHash}(M')\big)$ then we can find a discrete logarithm. So either $\mathsf{SinsemillaHash}(M) = \mathsf{SinsemillaHash}(M')$ (in which case use the original Pedersen hash proof) or $\mathsf{SinsemillaHash}(M) = -\mathsf{SinsemillaHash}(M')$. In the latter case,

$$\begin{aligned} [2^n]\, Q + \sum_{j=0}^{2^k-1} [x_j]\, P[j] &= -\left( [2^n]\, Q + \sum_{j'=0}^{2^k-1} [x'_{j'}]\, P[j' \right) \\ [2^{n+1}]\, Q + \sum_{j=0}^{2^k-1} [x_j + x'_j]\, P[j] &= 0 \end{aligned}$$

Because the coefficients $(\bmod\ q)$ are not all zero, this is a discrete log relation between independent bases. This argument also extends straightforwardly to the binding property of SinsemillaCommit.

SinsemillaCommit is perfectly hiding because the output distribution is perfectly indistinguishable from a random $\mathbb{P}$ element, given that $r$ is a random scalar on $[0, q)$. It follows that SinsemillaShortCommit is also perfectly hiding, since hiding cannot be affected by applying any fixed function to the *output* of $Commit$.

**Non-normative note:** The above theorem covers the case where additional terms may be added to the SinsemillaHashToPoint output before applying $\mathsf{Extract}_{\mathbb{P}}$. This is needed to show security of the SinsemillaShortCommit *commitment scheme* defined in § 5.4.8.4 ***'Sinsemilla commitments'*** on p. 90. It is also needed to show security of the *nullifier* derivation defined in § 4.16 ***'Note Commitments and Nullifiers'*** on p. 53 against Faerie Gold attacks, as described in § 8.4 ***'Faerie Gold attack and fix'*** on p. 133.

**Theorem 5.4.4.** *A* $\perp$ *output from* SinsemillaHashToPoint *allows computing a nontrivial discrete logarithm.*

*Proof.* We now aim to show that when Sinsemilla is instantiated over a short Weierstrass curve, the additions can be implemented as incomplete additions.

For convenience of reference, we repeat the algorithm including the $(A + P) + A$ optimization:

$\mathsf{Hash}(M)$: Split $M$ into $n$ groups of $k$ bits. Interpret each group as a $k$-bit little-endian integer $m_i$. $A_0 := Q$ for $i$ from 0 up to $n - 1$: $A_{i+1} := (A_i + P[m_i]) + A_i$ return $A_n$

We have an exceptional case iff $A_i = \pm P[m_i]$ or $A_i + P[m_i] = \pm A_i$. (Since none of $Q$, $P[0..2^k - 1]$ are $\mathcal{O}$, no intermediate results can be $\mathcal{O}$ unless one of the preceding conditions occurs.)

If $A_i + P[m_i] = A_i$, then we have $P[m_i] = \mathcal{O}$ contrary to assumption. So exceptional cases occur only if $[\alpha]A_i + P[m_i] = \mathcal{O}$ for some $i \in [0, n)$ and some $\alpha \in \{-1, 1, 2\}$.

$A_i$ has a representation $[2^i]Q + \sum_{j=0}^{i-1}[x_j]P[j]$ for some $x_{0..i-1}$. So given $M$ that results in an exceptional case, the nontrivial discrete log relation $[\alpha 2^i]Q + \left(\sum_{j=0}^{i-1}[\alpha x_j]P[j]\right) + P[i] = \mathcal{O}$ is easily computable from $M$. (The coefficients in this representation do not overflow since $|\alpha \cdot 2^i| \leq q - 1$ for all $i < n$ and $\alpha \in \{-1, 1, 2\}$.)

Since by assumption it is hard to find a nontrivial discrete log relation, we can argue that it is safe to use incomplete additions when computing Sinsemilla inside a circuit. When computing the hash outside a circuit, we can either abort if an exceptional case occurs, or just compute it using complete formulae.

#### 5.4.1.10 PoseidonHash **Function**

Poseidon is a cryptographic permutation described in [GKRRS2019]. It operates over a sequence of finite field elements, which we instantiate as $\mathbb{F}_{q_{\mathbb{P}}}^{[3]}$.

The S-box function is $x \mapsto x^5$. The number of full rounds $R_F$ is 8, and the number of partial rounds $R_P$ is 58.

We use Poseidon in a sponge configuration [BDPA2011] (with elementwise addition in $\mathbb{F}_{q_{\mathbb{P}}}$ replacing exclusive-or of bit strings[6]) to construct a *hash function*. The sponge capacity is one field element, the rate is two field elements, and the output is one field element. We do not append any padding to the input message; this does not affect security because the input length is fixed.

That is, if $f : \mathbb{F}_{q_{\mathbb{P}}}^{[3]} \to \mathbb{F}_{q_{\mathbb{P}}}^{[3]}$ is the Poseidon permutation, then the *hash function* PoseidonHash $: \mathbb{F}_{q_{\mathbb{P}}} \times \mathbb{F}_{q_{\mathbb{P}}} \to \mathbb{F}_{q_{\mathbb{P}}}$ is specified as:

$\mathsf{PoseidonHash}(x, y) = f([x, y, 2^{65}])_1$ (using 1-based indexing).

TODO: Specify the MDS matrix.

[6] The sponge construction was originally proposed as operating on an arbitrary group. [BDPA2007]

**Non-normative notes:**

- The choice of MDS matrix and the number of rounds take into account cryptanalytic results in [KR2020] and [BCD+2020]. TODO: check.
- [BCD+2020] says that "... finite fields $\mathbb{F}_q$ with a limited number of multiplicative subgroups might be preferable, i.e. one might want to avoid $q-1$ being smooth. This implies that the fields which are suitable for implementing FFT may be more vulnerable to integral attacks." $\mathbb{F}_{q_\mathbb{P}}$ is such a field; the factorization of $q_\mathbb{P} - 1$ is $2^{32} \cdot 3 \cdot 463 \cdot 539204044132271846773 \cdot 8999194758858563409123804352480028797519453$.

  Furthermore, cryptanalysis of Poseidon has focussed mainly on the case of S-box $x \mapsto x^3$. That variant cannot be used in $\mathbb{F}_{q_\mathbb{P}}$ because $x \mapsto x^3$ would not be a permutation. $\alpha = 5$ is the smallest integer for which $x \mapsto x^\alpha$ is a permutation in $\mathbb{F}_{q_\mathbb{P}}$.

  On the other hand, the number of rounds chosen includes a significant security margin, even taking into account these considerations. For small $t$, such as $t = 3$ as used here, the results of [KR2020] are positive for security since they indicate that the number of active S-boxes through the middle rounds is larger than originally estimated by the Poseidon designers (and the number of rounds is based on this original conservative estimate).

  Also note that the use of Poseidon in **Orchard** is very conservative. First, the sponge mode limits an adversary to only being able to influence part of the Poseidon permutation input, and we use it only to construct a PRF ($\mathsf{PRF}^{\mathsf{nfOrchard}}$ as described in § 5.4.2 ***'Pseudo Random Functions'*** on p. 79). Half of the sponge input is a random key $\mathsf{nk}$, known only to holders of a *full viewing key*, and the remaining half $\rho\star$ is also chosen randomly by the *note* creator (both are derived using $\mathsf{PRF}^{\mathsf{expand}}$, from $\mathsf{sk}$ and $\mathsf{rseed}$ respectively). Then the PRF is used to enhance the security of a discrete-log-based nullifier construction (described in **§?** **'??'** on p. **??**) against a potential discrete-log-breaking adversary. Given the weak assumption that the PoseidonHash sponge produces output that preserves sufficient entropy from the inputs $\mathsf{nk}$ and $\rho\star$, this nullifier construction would still be secure under a decisional Diffie–Hellman assumption on the Pallas curve, even if the Poseidon-based PRF were distinguishable from an ideal PRF.

  The recommended number of partial rounds for these parameters in the Poseidon paper is 57, but we prefer an even number of partial rounds for circuit efficiency.
- The constant $2^{65}$ comes from [GKRRS2019, section 4.2]: "Constant-Input-Length Hashing. The capacity value is $\mathit{length} \cdot (2^{64}) + (o - 1)$ where $o$ is the output length." In this case the input length ($\mathit{length}$) is 2 field elements, and the output length is 1 field element.

#### 5.4.1.11 Equihash Generator

$\mathsf{EquihashGen}_{n,k}$ is a specialized *hash function* that maps an input and an index to an output of length $n$ bits. It is used in § 7.7.1 ***'Equihash'*** on p. 124.

Let powtag :=

| | | |
|---|---|---|
| 64-bit "ZcashPoW" | 32-bit $n$ | 32-bit $k$ |

.

Let powcount$(g)$ :=

| |
|---|
| 32-bit $g$ |

.

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

$$m = \mathsf{floor}\big(\tfrac{512}{n}\big);$$

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

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

Indices of bits in $T$ are 1-based.

$\mathsf{BLAKE2b}\text{-}\ell(p, x)$ is defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69.

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

**Note:** When EquihashGen is evaluated for sequential indices, as in the *Equihash* solving process (§ 7.7.1 ***'Equihash'*** on p. 124), the number of calls to BLAKE2b can be reduced by a factor of $\mathsf{floor}\left(\frac{512}{n}\right)$ in the best case (which is a factor of 2 for $n = 200$).

### 5.4.2 Pseudo Random Functions

Let SHA256Compress be as given in § 5.4.1.1 *'SHA-256, SHA-256d, SHA256Compress, and SHA-512 Hash Functions'* on p. 68.

The *Pseudo Random Functions* $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nfSprout}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and $\mathsf{PRF}^{\rho}$ from § 4.1.2 ***'Pseudo Random Functions'*** on p. 21, are all instantiated using SHA256Compress:

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

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

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

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

**Security requirements:**

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

**Note:** The first four bits –i.e. the most significant four bits of the first byte– are used to separate distinct uses of SHA256Compress, ensuring that the functions are independent. As well as the inputs shown here, bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function; see § 5.4.8.1 ***'Sprout Note Commitments'*** on p. 88.

(The specific bit patterns chosen here were motivated by the possibility of future extensions that might have increased $\mathrm{N}^{\mathsf{old}}$ and/or $\mathrm{N}^{\mathsf{new}}$ to 3, or added an additional bit to $\mathsf{a_{sk}}$ to encode a new key type, or that would have required an additional *PRF*. In fact since **Sapling** switches to non-SHA256Compress-based cryptographic primitives, these extensions are unlikely to be necessary.)

$\mathsf{PRF}^{\mathsf{expand}}$ is used in § 4.2.2 ***'Sapling Key Components'*** on p. 32 to derive the *Spend authorizing key* ask and the *proof authorizing key* nsk.

It is instantiated using the BLAKE2b *hash function* defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69:

$$\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(t) := \mathsf{BLAKE2b\text{-}512}(\texttt{"Zcash\_ExpandSeed"}, \mathsf{LEBS2OSP}_{256}(\mathsf{sk}) \,||\, t)$$

**Security requirement:** $\mathsf{BLAKE2b\text{-}512}(\texttt{"Zcash\_ExpandSeed"}, \mathsf{LEBS2OSP}_{256}(\mathsf{sk}) \,||\, t)$ must be a *PRF* for output range $\mathbb{BY}^{[\ell_{\mathsf{PRFexpand}}/8]}$ when keyed by the bits corresponding to sk, with input in the bits corresponding to $t$.

$\mathsf{PRF}^{\mathsf{ockSapling}}$ is used in § 4.19.1 ***'Encryption (Sapling and Orchard)'*** on p. 61 to derive the *outgoing cipher key* ock used to encrypt an *Output ciphertext*.

It is instantiated using the BLAKE2b *hash function* defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69:

$$\mathsf{PRF}^{\mathsf{ockSapling}}_{\mathsf{ovk}}(\texttt{cv}, \texttt{cmu}, \texttt{ephemeralKey}) := \mathsf{BLAKE2b\text{-}256}(\texttt{"Zcash\_Derive\_ock"}, \mathsf{ockInput})$$

$$\text{where } \mathsf{ockInput} = \begin{array}{|c|c|c|c|}\hline \mathsf{LEBS2OSP}_{256}(\mathsf{ovk}) & \text{32-byte } \texttt{cv} & \text{32-byte } \texttt{cmu} & \text{32-byte } \texttt{ephemeralKey} \\ \hline \end{array}\,.$$

**Security requirement:** BLAKE2b-512(**“`Zcash_Derive_ock`”**, $\mathsf{ockInput}$) must be a PRF for output range $\mathsf{Sym}.\mathbf{K}$ (defined in § 5.4.3 *‘Symmetric Encryption’* on p. 81) when keyed by the bits corresponding to $\mathsf{ovk}$, with input in the bits corresponding to `cv`, `cmu`, and `ephemeralKey`.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used to derive the *nullifier* for a **Sapling** *note*. It is instantiated using the BLAKE2s *hash function* defined in § 5.4.1.2 *‘BLAKE2 Hash Functions’* on p. 69:

$$\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star) := \text{BLAKE2s-256}\Big(\text{“}\mathtt{Zcash\_nf}\text{”},\ \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{nk}\star)}\boxed{\mathsf{LEBS2OSP}_{256}(\rho\star)}\Big).$$

**Security requirement:** $\text{BLAKE2s-256}\Big(\text{“}\mathtt{Zcash\_nf}\text{”},\ \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{nk}\star)}\boxed{\mathsf{LEBS2OSP}_{256}(\rho\star)}\Big)$ must be a *collision-resistant PRF* for output range $\mathbb{BY}^{[32]}$ when keyed by the bits corresponding to $\mathsf{nk}\star$, with input in the bits corresponding to $\rho\star$. Note that $\mathsf{nk}\star \colon \mathbb{J}^{(r)}_\star$ is a representation of a point in the $r_\mathbb{J}$-order subgroup of the Jubjub curve, and therefore is not uniformly distributed on $\mathbb{B}^{[\ell_\mathbb{J}]}$. $\mathbb{J}^{(r)}_\star$ is defined in § 5.4.9.3 ‘Jubjub’ on p. 94.

$\mathsf{PRF}^{\mathsf{ockOrchard}}$ is used in **§?** ‘??’ on p. **??** to derive the *outgoing cipher key* $\mathsf{ock}$ used to encrypt an *Output ciphertext*.

It is instantiated using the BLAKE2b *hash function* defined in § 5.4.1.2 *‘BLAKE2 Hash Functions’* on p. 69:

$$\mathsf{PRF}^{\mathsf{ockOrchard}}_{\mathsf{ovk}}(\mathtt{cv}, \mathtt{cmx}, \mathtt{ephemeralKey}) := \text{BLAKE2b-256}(\text{“}\mathtt{Zcash\_Orchardock}\text{”}, \mathsf{ockInput})$$

$$\text{where } \mathsf{ockInput} = \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{ovk})}\boxed{\text{32-byte } \mathtt{cv}}\boxed{\text{32-byte } \mathtt{cmx}}\boxed{\text{32-byte } \mathtt{ephemeralKey}}\,.$$

**Security requirement:** BLAKE2b-512(**“`Zcash_Orchardock`”**, $\mathsf{ockInput}$) must be a PRF for output range $\mathsf{Sym}.\mathbf{K}$ (defined in § 5.4.3 *‘Symmetric Encryption’* on p. 81) when keyed by the bits corresponding to $\mathsf{ovk}$, with input in the bits corresponding to `cv`, `cmx`, and `ephemeralKey`.

Let $q_\mathbb{P}$ be as defined in § 5.4.9.6 ‘Pallas *and* Vesta’ on p. 97.

$\mathsf{PRF}^{\mathsf{nfOrchard}} \colon \mathbb{F}_{q_\mathbb{P}} \times \mathbb{F}_{q_\mathbb{P}} \to \mathbb{F}_{q_\mathbb{P}}$ is used as part of deriving the *nullifier* for an **Orchard** *note*.

It is instantiated using the PoseidonHash *hash function* [GKRRS2019] defined in § 5.4.1.10 ‘PoseidonHash *Function*’ on p. 77:

$$\mathsf{PRF}^{\mathsf{nfOrchard}}_{\mathsf{nk}}(\rho) := \mathsf{Poseidon}(\mathsf{nk}, \rho).$$

**Security requirement:** $\mathsf{Poseidon} \colon \mathbb{F}_{q_\mathbb{P}} \times \mathbb{F}_{q_\mathbb{P}} \to \mathbb{F}_{q_\mathbb{P}}$ must be a PRF when keyed by its first argument, with its second argument as input.

**Non-normative notes:**

- This construction of a PRF from a sponge is described in [BDPA2011, section 3.12]. It is called “outer-keyed sponge” in [ADMA2015], or “black-box keying” in [GPT2015]. The results of these papers do not directly apply because the key is smaller than the rate. However, the result of [GG2015] does apply.
- See § 5.4.1.10 ‘PoseidonHash *Function*’ on p. 77 for further security discussion of how **Orchard** uses Poseidon.

### 5.4.3 Symmetric Encryption

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

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

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

### 5.4.4 Pseudo Random Permutations

Let $\ell_{\mathsf{dk}}$ and $\ell_{\mathsf{d}}$ be as defined in § 5.3 *'Constants'* on p. 67.

$\mathsf{PRP}^{\mathsf{d}} ⦂ \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{dk}}/8]}} \times \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is a *Pseudo Random Permutation* specified in § 4.1.3 *'Pseudo Random Permutations'* on p. 22. In this specification, it is used to generate *diversifiers* for **Orchard** *shielded payment addresses* in § 4.2.3 *'**Orchard** Key Components'* on p. 34. ([ZIP-32] uses an identical construction to generate *diversifiers* for **Sapling** *shielded payment addresses*.)

Let $\mathsf{FF1\text{-}AES256}_K(\mathit{tweak}, x)$ be the FF1 format-preserving encryption algorithm [NIST2016] using AES with a $256$-bit key $K$, and parameters $\mathit{radix} = 2$, $\mathit{minlen} = 88$, $\mathit{maxlen} = 88$. It will be used only with the empty string **""** as the $\mathit{tweak}$. $x$ is a sequence of $88$ bits, as is the output.

Define $\mathsf{PRP}^{\mathsf{d}}_K(\mathsf{d}) := \mathsf{FF1\text{-}AES256}_K(\text{“”}, \mathsf{d})$.

### 5.4.5 Key Agreement And Derivation

#### 5.4.5.1 Sprout Key Agreement

$\mathsf{KA}^{\mathsf{Sprout}}$ is a *key agreement scheme* as specified in § 4.1.5 *'Key Agreement'* on p. 23.

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, section 2].

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

Let $\mathsf{clamp}_{\mathsf{Curve25519}}(\underline{x})$ take a 32-byte sequence $\underline{x}$ as input and return a byte sequence representing a Curve25519 *private key*, with bits "clamped" as described in [Bernstein2006, section 3]: "clear bits $0, 1, 2$ of the first byte, clear bit $7$ of the last byte, and set bit $6$ of the last byte." Here the bits of a byte are numbered such that bit $b$ has numeric weight $2^b$.

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

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

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

#### 5.4.5.2 Sprout Key Derivation

$\mathsf{KDF}^{\mathsf{Sprout}}$ is a *Key Derivation Function* as specified in § 4.1.6 ***'Key Derivation'*** on p. 23.

It is instantiated using BLAKE2b-256 as follows:

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

where:

$$\mathsf{kdftag} := \begin{array}{|c|c|c|}\hline \text{64-bit “}\mathtt{ZcashKDF}\text{”} & \text{8-bit } i-1 & [0]^{56} \\ \hline\end{array}$$

$$\mathsf{kdfinput} := \begin{array}{|c|c|c|c|}\hline \text{256-bit } \mathsf{h_{Sig}} & \text{256-bit } \mathsf{sharedSecret}_i & \text{256-bit } \mathsf{epk} & \text{256-bit } \mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},i} \\ \hline\end{array}\,.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69.

#### 5.4.5.3 Sapling Key Agreement

$\mathsf{KA}^{\mathsf{Sapling}}$ is a *key agreement scheme* as specified in § 4.1.5 ***'Key Agreement'*** on p. 23.

It is instantiated as Diffie–Hellman with cofactor multiplication on Jubjub as follows:

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and the cofactor $h_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public} := \mathbb{J}$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup} := \mathbb{J}^{(r)}$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret} := \mathbb{J}^{(r)}$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private} := \mathbb{F}_{r_{\mathbb{J}}}$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{sk}, B) := [\mathsf{sk}]\, B$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{sk}, P) := [h_{\mathbb{J}} \cdot \mathsf{sk}]\, P$.

#### 5.4.5.4 Sapling Key Derivation

$\mathsf{KDF}^{\mathsf{Sapling}}$ is a *Key Derivation Function* as specified in § 4.1.6 ***'Key Derivation'*** on p. 23.

It is instantiated using BLAKE2b-256 as follows:

$$\mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{sharedSecret}, \mathtt{ephemeralKey}) := \mathsf{BLAKE2b\text{-}256}(\text{“}\mathtt{Zcash\_SaplingKDF}\text{”}, \mathsf{kdfinput}).$$

where:

$$\mathsf{kdfinput} := \begin{array}{|c|c|}\hline \mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{sharedSecret})\big) & \mathsf{LEBS2OSP}_{256}(\mathtt{ephemeralKey}) \\ \hline\end{array}\,.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69.

#### 5.4.5.5 Orchard Key Agreement

$\mathsf{KA}^{\mathsf{Orchard}}$ is a *key agreement scheme* as specified in § 4.1.5 ***'Key Agreement'*** on p. 23.

It is instantiated as Diffie–Hellman on Pallas as follows:

Let $\mathbb{P}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Define $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public} := \mathbb{P}$.

Define $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{SharedSecret} := \mathbb{P}$.

Define $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Private} := \mathbb{F}_{r_{\mathbb{P}}}$.

Define $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{DerivePublic}(\mathsf{sk}, B) := [\mathsf{sk}]\, B$.

Define $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Agree}(\mathsf{sk}, P) := [\mathsf{sk}]\, P$.

#### 5.4.5.6 Orchard Key Derivation

$\mathsf{KDF}^{\mathsf{Orchard}}$ is a *Key Derivation Function* as specified in § 4.1.6 ***'Key Derivation'*** on p. 23.

It is instantiated using BLAKE2b-256 as follows:

$$\mathsf{KDF}^{\mathsf{Orchard}}(\mathsf{sharedSecret}, \mathtt{ephemeralKey}) := \mathsf{BLAKE2b\text{-}256}(\text{“}\mathtt{Zcash\_OrchardKDF}\text{”}, \mathsf{kdfinput}).$$

where:

$$\mathsf{kdfinput} := \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{repr}_{\mathbb{P}}(\mathsf{sharedSecret}))}\boxed{\mathsf{LEBS2OSP}_{256}(\mathtt{ephemeralKey})}\,.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in § 5.4.1.2 ***'BLAKE2 Hash Functions'*** on p. 69.

### 5.4.6 Ed25519

Ed25519 is a *signature scheme* as specified in § 4.1.7 ***'Signature'*** on p. 24. It is used to instantiate JoinSplitSig as described in § 4.11 ***'Non-malleability (Sprout)'*** on p. 46.

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

[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00],

[0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 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, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f],

[0xed, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f],

[0xee, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f],

[0xd9, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff],

[0xda, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff]

$\}$.

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

Let $a = -1$.

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

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

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

Define I2LEOSP, LEOS2BSP, and LEBS2IP as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Define $\mathsf{reprBytes}_{\mathsf{Ed25519}} : \mathsf{Ed25519} \to \mathbb{B}^{\mathbb{Y}[32]}$ such that $\mathsf{reprBytes}_{\mathsf{Ed25519}}(x, y) = \mathsf{I2LEOSP}_{256}\big(y + 2^{255} \cdot \tilde{x}\big)$, where $\tilde{x} = x \bmod 2$.[7]

[7] 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 § 5.4.9.3 'Jubjub' on p. 94 and in § A.2 ***'Elliptic curve background'*** on p. 173.

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

let $y\star ⦂ \mathbb{B}^{[255]}$ be the first 255 bits of $\mathsf{LEOS2BSP}_{256}(\underline{P})$ and let $\tilde{x} ⦂ \mathbb{B}$ be the last bit.

let $y ⦂ \mathbb{F}_p = \mathsf{LEBS2IP}_{255}(y\star) \pmod{p}$.

let $x = \sqrt[?]{\frac{1 - y^2}{a - d \cdot y^2}}$. (The denominator $a - d \cdot y^2$ cannot be zero, since $\frac{a}{d}$ is not square in $\mathbb{F}_p$.)

if $x = \bot$, return $\bot$.

if $x \bmod 2 = \tilde{x}$ then return $(x, y)$ else return $(p - x, y)$.

**Note:** This definition of point decoding differs from that of [RFC-8032, section 5.1.3, as corrected by the errata]. In the latter there is an additional step "`If x = 0, and x_0 = 1, decoding fails.`", which rejects the encodings {

`[0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80],`

`[0xee, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff],`

`[0xec, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff]`

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

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

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

- $\underline{S}$ **MUST** represent an integer less than $\ell$.
- $\underline{R}$ and $\underline{A}$ **MUST** be encodings of points $R$ and $A$ respectively on the Ed25519 curve;
- [Pre-**Canopy**] $\underline{R}$ **MUST NOT** be in ExcludedPointEncodings;
- [Pre-**Canopy**] The validation equation **MUST** be equivalent to $[S]\,B = R + [c]\,A$.
- [**Canopy** onward] The validation equation **MUST** be equivalent to $[8]\,[S]\,B = [8]\,R + [8]\,[c]\,A$ for single-signature validation.

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[7] of the points represented by $\underline{R}$ or $\underline{A}$ are less than $2^{255} - 19$.
- It is *not* required that $\underline{A} \notin$ ExcludedPointEncodings.
- [**Canopy** onward] Appendix § B.3 'Ed25519 *batch validation*' on p. 196 describes an optimization that **MAY** be used to speed up validation of batches of Ed25519 signatures.

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

### 5.4.7 RedDSA, RedJubjub, and RedPallas

RedDSA is a Schnorr-based *signature scheme*, optionally supporting key re-randomization as described in § 4.1.7.1 ***'Signature with Re-Randomizable Keys'*** on p. 25. It also supports a Secret Key to Public Key Monomorphism as described in § 4.1.7.2 ***'Signature with Signing Key to Validating Key Monomorphism'*** on p. 26. It is based on a scheme from [FKMSSS2016, section 3], with some ideas from EdDSA [BJLSY2015].

RedJubjub is a specialization of RedDSA to the Jubjub curve (§ 5.4.9.3 'Jubjub' on p. 94), using the BLAKE2b-512 hash function.

The *spend authorization signature scheme* $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ is instantiated by RedJubjub, using parameters defined in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.

The *binding signature scheme* $\mathsf{BindingSig}^{\mathsf{Sapling}}$ is instantiated by RedJubjub without key re-randomization, using parameters defined in § 5.4.7.2 ***'Binding Signature (Sapling and Orchard)'*** on p. 88.

RedPallas is a specialization of RedDSA to the Pallas curve (§ 5.4.9.6 'Pallas *and* Vesta' on p. 97), using the BLAKE2b-512 hash function.

The *spend authorization signature scheme* $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ is instantiated by RedPallas, using parameters defined in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.

The *binding signature scheme* $\mathsf{BindingSig}^{\mathsf{Orchard}}$ is instantiated by RedPallas without key re-randomization, using parameters defined in § 5.4.7.2 ***'Binding Signature (Sapling and Orchard)'*** on p. 88.

Let I2LEBSP, I2LEOSP, LEOS2IP, and LEBS2OSP be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

We first describe the scheme RedDSA over a general *represented group*. Its parameters are:

- a *represented group* $\mathbb{G}$, which also defines 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 $\mathsf{abst}_{\mathbb{G}}$, as specified in § 4.1.9 ***'Represented Group'*** on p. 29;
- $\mathcal{P}_{\mathbb{G}}$, a generator of $\mathbb{G}^{(r)}$;
- a bit-length $\ell_{\mathsf{H}} ⦂ \mathbb{N}$ such that $2^{\ell_{\mathsf{H}}-128} \geq r_{\mathbb{G}}$ and $\ell_{\mathsf{H}} \bmod 8 = 0$;
- a cryptographic *hash function* $\mathsf{H} ⦂ \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{BY}^{[\ell_{\mathsf{H}}/8]}$.

Its associated types are defined as follows:

$\mathsf{RedDSA.Message} := \mathbb{BY}^{[\mathbb{N}]}$

$\mathsf{RedDSA.Signature} := \mathbb{BY}^{[\mathsf{ceiling}(\ell_{\mathbb{G}}/8)\,+\,\mathsf{ceiling}(\mathsf{bitlength}(r_{\mathbb{G}})/8)]}$

$\mathsf{RedDSA.Public} := \mathbb{G}$

$\mathsf{RedDSA.Private} := \mathbb{F}_{r_{\mathbb{G}}}$.

$\mathsf{RedDSA.Random} := \mathbb{F}_{r_{\mathbb{G}}}$.

Define $\mathsf{H}^{\circledast} ⦂ \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{F}_{r_{\mathbb{G}}}$ by:

$$\mathsf{H}^{\circledast}(B) = \mathsf{LEOS2IP}_{\ell_{\mathsf{H}}}\big(\mathsf{H}(B)\big) \pmod{r_{\mathbb{G}}}$$

Define $\mathsf{RedDSA.GenPrivate} ⦂ () \xrightarrow{\mathsf{R}} \mathsf{RedDSA.Private}$ as:

Return $\mathsf{sk} \overset{\mathsf{R}}{\leftarrow} \mathbb{F}_{r_{\mathbb{G}}}$.

Define $\mathsf{RedDSA.DerivePublic} ⦂ \mathsf{RedDSA.Private} \to \mathsf{RedDSA.Public}$ by:

$\mathsf{RedDSA.DerivePublic}(\mathsf{sk}) := [\mathsf{sk}]\,\mathcal{P}_{\mathbb{G}}$.

Define $\mathsf{RedDSA.GenRandom} ⦂ () \xrightarrow{\mathsf{R}} \mathsf{RedDSA.Random}$ as:

```
    Choose a byte sequence $T$ uniformly at random on $\mathbb{BY}^{[(\ell_{\mathsf{H}}+128)/8]}$.
    Return $\mathsf{H}^{\circledast}(T)$.
```

Define $\mathcal{O}_{\mathsf{RedDSA.Random}} := 0 \pmod{r_{\mathbb{G}}}$.

Define $\mathsf{RedDSA.RandomizePrivate} ⦂ \mathsf{RedDSA.Random} \times \mathsf{RedDSA.Private} \to \mathsf{RedDSA.Private}$ by:

```
    $\mathsf{RedDSA.RandomizePrivate}(\alpha, \mathsf{sk}) := \mathsf{sk} + \alpha \pmod{r_{\mathbb{G}}}$.
```

Define $\mathsf{RedDSA.RandomizePublic} ⦂ \mathsf{RedDSA.Random} \times \mathsf{RedDSA.Public} \to \mathsf{RedDSA.Public}$ as:

```
    $\mathsf{RedDSA.RandomizePublic}(\alpha, \mathsf{vk}) := \mathsf{vk} + [\alpha]\,\mathcal{P}_{\mathbb{G}}$.
```

Define $\mathsf{RedDSA.Sign} ⦂ (\mathsf{sk} ⦂ \mathsf{RedDSA.Private}) \times (M ⦂ \mathsf{RedDSA.Message}) \xrightarrow{\mathsf{R}} \mathsf{RedDSA.Signature}$ as:

```
    Choose a byte sequence $T$ uniformly at random on $\mathbb{BY}^{[(\ell_{\mathsf{H}}+128)/8]}$.
    Let $\underline{\mathsf{vk}} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{RedDSA.DerivePublic}(\mathsf{sk}))\big)$.
    Let $r = \mathsf{H}^{\circledast}(T \,||\, \underline{\mathsf{vk}} \,||\, M)$.
    Let $R = [r]\,\mathcal{P}_{\mathbb{G}}$.
    Let $\underline{R} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(R)\big)$.
    Let $S = (r + \mathsf{H}^{\circledast}(\underline{R} \,||\, \underline{\mathsf{vk}} \,||\, M) \cdot \mathsf{sk}) \bmod r_{\mathbb{G}}$.
    Let $\underline{S} = \mathsf{I2LEOSP}_{\mathsf{bitlength}(r_{\mathbb{G}})}(S)$.
    Return $\underline{R} \,||\, \underline{S}$.
```

Define $\mathsf{RedDSA.Validate} ⦂ (\mathsf{vk} ⦂ \mathsf{RedDSA.Public}) \times (M ⦂ \mathsf{RedDSA.Message}) \times (\sigma ⦂ \mathsf{RedDSA.Signature}) \to \mathbb{B}$ as:

```
    Let $\underline{R}$ be the first $\mathsf{ceiling}\big(\ell_{\mathbb{G}}/8\big)$ bytes of $\sigma$, and let $\underline{S}$ be the remaining $\mathsf{ceiling}\big(\mathsf{bitlength}(r_{\mathbb{G}})/8\big)$ bytes.
    Let $R = \mathsf{abst}_{\mathbb{G}}\big(\mathsf{LEOS2BSP}_{\ell_{\mathbb{G}}}(\underline{R})\big)$, and let $S = \mathsf{LEOS2IP}_{8\cdot\mathsf{length}(\underline{S})}(\underline{S})$.
    Let $\underline{\mathsf{vk}} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{vk})\big)$.
    Let $c = \mathsf{H}^{\circledast}(\underline{R} \,||\, \underline{\mathsf{vk}} \,||\, M)$.
    [NU5 onward] If $\mathsf{repr}_{\mathbb{G}}(R) \neq \underline{R}$, return $0$.
    Return $1$ if $R \neq \bot$ and $S < r_{\mathbb{G}}$ and $[h_{\mathbb{G}}]\big(-[S]\,\mathcal{P}_{\mathbb{G}} + R + [c]\,\mathsf{vk}\big) = \mathcal{O}_{\mathbb{G}}$, otherwise $0$.
```

**Notes:**

- The validation algorithm ***does not*** check that $R$ is a point of order at least $r_{\mathbb{G}}$.
- After activation of [ZIP-216], validation returns $0$ if $\underline{R}$ is a *non-canonical* compressed point encoding.
- The value $\underline{R}$ used as part of the input to $\mathsf{H}^{\circledast}$ **MUST** be exactly as encoded in the signature.
- Appendix § B.1 'RedDSA ***batch validation***' on p. 193 describes an optimization that **MAY** be used to speed up validation of batches of RedDSA signatures.

**Non-normative notes:**

- The randomization used in RedDSA.RandomizePrivate and RedDSA.RandomizePublic may interact with other uses of additive properties of keys for Schnorr-based signature schemes. In the **Zcash** protocol, such properties are used for *binding signatures* but not at the same time as key randomization. They are also used in [ZIP-32] when deriving child extended keys, but this does not result in any practical security weakness as long as the security recommendations of ZIP-32 are followed. If RedDSA is reused in other protocols making use of these additive properties, careful analysis of potential interactions is required.
- It is **RECOMMENDED** that, for deployments of RedDSA in other protocols than **Zcash**, the requirement for $\underline{R}$ to be canonically encoded is always enforced (which was the original intent of the design).

The two abelian groups specified in § 4.1.7.2 *'Signature with Signing Key to Validating Key Monomorphism'* on p. 26 are instantiated for RedDSA as follows:

- $\mathcal{O}_{\boxplus} := 0 \pmod{r_{\mathbb{G}}}$
- $\mathsf{sk}_1 \boxplus \mathsf{sk}_2 := \mathsf{sk}_1 + \mathsf{sk}_2 \pmod{r_{\mathbb{G}}}$
- $\mathcal{O}_{\diamond} := \mathcal{O}_{\mathbb{G}}$
- $\mathsf{vk}_1 \diamond \mathsf{vk}_2 := \mathsf{vk}_1 + \mathsf{vk}_2$.

As required, RedDSA.DerivePublic is a group monomorphism, since it is injective and:

$$\begin{aligned}\mathsf{RedDSA.DerivePublic}(\mathsf{sk}_1 \boxplus \mathsf{sk}_2) &= [\mathsf{sk}_1 + \mathsf{sk}_2 \pmod{r_{\mathbb{G}}}]\,\mathcal{P}_{\mathbb{G}} \\ &= [\mathsf{sk}_1]\,\mathcal{P}_{\mathbb{G}} + [\mathsf{sk}_2]\,\mathcal{P}_{\mathbb{G}} \;\; (\text{since } \mathcal{P}_{\mathbb{G}} \text{ has order } r_{\mathbb{G}}) \\ &= \mathsf{RedDSA.DerivePublic}(\mathsf{sk}_1) \diamond \mathsf{RedDSA.DerivePublic}(\mathsf{sk}_2).\end{aligned}$$

A RedDSA *validating key* vk can be encoded as a bit sequence $\mathsf{repr}_{\mathbb{G}}(\mathsf{vk})$ of length $\ell_{\mathbb{G}}$ bits (or as a corresponding byte sequence $\underline{\mathsf{vk}}$ by then applying $\mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}$).

The scheme RedJubjub specializes RedDSA with:

- $\mathbb{G} := \mathbb{J}$ as defined in § 5.4.9.3 'Jubjub' on p. 94;
- $\ell_{\mathsf{H}} := 512$;
- $\mathsf{H}(x) := \mathsf{BLAKE2b\text{-}512}(\text{“}\mathtt{Zcash\_RedJubjubH}\text{”}, x)$ as defined in § 5.4.1.2 *'BLAKE2 Hash Functions'* on p. 69.

The scheme RedPallas specializes RedDSA with:

- $\mathbb{G} := \mathbb{P}$ as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97;
- $\ell_{\mathsf{H}} := 512$;
- $\mathsf{H}(x) := \mathsf{BLAKE2b\text{-}512}(\text{“}\mathtt{Zcash\_RedPallasH}\text{”}, x)$ as defined in § 5.4.1.2 *'BLAKE2 Hash Functions'* on p. 69.

The generator $\mathcal{P}_{\mathbb{G}} : \mathbb{G}^{(r)}$ is left as an unspecified parameter, different between $\mathsf{BindingSig}^{\mathsf{Sapling}}$, $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$, $\mathsf{BindingSig}^{\mathsf{Orchard}}$, and $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$.

#### 5.4.7.1 Spend Authorization Signature (Sapling and Orchard)

Let RedJubjub be as defined in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85.

Define $\mathcal{G}^{\mathsf{Sapling}} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\mathtt{Zcash\_G\_}\text{”}, \text{“”})$.

The *spend authorization signature scheme* $\mathsf{SpendAuthSig}^{\mathsf{Sapling}}$ is instantiated as RedJubjub with key re-randomization and with generator $\mathcal{P}_{\mathbb{G}} = \mathcal{G}^{\mathsf{Sapling}}$.

Let RedPallas be as defined in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85.

Define $\mathcal{G}^{\mathsf{Orchard}} := \mathsf{GroupHash}^{\mathbb{P}}(\text{“}\mathtt{z.cash:Orchard}\text{”}, \text{“}\mathtt{G}\text{”})$.

The *spend authorization signature scheme* $\mathsf{SpendAuthSig}^{\mathsf{Orchard}}$ is instantiated as RedPallas with key re-randomization and with generator $\mathcal{P}_{\mathbb{G}} = \mathcal{G}^{\mathsf{Orchard}}$.

See § 4.15 *'Spend Authorization Signature (Sapling and Orchard)'* on p. 52 for details on the use of this *signature scheme*.

**Security requirement:** Each instantiation of SpendAuthSig must be a SURK-CMA secure *signature scheme with re-randomizable keys* as defined in § 4.1.7.1 *'Signature with Re-Randomizable Keys'* on p. 25.

#### 5.4.7.2 Binding Signature (Sapling and Orchard)

Let RedJubjub and RedPallas be as defined in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85.

The **Sapling** *binding signature scheme*, $\mathsf{BindingSig}^{\mathsf{Sapling}}$, is instantiated as RedJubjub without key re-randomization, using generator $\mathcal{P}_{\mathbb{G}} = \mathcal{R}^{\mathsf{Sapling}}$ defined in § 5.4.8.3 *'Homomorphic Pedersen commitments (**Sapling** and **Orchard**)'* on p. 89. See § 4.13 *'Balance and Binding Signature (**Sapling**)'* on p. 47 for details on the use of this *signature scheme*.

The **Orchard** *binding signature scheme*, $\mathsf{BindingSig}^{\mathsf{Orchard}}$, is instantiated as RedPallas without key re-randomization, using generator $\mathcal{P}_{\mathbb{G}} = \mathcal{R}^{\mathsf{Orchard}}$ defined in § 5.4.8.3 *'Homomorphic Pedersen commitments (**Sapling** and **Orchard**)'* on p. 89. See § 4.14 *'Balance and Binding Signature (**Orchard**)'* on p. 50 for details on the use of this *signature scheme*.

**Security requirement:** Each instantiation of BindingSig must be a SUF-CMA secure *signature scheme with key monomorphism* as defined in § 4.1.7.2 *'Signature with Signing Key to Validating Key Monomorphism'* on p. 26. A signature must prove knowledge of the discrete logarithm of the *validating key* with respect to the base $\mathcal{R}^{\mathsf{Sapling}}$ or $\mathcal{R}^{\mathsf{Orchard}}$.

### 5.4.8 Commitment schemes

#### 5.4.8.1 Sprout Note Commitments

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ specified in § 4.1.8 *'Commitment'* on p. 27 is instantiated using SHA-256 as follows:

$$\mathsf{NoteCommit}^{\mathsf{Sprout}}_{\mathsf{rcm}}(\mathsf{a_{pk}}, \mathsf{v}, \rho) := \text{SHA-256}\left(\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{256-bit } \mathsf{a_{pk}} & \text{64-bit } \mathsf{v} & \text{256-bit } \rho & \text{256-bit } \mathsf{rcm} \\ \hline\end{array}\right)$$

$\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{GenTrapdoor}()$ 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.

#### 5.4.8.2 Windowed Pedersen commitments

§ 5.4.1.7 *'Pedersen Hash Function'* on p. 72 defines a *Pedersen hash* construction. We construct "*windowed*" *Pedersen commitments* by reusing that construction, and adding a randomized point on the Jubjub curve (see § 5.4.9.3 'Jubjub' on p. 94):

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

See § A.3.5 *'Windowed Pedersen Commitment'* on p. 186 for rationale and efficient circuit implementation of this function.

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ specified in § 4.1.8 *'Commitment'* on p. 27 is instantiated as follows using WindowedPedersenCommit:

$$\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g}\star_{\mathsf{d}}, \mathsf{pk}\star_{\mathsf{d}}, \mathsf{v}) := \mathsf{WindowedPedersenCommit}_{\mathsf{rcm}}\left([1]^6 \,||\, \mathsf{I2LEBSP}_{64}(\mathsf{v}) \,||\, \mathsf{g}\star_{\mathsf{d}} \,||\, \mathsf{pk}\star_{\mathsf{d}}\right)$$

$\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_{\mathbb{J}}}$.

**Security requirements:**

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

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

**Notes:**

- $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ is also defined in terms of $\mathsf{PedersenHashToPoint}$ (see § 5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 69). The prefix $[1]^6$ distinguishes the use of $\mathsf{WindowedPedersenCommit}$ in $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ from the layer prefix used in $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$. That layer prefix is a 6-bit little-endian encoding of an integer in the range $\{0\,..\,\mathsf{MerkleDepth}^{\mathsf{Sapling}}-1\}$; because $\mathsf{MerkleDepth}^{\mathsf{Sapling}} < 64$, it cannot collide with $[1]^6$.
- The arguments to $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ are in a different order to their encodings in $\mathsf{WindowedPedersenCommit}$. There is no particularly good reason for this.

**Theorem 5.4.5.** $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ *is not in the range of* $\mathsf{NoteCommit}^{\mathsf{Sapling}}$.

*Proof.* $\mathsf{Uncommitted}^{\mathsf{Sapling}}$ is defined as $\mathsf{I2LEBSP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}(1)$. By injectivity of $\mathsf{I2LEBSP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}$ and definitions of $\mathsf{Extract}_{\mathbb{J}^{(r)}}$, $\mathsf{WindowedPedersenCommit}$, and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathsf{I2LEBSP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}(1)$ can be in the range of $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ only if there exist $\mathsf{rcm} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}$, $D ⦂ \mathbb{B}^{\mathbb{Y}^{[8]}}$, and $M ⦂ \mathbb{B}^{[\mathbb{N}^+]}$ such that $\mathcal{U}(\mathsf{WindowedPedersenCommit}_{\mathsf{rcm}}(D, M)) = 1$. The latter can only be the *affine-ctEdwards* $u$-coordinate of a point in $\mathbb{J}$. We show that there are no points in $\mathbb{J}$ with *affine-ctEdwards* $u$-coordinate 1. Suppose for a contradiction that $(u, \mathsf{v}) \in \mathbb{J}$ for $u = 1$ and some $\mathsf{v} ⦂ \mathbb{F}_{r_{\mathbb{S}}}$. By writing the curve equation as $\mathsf{v}^2 = (1 - a_{\mathbb{J}}\cdot u^2)/(1 - d_{\mathbb{J}}\cdot u^2)$, and noting that $1 - d_{\mathbb{J}}\cdot u^2 \neq 0$ because $d_{\mathbb{J}}$ is nonsquare, we have $\mathsf{v}^2 = (1 - a_{\mathbb{J}})/(1 - d_{\mathbb{J}})$. The right-hand-side is a nonsquare in $\mathbb{F}_{r_{\mathbb{S}}}$ (for the Jubjub curve parameters), so there are no solutions for $\mathsf{v}$ (contradiction). □

### 5.4.8.3 Homomorphic Pedersen commitments (Sapling and Orchard)

The windowed Pedersen commitments defined in the preceding section are highly efficient, but they do not support the homomorphic property we need when instantiating $\mathsf{ValueCommit}$.

For more details on the use of this property, see § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47, § 4.14 ***'Balance and Binding Signature (Orchard)'*** on p. 50, and § 3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 17.

In order to support this property, we also define *homomorphic Pedersen commitments* for **Sapling**:

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

$\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_{\mathbb{J}}}$.

See § A.3.6 ***'Homomorphic Pedersen Commitment'*** on p. 186 for rationale and efficient circuit implementation of this function.

We also define *homomorphic Pedersen commitments* for **Orchard**:

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

$\mathsf{ValueCommit}^{\mathsf{Orchard}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_{\mathbb{P}}}$.

Define:

$$\begin{aligned}
\mathcal{V}^{\mathsf{Sapling}} &:= \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\texttt{Zcash\_cv}\text{”}, \text{“}\texttt{v}\text{”}) \\
\mathcal{R}^{\mathsf{Sapling}} &:= \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{“}\texttt{Zcash\_cv}\text{”}, \text{“}\texttt{r}\text{”}) \\
\mathcal{V}^{\mathsf{Orchard}} &:= \mathsf{GroupHash}^{\mathbb{P}}(\text{“}\texttt{z.cash:Orchard-cv}\text{”}, \text{“}\texttt{v}\text{”}) \\
\mathcal{R}^{\mathsf{Orchard}} &:= \mathsf{GroupHash}^{\mathbb{P}}(\text{“}\texttt{z.cash:Orchard-cv}\text{”}, \text{“}\texttt{r}\text{”})
\end{aligned}$$

The commitment scheme $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ specified in § 4.1.8 ‘*Commitment*’ on p. 27 is instantiated as follows using $\mathsf{HomomorphicPedersenCommit}^{\mathsf{Sapling}}$ on the Jubjub curve:

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

which is equivalent to:

$$\mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v}) := [\mathsf{v}]\, \mathcal{V}^{\mathsf{Sapling}} + [\mathsf{rcv}]\, \mathcal{R}^{\mathsf{Sapling}}.$$

The commitment scheme $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ specified in § 4.1.8 ‘*Commitment*’ on p. 27 is instantiated as follows using $\mathsf{HomomorphicPedersenCommit}^{\mathsf{Orchard}}$ on the Pallas curve:

$$\mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v}) := \mathsf{HomomorphicPedersenCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\text{“}\texttt{z.cash:Orchard-cv}\text{”}, \mathsf{v}).$$

which is equivalent to:

$$\mathsf{ValueCommit}^{\mathsf{Orchard}}_{\mathsf{rcv}}(\mathsf{v}) := [\mathsf{v}]\, \mathcal{V}^{\mathsf{Orchard}} + [\mathsf{rcv}]\, \mathcal{R}^{\mathsf{Orchard}}.$$

**Security requirements:**

- $\mathsf{HomomorphicPedersenCommit}^{\mathsf{Sapling}}$ and $\mathsf{HomomorphicPedersenCommit}^{\mathsf{Orchard}}$ must be computationally binding and at least computationally hiding *commitment schemes*, for a given personalization input $D$.
- $\mathsf{ValueCommit}^{\mathsf{Sapling}}$ and $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ must be computationally binding and at least computationally hiding *commitment schemes*.

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

### 5.4.8.4 Sinsemilla commitments

Let $\ell^{\mathsf{Orchard}}_{\mathsf{base}}$ be as defined in § 5.3 ‘*Constants*’ on p. 67.

Let $\mathsf{Extract}_{\mathbb{P}}$ be as defined in § 5.4.9.7 ‘*Coordinate Extractor for* Pallas’ on p. 98.

§ 5.4.1.9 ‘*Sinsemilla Hash Function*’ on p. 74 defines a *Sinsemilla hash* construction. We construct *Sinsemilla commitments* by reusing that construction, and adding a randomized point on the Pallas curve (see § 5.4.9.6 ‘Pallas *and* Vesta’ on p. 97):

$$\begin{aligned}
&\mathsf{SinsemillaCommit}_r(D, M) := \mathsf{SinsemillaHashToPoint}(D \,||\, \text{“}\texttt{-M}\text{”}, M) + [r]\, \mathsf{GroupHash}^{\mathbb{P}}(D \,||\, \text{“}\texttt{-r}\text{”}, \text{“”}) \\
&\mathsf{SinsemillaShortCommit}_r(D, M) := \mathsf{Extract}_{\mathbb{P}}\big(\mathsf{SinsemillaCommit}_r(D, M)\big).
\end{aligned}$$

See [Zcash-Orchard, Section TODO] for rationale and efficient circuit implementation of this function.

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ specified in § 4.1.8 ‘*Commitment*’ on p. 27 is instantiated as follows using $\mathsf{SinsemillaCommit}$:

$$\begin{aligned}
&\mathsf{NoteCommit}^{\mathsf{Orchard}}_{\mathsf{rcm}}(\mathsf{g}\star_{\mathsf{d}}, \mathsf{pk}\star_{\mathsf{d}}, \mathsf{v}, \rho, \psi) := \\
&\quad \mathsf{SinsemillaCommit}_{\mathsf{rcm}}\big(\text{“}\texttt{z.cash:Orchard-NoteCommit}\text{”}, \\
&\qquad\qquad \mathsf{g}\star_{\mathsf{d}} \,||\, \mathsf{pk}\star_{\mathsf{d}} \,||\, \mathsf{I2LEBSP}_{64}(\mathsf{v}) \,||\, \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{base}}}(\rho) \,||\, \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{base}}}(\psi)\big)
\end{aligned}$$

$\mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_{\mathbb{P}}}$.

The commitment scheme $\mathsf{Commit}^{\mathsf{ivk}}$ specified in § 4.1.8 ***'Commitment'*** on p. 27 is instantiated as follows using SinsemillaCommit:

$$\mathsf{Commit}^{\mathsf{ivk}}_{\mathsf{rivk}}(\mathsf{ak}, \mathsf{nk}) := \mathsf{SinsemillaShortCommit}_{\mathsf{rivk}}\big(\text{“}\texttt{z.cash:Orchard-CommitIvk}\text{”}, \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{base}}}(\mathsf{ak}) \,||\, \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{base}}}(\mathsf{nk})\big) \pmod{r_{\mathbb{P}}}$$

$\mathsf{Commit}^{\mathsf{ivk}}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_{\mathbb{P}}}$.

**Security requirements:**

- SinsemillaCommit and SinsemillaShortCommit, and hence $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ and $\mathsf{Commit}^{\mathsf{ivk}}$, must be computationally binding and at least computationally hiding *commitment schemes*.

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

**Notes:**

- $\mathsf{MerkleCRH}^{\mathsf{Orchard}}$ is also defined in terms of SinsemillaHashToPoint (see § 5.4.1.3 ***'Merkle Tree Hash Function'*** on p. 69).
- The arguments to $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ are the same order as their encodings in the input to SinsemillaCommit; this is different to $\mathsf{NoteCommit}^{\mathsf{Sapling}}$.

**Theorem 5.4.6.** $\mathsf{Uncommitted}^{\mathsf{Orchard}}$ *is not in the range of* $\mathsf{NoteCommit}^{\mathsf{Orchard}}$.

*Proof.* $\mathsf{Uncommitted}^{\mathsf{Orchard}}$ is defined as $\mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(2)$. By injectivity of $\mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}$ and definitions of $\mathsf{Extract}_{\mathbb{P}}$, SinsemillaShortCommit, and $\mathsf{NoteCommit}^{\mathsf{Orchard}}$, $\mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(2)$ can be in the range of $\mathsf{NoteCommit}^{\mathsf{Orchard}}$ only if there exist $\mathsf{rcm} \,⦂\, \mathsf{NoteCommit}^{\mathsf{Orchard}}.\mathsf{Trapdoor}$, $D \,⦂\, \mathbb{BY}^{[\mathbb{N}]}$, and $M \,⦂\, \mathbb{B}^{[\mathbb{N}^+]}$ such that $\mathsf{Extract}_{\mathbb{P}}\big(\mathsf{SinsemillaCommit}_{\mathsf{rcm}}(D, M)\big) = 2$. $\mathsf{Extract}_{\mathbb{P}}\big(\mathsf{SinsemillaHashToPoint}(D, M)\big)$ can only be $0$ or the *affine-short-Weierstrass* $x$-coordinate of a point in $\mathbb{P}$. But $0 \neq 2 \pmod{q_{\mathbb{P}}}$, and there are no points in $\mathbb{P}$ with *affine-short-Weierstrass* $x$-coordinate $2 \pmod{q_{\mathbb{P}}}$, since $2^3 + b_{\mathbb{P}} = 13$ is not square in $\mathbb{F}_{q_{\mathbb{P}}}$. □

**Non-normative note:** There are also no points in $\mathbb{P}$ with *affine-short-Weierstrass* $x$-coordinate $0 \pmod{q_{\mathbb{P}}}$. We do not choose $\mathsf{Uncommitted}^{\mathsf{Orchard}} = \mathsf{I2LEBSP}_{\ell^{\mathsf{Orchard}}_{\mathsf{Merkle}}}(0)$ because we define $\mathsf{Extract}_{\mathbb{P}}(\mathcal{O}_{\mathbb{P}}) = 0$, and it is technically possible (with negligible probability) that SinsemillaHashToPoint could return $\mathcal{O}_{\mathbb{P}}$.

## 5.4.9 Represented Groups and Pairings

### 5.4.9.1 BN-254

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

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

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

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

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

Let $\mathbb{G}^{(r)}_1$ 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, $\xi$ is given by $t + 9$.

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

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

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

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

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

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

Define I2BEBSP $: (\ell : \mathbb{N}) \times \{0 \,..\, 2^\ell - 1\} \rightarrow \mathbb{B}^{[\ell]}$ as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

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 $: \mathbb{F}_{q_\mathbb{G}}[t]/(t^2+1) \rightarrow \{0 \,..\, q_\mathbb{G}^2 - 1\}$ 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, section 5.5.6.2]. The LSB compressed form (i.e. EC2OSP-XL) is used for points in $\mathbb{G}_1^{(r)*}$, and the SORT compressed form (i.e. EC2OSP-XS) for points in $\mathbb{G}_2^{(r)*}$.

- Testing $y > y'$ for the compression of $\mathbb{G}_2^{(r)*}$ points is equivalent to testing whether $(a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})$ in lexicographic order.
- Algorithms for decompressing points from the above encodings are given in [IEEE2000, Appendix A.12.8] for $\mathbb{G}_1^{(r)*}$, and [IEEE2004, Appendix A.12.11] for $\mathbb{G}_2^{(r)*}$.

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

#### 5.4.9.2 BLS12-381

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

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

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

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

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

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

Let $\mathbb{S}_1^{(r)}$ be the subgroup of order $r_{\mathbb{S}}$ of the group of rational points on a Barreto–Lynn–Scott ([BLS2002]) curve $E_{\mathbb{S}_1}$ over $\mathbb{F}_{q_{\mathbb{S}}}$ with equation $y^2 = x^3 + b_{\mathbb{S}}$. This curve has embedding degree 12 with respect to $r_{\mathbb{S}}$.

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

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

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

Let $\hat{e}_{\mathbb{S}}$ be the optimal ate pairing of type $\mathbb{S}_1^{(r)} \times \mathbb{S}_2^{(r)} \to \mathbb{S}_T^{(r)}$.

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

Let $\mathcal{P}_{\mathbb{S}_1} : \mathbb{S}_1^{(r)*} :=$

$$(3685416753713387016781088315183077757961620795782546409894578378688607592378376318836054947676345821548104185464507,$$
$$1339506544944476473020471379941921221584933875938349620426543736416511423956333506472724655353366534992391756441569).$$

Let $\mathcal{P}_{\mathbb{S}_2} : \mathbb{S}_2^{(r)*} :=$

$$(3059144344244213709971259814753781636986470325476647558659373206291635324768958432433509563104347017837885763365758 \cdot t +$$
$$352701069587466618187139116011060144890029952792775240219908644239793785735715026873347600343865175952761926303160,$$
$$927553665492332455747201965776037880757740193453592970025027978793976877002675564980949289727957565575433344219582 \cdot t +$$
$$1985150602287291935568054521177171638300868978215655730859378665066344726373823718423869104263333984641494340347905).$$

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

Define $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0 \mathinner{..} 2^\ell{-}1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

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

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

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

.

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

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

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

.

**Non-normative notes:**

- Only the $r_\mathbb{S}$-order subgroups $\mathbb{S}_{1,2,T}^{(r)}$ are used in the protocol, not their containing groups $\mathbb{S}_{1,2,T}$. Points in $\mathbb{S}_{1,2}^{(r)*}$ are ***always*** checked to be of order $r_\mathbb{S}$ when decoding from external representation. (Elements of $\mathbb{S}_T^{(r)}$ are never represented externally.) The $(r)$ superscripts on $\mathbb{S}_{1,2,T}^{(r)}$ are used for consistency with notation elsewhere in this specification.
- The points at infinity $\mathcal{O}_{\mathbb{S}_{1,2}}$ never occur in proofs and have no defined encodings in this protocol.
- In contrast to the corresponding BN-254 curve, $E_{\mathbb{S}_1}$ over $\mathbb{F}_{q_\mathbb{S}}$ is ***not*** of prime order.
- A rational point $P \neq \mathcal{O}_{\mathbb{S}_i}$ on the curve $E_{\mathbb{S}_i}$ for $i \in \{1, 2\}$ can be verified to be of order $r_\mathbb{S}$, and therefore in $\mathbb{S}_i^{(r)*}$, by checking that $r_\mathbb{S} \cdot P = \mathcal{O}_{\mathbb{S}_i}$.
- The encodings for $\mathbb{S}_{1,2}^{(r)*}$ are specific to **Zcash**.
- Algorithms for decompressing points from the encodings of $\mathbb{S}_{1,2}^{(r)*}$ are defined analogously to those for $\mathbb{G}_{1,2}^{(r)*}$ in § 5.4.9.1 'BN-254' on p. 91, taking into account that the SORT compressed form (not the LSB compressed form) is used for $\mathbb{S}_1^{(r)*}$.

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

#### 5.4.9.3 Jubjub

> *"You boil it in sawdust: you salt it in glue:*
> *You condense it with locusts and tape:*
> *Still keeping one principal object in view—*
> *To preserve its symmetrical shape."*
>
> — Lewis Carroll, "The Hunting of the Snark" [Carroll1876]

**Sapling** uses an elliptic curve, Jubjub, designed to be efficiently implementable in *zk-SNARK circuits*. The *represented group* $\mathbb{J}$ of points on this curve is defined in this section.

A *complete twisted Edwards elliptic curve*, as defined in [BL2017, section 4.3.4], is an elliptic curve $E$ over a non-binary field $\mathbb{F}_q$, parameterized by distinct $a, d \mathrel{⦂} \mathbb{F}_q \setminus \{0\}$ such that $a$ is square and $d$ is nonsquare, with equation $E : a \cdot u^2 + v^2 = 1 + d \cdot u^2 \cdot v^2$. We use the abbreviation "*ctEdwards*" to refer to *complete twisted Edwards elliptic curves* and coordinates.

Let $q_{\mathbb{J}} := r_{\mathbb{S}}$, as defined in § 5.4.9.2 'BLS12-381' on p. 93.

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

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

Let $h_{\mathbb{J}} := 8$.

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

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

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

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

Define $\mathsf{I2LEBSP} \colon (\ell \colon \mathbb{N}) \times \{0 \,..\, 2^{\ell}-1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66, and similarly for $\mathsf{LEBS2IP} \colon (\ell \colon \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \{0 \,..\, 2^{\ell}-1\}$.

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

Define $\mathsf{abst}_{\mathbb{J}} \colon \mathbb{B}^{[\ell_{\mathbb{J}}]} \to \mathbb{J} \cup \{\bot\}$ such that $\mathsf{abst}_{\mathbb{J}}(P\star)$ is computed as follows:

```
let $v\star \colon \mathbb{B}^{[255]}$ be the first 255 bits of $P\star$ and let $\tilde{u} \colon \mathbb{B}$ be the last bit.
if $\mathsf{LEBS2IP}_{255}(v\star) \geq q_{\mathbb{J}}$ then return $\bot$, otherwise let $v \colon \mathbb{F}_{q_{\mathbb{J}}} = \mathsf{LEBS2IP}_{255}(v\star) \pmod{q_{\mathbb{J}}}$.
let $u = \sqrt[?]{\dfrac{1 - v^2}{a_{\mathbb{J}} - d_{\mathbb{J}} \cdot v^2}}$. (The denominator $a_{\mathbb{J}} - d_{\mathbb{J}} \cdot v^2$ cannot be zero, since $\frac{a_{\mathbb{J}}}{d_{\mathbb{J}}}$ is not square in $\mathbb{F}_{q_{\mathbb{J}}}$.)
if $u = \bot$, return $\bot$.
if $u \bmod 2 = \tilde{u}$ then return $(u, v)$ else return $(q_{\mathbb{J}} - u, v)$.
```

**Note:** In earlier versions of this specification, $\mathsf{abst}_{\mathbb{J}}$ was defined as the left inverse of $\mathsf{repr}_{\mathbb{J}}$ such that if $S$ is not in the range of $\mathsf{repr}_{\mathbb{J}}$, then $\mathsf{abst}_{\mathbb{J}}(S) = \bot$. This differs from the specification above:

- Previously, $\mathsf{abst}_{\mathbb{J}}\Big(\mathsf{I2LEBSP}_{256}\big(2^{255} + 1\big)\Big)$ and $\mathsf{abst}_{\mathbb{J}}\Big(\mathsf{I2LEBSP}_{256}\big(2^{255} + q_{\mathbb{J}} - 1\big)\Big)$ were defined as $\bot$.
- In the current specification, $\mathsf{abst}_{\mathbb{J}}\Big(\mathsf{I2LEBSP}_{256}\big(2^{255} + 1\big)\Big) = \mathsf{abst}_{\mathbb{J}}\big(\mathsf{I2LEBSP}_{256}(1)\big) = (0, 1) = \mathcal{O}_{\mathbb{J}}$, and also $\mathsf{abst}_{\mathbb{J}}\Big(\mathsf{I2LEBSP}_{256}\big(2^{255} + q_{\mathbb{J}} - 1\big)\Big) = \mathsf{abst}_{\mathbb{J}}\big(\mathsf{I2LEBSP}_{256}(q_{\mathbb{J}} - 1)\big) = (0, -1)$.

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

Define $\mathbb{J}^{(r)}_{\star} := \Big\{ \mathsf{repr}_{\mathbb{J}}(P) \colon \mathbb{B}^{[\ell_{\mathbb{J}}]} \mid P \in \mathbb{J}^{(r)} \Big\}$.

**Non-normative notes:**

- The *ctEdwards compressed encoding* used here is consistent with that used in EdDSA [BJLSY2015] for *validating keys* and the $R$ element of a signature.
- [BJLSY2015, "Encoding and parsing curve points"] gives algorithms for decompressing points from the encoding of $\mathbb{J}$.
- [BJLSY2015, "Encoding and parsing integers"] describes several possibilities for parsing of integers; the specification of $\mathsf{abst}_{\mathbb{J}}$ above requires "strict" parsing.

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

Note that algorithms elsewhere in this specification that use Jubjub may impose other conditions on points, for example that they have order at least $r_{\mathbb{J}}$.

#### 5.4.9.4 Coordinate Extractor for Jubjub

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

Define $\mathsf{Extract}_{\mathbb{J}^{(r)}} ⦂ \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]}$ by

$\mathsf{Extract}_{\mathbb{J}^{(r)}}(P) := \mathsf{I2LEBSP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}(\mathcal{U}(P))$.

**Facts:** The point $(0, 1) = \mathcal{O}_{\mathbb{J}}$, and the point $(0, -1)$ has order $2$ in $\mathbb{J}$. $\mathbb{J}^{(r)}$ is of odd-prime order.

**Lemma 5.4.7.** *Let $P = (u, \mathsf{v}) \in \mathbb{J}^{(r)}$. Then $(u, -\mathsf{v}) \notin \mathbb{J}^{(r)}$.*

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

**Theorem 5.4.8.** *$\mathcal{U}$ is injective on $\mathbb{J}^{(r)}$.*

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

Since $\mathsf{I2LEBSP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}$ is injective, it follows that $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ is injective on $\mathbb{J}^{(r)}$.

#### 5.4.9.5 Group Hash into Jubjub

Let $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}.\mathsf{Input} := \mathbb{BY}^{[8]} \times \mathbb{BY}^{[\mathbb{N}]}$, and let $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}.\mathsf{URSType} := \mathbb{BY}^{[64]}$.

(The input element with type $\mathbb{BY}^{[8]}$ is intended to act as a “personalization” parameter to distinguish uses of the *group hash* for different purposes.)

Let URS be the MPC randomness beacon defined in § 5.9 *‘Randomness Beacon’* on p. 112.

Let BLAKE2s-256 be as defined in § 5.4.1.2 *‘BLAKE2 Hash Functions’* on p. 69.

Let LEOS2IP be as defined in § 5.2 *‘Integers, Bit Sequences, and Endianness’* on p. 66.

Let $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $\mathsf{abst}_{\mathbb{J}}$ be as defined in § 5.4.9.3 ‘Jubjub’ on p. 94.

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

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

```
let $\underline{H} = \mathsf{BLAKE2s\text{-}256}(D,\ \mathsf{URS} \,||\, M)$
let $P = \mathsf{abst}_{\mathbb{J}}(\mathsf{LEOS2BSP}_{256}(\underline{H}))$
if $P = \bot$ then return $\bot$
let $Q = [h_{\mathbb{J}}]\, P$
if $Q = \mathcal{O}_{\mathbb{J}}$ then return $\bot$, else return $Q$.
```

**Notes:**

- The use of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$ for $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ and to generate independent bases needs a *random oracle* (for inputs on which $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$ does not return $\bot$); here we show that it is sufficient to employ a simpler *random oracle* instantiated by BLAKE2s-256 in the security analysis.

  $\underline{H} \,⦂\, \mathbb{BY}^{[32]} \mapsto_{\notin \{\bot,\, \mathcal{O}_{\mathbb{J}},\, (0,-1)\}} \mathsf{abst}_{\mathbb{J}}\big(\mathsf{LEOS2BSP}_{256}(\underline{H})\big) \,⦂\, \mathbb{J}$ is injective, and both it and its inverse are efficiently computable.

  $P \,⦂\, \mathbb{J} \mapsto_{\notin \{\mathcal{O}_{\mathbb{J}}\}} [h_{\mathbb{J}}]\, P \,⦂\, \mathbb{J}^{(r)*}$ is exactly $h_{\mathbb{J}}$-to-1, and both it and its inverse relation are efficiently computable.

  It follows that when $\big(D \,⦂\, \mathbb{BY}^{[8]}, M \,⦂\, \mathbb{BY}^{[\mathbb{N}]}\big) \mapsto \mathsf{BLAKE2s\text{-}256}(D,\, \mathsf{URS} \,||\, M) \,⦂\, \mathbb{BY}^{[32]}$ is modelled as a *random oracle*, $\big(D \,⦂\, \mathbb{BY}^{[8]}, M \,⦂\, \mathbb{BY}^{[\mathbb{N}]}\big) \mapsto_{\notin \{\bot\}} \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}\big(D, M\big) \,⦂\, \mathbb{J}^{(r)*}$ also acts as a *random oracle*.
- The BLAKE2s-256 chaining variable after processing URS may be precomputed.

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

Define $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}\big(D, M\big) := \mathsf{first}(i \,⦂\, \mathbb{BY} \mapsto \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(D, M \,||\, [i]) \,⦂\, \mathbb{J}^{(r)*} \cup \{\bot\})$.

**Note:** For random input, $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ returns $\bot$ with probability approximately $2^{-256}$. In the **Zcash** protocol, most uses of $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ are for constants and do not return $\bot$; the only use that could potentially return $\bot$ is in the computation of a *default diversified payment address* in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

### 5.4.9.6 Pallas and Vesta

**Orchard** uses two elliptic curves, Pallas and Vesta, that form a cycle: the base field of each is the scalar field of the other. In **Orchard**, we use Vesta for the proof system (playing a similar rôle to BLS12-381 in **Sapling**), and Pallas for the application circuit (similar to Jubjub curve in **Sapling**). Both curves are designed to be efficiently implementable in *zk-SNARK circuits*, although we only use Pallas in that way for **Orchard**.

The *represented groups* $\mathbb{P}$ and $\mathbb{V}$ of points on Pallas and Vesta respectively are defined in this section.

A *short Weierstrass elliptic curve*, as defined for example in [Hışıl2010, Definition 2.3.1], is an elliptic curve $E$ over a field $\mathbb{F}_q$, parameterized by $a, b \,⦂\, \mathbb{F}_q$ such that $4 \cdot a^3 + 27 \cdot b^2 \neq 0$, with equation $E : y^2 = x^3 + a \cdot x + b$. The curve has a distinguished zero point $\mathcal{O}$, also called the *"point at infinity"*. For Pallas and Vesta we have $a = 0$ and so we will omit that term below.

Let $q_{\mathbb{P}} :=$ `0x40000000000000000000000000000000224698fc094cf91b992d30ed00000001`.

Let $q_{\mathbb{V}} :=$ `0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001`.

($q_{\mathbb{P}}$ and $q_{\mathbb{V}}$ are prime.)

Let $r_{\mathbb{P}} := q_{\mathbb{V}}$ and $r_{\mathbb{V}} := q_{\mathbb{P}}$.

Let $b_{\mathbb{P}} = b_{\mathbb{V}} := 5$.

Let $\mathbb{P}$ be the group of points $(x, y)$ with zero point $\mathcal{O}_{\mathbb{P}}$, on a *short Weierstrass curve* $E_{\mathbb{P}}$ over $\mathbb{F}_{q_{\mathbb{P}}}$ with equation $y^2 = x^3 + b_{\mathbb{P}}$. $\mathbb{P}$ has order $r_{\mathbb{P}}$.

Let $\mathbb{V}$ be the group of points $(x, y)$ with zero point $\mathcal{O}_{\mathbb{V}}$, on a *short Weierstrass curve* $E_{\mathbb{V}}$ over $\mathbb{F}_{q_{\mathbb{V}}}$ with equation $y^2 = x^3 + b_{\mathbb{V}}$. $\mathbb{V}$ has order $r_{\mathbb{V}}$.

For the set of points on Pallas of order $r_{\mathbb{P}}$ (which excludes $\mathcal{O}_{\mathbb{P}}$), we write $\mathbb{P}^*$.

For the set of points on Vesta of order $r_{\mathbb{V}}$ (which excludes $\mathcal{O}_{\mathbb{V}}$), we write $\mathbb{V}^*$.

Let $\ell_{\mathbb{P}} = \ell_{\mathbb{V}} := 256$.

Define $\mathsf{I2LEBSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^{\ell}-1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66, and similarly for $\mathsf{LEBS2IP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \{0\,..\,2^{\ell}-1\}$.

Define $\mathsf{repr}_{\mathbb{G}} : \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]}$ such that

$$\mathsf{repr}_{\mathbb{G}}\big(\mathcal{O}_{\mathbb{G}}\big) = \mathsf{I2LEBSP}_{256}(0)$$
$$\mathsf{repr}_{\mathbb{G}}\big((x, y)\big) = \mathsf{I2LEBSP}_{256}\big(x + 2^{255} \cdot \tilde{y}\big), \text{ where } \tilde{y} = y \bmod 2.$$

Define $\mathsf{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\}$ such that $\mathsf{abst}_{\mathbb{J}}(P\star)$is computed as follows:

let $x\star : \mathbb{B}^{[255]}$ be the first 255 bits of $P\star$ and let $\tilde{y} : \mathbb{B}$ be the last bit.

if $\mathsf{LEBS2IP}_{255}(x\star) \geq q_{\mathbb{G}}$ then return $\bot$, otherwise let $x : \mathbb{F}_{q_{\mathbb{G}}} = \mathsf{LEBS2IP}_{255}(x\star) \pmod{q_{\mathbb{G}}}$.

let $y = \sqrt[?]{x^3 + b_{\mathbb{G}}}$.

if $x = 0$ and $\tilde{y} = 0$, return $\mathcal{O}_{\mathbb{G}}$.

if $y = \bot$, return $\bot$.

if $y \bmod 2 = \tilde{y}$ then return $(x, y)$ else return $(x, q_{\mathbb{G}} - y)$.

**Notes:**

- There is no solution to $0 = x^3 + 5$ in either $\mathbb{F}_{q_{\mathbb{P}}}$ or $\mathbb{F}_{q_{\mathbb{V}}}$, and so $y$ cannot be zero. Therefore there is only one valid representation of each point on Pallas and of each point on Vesta; in particular $\mathsf{abst}_{\mathbb{P}}(\mathsf{nc}) = \bot$ and $\mathsf{abst}_{\mathbb{V}}(\mathsf{nc}) = \bot$ for $\mathsf{nc} = \mathsf{I2LEBSP}_{256}\big(2^{255}\big)$. This differs from the corresponding case of $\mathsf{abst}_{\mathbb{J}}(\mathsf{nc})$for Jubjub, for example.
- When computing square roots in $\mathbb{F}_{q_{\mathbb{P}}}$ or $\mathbb{F}_{q_{\mathbb{V}}}$ 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.

### 5.4.9.7 Coordinate Extractor for Pallas

Let $\mathbb{P}$, $\mathcal{O}_{\mathbb{P}}$, $q_{\mathbb{P}}$, and $b_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Define $\mathbb{P}_x^*$ be the set of $x$-coordinates of points on the Pallas curve, i.e. $\{x : \mathbb{F}_{q_{\mathbb{P}}} \mid x^3 + b_{\mathbb{P}} \text{ is square in } \mathbb{F}_{q_{\mathbb{P}}}\}$.

Define $\mathbb{P}_x := \mathbb{P}_x^* \cup \{0\}$.

Define $\mathsf{Extract}_{\mathbb{P}} : \mathbb{P} \to \mathbb{P}_x$ such that

$$\mathsf{Extract}_{\mathbb{P}}\big(\mathcal{O}_{\mathbb{P}}\big) = 0$$
$$\mathsf{Extract}_{\mathbb{P}}\big((x, y)\big) = x.$$

We also define $\mathsf{Extract}_{\mathbb{P}}^{\bot} : \mathbb{P} \cup \{\bot\} \to \mathbb{P}_x \cup \{\bot\}$ such that

$$\mathsf{Extract}_{\mathbb{P}}^{\bot}\big(\bot\big) = 0$$
$$\mathsf{Extract}_{\mathbb{P}}^{\bot}\big(P : \mathbb{P}\big) = \mathsf{Extract}_{\mathbb{P}}(P).$$

**Non-normative note:** $\mathsf{Extract}_{\mathbb{P}}$ returns the type $\mathbb{P}_x$ which is precise for its range, unlike $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ which returns a bit sequence.

### 5.4.9.8 Group Hash into Pallas and Vesta

**Orchard** uses the "simplified SWU" algorithm for *random-oracle* hashing to elliptic curves with $j$-invariant $0$, consistent with [ID-hashtocurve, section 6.6.3], based on a method by Riad Wahby and Dan Boneh [WB2019]. It is adapted from work of Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi in [BCIMRT2010]; Andrew Shallue and Christiaan van de Woestijne in [SvdW2006]; and Maciej Ulas in [Ulas2007].

Let $\mathbb{P}$ and $\mathbb{V}$ be the represented groups of points on the Pallas curve and the Vesta curve respectively, as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97. Let $\mathbb{G}$ be either $\mathbb{P}$ or $\mathbb{V}$ according to the desired target curve.

Also define $\mathcal{O}_\mathbb{G}$, $\mathbb{G}^*$, $q_\mathbb{G}$, and $\mathsf{abst}_\mathbb{G}$ by replacing $\mathbb{G}$ with $\mathbb{P}$ or $\mathbb{V}$, using definitions from § 5.4.9.6 'Pallas *and* Vesta' on p. 97. Let $\mathsf{curveName}_\mathbb{G}$ be **"`pallas`"** when $\mathbb{G} = \mathbb{P}$, or **"`vesta`"** when $\mathbb{G} = \mathbb{V}$.

The algorithm makes use of a curve $E_{\mathsf{iso}\text{-}\mathbb{P}}$, called iso-Pallas, that is isogenous[8] to $E_\mathbb{P}$; or $E_{\mathsf{iso}\text{-}\mathbb{V}}$, called iso-Vesta, that is isogenous to $E_\mathbb{V}$.

Let $a_{\mathsf{iso}\text{-}\mathbb{P}} :=$ `0x18354a2eb0ea8c9c49be2d7258370742b74134581a27a59f92bb4b0b657a014b`.

Let $a_{\mathsf{iso}\text{-}\mathbb{V}} :=$ `0x267f9b2ee592271a81639c4d96f787739673928c7d01b212c515ad7242eaa6b1`.

Let $b_{\mathsf{iso}\text{-}\mathbb{P}} = b_{\mathsf{iso}\text{-}\mathbb{V}} := 1265$.

Let $\mathsf{iso}\text{-}\mathbb{P}$ be the group of points $(x, y)$ with zero point $\mathcal{O}_{\mathsf{iso}\text{-}\mathbb{P}}$, on a *short Weierstrass curve* $E_{\mathsf{iso}\text{-}\mathbb{P}}$ over $\mathbb{F}_{q_\mathbb{P}}$ with equation $y^2 = x^3 + a_{\mathsf{iso}\text{-}\mathbb{P}} \cdot x + b_{\mathsf{iso}\text{-}\mathbb{P}}$. Since $E_{\mathsf{iso}\text{-}\mathbb{P}}$ is isogenous to $E_\mathbb{P}$, it has the same order $r_{\mathsf{iso}\text{-}\mathbb{P}} = r_\mathbb{P} = q_\mathbb{V}$.

Let $\mathsf{iso}\text{-}\mathbb{V}$ be the group of points $(x, y)$ with zero point $\mathcal{O}_{\mathsf{iso}\text{-}\mathbb{V}}$, on a *short Weierstrass curve* $E_{\mathsf{iso}\text{-}\mathbb{V}}$ over $\mathbb{F}_{q_\mathbb{V}}$ with equation $y^2 = x^3 + a_{\mathsf{iso}\text{-}\mathbb{V}} \cdot x + b_{\mathsf{iso}\text{-}\mathbb{V}}$. Since $E_{\mathsf{iso}\text{-}\mathbb{V}}$ is isogenous to $E_\mathbb{V}$, it has the same order $r_{\mathsf{iso}\text{-}\mathbb{V}} = r_\mathbb{V} = q_\mathbb{P}$.

Let $\mathcal{C}^\mathbb{P} : {\mathbb{F}_{q_\mathbb{P}}}^{[13]} :=$ [

```
  0x0e38e38e38e38e38e38e38e38e38e38e4081775473d8375b775f6034aaaaaaab,
  0x3509afd51872d88e267c7ffa51cf412a0f93b82ee4b994958cf863b02814fb76,
  0x17329b9ec525375398c7d7ac3d98fd13380af066cfeb6d690eb64faef37ea4f7,
  0x1c71c71c71c71c71c71c71c71c71c71c8102eea8e7b06eb6eebec06955555580,
  0x1d572e7ddc099cff5a607fcce0494a799c434ac1c96b6980c47f2ab668bcd71f,
  0x325669becaecd5d11d13bf2a7f22b105b4abf9fb9a1fc81c2aa3af1eae5b6604,
  0x1a12f684bda12f684bda12f684bda12f7642b01ad461bad25ad985b5e38e38e4,
  0x1a84d7ea8c396c47133e3ffd28e7a09507c9dc17725cca4ac67c31d8140a7dbb,
  0x3fb98ff0d2ddcadd303216cce1db9ff11765e924f745937802e2be87d225b234,
  0x025ed097b425ed097b425ed097b425ed0ac03e8e134eb3e493e53ab371c71c4f,
  0x0c02c5bcca0e6b7f0790bfb3506defb65941a3a4a97aa1b35a28279b1d1b42ae,
  0x17033d3c60c68173573b3d7f7d681310d976bbfabbc5661d4d90ab820b12320a,
  0x40000000000000000000000000000000224698fc094cf91b992d30ecffffde5
```

].

Let $\mathcal{C}^\mathbb{V} : {\mathbb{F}_{q_\mathbb{V}}}^{[13]} :=$ [

```
  0x38e38e38e38e38e38e38e38e38e38e390205dd51cfa0961a43cd42c800000001,
  0x1d935247b4473d17acecf10f5f7c09a2216b8861ec72bd5d8b95c6aaf703bcc5,
  0x18760c7f7a9ad20ded7ee4a9cdf78f8fd59d03d23b39cb11aeac67bbeb586a3d,
  0x31c71c71c71c71c71c71c71c71c71c71e1c521a795ac8356fb539a6f0000002b,
  0x0a2de485568125d51454798a5b5c56b2a3ad678129b604d3b7284f7eaf21a2e9,
  0x14735171ee5427780c621de8b91c242a30cd6d53df49d235f169c187d2533465,
  0x12f684bda12f684bda12f684bda12f685601f4709a8adcb36bef1642aaaaaaab,
  0x2ec9a923da239e8bd6767887afbe04d121d910aefb03b31d8bee58e5fb81de63,
  0x19b0d87e16e2578866d1466e9de10e6497a3ca5c24e9ea634986913ab4443034,
  0x1ed097b425ed097b425ed097b425ed098bc32d36fb21a6a38f64842c55555533,
  0x2f44d6c801c1b8bf9e7eb64f890a820c06a767bfc35b5bac58dfecce86b2745e,
  0x3d59f455cafc7668252659ba2b546c7e926847fb9ddd76a1d43d449776f99d2f,
  0x40000000000000000000000000000000224698fc0994a8dd8c46eb20ffffde5
```

].

[8] For a brief introduction to isogenies between elliptic curves, see [Cook2019]. For deeper mathematical background, see the notes for lectures 5, 6, and 7 at [Sutherland2019].

Let $\mathsf{iso\_map}^{\mathbb{G}} \colon \mathsf{iso}\text{-}\mathbb{G} \to \mathbb{G}$ be the isogeny map given by:

$$\begin{aligned}
\mathsf{iso\_map}^{\mathbb{G}}\big(\mathcal{O}_{\mathsf{iso}\text{-}\mathbb{G}}\big) &= \mathcal{O}_{\mathbb{G}} \\
\mathsf{iso\_map}^{\mathbb{G}}\big((x, y)\big) &= \left( \frac{\mathcal{C}_1^{\mathbb{G}} \cdot x^3 + \mathcal{C}_2^{\mathbb{G}} \cdot x^2 + \mathcal{C}_3^{\mathbb{G}} \cdot x + \mathcal{C}_4^{\mathbb{G}}}{x^2 + \mathcal{C}_5^{\mathbb{G}} \cdot x + \mathcal{C}_6^{\mathbb{G}}}, \frac{(\mathcal{C}_7^{\mathbb{G}} \cdot x^3 + \mathcal{C}_8^{\mathbb{G}} \cdot x^2 + \mathcal{C}_9^{\mathbb{G}} \cdot x + \mathcal{C}_{10}^{\mathbb{G}}) \cdot y}{x^3 + \mathcal{C}_{11}^{\mathbb{G}} \cdot x^2 + \mathcal{C}_{12}^{\mathbb{G}} \cdot x + \mathcal{C}_{13}^{\mathbb{G}}} \right).
\end{aligned}$$

Let $\mathsf{BLAKE2b\text{-}512} \colon \mathbb{BY}^{[16]} \times \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{BY}^{[\ell/8]}$ be as defined in § 5.4.1.2 *'BLAKE2 Hash Functions'* on p. 69.

Let BEOS2IP be as defined in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

Define $\mathsf{hash\_to\_field}^{\mathbb{F}_{q_{\mathbb{G}}}^{[2]}}_{\mathsf{XMD:BLAKE2b}}(\mathsf{msg} \colon \mathbb{BY}^{[\mathbb{N}]}, \mathsf{DST} \colon \mathbb{BY}^{[\{0\,..\,255\}]}) \to \mathbb{F}_{q_{\mathbb{G}}}^{[2]}$ as follows:

```
let $\mathsf{DST}' = \mathsf{DST} \,||\, [\,\mathsf{length}(\mathsf{DST})\,]$
let $\mathsf{msg}' = [\mathtt{0x00}]^{64} \,||\, \mathsf{msg} \,||\, [\,0, 128\,] \,||\, [\,0\,] \,||\, \mathsf{DST}'$
let $b_0 = \mathsf{BLAKE2b\text{-}512}([\mathtt{0x00}]^{16}, \mathsf{msg}')$
let $b_1 = \mathsf{BLAKE2b\text{-}512}([\mathtt{0x00}]^{16}, b_0 \,||\, [\,1\,] \,||\, \mathsf{DST}')$
let $b_2 = \mathsf{BLAKE2b\text{-}512}([\mathtt{0x00}]^{16}, (b_0 \oplus b_1) \,||\, [\,2\,] \,||\, \mathsf{DST}')$
return $[\,\mathsf{BEOS2IP}_{512}(b_1) \pmod{q_{\mathbb{G}}}, \mathsf{BEOS2IP}_{512}(b_2) \pmod{q_{\mathbb{G}}}\,]$.
```

**Non-normative notes:**

- This algorithm is intended to correspond to $\mathsf{hash\_to\_field}(\mathsf{msg}, 2)$ defined in [ID-hashtocurve, section 5.3], using as its $\mathsf{expand\_message}$ parameter the function XMD:BLAKE2b corresponding to $\mathsf{expand\_message\_xmd}$ defined in [ID-hashtocurve, section 5.4.1], and with domain separation tag DST. In $\mathsf{expand\_message\_xmd}$, H is instantiated as BLAKE2b-512 with $\mathsf{b\_in\_bytes} = 64$, and we specialize to $\mathsf{len\_in\_bytes} = 128$ since that is the only case we need. In the event of any discrepancy or change to the Internet Draft, the definition here takes precedence.
- Unlike other uses of BLAKE2b in **Zcash**, zero bytes are used for the BLAKE2b personalization, in order to follow the Internet Draft which encodes DST in the hash inputs instead.
- The conversion from bytes to field elements uses big-endian order, again in order to follow the Internet Draft.
- A minor optimization is to cache the state of the BLAKE2b-512 instance used to compute $b_0$ after processing $[\mathtt{0x00}]^{64}$, since this state does not depend on the message.

Let $\lambda_{\mathbb{G}}$ be any fixed nonsquare in $\mathbb{F}_{q_{\mathbb{G}}}$. Define $\mathsf{sqrt\_ratio}_{\mathbb{F}_{q_{\mathbb{G}}}}(\mathsf{num}, \mathsf{div}) \colon \mathbb{F}_{q_{\mathbb{G}}} \times \mathbb{F}^*_{q_{\mathbb{G}}} \to \mathbb{F}_{q_{\mathbb{G}}}$ as follows:

$$\mathsf{sqrt\_ratio}_{\mathbb{F}_{q_{\mathbb{G}}}}(\mathsf{num}, \mathsf{div}) = \begin{cases} \big(\sqrt[?]{\mathsf{num}/\mathsf{div}}, \quad 1\big), & \text{if } \mathsf{num}/\mathsf{div} \text{ is square in } \mathbb{F}_{q_{\mathbb{G}}} \\ \big(\sqrt[?]{\lambda_{\mathbb{G}} \cdot \mathsf{num}/\mathsf{div}}, 0\big), & \text{otherwise.} \end{cases}$$

**Non-normative notes:**

- An arbitrary square root may be chosen in either case of the definition. The result is never $\bot$.
- The computation of $\mathsf{sqrt\_ratio}_{\mathbb{F}_{q_{\mathbb{G}}}}$ can be optimized as described in TODO: .

Define $Z_{\mathsf{iso}\text{-}\mathbb{G}} := -13 \pmod{q_{\mathbb{G}}}$. (This value is suitable for both iso-Pallas and iso-Vesta.)

Precompute $\theta_{\mathsf{iso}\text{-}\mathbb{G}} := \sqrt[?]{Z_{\mathsf{iso}\text{-}\mathbb{G}}/\lambda_{\mathbb{G}}}$, which is not $\bot$.[9]

Precompute $b_{\mathsf{iso}\text{-}\mathbb{G}}/(Z_{\mathsf{iso}\text{-}\mathbb{G}} \cdot a_{\mathsf{iso}\text{-}\mathbb{G}})$.

By definition we have that $E_{\mathbb{G}}$ is the *short Weierstrass curve* with equation $y^2 = x^3 + b_{\mathbb{G}}$, and $E_{\mathsf{iso}\text{-}\mathbb{G}}$ is the *short Weierstrass curve* with equation $y^2 = x^3 + a_{\mathsf{iso}\text{-}\mathbb{G}} \cdot x + b_{\mathsf{iso}\text{-}\mathbb{G}}$.

[9] Both $Z_{\mathsf{iso}\text{-}\mathbb{G}}$ and $\lambda_{\mathbb{G}}$ are nonsquare, and so their ratio is square in $\mathbb{F}_{q_{\mathbb{G}}}$. An arbitrary square root may be chosen.

Define $\mathsf{map\_to\_curve\_simple\_swu}^{\mathsf{iso}\text{-}\mathbb{G}}(u \,⦂\, \mathbb{F}_{q_{\mathbb{G}}}) \to \mathsf{iso}\text{-}\mathbb{G}$ as follows:

```
    let $\mathsf{Zuu} = Z_{\mathsf{iso}\text{-}\mathbb{G}} \cdot u^2$
    let $\mathsf{ta} = \mathsf{Zuu}^2 + \mathsf{Zuu}$
    let $\mathsf{x1}_{\mathsf{num}} = b_{\mathsf{iso}\text{-}\mathbb{G}} \cdot (\mathsf{ta} + 1)$
    let $\mathsf{x}_{\mathsf{div}} = a_{\mathsf{iso}\text{-}\mathbb{G}} \cdot \big((\mathsf{ta} = 0) \;?\; Z_{\mathsf{iso}\text{-}\mathbb{G}} : -\mathsf{ta}\big)$
    compute $\mathsf{x}_{\mathsf{div}}^2$ and $\mathsf{x}_{\mathsf{div}}^3$
    let $\mathsf{U} = (\mathsf{x1}_{\mathsf{num}}^2 + a_{\mathsf{iso}\text{-}\mathbb{G}} \cdot \mathsf{x}_{\mathsf{div}}^2) \cdot \mathsf{x1}_{\mathsf{num}} + b_{\mathsf{iso}\text{-}\mathbb{G}} \cdot \mathsf{x}_{\mathsf{div}}^3$
    let $\mathsf{x2}_{\mathsf{num}} = \mathsf{Zuu} \cdot \mathsf{x1}_{\mathsf{num}}$
    let $(\mathsf{y1},\, \mathsf{is\_gx1\_square}) = \mathsf{sqrt\_ratio}_{\mathbb{F}_{q_{\mathbb{G}}}}(\mathsf{U},\, \mathsf{x}_{\mathsf{div}}^3)$
    let $\mathsf{y2} = \theta_{\mathsf{iso}\text{-}\mathbb{G}} \cdot \mathsf{Zuu} \cdot u \cdot \mathsf{y1}$
    let $\mathsf{x}_{\mathsf{num}} = \mathsf{is\_gx1\_square} \;?\; \mathsf{x1}_{\mathsf{num}} : \mathsf{x2}_{\mathsf{num}}$
    let $\mathsf{y}' = \mathsf{is\_gx1\_square} \;?\; \mathsf{y1} : \mathsf{y2}$
    let $\mathsf{y} = (u \bmod 2 = y \bmod 2) \;?\; \mathsf{y}' : -\mathsf{y}'$
    return the $E_{\mathsf{iso}\text{-}\mathbb{G}}$ point with affine coordinates $(\mathsf{x}_{\mathsf{num}}/\mathsf{x}_{\mathsf{div}},\, \mathsf{y})$.
```

Let $\mathsf{GroupHash}^{\mathbb{G}}.\mathsf{Input} := \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{BY}^{[\mathbb{N}]}$. The first input element acts as a domain separator to distinguish uses of the *group hash* for different purposes; the second input element is the message.

This hash-to-curve algorithm does not have a URS, i.e. $\mathsf{GroupHash}^{\mathbb{G}}.\mathsf{URSType} := ()$.

The hash $\mathsf{GroupHash}^{\mathbb{G}}(D \,⦂\, \mathbb{BY}^{[\mathbb{N}]},\, M \,⦂\, \mathbb{BY}^{[\mathbb{N}]}) \,⦂\, \mathbb{G}$ is calculated as follows:

```
    let $\mathsf{DST} = D \,||\, \text{“}\texttt{-}\text{”} \,||\, \mathsf{curveName}_{\mathbb{G}} \,||\, \text{“}\texttt{\_XMD:BLAKE2b\_SSWU\_RO\_}\text{”}$
    let $[\, u_0, u_1 \,] = \mathsf{hash\_to\_field}^{\mathbb{F}_{q_{\mathbb{G}}}^{[2]}}_{\mathsf{XMD:BLAKE2b}}(\mathsf{msg}, \mathsf{DST})$
    let $Q_i = \mathsf{map\_to\_curve\_simple\_swu}^{\mathsf{iso}\text{-}\mathbb{G}}(u_i)$ for $i \in \{0, 1\}$
    return $\mathsf{iso\_map}^{\mathbb{G}}(Q_0 + Q_1)$.
```

**Non-normative notes:**

- $\mathsf{GroupHash}^{\mathbb{P}}$ and $\mathsf{GroupHash}^{\mathbb{V}}$ are intended to be instantiations of `hash_to_curve` using "Simplified SWU for $AB = 0$" described in [ID-hashtocurve, section 6.6.3]. In the event of any discrepancy or change to the Internet Draft, the definition here takes precedence.
- It is not necessary to use the $\mathsf{clear\_cofactor}$ function specified in the Internet Draft, because Pallas and Vesta (and therefore iso-Pallas and iso-Vesta) are prime-order.
- The above description incorporates optimizations from [WB2019] that avoid inversions and unnecessary square tests in the computation of $\mathsf{map\_to\_curve\_simple\_swu}^{\mathsf{iso}\text{-}\mathbb{G}}$. In order to fully avoid inversions, the output of $\mathsf{map\_to\_curve\_simple\_swu}^{\mathsf{iso}\text{-}\mathbb{G}}$ can be expressed in Jacobian coordinates, as can the input and output of $\mathsf{iso\_map}^{\mathbb{G}}$. It is outside the scope of this document to describe Jacobian coordinates, but for example, the $E_{\mathsf{iso}\text{-}\mathbb{G}}$ point with affine coordinates $(\mathsf{x}_{\mathsf{num}}/\mathsf{x}_{\mathsf{div}}, \mathsf{y})$, has Jacobian coordinates $(\mathsf{x}_{\mathsf{num}} \cdot \mathsf{x}_{\mathsf{div}} : \mathsf{y} \cdot \mathsf{x}_{\mathsf{div}}^3 : \mathsf{x}_{\mathsf{div}})$.

**Note:** The uses of $\mathsf{GroupHash}^{\mathbb{P}}$ for $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$, and of both $\mathsf{GroupHash}^{\mathbb{P}}$ and $\mathsf{GroupHash}^{\mathbb{V}}$ to generate independent bases, need a *random oracle*. The $\mathsf{hash\_to\_curve}$ algorithm in [ID-hashtocurve] is designed to be indifferentiable from a *random oracle* (in the framework of [MRH2003]), given that XMD:BLAKE2b satisfies the requirements of [ID-hashtocurve, section 5.5.4]. The security of the Brier et al. construction on which this algorithm is based is analysed in [FFSTV2013] and [KT2015], with a verified proof in [BGHOZ2013].

### 5.4.10 Zero-Knowledge Proving Systems

#### 5.4.10.1 BCTV14

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

A BCTV14 proof comprises $(\pi_A : \mathbb{G}_1^{(r)*}, \pi'_A : \mathbb{G}_1^{(r)*}, \pi_B : \mathbb{G}_2^{(r)*}, \pi'_B : \mathbb{G}_1^{(r)*}, \pi_C : \mathbb{G}_1^{(r)*}, \pi'_C : \mathbb{G}_1^{(r)*}, \pi_K : \mathbb{G}_1^{(r)*}, \pi_H : \mathbb{G}_1^{(r)*})$. It is computed as described in [BCTV2014a, Appendix B], using the pairing parameters specified in § 5.4.9.1 'BN-254' on p. 91.

**Note:** Many details of the *proving system* are beyond the scope of this protocol document. For example, the *quadratic constraint program* verifying the *JoinSplit statement*, or its translation to a *Quadratic Arithmetic Program* [BCTV2014a, section 2.3], are not specified in this document. In 2015, Bryan Parno found a bug in this translation, which is corrected by the *libsnark* implementation[10] [WCBTV2015] [Parno2015] [BCTV2014a, Remark 2.5]. 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 § 5.7 'BCTV14 *zk-SNARK Parameters*' on p. 111, together with a *proving system* implementation that is interoperable with the **Zcash** fork of *libsnark*, to ensure compatibility.

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

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

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

[**Sapling** onward] An implementation of **Zcash** that checkpoints on a *block* after **Sapling MAY** choose to skip verification of BCTV14 proofs. In this case, the implementation **MUST** only accept *blocks* that are descendants of the known **Sapling** *activation block* on the appropriate *network*.

#### Encoding of BCTV14 Proofs

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 264-bit $\pi_A$ | 264-bit $\pi'_A$ | 520-bit $\pi_B$ | 264-bit $\pi'_B$ | 264-bit $\pi_C$ | 264-bit $\pi'_C$ | 264-bit $\pi_K$ | 264-bit $\pi_H$ |

The resulting proof size is 296 bytes.

In addition to the steps to verify a proof given in [BCTV2014a, Appendix B], the verifier **MUST** check, for the encoding of each element, that:

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

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

#### 5.4.10.2 Groth16

After **Sapling** activation, **Zcash** uses *zk-SNARKs* with the Groth16 *proving system* described in [BGM2017], which is a modification of the system in [Groth2016]. An independent security proof of this system and its setup is given in [Maller2018].

Groth16 *zk-SNARK proofs* are used in *transaction version* 4 and later (§ 7.1 *'Transaction Encoding and Consensus'* on p. 114), both in **Sprout** *JoinSplit descriptions* and in **Sapling** *Spend descriptions* and *Output descriptions*. They are generated by the *bellman* library [Bowe-bellman].

A Groth16 proof comprises $(\pi_A ⦂ \mathbb{S}_1^{(r)*},\ \pi_B ⦂ \mathbb{S}_2^{(r)*},\ \pi_C ⦂ \mathbb{S}_1^{(r)*})$. It is computed as described in [Groth2016, section 3.2], using the pairing parameters specified in § 5.4.9.2 'BLS12-381' on p. 93. The proof elements are in a different order to the presentation in [Groth2016].

**Note:** The *quadratic constraint programs* verifying the *Spend statement* and *Output statement* are described in Appendix § A *'Circuit Design'* on p. 173. However, many other details of the *proving system* are beyond the scope of this protocol document. For example, certain details of the translations of the *Spend statement* and *Output statement* to *Quadratic Arithmetic Programs* are not specified in this document. In practice it will be necessary to use the specific proving and verifying keys generated for the **Zcash** production *block chain* (see § 5.8 'Groth16 *zk-SNARK Parameters*' on p. 112), and a *proving system* implementation that is interoperable with the *bellman* library used by **Zcash**, to ensure compatibility.

#### Encoding of Groth16 Proofs

A Groth16 proof is encoded by concatenating the encodings of its elements; for the BLS12-381 pairing this is:

| | | |
|---|---|---|
| 384-bit $\pi_A$ | 768-bit $\pi_B$ | 384-bit $\pi_C$ |

The resulting proof size is 192 bytes.

In addition to the steps to verify a proof given in [Groth2016], the verifier **MUST** check, for the encoding of each element, that:

- the leading bitfield is of the required form;
- the remaining bits encode a big-endian representation of an integer in $\{0\,..\,q_{\mathbb{S}}-1\}$ or (in the case of $\pi_B$) two integers in that range;
- the encoding represents a point in $\mathbb{S}_1^{(r)*}$ or (in the case of $\pi_B$) $\mathbb{S}_2^{(r)*}$, including checking that it is of order $r_{\mathbb{S}}$ in each case.

#### 5.4.10.3 Halo 2

For **Orchard** *Action descriptions* in version 5 *transactions*, **Zcash** uses *zk-SNARKs* with the Halo 2 *proving system* described in TODO: .

#### Encoding of Halo 2 Proofs

Halo 2 proofs are defined as byte sequences, and so the encoding is the proof itself.

## 5.5 Encodings of Note Plaintexts and Memo Fields

As explained in § 3.2.1 *'Note Plaintexts and Memo Fields'* on p. 15, 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:

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

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

$$(\mathsf{leadByte} ⦂ \mathbb{B}^{\mathbb{Y}}, \mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0 \,..\, 2^{\ell_{\mathsf{value}}}-1\}, \mathsf{rseed} ⦂ \mathbb{B}^{\mathbb{Y}[32]}, \mathsf{memo} ⦂ \mathbb{B}^{\mathbb{Y}[512]})$$

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 § 3.2 *'Notes'* on p. 13.

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.

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

| | | | | |
|---|---|---|---|---|
| 8-bit leadByte | 88-bit d | 64-bit v | 256-bit rseed | memo (512 bytes) |

- A byte indicating this version of the encoding of a **Sapling** *note plaintext*. This will be 0x01 before activation of the **Canopy** *network upgrade*, and 0x02 afterward.
- 11 bytes specifying d.
- 8 bytes specifying v.
- 32 bytes specifying rseed.
- 512 bytes specifying memo.

## 5.6 Encodings of Addresses and Keys

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

Addresses and keys can be encoded as a byte sequence; this is called the *raw encoding*. 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].

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

**Non-normative note:** ZIP 173 is similar to **Bitcoin**'s BIP 173, except for dropping the limit of 90 characters on an encoded *Bech32* string (which does not hold for **Sapling** viewing keys, for example), and requirements specific to Bitcoin's Segwit addresses.

**Orchard** introduces a new address format called a *unified payment address*. This can encode an **Orchard** address, but also a **Sapling** address, a *transparent address*, and potentially future address formats, all in the same *unified payment address*. It is **RECOMMENDED** to use *unified payment addresses* for all new applications, unless compatibility with software that only accepts previous address formats is required.

A *unified payment address* is encoded with *Bech32m* [BIP-350] rather than *Bech32*.

*Payment addresses* **MAY** be encoded as QR codes; in this case, the **RECOMMENDED** format for a **Sapling** or unified *payment address* is to convert the *Bech32* or *Bech32m* form to uppercase and use the Alphanumeric mode [ISO2015, sections 7.3.4 and 7.4.4].

## 5.6.1 Transparent Encodings

### 5.6.1.1 Transparent Addresses

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

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

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

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

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

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

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

**Notes:**

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

### 5.6.1.2 Transparent Private Keys

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

### 5.6.2 Sprout Encodings

#### 5.6.2.1 Sprout Payment Addresses

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81.

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

$\mathsf{a_{pk}}$ is a SHA256Compress output. $\mathsf{pk_{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ key, for use with the encryption scheme defined in § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59. These components are derived from a *spending key* as described in § 4.2.1 ***'Sprout Key Components'*** on p. 32.

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

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

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

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

#### 5.6.2.2 Sprout Incoming Viewing Keys

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as defined in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81.

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

$\mathsf{a_{pk}}$ is a SHA256Compress output. $\mathsf{sk_{enc}}$ is a $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$ key, for use with the encryption scheme defined in § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59. These components are derived from a *spending key* as described in § 4.2.1 ***'Sprout Key Components'*** on p. 32.

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

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

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

$\mathsf{sk_{enc}}$ **MUST** be "clamped" using $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{FormatPrivate}$ as specified in § 4.2.1 ***'Sprout Key Components'*** on p. 32. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{sk_{enc}} \neq \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{FormatPrivate}(\mathsf{sk_{enc}})$.

$\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{FormatPrivate}$ is defined in § 5.4.5.1 ***'Sprout Key Agreement'*** on p. 81.

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

#### 5.6.2.3 Sprout Spending Keys

A **Sprout** *spending key* consists of $\mathsf{a_{sk}}$, which is a sequence of 252 bits (see § 4.2.1 ***'Sprout Key Components'*** on p. 32).

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

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

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

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

**Notes:**

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

### 5.6.3 Sapling Encodings

#### 5.6.3.1 Sapling Payment Addresses

Let $\mathsf{KA^{Sapling}}$ be as defined in § 5.4.5.3 ***'Sapling Key Agreement'*** on p. 82.

Let $\ell_{\mathsf{d}}$ be as defined in § 5.3 ***'Constants'*** on p. 67.

Let $\mathbb{J}^{(r)}$, $\mathsf{abst}_{\mathbb{J}}$, and $\mathsf{repr}_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let $\mathsf{LEBS2OSP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{B}^{\mathbb{Y}[\mathsf{ceiling}(\ell/8)]}$ be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

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

$\mathsf{pk_d}$ is an encoding of a $\mathsf{KA^{Sapling}}$ *public key* of type $\mathsf{KA^{Sapling}.PublicPrimeSubgroup}$, for use with the encryption scheme defined in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60. $\mathsf{d}$ is a *diversifier*. These components are derived as described in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

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

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

- 11 bytes specifying $\mathsf{d}$.
- 32 bytes specifying the *ctEdwards compressed encoding* of $\mathsf{pk_d}$ (see § 5.4.9.3 'Jubjub' on p. 94).

When decoding the representation of $\mathsf{pk_d}$, the address **MUST** be considered invalid if $\mathsf{abst}_{\mathbb{J}}$ returns $\bot$.

[ZIP-216] specifies that the address **MUST** also be considered invalid if the resulting $\mathsf{pk_d}$ is not in the prime-order subgroup $\mathbb{J}^{(r)}$, or if it is a non-canonical encoding as defined in § 4.1.9 ***'Represented Group'*** on p. 29. This **MAY** be enforced in advance of activation of **NU5**.

For addresses on *Mainnet*, the *Human-Readable Part* (as defined in [ZIP-173]) is **"`zs`"**. For addresses on *Testnet*, the *Human-Readable Part* is **"`ztestsapling`"**.

### 5.6.3.2 Sapling Incoming Viewing Keys

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in § 5.4.5.3 ***'Sapling Key Agreement'*** on p. 82.

Let $\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}$ be as defined in § 5.3 *'Constants'* on p. 67.

A **Sapling** *incoming viewing key* consists of $\mathsf{ivk} \mathrel{⦂} \{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}} - 1\}$.

$\mathsf{ivk}$ is a $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private}$ key (restricted to $\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}$ bits), derived as described in § 4.2.2 ***'Sapling Key Components'*** on p. 32. It is used with the encryption scheme defined in § 4.19 *'In-band secret distribution (**Sapling** and **Orchard**)'* on p. 60.

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

| |
|---|
| 256-bit ivk |

- 32 bytes (little-endian) specifying $\mathsf{ivk}$, padded with zeros in the most significant bits.

$\mathsf{ivk}$ **MUST** be in the range $\{0 \,..\, 2^{\ell^{\mathsf{Sapling}}_{\mathsf{ivk}}} - 1\}$ as specified in § 4.2.2 ***'Sapling Key Components'*** on p. 32. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{ivk}$ is not in this range.

For *incoming viewing keys* on *Mainnet*, the *Human-Readable Part* is **"`zivks`"**. For *incoming viewing keys* on *Testnet*, the *Human-Readable Part* is **"`zivktestsapling`"**.

### 5.6.3.3 Sapling Full Viewing Keys

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in § 5.4.5.3 ***'Sapling Key Agreement'*** on p. 82.

A **Sapling** *full viewing key* consists of $\mathsf{ak} \mathrel{⦂} \mathbb{J}^{(r)*}$, $\mathsf{nk} \mathrel{⦂} \mathbb{J}^{(r)}$, and $\mathsf{ovk} \mathrel{⦂} \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]}$.

$\mathsf{ak}$ and $\mathsf{nk}$ are points on the Jubjub curve (see § 5.4.9.3 'Jubjub' on p. 94). They are derived as described in § 4.2.2 ***'Sapling Key Components'*** on p. 32.

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

| | | |
|---|---|---|
| $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})\big)$ | $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big)$ | 32-byte ovk |

- 32 bytes specifying the *ctEdwards compressed encoding* of $\mathsf{ak}$ (see § 5.4.9.3 'Jubjub' on p. 94).
- 32 bytes specifying the *ctEdwards compressed encoding* of $\mathsf{nk}$.
- 32 bytes specifying the *outgoing viewing key* $\mathsf{ovk}$.

When decoding this representation, the key **MUST** be considered invalid if $\mathsf{abst}_{\mathbb{J}}$ returns $\bot$ for either $\mathsf{ak}$ or $\mathsf{nk}$, or if $\mathsf{ak} \notin \mathbb{J}^{(r)*}$, or if $\mathsf{nk} \notin \mathbb{J}^{(r)}$.

For *incoming viewing keys* on *Mainnet*, the *Human-Readable Part* is **"`zviews`"**. For *incoming viewing keys* on *Testnet*, the *Human-Readable Part* is **"`zviewtestsapling`"**.

### 5.6.3.4 Sapling Spending Keys

A **Sapling** *spending key* consists of $\mathsf{sk} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$ (see § 4.2.2 ***'Sapling Key Components'*** on p. 32).

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

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

- 32 bytes specifying $\mathsf{sk}$.

For *spending keys* on *Mainnet*, the *Human-Readable Part* is **"`secret-spending-key-main`"**. For *spending keys* on *Testnet*, the *Human-Readable Part* is **"`secret-spending-key-test`"**.

### 5.6.4 Unified and Orchard Encodings

#### 5.6.4.1 Unified Payment Addresses

Rather than defining a *Bech32* string encoding of **Orchard** *shielded payment addresses*, we instead define a *unified payment address* format that is able to encode a set of *payment addresses* of different types. This enables the consumer of an address to choose the best address type it supports, providing a better user experience as new formats are added in the future.

Assume that we are given a set of one or more *raw encodings* of *payment addresses* of distinct types. That is, the set may optionally contain one of each of the *payment address* types in the following list:

- typecode `0x03` – § 5.6.4.2 ***'Orchard Raw Payment Addresses'*** on p. 110;
- typecode `0x02` – § 5.6.3.1 ***'Sapling Payment Addresses'*** on p. 107;
- typecode `0x01` – *transparent* P2SH address, ***or*** typecode `0x00` – *transparent* P2PKH address.

The intended semantics is that the consumer of a *unified payment address* **SHOULD** take the "best" address type that it supports from the set, i.e. the first in the above list. For example, if the *unified payment address* includes an **Orchard** address, and the consumer supports sending funds to **Orchard** addresses, and no more recent address format has been defined at the time of use, then the **Orchard** address **SHOULD** be used.

The raw encoding of a *unified payment address* is a concatenation of (`typecode`, `length`, `addr`) encodings of the consituent addresses:

- `typecode` : $\mathbb{B}^{\mathbb{Y}}$ – the typecode from the above list;
- `length` : $\mathbb{B}^{\mathbb{Y}}$ – the length in bytes of `addr`;
- `addr` : $\mathbb{B}^{\mathbb{Y}[\mathtt{length}]}$ – the raw encoding of a *shielded payment address*, or the 160-bit script hash of a P2SH address [Bitcoin-P2SH], or the 160-bit *validating key* hash of a P2PKH address [Bitcoin-P2PKH].

The result of the concatenation is then encoded with *Bech32m* [BIP-350], ignoring any length restrictions. This is chosen over *Bech32* in order to better handle variable-length inputs.

For *unified payment addresses* on *Mainnet*, the *Human-Readable Part* (as defined in [ZIP-173]) is **"`u`"**. For *unified payment addresses* on *Testnet*, the *Human-Readable Part* is **"`utest`"**.

**Notes:**

- The `length` field is always encoded as a single byte, ***not*** as a `compactSize`.
- For *transparent addresses*, the `addr` field does not include the first two bytes of a *raw encoding*.
- There is intentionally no typecode defined for a **Sprout** *shielded payment address*. Since it is no longer possible (since activation of [ZIP-211] in the **Canopy** *network upgrade*) to send funds into the **Sprout** *chain value pool*, this would not be generally useful.
- Consumers **MUST** ignore constituent addresses with typecodes they do not recognize.

- Consumers **MUST** reject *unified payment addresses* in which the same typecode appears more than once, or that include both P2SH and P2PKH *transparent addresses*.
- Producers **SHOULD** order the constituent addresses in the same order as the list of address types above. However, consumers **MUST NOT** assume this ordering, and it does not affect which address should be used by a consumer.
- There **MUST NOT** be additional bytes at the end of the encoding that cannot be interpreted as specified above.

### 5.6.4.2 Orchard Raw Payment Addresses

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as defined in § 5.4.5.5 ***'Orchard Key Agreement'*** on p. 82.

An **Orchard** *shielded payment address* consists of $\mathsf{d} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{d}}]}$ and $\mathsf{pk_d} \mathrel{⦂} \mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public}$.

$\mathsf{pk_d}$ is an encoding of a $\mathsf{KA}^{\mathsf{Orchard}}$ *public key* of type $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Public}$, for use with the encryption scheme defined in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60. $\mathsf{d}$ is a sequence of 11 bytes. These components are derived as described in § 4.2.3 ***'Orchard Key Components'*** on p. 34.

The *raw encoding* of an **Orchard** *shielded payment address* consists of:

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

- 11 bytes specifying $\mathsf{d}$.
- 32 bytes specifying the *short Weierstrass compressed encoding* of $\mathsf{pk_d}$ (see § 5.4.9.6 'Pallas *and* Vesta' on p. 97).

When decoding the representation of $\mathsf{pk_d}$, the address **MUST** be considered invalid if $\mathsf{abst}_{\mathbb{P}}$ returns $\bot$.

There is no *Bech32* encoding defined for an individual **Orchard** *shielded payment address*; instead use a *unified payment address* as defined in § 5.6.4.1 ***'Unified Payment Addresses'*** on p. 109.

### 5.6.4.3 Orchard Incoming Viewing Keys

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as defined in § 5.4.5.5 ***'Orchard Key Agreement'*** on p. 82.

An **Orchard** *incoming viewing key* consists of a $\mathsf{KA}^{\mathsf{Orchard}}.\mathsf{Private}$ key $\mathsf{ivk}$, restricted to the range $\{0\,..\,q_{\mathbb{P}}-1\}$. It is derived as described in § 4.2.3 ***'Orchard Key Components'*** on p. 34, and is used with the encryption scheme defined in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60.

Let $\mathsf{I2LEOSP}$ be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

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

| |
|---|
| $\mathsf{I2LEOSP}_{256}(\mathsf{ivk})$ |

- 32 bytes (little-endian) specifying $\mathsf{ivk}$.

$\mathsf{ivk}$ **MUST** be in the range $\{0\,..\,q_{\mathbb{P}}-1\}$ as specified in § 4.2.3 ***'Orchard Key Components'*** on p. 34. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{ivk}$ is not in this range.

For *incoming viewing keys* on *Mainnet*, the *Human-Readable Part* is **"`zivko`"**. For *incoming viewing keys* on *Testnet*, the *Human-Readable Part* is **"`zivktestorchard`"**.

#### 5.6.4.4 Orchard Full Viewing Keys

Let $\mathsf{KA}^{\mathsf{Orchard}}$ be as defined in § 5.4.5.5 *'Orchard Key Agreement'* on p. 82.

Let $\mathsf{Extract}_{\mathbb{P}}$ be as defined in § 5.4.9.7 *'Coordinate Extractor for* Pallas' on p. 98.

An **Orchard** *full viewing key* consists of $\mathsf{ak} ⦂ \mathbb{P}_x$, $\mathsf{nk} ⦂ \mathbb{F}_{q_{\mathbb{P}}}$, and $\mathsf{rivk} ⦂ \mathbb{F}_{r_{\mathbb{P}}}$.

$\mathsf{ak}$ is the *Spend validating key*, a result of applying $\mathsf{Extract}_{\mathbb{P}}$ to a point on the Pallas curve (see § 5.4.9.6 'Pallas *and* Vesta' on p. 97). $\mathsf{nk}$ is the *nullifier deriving key*, a field element in $\mathbb{F}_{q_{\mathbb{P}}}$. $\mathsf{rivk}$ is the $\mathsf{Commit}^{\mathsf{ivk}}$ randomness, a field element in $\mathbb{F}_{r_{\mathbb{P}}}$. They are derived as described in § 4.2.3 *'Orchard Key Components'* on p. 34.

Let $\mathsf{I2LEOSP}$ be as defined in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

The *raw encoding* of an **Orchard** *full viewing key* consists of:

| | | |
|---|---|---|
| $\mathsf{I2LEOSP}_{256}(\mathsf{ak})$ | $\mathsf{I2LEOSP}_{256}(\mathsf{nk})$ | $\mathsf{I2LEOSP}_{256}(\mathsf{rivk})$ |

- 32 bytes (little-endian) specifying $\mathsf{ak}$.
- 32 bytes (little-endian) specifying $\mathsf{nk}$.
- 32 bytes (little-endian) specifying $\mathsf{rivk}$.

When decoding this representation, the key **MUST** be considered invalid if $\mathsf{ak}$, $\mathsf{nk}$, or $\mathsf{rivk}$ are not canonically encoded elements of their respective fields, or if $\mathsf{ak} \notin \mathbb{P}_x$.

For *incoming viewing keys* on *Mainnet*, the *Human-Readable Part* is "**`zviewo`**". For *incoming viewing keys* on *Testnet*, the *Human-Readable Part* is "**`zviewtestorchard`**".

#### 5.6.4.5 Orchard Spending Keys

An **Orchard** *spending key* consists of $\mathsf{sk} ⦂ \mathbb{B}^{[\ell_{\mathsf{sk}}]}$ (see § 4.2.3 *'Orchard Key Components'* on p. 34).

The *raw encoding* of an **Orchard** *spending key* consists of:

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

- 32 bytes specifying $\mathsf{sk}$.

For *spending keys* on *Mainnet*, the *Human-Readable Part* is "**`secret-orchard-sk-main`**". For *spending keys* on *Testnet*, the *Human-Readable Part* is "**`secret-orchard-sk-test`**".

### 5.7 BCTV14 zk-SNARK Parameters

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

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

These parameters were obtained by a multi-party computation described in [BGG-mpc] and [BGG2017]. They are used only before **Sapling** activation. Due to the security vulnerability described in § 5.4.10.1 'BCTV14' on p. 102, 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.

## 5.8 Groth16 **zk-SNARK Parameters**

*bellman* [Bowe-bellman] encodes the *proving key* and *verifying key* for a *zk-SNARK circuit* in a single parameters file. The BLAKE2b-512 hashes of this file for the **Sapling** *Spend circuit* and *Output circuit*, and for the implementation of the **Sprout** *JoinSplit circuit* used after **Sapling** activation, are respectively:

```
8270785a1a0d0bc77196f000ee6d221c9c9894f55307bd9357c3f0105d31ca63
991ab91324160d8f53e2bbd3c2633a6eb8bdf5205d822e7f3f73edac51b2b70c sapling-spend.params
657e3d38dbb5cb5e7dd2970e8b03d69b4787dd907285b5a7f0790dcc8072f60b
f593b32cc2d1c030e00ff5ae64bf84c5c3beb84ddc841d48264b4a171744d028 sapling-output.params
e9b238411bd6c0ec4791e9d04245ec350c9c5744f5610dfcce4365d5ca49dfef
d5054e371842b3f88fa1b9d7e8e075249b3ebabd167fa8b0f3161292d36c180a sprout-groth16.params
```

These parameters were obtained by a multi-party computation described in [BGM2017].

## 5.9 Randomness Beacon

Let URS := "`096b36a5804bfacef1691e173c366a47ff5ba84a44f26ddd7e8d9f79d5b42df0`".

This value is used in the definition of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ in § 5.4.9.5 *'Group Hash into* Jubjub' on p. 96, and in the multi-party computation to obtain the **Sapling** parameters given in § 5.8 'Groth16 ***zk-SNARK Parameters***' on p. 112.

It is derived as described in [Bowe2018]:

- Take the hash of the **Bitcoin** *block* at height 514200 in *RPC byte order*, i.e. the big-endian 32-byte representation of `0x00000000000000000034b33e842ac1c50456abe5fa92b60f6b3dfc5d247f7b58`.
- Apply SHA-256 $2^{42}$ times.
- Convert to a US-ASCII lowercase hexadecimal string.

**Note:** URS is a 64-byte US-ASCII string, i.e. the first byte is `0x30`, not `0x09`.

# 6 Network Upgrades

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

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

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

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

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

The specifications of the **Sapling** upgrade are described in this document, [ZIP-205], and [ZIP-243].

The specifications of the **Blossom** upgrade are described in this document, [ZIP-206], and [ZIP-208].

The specifications of the **Heartwood** upgrade are described in this document, [ZIP-250], [ZIP-213], and [ZIP-221].

The specifications of the **Canopy** upgrade are described in this document, [ZIP-251], [ZIP-207], [ZIP-211], [ZIP-212], [ZIP-214], and [ZIP-215].

The specifications of the **NU5** upgrade will be described in this document, [ZIP-216], [ZIP-224], [ZIP-225], and [ZIP-244]. The contents of this *network upgrade* are subject to change as a result of discussions in the Network Upgrade Process and the results of audits.

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. See [ZIP-201] for how this applies to the **Overwinter** upgrade, for example.

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

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

# 7 Consensus Changes from Bitcoin

## 7.1 Transaction Encoding and Consensus

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

| Version* | Bytes | Name | Data Type | Description |
|---|---|---|---|---|
| 1 .. 4 | 4 | `header` | `uint32` | Contains:<br>· `fOverwintered` flag (bit 31)<br>· `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* in the range $\{1 .. 499999999\}$ 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 § 7.3 *'Spend Description Encoding and Consensus'* on p. 119. |
| 4 | *Varies* | `nOutputsSapling` | `compactSize` | The number of *Output descriptions* in `vOutputsSapling`. |
| 4 | 948· `nOutputsSapling` | `vOutputsSapling` | `OutputDescriptionV4 [nOutputsSapling]` | A sequence of *Output descriptions*, encoded per § 7.4 *'Output Description Encoding and Consensus'* on p. 120. |
| 2 .. 4 | *Varies* | `nJoinSplit` | `compactSize` | The number of *JoinSplit descriptions* in `vJoinSplit`. |
| 2 .. 3 | 1802· `nJoinSplit` | `vJoinSplit` | `JSDescriptionBCTV14 [nJoinSplit]` | A sequence of *JoinSplit descriptions* using BCTV14 proofs, encoded per § 7.2 *'JoinSplit Description Encoding and Consensus'* on p. 119. |
| 4 | 1698· `nJoinSplit` | `vJoinSplit` | `JSDescriptionGroth16 [nJoinSplit]` | A sequence of *JoinSplit descriptions* using Groth16 proofs, encoded per § 7.2 *'JoinSplit Description Encoding and Consensus'* on p. 119. |
| 2 .. 4 † | 32 | `joinSplitPubKey` | `byte[32]` | An encoding of a JoinSplitSig public *validating key*. |
| 2 .. 4 † | 64 | `joinSplitSig` | `byte[64]` | A signature on a prefix of the *transaction* encoding, validated using `joinSplitPubKey` as specified in § 4.11 *'Non-malleability (**Sprout**)'* on p. 46. |
| 4 ‡ | 64 | `bindingSigSapling` | `byte[64]` | A *Sapling binding signature* on the *SIGHASH transaction hash*, validated as specified in § 5.4.7.2 *'Binding Signature (**Sapling** and **Orchard**)'* on p. 88. |

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

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

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

Note that the `valueBalanceSapling` field is always present for these *transaction versions*.

Several **Sapling** fields have been renamed from previous versions of this specification:
`valueBalance` → `valueBalanceSapling`; `nShieldedSpend` → `nSpendsSapling`; `vShieldedSpend` → `vSpendsSapling`; `nShieldedOutput` → `nOutputsSapling`; `vShieldedOutput` → `vOutputsSapling`; `bindingSig` → `bindingSigSapling`.

The **Zcash** *transaction* format for *transaction version* 5 is as follows (this should be read in the context of consensus rules later in the section):

| Note | Bytes | Name | Data Type | Description |
|---|---|---|---|---|
| | 4 | header | uint32 | Contains:<br>· fOverwintered flag (bit 31, always set)<br>· version (bits 30 .. 0) – *transaction version*. |
| | 4 | nVersionGroupId | uint32 | Version group ID (nonzero). |
| | 4 | lock_time | uint32 | Unix-epoch UTC time or *block height*, encoded as in **Bitcoin**. |
| | 4 | nExpiryHeight | uint32 | A *block height* in the range $\{1 \,..\, 499999999\}$ after which the *transaction* will expire, or 0 to disable expiry. [ZIP-203] |
| | *Varies* | tx_in_count | compactSize | Number of *transparent* inputs. |
| | *Varies* | tx_in | tx_in | *Transparent* inputs, encoded as in **Bitcoin**. |
| | *Varies* | tx_out_count | compactSize | Number of *transparent* outputs. |
| | *Varies* | tx_out | tx_out | *Transparent* outputs, encoded as in **Bitcoin**. |
| | *Varies* | nSpendsSapling | compactSize | The number of *Spend descriptions* in vSpendsSapling. |
| | 362· nSpendsSapling | vSpendsSapling | SpendDescriptionV5 [nSpendsSapling] | A sequence of *Spend descriptions*, encoded per § 7.3 *'Spend Description Encoding and Consensus'* on p. 119. |
| | *Varies* | nOutputsSapling | compactSize | The number of *Output descriptions* in vOutputsSapling. |
| | 948· nOutputsSapling | vOutputsSapling | OutputDescriptionV5 [nOutputsSapling] | A sequence of *Output descriptions*, encoded per § 7.4 *'Output Description Encoding and Consensus'* on p. 120. |
| ‡ | 8 | valueBalanceSapling | int64 | The net value of **Sapling** spends minus outputs. |
| ‡ | 32 | anchorSapling | byte[32] | A *root* of the **Sapling** *note commitment tree* at some *block height* in the past, $\mathsf{LEBS2OSP}_{256}(\mathsf{rt}^{\mathsf{Sapling}})$. |
| ‡ | 192· nSpendsSapling | vSpendProofsSapling | byte[192] [nSpendsSapling] | Encodings of the *zk-SNARK proofs* for each **Sapling** *Spend description*. |
| ‡ | 64· nSpendsSapling | vSpendAuthSigsSapling | byte[64] [nSpendsSapling] | Authorizing signatures for each **Sapling** *Output description*. |
| ‡ | 192· nOutputsSapling | vOutputProofsSapling | byte[192] [nOutputsSapling] | Encodings of the *zk-SNARK proofs* for each **Sapling** *Output description*. |
| ‡ | 64 | bindingSigSapling | byte[64] | A *Sapling binding signature* on the *SIGHASH transaction hash*, validated per § 5.4.7.2 *'Binding Signature (**Sapling** and **Orchard**)'* on p. 88. |
| | *Varies* | nActionsOrchard | compactSize | The number of *Action descriptions* in vActionsOrchard. |
| | 884· nActionsOrchard | vActionsOrchard | ActionDescription [nActionsOrchard] | A sequence of *Action descriptions*, encoded per § 7.5 *'Action Description Encoding and Consensus'* on p. 121. |
| § | 1 | flagsOrchard | byte | Contains:<br>· enableSpendsOrchard flag (bit 0)<br>· enableOutputsOrchard flag (bit 1)<br>· Reserved, zeros (bits 2 .. 7). |
| § | 8 | valueBalanceOrchard | int64 | The net value of **Orchard** spends minus outputs. |
| § | 32 | anchorOrchard | byte[32] | A *root* of the **Orchard** *note commitment tree* at some *block height* in the past, $\mathsf{LEBS2OSP}_{256}(\mathsf{rt}^{\mathsf{Orchard}})$. |
| § | *Varies* | sizeProofsOrchard | compactSize | The length of the aggregated *zk-SNARK proof* $\pi_{\mathsf{ZKAction}}$. |
| § | sizeProofsOrchard | proofsOrchard | byte[sizeProofsOrchard] | The aggregated *zk-SNARK proof* $\pi_{\mathsf{ZKAction}}$ (see § 5.4.10.3 'Halo 2' on p. 103). |
| § | 64· nActionsOrchard | vSpendAuthSigsOrchard | byte[64] [nActionsOrchard] | Authorizing signatures for each spend of an **Orchard** *Action description*. |
| § | 64 | bindingSigOrchard | byte[64] | An *Orchard binding signature* on the *SIGHASH transaction hash*, validated per § 5.4.7.2 *'Binding Signature (**Sapling** and **Orchard**)'* on p. 88. |

‡ The fields valueBalanceSapling, anchorSapling, vSpendProofsSapling, vSpendAuthSigsSapling, vOutputProofsSapling, and bindingSigSapling are present if and only if nSpendsSapling + nOutputsSapling $> 0$. If valueBalanceSapling is not present, then $\mathsf{v}^{\mathsf{balanceSapling}}$ is defined to be 0.

§ The fields flagsOrchard, valueBalanceOrchard, anchorOrchard, sizeProofsOrchard, proofsOrchard, vSpendAuthSigsOrchard, and bindingSigOrchard are present if and only if nActionsOrchard $> 0$. If valueBalanceOrchard is not present, then $\mathsf{v}^{\mathsf{balanceOrchard}}$ is defined to be 0.

*Transaction version* 5 does not support *JoinSplit transfers*. Several fields are reordered and/or renamed relative to prior versions.

**Consensus rules:**

- The *transaction version number* **MUST** be greater than or equal to 1.
- [Pre-**Overwinter**] The `fOverwintered` flag **MUST NOT** be set.
- [**Overwinter** onward] The `fOverwintered` flag **MUST** be set.
- [**Overwinter** onward] The *version group ID* **MUST** be recognized.
- [**Overwinter** only, pre-**Sapling**] The *transaction version number* **MUST** be 3, and the *version group ID* **MUST** be `0x03C48270`.
- [**Sapling** onward] The *transaction version number* **MUST** be 4, and the *version group ID* **MUST** be `0x892F2085`.
- [**NU5** onward] The *transaction version number* **MUST** be 4 or 5. If the *transaction version number* is 4 then the *version group ID* **MUST** be `0x892F2085`. If the *transaction version number* is 5 then the *version group ID* **MUST** be `0x26A7270A`.
- [Pre-**Sapling**] The encoded size of the *transaction* **MUST** be less than or equal to 100000 bytes.
- [**NU5** onward] `nSpendsSapling`, `nOutputsSapling`, and `nActionsOrchard` **MUST** all be less than $2^{16}$.
- [Pre-**Sapling**] If effectiveVersion $= 1$ or `nJoinSplit` $= 0$, then both `tx_in_count` and `tx_out_count` **MUST** be nonzero.
- [**Sapling** onward] If effectiveVersion $< 5$, then at least one of `tx_in_count`, `nSpendsSapling`, and `nJoinSplit` **MUST** be nonzero.
- [**Sapling** onward] If effectiveVersion $< 5$, then at least one of `tx_out_count`, `nOutputsSapling`, and `nJoinSplit` **MUST** be nonzero.
- [**NU5** onward] If effectiveVersion $\geq 5$, then at least one of `tx_in_count`, `nSpendsSapling`, and `nActionsOrchard` **MUST** be nonzero.
- [**NU5** onward] If effectiveVersion $\geq 5$, then at least one of `tx_out_count`, `nOutputsSapling`, and `nActionsOrchard` **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 and *funding stream* outputs.
- If effectiveVersion $\geq 2$ and `nJoinSplit` $> 0$, then:
  - `joinSplitPubKey` **MUST** be a valid encoding (see § 5.4.6 'Ed25519' on p. 83) of an Ed25519 *validating key*.
  - `joinSplitSig` **MUST** represent a valid signature under `joinSplitPubKey` of dataToBeSigned, as defined in § 4.11 *'Non-malleability (Sprout)'* on p. 46.
- [**Sapling** onward] If effectiveVersion $\geq 4$ and `nSpendsSapling` + `nOutputsSapling` $> 0$, then:
  - let $\mathsf{bvk}^{\mathsf{Sapling}}$ and SigHash be as defined in § 4.13 *'Balance and Binding Signature (Sapling)'* on p. 47;
  - `bindingSigSapling` **MUST** represent a valid signature under the *transaction binding validating key* $\mathsf{bvk}^{\mathsf{Sapling}}$ of SigHash — i.e. $\mathsf{BindingSig}^{\mathsf{Sapling}}.\mathsf{Validate}_{\mathsf{bvk}^{\mathsf{Sapling}}}(\mathsf{SigHash}, \texttt{bindingSigSapling}) = 1$. [**NU5** onward] As specified in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85, the validation of the $\underline{R}$ component of the signature changes to prohibit *non-canonical* encodings.
- [**Sapling** onward] If effectiveVersion $= 4$ and there are no *Spend descriptions* or *Output descriptions*, then `valueBalanceSapling` **MUST** be 0.
- [**NU5** onward] If effectiveVersion $\geq 5$ and `nActionsOrchard` $> 0$, then:
  - let $\mathsf{bvk}^{\mathsf{Orchard}}$ and SigHash be as defined in § 4.14 *'Balance and Binding Signature (Orchard)'* on p. 50;
  - `bindingSigOrchard` **MUST** represent a valid signature under the *transaction binding validating key* $\mathsf{bvk}^{\mathsf{Orchard}}$ of SigHash — i.e. $\mathsf{BindingSig}^{\mathsf{Orchard}}.\mathsf{Validate}_{\mathsf{bvk}^{\mathsf{Orchard}}}(\mathsf{SigHash}, \texttt{bindingSigOrchard}) = 1$. As specified in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85, validation of the $\underline{R}$ component of the signature prohibits *non-canonical* encodings.

- The total value in *zatoshi* of *transparent outputs* from a *coinbase transaction*, minus $\mathsf{v^{balanceSapling}}$, minus $\mathsf{v^{balanceOrchard}}$, **MUST NOT** be greater than the value in *zatoshi* of *miner subsidy* plus the *transaction fees* paid by *transactions* in this *block*.
- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions* or *Spend descriptions*.
- [Pre-**Heartwood**] A *coinbase transaction* also **MUST NOT** have any *Output descriptions*.
- [**NU5** onward] In a version 5 *coinbase transaction*, the `enableSpendsOrchard`flag **MUST** be $0$.
- A *coinbase transaction* for a *block* at *block height* greater than $0$ **MUST** have a script that, as its first item, encodes the *block height* as follows. Let heightBytes be the signed little-endian representation of the number, using the minimum number of bytes such that the most significant byte is $<$ `0x80`. Then the encoding is the length of heightBytes encoded as one byte, followed by heightBytes itself. This matches the encoding used by **Bitcoin** in the implementation of [BIP-34] (but the description here is to be considered normative).
- A *transaction* **MUST NOT** spend a *transparent* output of a *coinbase transaction* from a *block* less than 100 *blocks* prior to the spend. Note that *transparent* outputs of *coinbase transactions* include *Founders' Reward* outputs and *transparent funding stream* 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*.)
- [**Overwinter** onward] `nExpiryHeight` **MUST** be less than or equal to 499999999.
- [**Overwinter** onward] If a *transaction* is not a *coinbase transaction* and its `nExpiryHeight` field is nonzero, then it **MUST NOT** be mined at a *block height* greater than its `nExpiryHeight`.
- [**Sapling** onward] `valueBalance` **MUST** be in the range $\{-\mathsf{MAX\_MONEY} .. \mathsf{MAX\_MONEY}\}$.
- [**Heartwood** onward] All **Sapling** and **Orchard** outputs in *coinbase transactions* **MUST** decrypt to a *note plaintext*, i.e. the procedure in § 4.19.3 ***'Decryption using a Full Viewing Key (Sapling and Orchard)'*** on p. 63 does not return $\bot$, using a sequence of 32 zero bytes as the *outgoing viewing key*.
- [**Canopy** onward] Any **Sapling** or **Orchard** output of a *coinbase transaction* decrypted to a *note plaintext* according to the preceding rule **MUST** have *note plaintext lead byte* equal to `0x02`. (This applies even during the "grace period" specified in [ZIP-212].)
- TODO: Other rules inherited from **Bitcoin**.

Consensus rules associated with each *JoinSplit description* (§ 7.2 ***'JoinSplit Description Encoding and Consensus'*** on p. 119), each *Spend description* (§ 7.3 ***'Spend Description Encoding and Consensus'*** on p. 119), each *Output description* (§ 7.4 ***'Output Description Encoding and Consensus'*** on p. 120), and each *Action description* (§ 7.5 ***'Action Description Encoding and Consensus'*** on p. 121) **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 *transaction version number* not equal to 1, 2, 3, 4, or 5 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*.
- [**Overwinter** onward] Once **Overwinter** has activated, limits on the maximum *transaction version number* are consensus rules.
- 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* or *version group ID*. It is likely that an upgrade that changes the *transaction version number* or *version group ID* will also change the *transaction* format, and software that parses *transactions* **SHOULD** take this into account.
- [**Overwinter** onward] The purpose of *version group ID* is to allow unambiguous parsing of *"loose" transactions*, independent of the context of a *block chain*. Code that parses *transactions* is likely to be reused between *block chain branches* as defined in [ZIP-200], and in that case the `fOverwintered` and `version` fields alone may be insufficient to determine the format to be used for parsing.
- A *transaction version number* of 2 does not have the same meaning as in **Bitcoin**, where it is associated with support for `OP_CHECKSEQUENCEVERIFY` as specified in [BIP-68]. **Zcash** was forked from **Bitcoin** v0.11.2 and does not currently support BIP 68.
- [**Sapling** onward] Because *coinbase transactions* have no *Spend descriptions*, the `valueBalanceSapling` field of a *coinbase transaction* must have a negative or zero value. The negative case can only occur after **Heartwood** activation, for *transactions* with [ZIP-213] *shielded outputs*.
- Prior to the **Heartwood** *network upgrade*, it was not possible for *coinbase transactions* to have *shielded* outputs, and therefore the "coinbase maturity" rule and the requirement to spend coinbase outputs only in *transactions* with no *transparent* outputs, applied to ***all*** coinbase outputs.
- The rule that **Sapling** outputs in *coinbase transactions* **MUST** decrypt to a *note plaintext* with lead byte `0x02`, also applies to *funding stream* outputs that specify **Sapling** *shielded payment addresses*, if there are any.
- The flags in `flagsOrchard` allow a version 5 *transaction* to declare that no funds are spent from **Orchard** *notes* (by setting `enableSpendsOrchard` to 0), or that no new **Orchard** *notes* with non-zero values are created (by setting `enableOutputsOrchard` to 0). This has two primary purposes. First, the `enableSpendsOrchard` flag is set to 0 in version 5 *coinbase transactions* to ensure that they cannot spend from existing **Orchard** outputs. This maintains a restriction present in *coinbase transactions* for *transparent*, **Sprout**, or **Sapling** funds, which would not otherwise be enforceable in the combined *Action transfer* design. Second, if a security vulnerability were found that affected only the input side, or only the output side of the *Action circuit*, it would be possible to use these flags in a soft fork (i.e. a strictly contracting consensus change) to effectively "switch off" non-zero-value transfers only on the relevant side.
- [**NU5** onward] Because the `enableSpendsOrchard` is set to 0 in version 5 *coinbase transactions* –which disables non-zero-valued **Orchard** spends– the `valueBalanceOrchard` field of a *coinbase transaction* must have a negative or zero value.
- [**NU5** onward] The rule that `nSpendsSapling`, `nOutputsSapling`, and `nActionsOrchard` **MUST** all be less than $2^{16}$, is technically redundant because a *transaction* that could violate this rule would not fit within the 2 MB *block* size limit. It is included in order to simplify the security argument for balance preservation.

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.
- [**Overwinter** onward] The field `nVersionGroupId` has been added.
- [**Sapling** onward] The following fields have been added: `nSpendsSapling`, `vSpendsSapling`, `nOutputsSapling`, `vOutputsSapling`, and `bindingSigSapling`.
- [**NU5** onward] In version 5 *transactions*, these fields have been added: `nActionsOrchard`, `vActionsOrchard`, `flagsOrchard`, `valueBalanceOrchard`, `anchorOrchard`, `sizeProofsOrchard`, `proofsOrchard`, `bindingSigOrchard`, and `vSpendAuthSigsOrchard`.
- In **Zcash** it is permitted for a *transaction* to have no *transparent* inputs, provided at least one of `nJoinSplit`, `nSpendsSapling`, `nOutputsSapling`, and `nActionsOrchard` 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

An abstract *JoinSplit description*, as described in § 3.5 ***'JoinSplit Transfers and Descriptions'*** on p. 16, is encoded in a *transaction* as an instance of a `JoinSplitDescription` type:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 8 | `vpub_old` | `uint64` | A value $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ that the *JoinSplit transfer* removes from the *transparent transaction value pool*. |
| 8 | `vpub_new` | `uint64` | A value $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ 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]` $[\mathrm{N}^{\mathsf{old}}]$ | A sequence of *nullifiers* of the input *notes* $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$. |
| 64 | `commitments` | `byte[32]` $[\mathrm{N}^{\mathsf{new}}]$ | A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$. |
| 32 | `ephemeralKey` | `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]` $[\mathrm{N}^{\mathsf{old}}]$ | A sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ binding $\mathsf{h}_{\mathsf{Sig}}$ to each $\mathsf{a}_{\mathsf{sk}}$ of the *JoinSplit description*, computed as described in § 4.11 ***'Non-malleability (Sprout)'*** on p. 46. |
| 296 † | `zkproof` | `byte[296]` | An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see § 5.4.10.1 'BCTV14' on p. 102). |
| 192 ‡ | `zkproof` | `byte[192]` | An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see § 5.4.10.2 'Groth16' on p. 103). |
| 1202 | `encCiphertexts` | `byte[601]` $[\mathrm{N}^{\mathsf{new}}]$ | A sequence of ciphertext components for the encrypted output *notes*, $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$. |

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

‡ Groth16 proofs are used when the *transaction version* is $\geq 4$, i.e. after **Sapling** activation.

The `ephemeralKey` and `encCiphertexts` fields together form the *transmitted notes ciphertext*, which is computed as described in § 4.18 ***'In-band secret distribution (Sprout)'*** on p. 59.

Consensus rules applying to a *JoinSplit description* are given in § 4.3 ***'JoinSplit Descriptions'*** on p. 36.

## 7.3 Spend Description Encoding and Consensus

Let LEBS2OSP be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Let $\mathsf{repr}_{\mathbb{J}}$ and $q_{\mathbb{J}}$ be as defined in § 5.4.9.3 'Jubjub' on p. 94.

Let spendAuthSig be the *spend authorization signature* for this *Spend transfer*. In a version 4 *transaction* this is encoded in the `spendAuthSig` field of the *Spend description*. In a version 5 *transaction*, *spend authorization signatures* in `vSpendAuthSigsSapling` are in one-to-one correspondence with *Spend descriptions* in `vSpendsSapling`.

An abstract *Spend description*, as described in § 3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 17, is encoded in a *transaction* as an instance of a `SpendDescriptionV4` or `SpendDecriptionV5` type:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 32 | `cv` | `byte[32]` | A *value commitment* to the value of the input *note*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{cv})\big)$. |
| 32 † | `anchor` | `byte[32]` | A *root* of the **Sapling** *note commitment tree* at some *block height* in the past, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{rt}^{\mathsf{Sapling}}\big)$. |
| 32 | `nullifier` | `byte[32]` | The *nullifier* of the input *note*, $\mathsf{nf}$. |
| 32 | `rk` | `byte[32]` | The randomized *validating key* for $\mathsf{spendAuthSig}$, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{rk})\big)$. |
| 192 † | `zkproof` | `byte[192]` | An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKSpend}}$ (see § 5.4.10.2 'Groth16' on p. 103). |
| 64 † | `spendAuthSig` | `byte[64]` | A signature authorizing this Spend. |

† The `anchorSapling`, `zkproof`, and `spendAuthSig` fields are only present in a *Spend description* if the *transaction version* is 4. For version 5 *transactions*, all *Spend descriptions* share the same *anchor*, which is encoded once as the `anchorSapling` field of the *transaction* as described in § 7.1 ***'Transaction Encoding and Consensus'*** on p. 114. The `zkproof` and `spendAuthSig` fields of a *Spend description* have been moved into the `vSpendProofsSapling` and `vSpendAuthSigsSapling` fields respectively of version 5 *transactions*.

**Consensus rule:** $\mathsf{LEOS2IP}_{256}(\texttt{anchorSapling})$, if present, **MUST** be less than $q_{\mathbb{J}}$.

Other consensus rules applying to a *Spend description* are given in § 4.4 ***'Spend Descriptions'*** on p. 37.

## 7.4 Output Description Encoding and Consensus

Let $\mathsf{LEBS2OSP}$ be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Let $\mathsf{repr}_{\mathbb{J}}$ and $q_{\mathbb{J}}$ be as in § 5.4.9.3 'Jubjub' on p. 94, and $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ as in § 5.4.9.4 ***'Coordinate Extractor for*** Jubjub' on p. 96.

An abstract *Output description*, described in § 3.6 ***'Spend Transfers, Output Transfers, and their Descriptions'*** on p. 17, is encoded in a *transaction* as an instance of an `OutputDescriptionV4` or `OutputDecriptionV5` type:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 32 | `cv` | `byte[32]` | A *value commitment* to the value of the output *note*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{cv})\big)$. |
| 32 | `cmu` | `byte[32]` | The $u$-coordinate of the *note commitment* for the output *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{cm}_u)$ where $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm})$. |
| 32 | `ephemeralKey` | `byte[32]` | An encoding of an ephemeral Jubjub *public key*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})\big)$. |
| 580 | `encCiphertext` | `byte[580]` | A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{enc}}$. |
| 80 | `outCiphertext` | `byte[80]` | A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{out}}$. |
| 192 † | `zkproof` | `byte[192]` | An encoding of the *zk-SNARK proof* $\pi_{\mathsf{ZKOutput}}$ (see § 5.4.10.2 'Groth16' on p. 103). |

† The `zkproof` field is only present in a *Spend description* if the *transaction version* is 4. This field has been moved into the `vOutputProofsSapling` field of version 5 *transactions*.

The `ephemeralKey`, `encCiphertext`, and `outCiphertext` fields together form the *transmitted note ciphertext*, which is computed as described in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60.

**Consensus rule:** $\mathsf{LEOS2IP}_{256}(\texttt{cmu})$ **MUST** be less than $q_{\mathbb{J}}$.

Other consensus rules applying to an *Output description* are given in § 4.5 ***'Output Descriptions'*** on p. 38.

## 7.5 Action Description Encoding and Consensus

Let LEBS2OSP be as defined in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

Let $\mathsf{repr}_{\mathbb{P}}$ and $q_{\mathbb{P}}$ be as defined in § 5.4.9.6 'Pallas *and* Vesta' on p. 97.

Let spendAuthSig be the *spend authorization signature* for this *Action transfer* from `vSpendAuthSigsOrchard`. *Spend authorization signatures* in `vSpendAuthSigsOrchard` are in one-to-one correspondence with *Action descriptions* in `vActionsOrchard`.

An abstract *Action description*, as described in § 3.7 ***'Action Transfers and their Descriptions'*** on p. 18, is encoded in a *transaction* as an instance of an `ActionDescription` type:

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 32 | `cv` | `byte[32]` | A *value commitment* to the net value of the input *note* minus the output *note*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{P}}(\mathsf{cv})\big)$. |
| 32 | `nullifier` | `byte[32]` | The *nullifier* of the input *note*, nf. |
| 32 | `rk` | `byte[32]` | The randomized *validating key* for spendAuthSig, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{P}}(\mathsf{rk})\big)$. |
| 32 | `cmx` | `byte[32]` | The $x$-coordinate of the *note commitment* for the output *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{cm}_x)$ where $\mathsf{cm}_x = \mathsf{Extract}_{\mathbb{P}}(\mathsf{cm})$. |
| 32 | `ephemeralKey` | `byte[32]` | An encoding of an ephemeral Jubjub *public key*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{P}}(\mathsf{epk})\big)$. |
| 580 | `encCiphertext` | `byte[580]` | A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{enc}}$. |
| 80 | `outCiphertext` | `byte[80]` | A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{out}}$. |

The `ephemeralKey`, `encCiphertext`, and `outCiphertext` fields together form the *transmitted note ciphertext*, which is computed as described in § 4.19 ***'In-band secret distribution (Sapling and Orchard)'*** on p. 60.

**Consensus rule:** $\mathsf{LEOS2IP}_{256}(\texttt{cmx})$ **MUST** be less than $q_{\mathbb{P}}$.

Other consensus rules applying to an *Action description* are given in § 4.6 ***'Action Descriptions'*** on p. 38.

## 7.6 Block Header Encoding and Consensus

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

| Bytes | Name | Data Type | Description |
|---|---|---|---|
| 4 | nVersion | int32 | The *block version number* indicates which set of *block* validation rules to follow. The current and only defined *block version number* for **Zcash** is 4. |
| 32 | hashPrevBlock | byte[32] | A SHA-256d hash in internal byte order of the previous *block*'s *header*. This ensures no previous *block* can be changed without also changing this *block*'s *header*. |
| 32 | hashMerkleRoot | byte[32] | A SHA-256d hash in internal byte order. The merkle root is derived from the hashes of all *transactions* included in this *block*, ensuring that none of those *transactions* can be modified without modifying the *header*. |
| 32 | hashReserved /<br>hashFinalSaplingRoot /<br>hashLightClientRoot /<br>hashBlockCommitments | byte[32] | [Pre-**Sapling**] A reserved field, to be ignored.<br>[**Sapling** and **Blossom** only, pre-**Heartwood**] The *root* $\mathsf{LEBS2OSP}_{256}(\mathsf{rt}^{\mathsf{Sapling}})$ of the **Sapling** *note commitment tree* corresponding to the final **Sapling** *treestate* of this *block*.<br>[**Heartwood** onward] The hashChainHistoryRoot of this *block* as defined in [ZIP-221].<br>[**NU5** onward] The hashBlockCommitments of this *block* as defined in [ZIP-244]. |
| 4 | nTime | uint32 | The *block timestamp* is a Unix epoch time (UTC) when the miner started hashing the *header* (according to the miner). |
| 4 | nBits | uint32 | An encoded version of the *target threshold* this *block*'s *header* hash must be less than or equal to, in the same nBits format used by **Bitcoin**. [Bitcoin-nBits] |
| 32 | nNonce | byte[32] | An arbitrary field that miners can change to modify the *header* hash in order to produce a hash less than or equal to the *target threshold*. |
| 3 | solutionSize | compactSize | The size of an *Equihash* solution in bytes (always 1344). |
| 1344 | solution | byte[1344] | The *Equihash* solution. |

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

Let ThresholdBits be as defined in § 7.7.3 *'Difficulty adjustment'* on p. 125, and let PoWMedianBlockSpan be the constant defined in § 5.3 *'Constants'* on p. 67.

Define the *median-time-past* of a *block* to be the median (as defined in § 7.7.3 *'Difficulty adjustment'* on p. 125) 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* $\mathsf{height}$, `nBits` **MUST** be equal to $\mathsf{ThresholdBits}(\mathsf{height})$.
- The *block* **MUST** pass the difficulty filter defined in § 7.7.2 ***'Difficulty filter'*** on p. 125.
- `solution` **MUST** represent a *valid Equihash solution* as defined in § 7.7.1 ***'Equihash'*** on p. 124.
- 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.
- [**Sapling** and **Blossom** only, pre-**Heartwood**] `hashLightClientRoot` **MUST** be $\mathsf{LEBS2OSP}_{256}\left(\mathsf{rt}^{\mathsf{Sapling}}\right)$ where $\mathsf{rt}^{\mathsf{Sapling}}$ is the *root* of the **Sapling** *note commitment tree* for the final **Sapling** *treestate* of this *block*.
- [**Heartwood** onward] `hashLightClientRoot` **MUST** be set to the value of `hashChainHistoryRoot` for this *block*, as specified in [ZIP-221].
- 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 $\mathsf{PoWMedianBlockSpan}$ *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*.
- There are no changes to the *block version number* or format for **Overwinter**.
- Although the *block version number* does not change for **Sapling**, the previously reserved (and ignored) field `hashReserved` has been repurposed for `hashFinalSaplingRoot`. There are no other format changes.
- There are no changes to the *block version number* or format for **Blossom**.

- For **Heartwood**, the hashFinalSaplingRoot field is renamed to hashLightClientRoot. Once **Heartwood** activates, the meaning of this field changes according to [ZIP-221].
- There are no changes to the *block version number* or format for **Canopy**.
- For **NU5**, the hashLightClientRoot field is renamed to hashBlockCommitments. Once **NU5** activates, the meaning of this field changes according to [ZIP-244].

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

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

## 7.7 Proof of Work

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

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

- The solution field encodes a *valid Equihash solution* according to § 7.7.1 ***'Equihash'*** on p. 124.
- The *block header* satisfies the difficulty check according to § 7.7.2 ***'Difficulty filter'*** on p. 125.

### 7.7.1 Equihash

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

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

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

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

Let powheader :=

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

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

EquihashGen is instantiated in § 5.4.1.11 ***'Equihash Generator'*** on p. 78.

Define $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0\,..\,2^\ell - 1\} \to \mathbb{B}^{[\ell]}$ as in § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.

A *valid Equihash solution* is then a sequence $i : \{1\,..\,N\}^{2^k}$ that satisfies the following conditions:

**Generalized Birthday condition** $\displaystyle\bigoplus_{j=1}^{2^k} X_{i_j} = 0.$

**Algorithm Binding conditions**

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

**Notes:**

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

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

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

Recall from § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66 that bits in the above diagram are ordered from most to least significant in each byte. For example, if the first $3$ elements of $i$ are $[69, 42, 2^{21}]$, then the corresponding bit array is:

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

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

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

### 7.7.2 Difficulty filter

Let ToTarget be as defined in § 7.7.4 ***'nBits conversion'*** on p. 127.

Difficulty is defined in terms of a *target threshold*, which is adjusted for each *block* according to the algorithm defined in § 7.7.3 ***'Difficulty adjustment'*** on p. 125.

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`).

### 7.7.3 Difficulty adjustment

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

The constants PoWLimit, PreBlossomHalvingInterval, PoWAveragingWindow, PoWMaxAdjustDown, PoWMaxAdjustUp, PoWDampingFactor, PreBlossomPoWTargetSpacing, and PostBlossomPoWTargetSpacing are specified in section § 5.3 ***'Constants'*** on p. 67.

Let ToCompact and ToTarget be as defined in § 7.7.4 ***'nBits conversion'*** on p. 127.

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

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

*Block header* fields are specified in § 7.6 ***'Block Header Encoding and Consensus'*** on p. 122.

Define:

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

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

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

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

$$\mathsf{IsBlossomActivated}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := (\mathsf{height} \geq \mathsf{BlossomActivationHeight})$$

$$\mathsf{BlossomPoWTargetSpacingRatio} := \frac{\mathsf{PreBlossomPoWTargetSpacing}}{\mathsf{PostBlossomPoWTargetSpacing}}$$

$$\mathsf{PostBlossomHalvingInterval} := \mathsf{floor}(\mathsf{PreBlossomHalvingInterval} \cdot \mathsf{BlossomPoWTargetSpacingRatio})$$

$$\mathsf{PoWTargetSpacing}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := \begin{cases} \mathsf{PreBlossomPoWTargetSpacing}, & \text{if not } \mathsf{IsBlossomActivated}(\mathsf{height}) \\ \mathsf{PostBlossomPoWTargetSpacing}, & \text{otherwise} \end{cases}$$

$$\mathsf{AveragingWindowTimespan}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := \mathsf{PoWAveragingWindow} \cdot \mathsf{PoWTargetSpacing}(\mathsf{height})$$

$$\mathsf{MinActualTimespan}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := \mathsf{floor}(\mathsf{AveragingWindowTimespan}(\mathsf{height}) \cdot (1 - \mathsf{PoWMaxAdjustUp}))$$

$$\mathsf{MaxActualTimespan}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := \mathsf{floor}(\mathsf{AveragingWindowTimespan}(\mathsf{height}) \cdot (1 + \mathsf{PoWMaxAdjustDown}))$$

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

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

$$\begin{aligned}&\mathsf{ActualTimespanDamped}(\mathsf{height} \mathbin{⦂} \mathbb{N}) := \\ &\qquad \mathsf{AveragingWindowTimespan}(\mathsf{height}) + \mathsf{trunc}\left(\frac{\mathsf{ActualTimespan}(\mathsf{height}) - \mathsf{AveragingWindowTimespan}(\mathsf{height})}{\mathsf{PoWDampingFactor}}\right)\end{aligned}$$

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

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

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

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

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

**Notes:**

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

On *Testnet* from *block height* 299188 onward, the difficulty adjustment algorithm is changed to allow minimum-difficulty *blocks*, as described in [ZIP-205]. The **Blossom** *network upgrade* changes the minimum-difficulty time threshold to 6 times the *block target spacing*, as described in [ZIP-208]. These changes do not apply to *Mainnet*.

### 7.7.4 nBits conversion

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

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

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

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

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

### 7.7.5 Definition of Work

As explained in § 3.3 *‘The Block Chain’* on p. 15, a node chooses the “best” *block chain* visible to it by finding the chain of valid *blocks* with the greatest total work.

Let ToTarget be as defined in § 7.7.4 *‘nBits conversion’* on p. 127.

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

## 7.8 Calculation of Block Subsidy, Funding Streams, and Founders’ Reward

§ 3.10 *‘Block Subsidy, Funding Streams, and Founders’ Reward’* on p. 19 defines the *block subsidy*, *miner subsidy*, *Founders’ Reward*, and *funding streams*. Their amounts in *zatoshi* are calculated from the *block height* using the formulae below.

Let the constants SlowStartInterval, PreBlossomHalvingInterval, PostBlossomHalvingInterval, BlossomActivationHeight, MaxBlockSubsidy, and FoundersFraction be as defined in § 5.3 *‘Constants’* on p. 67.

Let FundingStreams be as specified in § 7.10.1 *‘ZIP 214 Funding Streams’* on p. 131.

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

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

$$\mathsf{Halving}(\mathsf{height} : \mathbb{N}) := \begin{cases} 0, & \text{if } \mathsf{height} < \mathsf{SlowStartShift} \\ \mathsf{floor}\left(\frac{\mathsf{height} - \mathsf{SlowStartShift}}{\mathsf{PreBlossomHalvingInterval}}\right), & \text{if not } \mathsf{IsBlossomActivated}(\mathsf{height}) \\ \mathsf{floor}\left(\frac{\mathsf{BlossomActivationHeight} - \mathsf{SlowStartShift}}{\mathsf{PreBlossomHalvingInterval}} + \frac{\mathsf{height} - \mathsf{BlossomActivationHeight}}{\mathsf{PostBlossomHalvingInterval}}\right), & \text{otherwise} \end{cases}$$

$$\mathsf{BlockSubsidy}(\mathsf{height} : \mathbb{N}) := \begin{cases} \mathsf{SlowStartRate} \cdot \mathsf{height}, & \text{if } \mathsf{height} < \mathsf{SlowStartShift} \\ \mathsf{SlowStartRate} \cdot (\mathsf{height} + 1), & \text{if } \mathsf{SlowStartShift} \leq \mathsf{height} \\ & \quad \text{and } \mathsf{height} < \mathsf{SlowStartInterval} \\ \mathsf{floor}\left(\frac{\mathsf{MaxBlockSubsidy}}{2^{\mathsf{Halving}(\mathsf{height})}}\right), & \text{if } \mathsf{SlowStartInterval} \leq \mathsf{height} \\ & \quad \text{and not } \mathsf{IsBlossomActivated}(\mathsf{height}) \\ \mathsf{floor}\left(\frac{\mathsf{MaxBlockSubsidy}}{\mathsf{BlossomPoWTargetSpacingRatio} \cdot 2^{\mathsf{Halving}(\mathsf{height})}}\right), & \text{otherwise} \end{cases}$$

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

$$\text{for } \mathsf{fs} \in \mathsf{FundingStreams},\ \mathsf{fs.Value}(\mathsf{height}) :=$$

$$\begin{cases} 0, & \text{if } \mathsf{height} < \mathsf{CanopyActivationHeight} \\ \mathsf{floor}\left(\mathsf{BlockSubsidy}(\mathsf{height}) \cdot \dfrac{\mathsf{fs.Numerator}}{\mathsf{fs.Denominator}}\right), & \text{if } \mathsf{fs.StartHeight} \leq \mathsf{height} \text{ and } \mathsf{height} < \mathsf{fs.EndHeight} \\ 0, & \text{otherwise} \end{cases}$$

$$\mathsf{MinerSubsidy}(\mathsf{height}) := \mathsf{BlockSubsidy}(\mathsf{height}) - \mathsf{FoundersReward}(\mathsf{height}) - \sum_{\mathsf{fs} \in \mathsf{FundingStreams}} \mathsf{fs.Value}(\mathsf{height}).$$

## 7.9 Payment of Founders' Reward

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

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

```
[ “t3Vz22vK5z2LcKEdg16Yv4FFneEL1zg9ojd”, “t3cL9AucCajm3HXDhb5jBnJK2vapVoXsop3”,
  “t3fqvkzrrNaMcamkQMwAyHRjfDdM2xQvDTR”, “t3TgZ9ZT2CTSK44AnUPi6qeNaHa2eC7pUyF”,
  “t3SpkcPQPfuRYHsP5vz3Pv86PgKo5m9KVmx”, “t3Xt4oQMRPagwbpQqkgAViQgtST4VoSWR6S”,
  “t3ayBkZ4w6kKXynwoHZFUSSgXRKtogTXNgb”, “t3adJBQuaa21u7NxbR8YMzp3km3TbSZ4MGB”,
  “t3K4aLYagSSBySdrfAGGeUd5H9z5Qvz88t2”, “t3RYnsc5nhEvKiva3ZPhfRSk7eyh1CrA6Rk”,
  “t3Ut4KUq2ZSMTPNE67pBU5LqYCi2q36KpXQ”, “t3ZnCNAvgu6CSyHm1vWtrx3aiN98dSAGpnD”,
  “t3fB9cB3eSYim64BS9xfwAHQUKLgQQroBDG”, “t3cwZfKNNj2vXMAHBQeewm6pXhKFdhk18kD”,
  “t3YcoujXfspWy7rbNUsGKxFEWZqNstGpeG4”, “t3bLvCLigc6rbNrUTS5NwkgyVrZcZumTRa4”,
  “t3VvHWa7r3oy67YtU4LZKGCWa2J6eGHvShi”, “t3eF9X6X2dSo7MCvTjfZEzwWrVzquxRLNeY”,
  “t3esCNwwmcyc8i9qQfyTbYhTqmYXZ9AwK3X”, “t3M4jN7hYE2e27yLsuQPPjuVek81WV3VbBj”,
  “t3gGWxdC67CYNoBbPjNvrrWLAWxPqZLxrVY”, “t3LTWeoxeWPbmdkUD3NWBquk4WkazhFBmvU”,
  “t3P5KKX97gXYFSaSjJPiruQEX84yF5z3Tjq”, “t3f3T3nCWsEpzmD35VK62JgQfFig74dV8C9”,
  “t3Rqonuzz7afkF7156ZA4vi4iimRSEn41hj”, “t3fJZ5jYsyxDtvNrWBeoMbvJaQCj4JJgbgX”,
  “t3Pnbg7XjP7FGPBUuz75H65aczphHgkpoJW”, “t3WeKQDxCijL5X7rwFem1MTL9ZwVJkUFhpF”,
  “t3Y9FNi26J7UtAUC4moaETLbMo8KS1Be6ME”, “t3aNRLLsL2y8xcjPheZZwFy3Pcv7CsTwBec”,
  “t3gQDEavk5VzAAHK8TrQu2BWDLxEiF1unBm”, “t3RbykhX1TUFrgXrmBYrAJe2STxRKFL7G9r”,
  “t3aaW4aTdP7a8d1VTE1Bod2yhbeggHgMajR”, “t3YEiAa6uEjXwFL2v5ztU1fn3yKgzMQqNyo”,
  “t3g1yUUwt2PbmDvMDevTCPWUcbDatL2iQGP”, “t3dPWnep6YqGPuY1CecgbeZrY9iUwH8Yd4z”,
  “t3QRZXHDPh2hwU46iQs2776kRuuWfwFp4dV”, “t3enhACRxi1ZD7e8ePomVGKn7wp7N9fFJ3r”,
  “t3PkLgT71TnF112nSwBToXsD77yNbx2gJJY”, “t3LQtHUDoe7ZhhvddRv4vnaoNAhCr2f4oFN”,
  “t3fNcdBUbycvbCtsD2n9q3LuxG7jVPvFB8L”, “t3dKojUU2EMjs28nHV84TvkVEUDu1M1FaEx”,
  “t3aKH6NiWN1ofGd8c19rZiqgYpkJ3n679ME”, “t3MEXDF9Wsi63KwpPuQdD6by32Mw2bNTbEa”,
  “t3WDhPfik343yNmPTqtkZAoQZeqA83K7Y3f”, “t3PSn5TbMMAEw7Eu36DYctFezRzpX1hzf3M”,
  “t3R3Y5vnBLrEn8L6wFjPjBLnxSUQsKnmFpv”, “t3Pcm737EsVkGTbhsu2NekKtJeG92mvYyoN” ]
```

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

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

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

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

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

Define:

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

$$\mathsf{FounderAddressAdjustedHeight}(\mathsf{height} \;⦂\; \mathbb{N}) :=$$

$$\begin{cases} \mathsf{height}, & \text{if not } \mathsf{IsBlossomActivated}(\mathsf{height}), \\ \mathsf{BlossomActivationHeight} + \mathsf{floor}\left(\frac{\mathsf{height} - \mathsf{BlossomActivationHeight}}{\mathsf{BlossomPoWTargetSpacingRatio}}\right), & \text{otherwise} \end{cases}$$

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

$$\mathsf{FoundersRewardLastBlockHeight} := \mathsf{max}(\{\mathsf{height} \;⦂\; \mathbb{N} \mid \mathsf{Halving}(\mathsf{height}) < 1\}).$$

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:** [Pre-**Canopy**] A *coinbase transaction* at 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*), or after **Canopy** activation.
- 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.

**Non-normative notes:**

- [**Blossom** onward] $\mathsf{FoundersRewardLastBlockHeight} = 1046399$.
- **Blossom** is not intended to change the total *Founders' Reward* or the effective period over which it is paid.

## 7.10 Payment of Funding Streams

The *funding streams* are paid by outputs in the *coinbase transaction*, to one of a pre-defined set of addresses, depending on the *block height*.

A *funding stream* $\mathsf{fs}$ is defined by a *block subsidy* fraction (represented as a numerator and denominator), a start *block height* (inclusive), an end *block height* (exclusive), and a sequence of address representations:

$$\mathsf{fs.Numerator} \colon \mathbb{N}^+$$

$$\mathsf{fs.Denominator} \colon \mathbb{N}^+$$

$$\mathsf{fs.StartHeight} \colon \mathbb{N}$$

$$\mathsf{fs.EndHeight} \colon \mathbb{N}$$

$$\mathsf{fs.AddressList} \colon \mathbb{BY}^{[\mathbb{N}]\,[\mathbb{N}^+]}.$$

Define:

$$\mathsf{HeightForHalving}(\mathsf{halving} \colon \mathbb{N}^+) := \mathsf{min}(\{\mathsf{height} \colon \mathbb{N} \mid \mathsf{Halving}(\mathsf{height}) = \mathsf{halving}\})$$

$$\mathsf{FundingStreamAddressChangeInterval} := \mathsf{PostBlossomHalvingInterval}/48$$

$$\mathsf{FundingStreamAddressPeriod}(\mathsf{height}) := \mathsf{floor}\left(\frac{\mathsf{height} - (\mathsf{HeightForHalving}(1) - \mathsf{PostBlossomHalvingInterval})}{\mathsf{FundingStreamAddressChangeInterval}}\right).$$

For each *funding stream* $\mathsf{fs}$, define:

$$\mathsf{fs.AddressIndex}(\mathsf{height}) := 1 + \mathsf{FundingStreamAddressPeriod}(\mathsf{height}) - \mathsf{FundingStreamAddressPeriod}(\mathsf{fs.StartHeight})$$

$$\mathsf{fs.NumAddresses} := \mathsf{fs.AddressIndex}(\mathsf{fs.EndHeight} - 1).$$

$\mathsf{fs.AddressList}$ **MUST** be of length $\mathsf{fs.NumAddresses}$. Each element of $\mathsf{fs.AddressList}$ **MUST** represent either a *transparent* P2SH address as specified in § 5.6.1.1 *'Transparent Addresses'* on p. 105, or a **Sapling** *shielded payment address* as specified in § 5.6.3.1 *'**Sapling** Payment Addresses'* on p. 107.

Recall from § 7.8 *'Calculation of Block Subsidy, Funding Streams, and Founders' Reward'* on p. 127 the definition of $\mathsf{fs.Value}$. A *funding stream* $\mathsf{fs}$ is "active" at *block height* $\mathsf{height}$ when $\mathsf{fs.Value}(\mathsf{height}) > 0$.

**Consensus rule:** [**Canopy** onward] The *coinbase transaction* at *block height* $\mathsf{height}$ **MUST** contain at least one output per *funding stream* $\mathsf{fs}$ active at $\mathsf{height}$, that pays $\mathsf{fs.Value}(\mathsf{height})$ *zatoshi* in the prescribed way to the stream's recipient address represented by $\mathsf{fs.AddressList}_{\mathsf{fs.AddressIndex}(\mathsf{height})}$.

- The "prescribed way" to pay a *transparent* P2SH address is to use a standard P2SH script of the form `OP_HASH160` $\mathsf{fs.RedeemScriptHash}(\mathsf{height})$ `OP_EQUAL` as the `scriptPubKey`. Here $\mathsf{fs.RedeemScriptHash}(\mathsf{height})$ is the standard redeem script hash for the recipient address given by $\mathsf{fs.AddressList}_{\mathsf{fs.AddressIndex}(\mathsf{height})}$ in *Base58Check* form. The standard redeem script hash is specified in [Bitcoin-Multisig] for P2SH multisig addresses, or [Bitcoin-P2SH] for other P2SH addresses.
- The "prescribed way" to pay a **Sapling** address is as defined in [ZIP-213], using the post-**Heartwood** consensus rules specified for **Sapling** outputs of *coinbase transactions* in § 7.1 *'Transaction Encoding and Consensus'* on p. 114.

**Notes:**

- The *funding stream* 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* to have other outputs, possibly to the same address, that do not meet the criterion in the above consensus rule, as long as at least one output meets it.

### 7.10.1 ZIP 214 Funding Streams

Let $\mathsf{CanopyActivationHeight}$ be as defined in § 5.3 *'Constants'* on p. 67.

[ZIP-214] defines these *funding streams* for *Mainnet*:

| Stream | Numerator | Denominator | Start height | End height |
|---|---|---|---|---|
| FS_ZIP214_ECC | 7 | 100 | 1046400 | 2726400 |
| FS_ZIP214_ZF | 5 | 100 | 1046400 | 2726400 |
| FS_ZIP214_MG | 8 | 100 | 1046400 | 2726400 |

It also defines these *funding streams* for *Testnet*:

| Stream | Numerator | Denominator | Start height | End height |
|---|---|---|---|---|
| FS_ZIP214_ECC | 7 | 100 | 1028500 | 2796000 |
| FS_ZIP214_ZF | 5 | 100 | 1028500 | 2796000 |
| FS_ZIP214_MG | 8 | 100 | 1028500 | 2796000 |

**Notes:**

- The *block heights* of *halvings* are different between *Testnet* and *Mainnet*, as a result of different *activation block heights* for the **Blossom** *network upgrade* (which changed the *block target spacing*). The end height of these *funding streams* corresponds to the second *halving* on each *network*.
- On *Testnet*, the *activation block height* of **Canopy** is before the first *halving*. Therefore, the consequence of the above rules for *Testnet* is that the amount sent to each **Zcash** Development Fund recipient address will initially (before *Testnet block height* 1116000) be double the number of currency units as the corresponding initial amount on *Mainnet*. This reduces to the same number of currency units as on *Mainnet*, from *Testnet block heights* 1116000 (inclusive) to 2796000 (exclusive).

## 7.11 Changes to the Script System

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

### 7.12 Bitcoin Improvement Proposals

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

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

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

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

The effect of [BIP-34] has been incorporated into the consensus rules (§ 7.1 *'Transaction Encoding and Consensus'* on p. 114). This excludes the *Mainnet* and *Testnet genesis blocks*, for which the "height in coinbase" was inadvertently omitted.

[BIP-13] applies with the changes to address version bytes described in § 5.6.1.1 *'Transparent Addresses'* on p. 105.

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

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

# 8 Differences from the Zerocash paper

## 8.1 Transaction Structure

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

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

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

Computation of *SIGHASH transaction hashes*, as described in § 4.10 *'SIGHASH Transaction Hashing'* on p. 45, was changed to clean up handling of an error case for SIGHASH_SINGLE, to remove the special treatment of `OP_CODESEPARATOR`, and to include **Zcash**-specific fields in the hash [ZIP-76].

## 8.2 Memo Fields

**Zcash** adds a *memo field* sent from the creator of a *JoinSplit description* to the recipient of each output *note*. This feature is described in more detail in § 5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 104.

## 8.3 Unification of Mints and Pours

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

- a "Mint" transaction takes value from *transparent* UTXOs as input and produces a new *shielded note* as output.

- a "Pour" transaction takes up to $\mathrm{N}^{\mathsf{old}}$ *shielded notes* as input, and produces up to $\mathrm{N}^{\mathsf{new}}$ *shielded notes* and a *transparent* UTXO as output.

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

[Pre-**Sapling**] In **Zcash**, the sequence of operations added to a *transaction* (see § 8.1 ***'Transaction Structure'*** on p. 132) consists only of *JoinSplit transfers*. A *JoinSplit transfer* is a Pour operation generalized to take a *transparent* UTXO as input, allowing *JoinSplit transfers* to subsume the functionality of Mints. An advantage of this is that a **Zcash** *transaction* that takes input from an UTXO can produce up to $\mathrm{N}^{\mathsf{new}}$ output *notes*, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the *JoinSplit transfer*: an unused (zero-value) input is indistinguishable from an input that takes value from a *note*.

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

[**Sapling** onward] In **Sapling**, there are still no "Mint" transactions. Instead of *JoinSplit transfers*, there are *Spend transfers* and *Output transfers*. These make use of *Pedersen value commitments* to represent the shielded values that are transferred. Because these commitments are additively homomorphic, it is possible to check that all *Spend transfers* and *Output transfers* balance; see § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47 for detail. This reduces the granularity of the circuit, allowing a substantial performance improvement (orthogonal to other **Sapling** circuit improvements) when the numbers of *shielded* inputs and outputs are significantly different. This comes at the cost of revealing the exact number of *shielded* inputs and outputs, but *dummy* (zero-valued) outputs are still possible.

## 8.4 Faerie Gold attack and fix

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

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are verified as valid by Receive (as defined in [BCGGMTV2014, Figure 2]), but only one of which can be spent.

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

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

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

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

It would be possible to address the attack by requiring that a recipient remember all of the $\rho$ 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 $\rho$, by making use of the fact that all of the *nullifiers* in *JoinSplit descriptions* that appear in a *valid block chain* must be distinct. This is true regardless of whether the *nullifiers* corresponded to real or *dummy notes* (see § 4.8.1 ***'Dummy Notes (Sprout)'*** on p. 43). The *nullifiers* are used as input to $\mathsf{hSigCRH}$ to derive a public value $\mathsf{h_{Sig}}$ which uniquely identifies the transaction, as described in § 4.3 ***'JoinSplit Descriptions'*** on p. 36. ($\mathsf{h_{Sig}}$ 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 $\rho$ value for each output *note* is then derived from a random private seed $\varphi$ and $\mathsf{h_{Sig}}$ using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of $\rho$ for each output *note* is enforced by § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54 in the *JoinSplit statement*.

[**Sprout**] Now even if the creator of a *JoinSplit description* does not choose $\varphi$ randomly, uniqueness of *nullifiers* and *collision resistance* of both $\mathsf{hSigCRH}$ and $\mathsf{PRF}^{\rho}$ will ensure that the derived $\rho$ 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 $\rho$. However, since the *nullifier* is computed as $\mathsf{PRF}^{\mathsf{nfSprout}}_{\mathsf{a_{sk}}}(\rho)$ or $\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}}(\rho\star)$ (for **Orchard**, see below); this is only possible if the adversary finds a collision across both inputs on $\mathsf{PRF}^{\mathsf{nfSprout}}$ or $\mathsf{PRF}^{\mathsf{nfSapling}}$, which is assumed to be infeasible — see § 4.1.2 ***'Pseudo Random Functions'*** on p. 21.

[**Sprout**] Crucially, "*nullifier* integrity" is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note* (§ 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54). 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 $\rho$ 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.

[**Sapling** onward] In **Sapling**, uniqueness of $\rho$ is ensured by making it dependent on the position of the *note commitment* in the **Sapling** *note commitment tree*. Specifically, $\rho = \mathsf{cm} + [\mathsf{pos}]\, \mathcal{J}^{\mathsf{Sapling}}$, where $\mathcal{J}^{\mathsf{Sapling}}$ is a generator independent of the generators used in $\mathsf{NoteCommit}^{\mathsf{Sapling}}$. Therefore, $\rho$ commits uniquely to the *note* and its position, and this commitment is *collision-resistant* by the same argument used to prove *collision resistance* of *Pedersen hashes*. Note that it is possible for two distinct **Sapling** *positioned notes* (having different $\rho$ values and *nullifiers*, but different *note positions*) to have the same *note commitment*, but this causes no security problem. Roadblock attacks are not possible because a given *note position* does not repeat for outputs of different *transactions* in the same *block chain*. Note that this depends on the fact that the value is bound by the *note commitment*: it could be the case that the adversary uses a *dummy note* that is not required to have a *note commitment* in the *note commitment tree* when it is spent. If this happens and the victim's *note* is not a *dummy*, the *note commitments* will differ and so will the *nullifiers*. If both *notes* are dummies, the adversary cannot know the inputs to the *note commitment* since they are generated at random for the victim's spend, regardless of the adversary's potential knowledge of viewing keys.

[**NU5** onward] In **Orchard**, the *nullifier* is computed as $\mathsf{DeriveNullifier}_{\mathsf{nk}}(\rho, \psi, \mathsf{cm})$ as described in § 4.16 ***'Note Commitments and*** on p. 53. This construction combines elliptic curve cryptography and the $\mathsf{Poseidon}$-based $\mathsf{PRF}^{\mathsf{nfOrchard}}$ in a way that, for privacy properties, aims to provide defence in depth against potential weaknesses in either. Resistance to Faerie Gold attacks, on the other hand, depends entirely on hardness of the Discrete Logarithm Problem. The $\rho$ value of a *note* created in a given *Action transfer* is obtained from the *nullifier* of the *note* spent in that *Action transfer*; this ensures (without any cryptographic assumption) that all $\rho$ values of *notes* added to the *note commitment tree* are unique. Then, the *nullifier* derivation can be considered as computing a modified Pedersen commitment on input that includes $\rho$, so that the binding property of that *commitment scheme* ensures that **Orchard** *nullifiers* will be unique. (Specifically, this is a Sinsemilla commitment with an additional term having base $\mathcal{K}^{\mathsf{Orchard}}$, truncated to its $x$-coordinate. The $x$-coordinate truncation cannot harm *collision resistance* because, assuming hardness of the

Discrete Logarithm Problem on the Pallas curve, the security proof in Theorem 5.4.2 on p. 76 covers the case where the additional term is added.) Roadblock attacks are not possible because $\rho$ does not repeat for *notes* in the *note commitment tree*, and by a corresponding argument to **Sapling** for *dummy notes*.

## 8.5 Internal hash collision attack and fix

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 $\mathsf{a_{pk}}$, $\mathsf{v}$, and $\rho$. 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 $\mathsf{a_{pk}}$ and $\rho$ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of $2^{64}$, to find distinct pairs $(\mathsf{a_{pk}}, \rho)$ and $(\mathsf{a_{pk}}', \rho')$ with colliding outputs of the truncated hash, and therefore the same *note commitment*. This would have allowed such an attacker to break the Balance property by double-spending *notes*, potentially creating arbitrary amounts of currency for themself [HW2016].

**Zcash** uses a simpler construction with a single hash evaluation for the commitment: SHA-256 for **Sprout**, and PedersenHash for **Sapling**. The motivation for the nested construction in **Zerocash** was to allow Mint transactions to be publically verified without requiring a *zk-SNARK proof* ([BCGGMTV2014, section 1.3, under step 3]). Since **Zcash** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers* (for **Sprout**), or *Spend transfers* and *Output transfers* (for **Sapling**), 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:** **Sprout** *note commitments* are not statistically hiding, so for **Sprout** notes, **Zcash** does not support the "everlasting anonymity" property described in [BCGGMTV2014, section 8.1], even when used as described in that section. While it is possible to define a statistically hiding, computationally binding commitment scheme for this use at a 128-bit security level, the overhead of doing so within the *JoinSplit statement* was not considered to justify the benefits.

[**Sapling** onward] In **Sapling**, *Pedersen commitments* are used instead of SHA256Compress. These commitments are statistically hiding, and so "everlasting anonymity" is supported for **Sapling** notes under the same conditions as in **Zerocash** (by the protocol, not necessarily by zcashd). Note that *diversified payment addresses* can be linked if the Discrete Logarithm Problem on the Jubjub curve can be broken.

## 8.6 Changes to PRF inputs and truncation

The format of inputs to the *PRFs* instantiated in § 5.4.2 ***'Pseudo Random Functions'*** on p. 79 has changed relative to **Zerocash**. There is also a requirement for another *PRF*, $\mathsf{PRF}^{\rho}$, which must be domain-separated from the others.

In the **Zerocash** protocol, $\rho_i^{\mathsf{old}}$ 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, $\mathsf{h_{Sig}}$ is truncated from 256 to 253 bits in the input to $\mathsf{PRF}^{\mathsf{pk}}$. These truncations are not taken into account in the security proofs.

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

In more detail:

- In the argument relating $\mathbf{H}$ and $\Game_2$, it is stated that in $\Game_2$, "for each $i \in \{1, 2\}, \mathsf{sn}_i := \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{a_{sk}}}(\rho)$ for a random (and not previously used) $\rho$". 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 $\rho$ 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 $\rho$ 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 $\mathsf{CRH}$ provided that $\mathsf{pk_{sig}}$ is chosen randomly, but no formal argument for this is presented.

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

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

[**Sprout**] For resistance to Faerie Gold attacks as described in § 8.4 ***'Faerie Gold attack and fix'*** on p. 133, **Zcash** depends on *collision resistance* of hSigCRH and $\mathsf{PRF}^{\rho}$ (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 $\rho$.

[**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 $\mathsf{PRF^{addr}}$, $\mathsf{PRF^{nfSprout}}$, and $\mathsf{PRF^{pk}}$, and to $\varphi$ (which does not exist in **Zerocash**) for $\mathsf{PRF}^{\rho}$, and so those values have been reduced to 252 bits. This is preferable to requiring reasoning about truncation, and 252 bits is quite sufficient for security of these cryptovalues.

**Sapling** uses *Pedersen hashes* and BLAKE2s where **Sprout** used SHA256Compress. *Pedersen hashes* can be efficiently instantiated for arbitrary input lengths. BLAKE2s has an input block size of 512 bits, and uses a finalization flag rather than padding of the last input block; it also supports domain separation via a personalization parameter distinct from the input. Therefore, there is no need for truncation in the inputs to any of these hashes. Note however that the ***output*** of $\mathsf{CRH^{ivk}}$ is truncated, requiring a security assumption on BLAKE2s truncated to 251 bits (see § 5.4.1.5 '$\mathsf{CRH^{ivk}}$ ***Hash Function'*** on p. 71).

## 8.7 In-band secret distribution

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

The motivations for this change were as follows:

- The **Zerocash** paper did not specify the curve to be used. We believe that Curve25519 has significant side-channel resistance, performance, implementation complexity, and robustness advantages over most other available curve choices, as explained in [Bernstein2006]. For **Sapling**, the Jubjub curve was designed according to a similar design process following the "Safe curves" criteria [BL-SafeCurves] [Hopwood2018]. This retains Curve25519's advantages while keeping *shielded payment address* sizes short, because the same *public key* material supports both encryption and spend authentication.
- ECIES permits many options, which were not specified. There are at least –counting conservatively– 576 possible combinations of options and algorithms over the four standards (ANSI X9.63, IEEE Std 1363a-2004, ISO/IEC 18033-2, and SEC 1) that define ECIES variants [MAEÁ2010].
- Although the **Zerocash** paper states that ECIES satisfies *key privacy* (as defined in [BBDP2001]), it is not clear that this holds for all curve parameters and key distributions. For example, if a group of non-prime order is used, the distribution of ciphertexts could be distinguishable depending on the order of the points representing the ephemeral and recipient *public keys*. Public key validity is also a concern. Curve25519 (and Jubjub) key agreement is defined in a way that avoids these concerns due to the curve structure and the "clamping" of *private keys* (or explicit cofactor multiplication and point validation for **Sapling**).
- Unlike the DHAES/DHIES proposal on which it is based [ABR1999], ECIES does not require a representation of the sender's *ephemeral public key* to be included in the input to the KDF, which may impair the security properties of the scheme. (The Std 1363a-2004 version of ECIES [IEEE2004] has a "DHAES mode" that allows this, but the representation of the key input is underspecified, leading to incompatible implementations.)

The scheme we use for **Sprout** has both the ephemeral and recipient *public key* encodings –which are unambiguous for Curve25519– and also $\mathsf{h_{Sig}}$ and a nonce as described below, as input to the KDF. For **Sapling**, it is only possible to include the ephemeral public key encoding, but this is sufficient to retain the original security properties of DHAES. Note that being able to break the Elliptic Curve Diffie–Hellman Problem on Curve25519 or Jubjub (without breaking AEAD_CHACHA20_POLY1305 as an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the *transmitted note(s) ciphertext* unless $\mathsf{pk_{enc}}$ is known or guessed.

- [**Sprout**] The KDF also takes a public seed $\mathsf{h_{Sig}}$ as input. This can be modeled as using a different "randomness extractor" for each *JoinSplit transfer*, which limits degradation of security with the number of *JoinSplit transfers*. This facilitates security analysis as explained in [DGKM2011] — see section 7 of that paper for a security proof that can be applied to this construction under the assumption that single-block BLAKE2b-256 is a "*weak PRF*". Note that $\mathsf{h_{Sig}}$ is authenticated, by the *zk-SNARK proof*, as having been chosen with knowledge of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N}^{\mathsf{old}}}$, so an adversary cannot modify it in a ciphertext from someone else's transaction for use in a chosen-ciphertext attack without detection. (In **Sapling**, there is no equivalent to $\mathsf{h_{Sig}}$, but the *binding signature* and *spend authorization signatures* prevent such modifications.)
- [**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**, $\mathsf{a_{sk}}$ is $252$ bits, and $\mathsf{sk_{enc}}$ is no shorter than $\mathsf{a_{sk}}$.

## 8.8 Omission in Zerocash security proof

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

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

The error is in the proof of Balance in [BCGGMTV2014, Appendix D.3]. For the "$\mathcal{A}$ violates Condition I" case, the proof says:

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

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

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

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

The proof can be straightforwardly repaired. The intuition is that we can rely on *collision resistance* of $\mathsf{PRF}^{\mathsf{addr}}$ (on both its arguments) to argue that distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}$, together with constraint 1(b) of the *JoinSplit statement* (see § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54), implies distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},2}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

## 8.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a}_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}}), \mathsf{v}, \rho, \mathsf{rcm}, \mathsf{s}, \mathsf{cm})$, 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 $\mathsf{s}$, 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** $\mathtt{POUR}$ *statement* or the **Zcash** *JoinSplit statement*. $\mathsf{cm}$ can be computed from the other fields. (The definition of *notes* for **Sapling** is different again.)
- The length of proof encodings given in the paper is $288$ bytes. [**Sprout**] This differs from the $296$ bytes specified in § 5.4.10.1 'BCTV14' on p. 102, 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*. In **Sapling**, a customized encoding is used for BLS12-381 points in Groth16 proofs to minimize length.
- The range of monetary values differs. In **Zcash** this range is $\{0 .. \mathsf{MAX\_MONEY}\}$, while in **Zerocash** it is $\{0 .. 2^{\ell_{\mathsf{value}}}-1\}$. (The *JoinSplit statement* still only directly enforces that the sum of amounts in a given *JoinSplit transfer* is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

# 9 Acknowledgements

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

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

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

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

The Faerie Gold attack was found by Zooko Wilcox; subsequent analysis of variations on the attack was performed by Daira Hopwood and Sean Bowe. The internal hash collision attack was found by Taylor Hornby. The error in the **Zerocash** proof of Balance relating to *collision resistance* of $\mathsf{PRF}^{\mathsf{addr}}$ was found by Daira Hopwood. The errors in the proof of Ledger Indistinguishability mentioned in § 8.6 ***'Changes to PRF inputs and truncation'*** on p. 135 were also found by Daira Hopwood.

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

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

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

The observation in § 5.4.1.6 '$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$ *and* $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ *Hash Functions*' on p. 71 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 § B.2 'Groth16 *batch verification*' on p. 194, by applying them to BCTV14.

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

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

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

Many of the ideas used in **Zcash** —including the use of zero-knowledge proofs to resolve the tension between privacy and auditability, Merkle trees over note commitments (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

### 2021.1.20 2021-03-18

- Correct the description of `length` in § 5.6.4.1 *'Unified Payment Addresses'* on p. 109.
- Correct the type signature of $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ in § 4.1.1 *'Hash Functions'* on p. 20.
- Remove support for building the **Sprout**-only specification (`sprout.pdf`).
- Remove magenta highlighting of differences from **Zerocash**.

### 2021.1.19 2021-03-17

- Correct the range of input to $\mathsf{ValueCommit}^{\mathsf{Orchard}}$ in the *Action statement*, and the corresponding security argument in § 4.14 *'Balance and Binding Signature (Orchard)'* on p. 50.
- Update the consensus rules that prevent trivial transactions (with no inputs or outputs) to take into account *Action transfers* in the v5 *transaction* format.
- Make $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$ total, by replacing an output of $\mathcal{O}_{\mathbb{P}}$ with another base.
- Fix a type error in the non-normative note at the end of § 5.4.8.4 *'Sinsemilla commitments'* on p. 90.

**2021.1.18** 2021-03-17

- Define *unified payment addresses* in place of the *Bech32* form of **Orchard** *shielded payment addresses*.
- Remove **Sprout**-specific fields from the v5 *transaction* format.
- The $\rho$ value for an **Orchard** output *note* was incorrectly described as being derived from $\mathsf{rseed}$, instead of being set to the nullifier from the same *Action description* as intended.
- The $\psi$ value is now derived using the $\mathsf{PRF}^{\mathsf{expand}}$ input [9], instead of [10].
- Correct a note about the range of the Merkle hash inputs in § 4.17.4 *'Action Statement (**Orchard**)'* on p. 57.
- Correct the validity condition for $\mathsf{ak}$ in § 5.6.4.4 *'**Orchard** Full Viewing Keys'* on p. 111.
- Add a definition for $\mathcal{K}^{\mathsf{Orchard}}$ in § 4.16 *'Note Commitments and Nullifiers'* on p. 53.
- Correct the number of full and partial rounds for $\mathsf{Poseidon}$.
- Add a note explaining the origin of the $2^{65}$ constant in the definition of $\mathsf{PoseidonHash}$.
- The subgroup check added to § 4.19.3 *'Decryption using a Full Viewing Key (**Sapling** and **Orchard**)'* on p. 63 for **Sapling** in version 2021.1.17 was applied to the wrong variable ($\mathsf{g_d}$, when it should have been $\mathsf{pk_d}$), despite being described correctly in the Change History entry below.

**2021.1.17** 2021-03-15

- Draft **NU5** specification.
- In the consensus rule that a *transaction* with one or more *transparent* inputs from *coinbase transactions* **MUST** have no *transparent* outputs, explicitly say that inputs from *coinbase transactions* include *funding stream* outputs.
- The definition of an abstraction function in § 4.1.9 *'Represented Group'* on p. 29 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.)
- Correct Theorem 5.4.5 on p. 89, which was proving the wrong thing. It needs to prove that $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ does not return $\mathsf{Uncommitted}^{\mathsf{Sapling}}$, but was previously proving that $\mathsf{PedersenHash}$ does not return that value.
- The note about non-canonical encodings in § 5.4.9.3 'Jubjub' on p. 94 gave incorrect values for the encodings of the point of order 2, by omitting a $q_{\mathbb{J}}$ term.
- The specification of decryption in § 4.19.3 *'Decryption using a Full Viewing Key (**Sapling** and **Orchard**)'* on p. 63 differed from its implementation in zcashd, in two respects:
  - The specification had a type error in that it failed to check whether $\mathsf{abst}_{\mathbb{J}}(\mathsf{pk}\star_{\mathsf{d}}) = \bot$, which is needed in order for its use as input to $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}$ to be well-typed.
  - The specification did not require $\mathsf{pk_d}$ to be in the subgroup $\mathbb{J}^{(r)}$, while the implementation in zcashd did. This check is not needed for security; however, since Jubjub public keys are normally of type $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup}$, we change the specification to match zcashd.
- Correct the procedure for generating *dummy* **Sapling** *notes* in § 4.8.2 *'Dummy Notes (**Sapling**)'* on p. 43.
- Add a note in § 5.4.10.1 'BCTV14' on p. 102 describing conditions under which an implementation that checkpoints on **Sapling** can omit verifying BCTV14 proofs.
- Rename "hash extractor" to *coordinate extractor*. This is a more accurate name since it is also used on commitments.
- Rename `char` to `byte` in field type declarations.

**2021.1.16** 2021-01-11

- Add macros and `Makefile` support for building the **NU5** draft specification.
- Clarify the encoding of *block heights* for the "height in coinbase" rule. The description of this rule has also moved from § 7.6 on p. 122 to § 7.1 *'Transaction Encoding and Consensus'* on p. 114.
- Include the activation dates of **Heartwood** and **Canopy** in § 6 *'Network Upgrades'* on p. 112.
- Section links in the **Heartwood** and **Canopy** versions of the specification now go to the correct document URL.
- Attempt to improve search and cut-and-paste behaviour for ligatures in some PDF readers.

**2020.1.15** 2020-11-06

- Add a missing consensus rule that has always been implemented in zcashd: there must be at least one *transparent output*, *Output description*, or *JoinSplit description* in a *transaction*.
- Add a consensus rule that the (zero-valued) *coinbase transaction* output of the *genesis block* cannot be spent.
- Define ***Sprout*** *chain value pool balance* and ***Sapling*** *chain value pool balance*, and include consensus rules from [ZIP-209].
- Correct the **Sapling** *note* decryption algorithms:
  - `ephemeralKey` is kept as a byte sequence rather than immediately converted to a curve point; this matters because of *non-canonical* encoding.
  - The representation of $\mathsf{pk_d}$ in a *note plaintext* may also be *non-canonical* and need not be in the prime subgroup.
  - Move checking of $\mathsf{cm}_u$ in decryption with $\mathsf{ivk}$ to the end of the algorithm, to more closely match the implementation.
  - The note about decryption of outputs in *mempool transactions* should have been normative.
- 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

- 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.)
- Fix a type error in the output of $\mathsf{PRF}^{\mathsf{nfSapling}}$; a **Sapling** *nullifier* is a sequence of 32 bytes, not a bit sequence.
- Correct an off-by-one in an expression used in the definition of $c$ in § 5.4.1.7 *'Pedersen Hash Function'* on p. 72 (this does not change the value of $c$).

**2020.1.13** 2020-08-11

- Rename the type of **Sapling** *transmission keys* from $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$ to $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup}$. This type is defined as $\mathbb{J}^{(r)}$, which reflects the implementation in zcashd (subject to the next point below); it was never enforced that a *transmission key* ($\mathsf{pk_d}$) cannot be $\mathcal{O}_{\mathbb{J}}$.
- Add a non-normative note saying that zcashd does not fully conform to the requirement to treat *transmission keys* not in $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeSubgroup}$ as invalid when importing *shielded payment addresses*.
- Set CanopyActivationHeight for *Testnet*.
- Modify the tables and notes in § 7.10.1 ***'ZIP 214 Funding Streams'*** on p. 131 to reflect changes in [ZIP-214].
- Updates to reflect [ZIP-211]: add a consensus rule on $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ in § 4.3 ***'JoinSplit Descriptions'*** on p. 36, and a rule about node and wallet support for sending to **Sprout** addresses in § 4.7.1 ***'Sending Notes (Sprout)'*** on p. 40.
- Refine the domain of HeightForHalving from $\mathbb{N}$ to $\mathbb{N}^+$.
- Make Halving(height) return $0$ (rather than $-1$) for height $<$ SlowStartShift. This has no effect on consensus since the Halving function is not used in that case, but it makes the definition match the intuitive meaning of the function.
- Rename sections under § 7 ***'Consensus Changes from Bitcoin'*** on p. 114 to clarify that these sections do not only concern encoding, but also consensus rules.
- Make the **Canopy** specification the default.

**2020.1.12** 2020-08-03

- Include SHA-512 in § 5.4.1.1 'SHA-256, SHA-256d, SHA256Compress, ***and*** SHA-512 ***Hash Functions'*** on p. 68.
- Add a reference to [BCCGLRT2014] in § 4.1.13 ***'Zero-Knowledge Proving System'*** on p. 31.
- Use $\mathsf{abstBytes}_{\mathsf{Ed25519}}$ and $\mathsf{reprBytes}_{\mathsf{Ed25519}}$ for conversions in § B.3 'Ed25519 ***batch validation'*** on p. 196, and fix a missing requirement that $S_j < \ell$ for all signatures.

**2020.1.11** 2020-07-13

- Change instances of "the production network" to "*Mainnet*", and "the test network" to *Testnet*. This follows the terminology used in ZIPs.
- Update stale references to **Bitcoin** documentation.
- Add changes for [ZIP-207] and [ZIP-214].

**2020.1.10** 2020-07-05

- Corrections to a note in § 5.4.6 'Ed25519' on p. 83.

**2020.1.9** 2020-07-05

- Add § 3.12 ***'Mainnet and Testnet'*** on p. 20.
- Acknowledge Jane Lusby and Teor.
- Precisely specify the encoding and decoding of Ed25519 points.
- Correct an error introduced in 2020.1.8; "$-\mathcal{O}_{\mathbb{J}}$" was incorrectly used when the point $(0, -1)$ on Jubjub was meant.
- Precisely specify the conversion from a bit sequence in $\mathsf{abst}_{\mathbb{J}}$.

**2020.1.8** 2020-07-04

- Add Ying Tong Lai and Kris Nuttycombe as **Zcash** protocol designers.
- Change the specification of $\mathsf{abst}_{\mathbb{J}}$ in § 5.4.9.3 **'Jubjub'** on p. 94 to match the implementation.
- Repair the argument for $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$ being usable as a *random oracle*, which previously depended on $\mathsf{abst}_{\mathbb{J}}$ being injective.
- In RedDSA verification, clarify that $\underline{R}$ used as part of the input to $\mathsf{H}^{\circledast}$ must be exactly as encoded in the signature.
- Specify that *shielded outputs* of *coinbase transactions* **MUST** use v2 *note plaintexts* after **Canopy** activation.
- Correct a bug in § 4.19.3 ***'Decryption using a Full Viewing Key (Sapling and Orchard)'*** on p. 63: $\mathsf{esk}$ is only to be checked against $\mathsf{ToScalar}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([4]))$ when $\mathsf{leadByte} \neq \mathtt{0x01}$.

**2020.1.7** 2020-06-26

- Delete some 'new' superscripts that only added notational clutter.
- Add an explicit lead byte field to **Sprout** *note plaintexts*, and clearly specify the error handling when it is invalid.
- Define a **Sapling** *note plaintext lead byte* as having type $\mathbb{B}^{\mathbb{Y}}$ (so that decoding to a *note plaintext* always succeeds, and error handling is more explicit).
- Fix a sign error in the fixed-base term of the batch validation equation in § B.1 **'RedDSA *batch validation*'** on p. 193.
- Fix a sign error in the fixed-base term of the batch validation equation in § B.3 **'Ed25519 *batch validation*'** on p. 196.

**2020.1.6** 2020-06-17

- Incorporate changes to **Sapling** *note* encryption from [ZIP-212].
- Correct an error in the specification of Ed25519 *validating keys*: they should not have been specified to be checked against $\mathsf{ExcludedPointEncodings}$, since libsodium v1.0.15 does not do so.
- Incorporate Ed25519 changes for **Canopy** from [ZIP-215].
- Add Appendix § B.3 **'Ed25519 *batch validation*'** on p. 196.
- Consistently use "validating" for signatures and "verifying" for proofs.
- Use the symbol $\sqrt[+]{\bullet}$ for positive square root.

**2020.1.5** 2020-06-02

- Reference [ZIP-173] instead of BIP 173.
- Mark more index entries as definitions.

**2020.1.4** 2020-05-27

- Reference [BIP-32] and [ZIP-32] when describing keys and their encodings.
- Network Upgrade 4 has been given the name **Canopy**.
- Reference [ZIP-211], [ZIP-212], and [ZIP-215] for the **Canopy** upgrade.
- Improve LaTeX portability of this specification.

### **2020.1.3** 2020-04-22

- Correct a wording error transposing *transparent inputs* and *transparent outputs* in § 4.12 ***'Balance (Sprout)'*** on p. 47.
- Minor wording clarifications.
- Reference [ZIP-251], [ZIP-207], and [ZIP-214] for the **Canopy** upgrade.

### **2020.1.2** 2020-03-20

- The implementation of **Sprout** Ed25519 signature validation in zcashd differed from what was specified in § 5.4.6 **'Ed25519'** on p. 83. The specification has been changed to match the implementation.
- Add consensus rules for **Heartwood**.
- Remove "pvc" `Makefile` targets.
- Make the **Heartwood** specification the default.
- Add macros and `Makefile` support for building the **Canopy** specification.

### **2020.1.1** 2020-02-13

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

### **2020.1.0** 2020-02-06

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

### **2019.0.9** 2019-12-27

- No changes to **Sprout** or **Sapling**.
- Specify the height at which **Blossom** activated.
- Add **Blossom** to § 6 ***'Network Upgrades'*** on p. 112.
- Add a non-normative note giving the explicit value of FoundersRewardLastBlockHeight.
- Clarify the effect of **Blossom** on *SIGHASH transaction hashes*.
- `Makefile` updates for **Heartwood**.

### **2019.0.8** 2019-09-24

- Fix a typo in the generator $\mathcal{P}_{\mathbb{S}_1}$ in § 5.4.9.2 'BLS12-381' on p. 93 found by magrady.
- Clarify the type of $\mathsf{v}^{\mathsf{new}}$ in § 4.7.2 ***'Sending Notes (Sapling)'*** on p. 41.

**2019.0.7** 2019-09-24

- Fix a discrepancy in the number of constraints for BLAKE2s found by QED-it.
- Fix an error in the expression for $\Delta$ in § A.3.3.9 ***'Pedersen hash'*** on p. 183, and add acknowledgement to Kobi Gurkan.
- Fix a typo in § 4.9 ***'Merkle Path Validity'*** on p. 45 and add acknowledgement to Weikeng Chen.
- Update references to ZIPs and to the Electric Coin Company blog.
- `Makefile` improvements to suppress unneeded output.

**2019.0.6** 2019-08-23

- No changes to **Sprout** or **Sapling**.
- Replace dummy **Blossom** *activation block height* with the *Testnet* height, and a reference to [ZIP-206].

**2019.0.5** 2019-08-23

- Note the change to the minimum-difficulty threshold time on *Testnet* for **Blossom**.
- Correct the packing of $\mathsf{nf}^{\mathsf{old}}$ into input elements in § A.4 ***'The Sapling Spend circuit'*** on p. 190.
- Add an epigraph from [Carroll1876] to the start of § 5.4.9.3 **'Jubjub'** on p. 94.
- Clarify how the constant $c$ in § 5.4.1.7 ***'Pedersen Hash Function'*** on p. 72 is obtained.
- Add a footnote that zcashd uses [ZIP-32] *extended spending keys* instead of the derivation from $\mathsf{sk}$ in § 3.1 ***'Payment Addresses and Keys'*** on p. 12.
- Remove "optimized" `Makefile` targets (which actually produced a larger PDF, with TeXLive 2019).
- Remove "html" `Makefile` targets.
- Make the **Blossom** spec the default.

**2019.0.4** 2019-07-23

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

**2019.0.3** 2019-07-08

- Experimental support for building using LuaTEX and XeTEX.
- Add an **Index**.

**2019.0.2** 2019-06-18

- Correct a misstatement in the security argument in § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47: binding for a commitment scheme does not imply that the commitment determines its randomness. The rest of the security argument did not depend on this; it is simpler to rely on knowledge soundness of the Spend and Output proofs.
- Give a definition for *complete twisted Edwards elliptic curves* in § 5.4.9.3 **'Jubjub'** on p. 94.
- Clarify that Theorem 5.4.5 on p. 89 depends on the parameters of the Jubjub curve.
- Ensure that this document builds correctly and without missing characters on recent versions of TEXLive.
- Update the `Makefile` to use Ghostscript for PDF optimization.
- Ensure that hyperlinks are preserved, and available as "Destination names" in URL fragments and links from other PDF documents.

**2019.0.1** 2019-05-20

- No changes to **Sprout** or **Sapling**.
- Minor fix to the list of integer constants in § 2 *'Notation'* on p. 9.
- Use IsBlossomActivated in the definition of FounderAddressAdjustedHeight for consistency.

**2019.0.0** 2019-05-01

- Fix a specification error in the *Founders' Reward* calculation during the slow start period.
- Correct an inconsistency in difficulty adjustment between the spec and zcashd implementation for the first PoWAveragingWindow − 1 *blocks* of the *block chain*. This inconsistency was pointed out by NCC Group in their **Blossom** specification audit.
- Revert changes for *funding streams* from Withdrawn ZIP 207.

**2019.0-beta-39** 2019-04-18

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

**2019.0-beta-38** 2019-04-18

- Update the following sections to match the current draft of [ZIP-208]:
  - § 7.7.3 *'Difficulty adjustment'* on p. 125
  - § 7.8 *'Calculation of Block Subsidy, Funding Streams, and Founders' Reward'* on p. 127
- Specify *funding streams*, along with the draft *funding streams* defined in the current draft of ZIP 207.
- Update the following sections to match the current draft of ZIP 207:
  - § 3.10 *'Block Subsidy, Funding Streams, and Founders' Reward'* on p. 19
  - § 3.11 *'Coinbase Transactions'* on p. 20
  - § 7.8 *'Calculation of Block Subsidy, Funding Streams, and Founders' Reward'* on p. 127
  - § 7.9 *'Payment of Founders' Reward'* on p. 128
- Correct the generators $\mathcal{P}_{\mathbb{S}_1}$ and $\mathcal{P}_{\mathbb{S}_2}$ for BLS12-381.
- Update `README.rst` to include `Makefile` targets for **Blossom**.
- `Makefile` updates:
  - Fix a typo for the `pvcblossom` target.
  - Update the pinned `git` hashes for `sam2p` and `pdfsizeopt`.

**2019.0-beta-37** 2019-02-22

- The rule that miners **SHOULD NOT** mine *blocks* that chain to other *blocks* with a *block version number* greater than 4, has been removed. This is because such *blocks* (mined nonconformantly) exist in the current *Mainnet* consensus *block chain*.
- Clarify that *Equihash* is based on a ***variation*** of the Generalized Birthday Problem, and cite [AR2017].
- Update reference [BGG2017] (previously [BGG2016]).
- Clarify which transaction fields are added by **Overwinter** and **Sapling**.
- Correct the rule about when a *transaction* is permitted to have no *transparent* inputs.
- Explain the differences between the system in [Groth2016] and what we refer to as Groth16.

- Reference Mary Maller's security proof for Groth16 [Maller2018].
- Correct [BGM2018] to [BGM2017].
- Fix a typo in § B.2 *'Groth16 batch verification'* on p. 194 and clarify the costs of Groth16 batch verification.
- Add macros and `Makefile` support for building the **Blossom** specification.

**2019.0-beta-36** 2019-02-09

- Correct isis agora lovecruft's name.

**2019.0-beta-35** 2019-02-08

- Cite [Gabizon2019] and acknowledge Ariel Gabizon.
- Correct [SBB2019] to [SWB2019].
- The [Gabizon2019] vulnerability affected Soundness of BCTV14 as well as Knowledge Soundness.
- Clarify the history of the [Parno2015] vulnerability and acknowledge Bryan Parno.
- Specify the difficulty adjustment change that occurred on *Testnet* at *block height* 299188.
- Add Eirik Ogilvie-Wigley and Benjamin Winston to acknowledgements.
- Rename zk-SNARK Parameters sections to be named according to the proving system (BCTV14 or Groth16), not the shielded protocol construction (**Sprout** or **Sapling**).
- In § 6 *'Network Upgrades'* on p. 112, say when **Sapling** activated.

**2019.0-beta-34** 2019-02-05

- Disclose a security vulnerability in BCTV14 that affected **Sprout** before activation of the **Sapling** *network upgrade* (see § 5.4.10.1 'BCTV14' on p. 102).
- Rename PHGR13 to BCTV2014.
- Rename reference [BCTV2015] to [BCTV2014a], and [BCTV2014] to [BCTV2014b].

**2018.0-beta-33** 2018-11-14

- Complete § A.4 *'The **Sapling** Spend circuit'* on p. 190.
- Add § A.5 *'The **Sapling** Output circuit'* on p. 192.
- Change the description of window lookup in § A.3.3.7 *'Fixed-base Affine-ctEdwards scalar multiplication'* on p. 181 to match sapling-crypto.
- Describe 2-bit window lookup with conditional negation in § A.3.3.9 *'Pedersen hash'* on p. 183.
- Fix or complete various calculations of constraint costs.
- Adjust the notation used for scalar multiplication in Appendix A to allow bit sequences as scalars.

**2018.0-beta-32** 2018-10-24

- Correct the input to $\mathsf{H}^{\circledast}$ used to derive the nonce $r$ in $\mathsf{RedDSA.Sign}$, from $T \,||\, M$ to $T \,||\, \underline{\mathsf{vk}} \,||\, M$. This matches the sapling-crypto implementation; the specification of this input was unintentionally changed in version 2018.0-beta-20.
- Clarify the description of the Merkle path check in § A.3.4 *'Merkle path check'* on p. 186.

**2018.0-beta-31** 2018-09-30

- Correct some uses of $r_{\mathbb{J}}$ that should have been $r_{\mathbb{S}}$ or $q$.
- Correct uses of $\mathsf{LEOS2IP}_{\ell}$ in RedDSA.Validate and RedDSA.BatchValidate to ensure that $\ell$ is a multiple of $8$ as required.
- Minor changes to avoid clashing notation for Edwards curves $E_{\mathsf{Edwards}(a,d)}$, *Montgomery curves* $E_{\mathsf{Mont}(A,B)}$, and extractors $\mathcal{E}_{\mathcal{A}}$.
- Correct a use of $\mathbb{J}$ that should have been $\mathbb{M}$ in the proof of Theorem A.3.4 on p. 179, and make a minor tweak to the theorem statement ($k_2 \neq \pm k_1$ instead of $k_1 \neq \pm k_2$) to make the contradiction derived by the proof clearer.
- Clarify notation in the proof of Theorem A.3.3 on p. 179.
- Address some of the findings of the QED-it report:
  - Improved cross-referencing in § 5.4.1.7 ***'Pedersen Hash Function'*** on p. 72.
  - Clarify the notes concerning domain separation of prefixes in § 5.4.1.3 'MerkleCRH$^{\mathsf{Sapling}}$ ***Hash Function'*** on p. 70 and § 5.4.8.2 ***'Windowed Pedersen commitments'*** on p. 88.
  - Correct the statement and proof of Theorem A.3.2 on p. 179.
- Add the QED-it report to the acknowledgements.

**2018.0-beta-30** 2018-09-02

- Give an informal security argument for Unlinkability of *diversified payment addresses* based on reduction to *key privacy* of ElGamal encryption, for which a security proof is given in [BBDP2001]. (This argument has gaps which will be addressed in a future version.)
- Add a reference to [BGM2017] for the **Sapling** *zk-SNARK* parameters.
- Write § A.4 ***'The Sapling Spend circuit'*** on p. 190 (draft).
- Add a reference to the ristretto_bulletproofs design notes [Dalek-notes] for the synthetic blinding factor technique.
- Ensure that the constraint costs in § A.3.3.1 ***'Checking that Affine-ctEdwards coordinates are on the curve'*** on p. 178 and § A.3.3.6 ***'Affine-ctEdwards nonsmall-order check'*** on p. 181 accurately reflect the implementation in sapling-crypto.
- Minor correction to the non-normative note in § A.3.2.2 ***'Range check'*** on p. 176.
- Clarify the non-normative note in § 4.1.8 ***'Commitment'*** on p. 27 about the definitions of ValueCommit$^{\mathsf{Sapling}}$.Output and NoteCommit$^{\mathsf{Sapling}}$.Output.
- Clarify that the signer of a *spend authorization signature* is supposed to choose the *spend authorization randomizer*, $\alpha$, itself. Only step 4 in the procedure in § 4.15 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 52 may securely be delegated.
- Add a non-normative note to § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85 explaining that RedDSA key randomization may interact with other uses of additive properties of Schnorr keys.
- 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

- Finish § A.3.2.2 ***'Range check'*** on p. 176.
- Change § A.3.7 ***'BLAKE2s hashes'*** on p. 187 to correct the constraint count and to describe batched equality checks performed by the sapling-crypto implementation.

**2018.0-beta-28** 2018-08-14

- Finish § A.3.7 *'BLAKE2s hashes'* on p. 187.
- Minor corrections to § A.3.3.8 *'Variable-base Affine-ctEdwards scalar multiplication'* on p. 182.

**2018.0-beta-27** 2018-08-12

- Notational changes:
  - Use a superscript $^{(r)}$ to mark the subgroup order, instead of a subscript.
  - Use $\mathbb{G}^{(r)*}$ for the set of $r_{\mathbb{G}}$-order points in $\mathbb{G}$.
  - Mark the subgroup order in pairing groups, e.g. use $\mathbb{G}_1^{(r)}$ instead of $\mathbb{G}_1$.
  - Make the bit-representation indicator $\star$ an affix instead of a superscript.
- Clarify that when validating a Groth16 proof, it is necessary to perform a subgroup check for $\pi_A$ and $\pi_C$ as well as for $\pi_B$.
- Correct the description of Groth16 batch verification to explicitly take account of how verification depends on *primary inputs*.
- 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

- Add § B.2 'Groth16 *batch verification'* on p. 194.

**2018.0-beta-25** 2018-08-05

- Add the hashes of parameter files for **Sapling**.
- Add cross references for parameters and functions used in RedDSA batch validation.
- `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

- Add a missing consensus rule for version 4 *transactions*: if there are no **Sapling** Spends or Outputs, then `valueBalanceSapling` **MUST** be 0.

**2018.0-beta-23** 2018-07-27

- Update RedDSA validation to use cofactor multiplication. This is necessary in order for the output of batch validation to match that of unbatched validation in all cases.
- Add § B.1 'RedDSA *batch validation'* on p. 193.

**2018.0-beta-22** 2018-07-18

- Update § 6 *'Network Upgrades'* on p. 112 to take account that **Overwinter** has activated.
- The recommendation for *transactions* without *JoinSplit descriptions* to be version 1 applies only before **Overwinter**, not before **Sapling**.
- Complete the proof of Theorem A.3.5 on p. 184.
- Add a note about redundancy in the nonsmall-order checking of $\mathsf{rk}$.
- Clarify the use of $\mathsf{cv}^{\mathsf{new}}$ and $\mathsf{cm}^{\mathsf{new}}$, and the selection of *outgoing viewing key*, in sending Sapling notes.
- Delete the description of optimizations for the affine twisted Edwards nonsmall-order check, since the **Sapling** circuit does not use them. Also clarify that some other optimizations are not used.

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

- Remove the consensus rule "If `nJoinSplit` $> 0$, the *transaction* **MUST NOT** use *SIGHASH types* other than SIGHASH_ALL.", which was never implemented.
- Add section on signature hashing.
- Briefly describe the changes to computation of *SIGHASH transaction hashes* in **Sprout**.
- Clarify that interstitial *treestates* form a tree for each *transaction* containing *JoinSplit descriptions*.
- Correct the description of P2PKH addresses in § 5.6.1.1 ***'Transparent Addresses'*** on p. 105 — they use a hash of a compressed, not an uncompressed ECDSA key representation.
- Clarify the wording of the caveat[4] 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.
- Use the more precise subgroup types $\mathbb{G}^{(r)}$ and $\mathbb{J}^{(r)}$ in preference to $\mathbb{G}$ and $\mathbb{J}$ where applicable.
- Change the types of *auxiliary inputs* to the *Spend statement* and *Output statement*, to be more faithful to the implementation.
- Rename the `cm` field of an *Output description* to `cmu`, reflecting the fact that it is a Jubjub curve $u$-coordinate.
- Add explicit consensus rules that the `anchorSapling` field of a *Spend description* and the `cmu` field of an *Output description* must be canonical encodings.
- Enforce that $\mathsf{esk}$ in `outCiphertext` is a canonical encoding.
- Add consensus rules that $\mathsf{cv}$ in a *Spend description*, and $\mathsf{cv}$ and $\mathsf{epk}$ in an *Output description*, are not of small order. Exclude $0$ from the range of $\mathsf{esk}$ when encrypting **Sapling** notes.
- Add a consensus rule that `valueBalanceSapling` is in the range $\{-\mathsf{MAX\_MONEY} \,..\, \mathsf{MAX\_MONEY}\}$.
- Enforce stronger constraints on the types of key components $\mathsf{pk_d}$, $\mathsf{ak}$, and $\mathsf{nk}$.
- Correct the conformance rule for `fOverwintered` (it must not be set before **Overwinter** has activated, not before **Sapling** has activated).
- Correct the argument that $\mathsf{v}^*$ is in range in § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47.
- Correct an error in the algorithm for $\mathsf{RedDSA.Validate}$: the *validating key* $\mathsf{vk}$ is given directly to this algorithm and should not be computed from the unknown *signing key* $\mathsf{sk}$.
- Correct or improve the types of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$, $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$, $\mathsf{Extract}_{\mathbb{J}^{(r)}}$, $\mathsf{PRF}^{\mathsf{expand}}$, $\mathsf{PRF}^{\mathsf{ockSapling}}$, and $\mathsf{CRH}^{\mathsf{ivk}}$.
- Instantiate $\mathsf{PRF}^{\mathsf{ockSapling}}$ using BLAKE2b-256.
- Change the syntax of a *commitment scheme* to add $\mathsf{COMM.GenTrapdoor}$. This is necessary because the intended distribution of *commitment trapdoors* may not be uniform on all values that are acceptable *trapdoor* inputs.
- Add notes on the purpose of *outgoing viewing keys*.
- Correct the encoding of a *full viewing key* ($\mathsf{ovk}$ was missing).
- Ensure that **Sprout** functions and values are given **Sprout**-specific types where appropriate.

- Improve cross-referencing.
- Clarify the use of BCTV14 vs Groth16 proofs in *JoinSplit statements*.
- Clarify that the $\sqrt[+]{a}$ notation refers to the positive square root. (This matters for the conversion in § A.3.3.3 ***'ctEdwards** ↔ **Montgomery conversion'*** on p. 178.)
- Model the group hash as a *random oracle*. This appears to be unavoidable in order to allow proving unlinkability of $\mathsf{DiversifyHash}^{\mathsf{Sapling}}$. Explain how this relates to the Discrete Logarithm Independence assumption used previously, and justify this modelling by showing that it follows from treating BLAKE2s-256 as a *random oracle* in the instantiation of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$.
- Rename CRS (Common Random String) to URS (*Uniform Random String*), to match the terminology adopted at the first zkproof workshop held in Boston, Massachusetts on May 10–11, 2018.
- Generalize $\mathsf{PRF}^{\mathsf{expand}}$ to accept an arbitrary-length input. (This specification does not use that generalization, but [ZIP-32] does.)
- Change the notation for a multiplication constraint in Appendix § A ***'Circuit Design'*** on p. 173 to avoid potential confusion with cartesian product.
- Clarify the wording of the abstract.
- Correct statements about which algorithms are instantiated by BLAKE2s and BLAKE2b.
- Add a note explaining which conformance requirements of BIP 173 (defining *Bech32*) apply.
- Add the Jubjub bird image to the title page. This image has been edited from a scan of Peter Newell's original illustration (as it appeared in [Carroll1902]) to remove the background and Bandersnatch, and to restore the bird's clipped right wing.
- Change the light yellow background to white (indicating that this **Overwinter** and **Sapling** specification is no longer a draft).

**2018.0-beta-20** 2018-05-22

- Add Michael Dixon and Andrew Poelstra to acknowledgements.
- Minor improvements to cross-references.
- Correct the order of arguments to RedDSA.RandomizePrivate and RedDSA.RandomizePublic.
- Correct a reference to RedDSA.RandomizePrivate that was intended to be RedDSA.RandomizePublic.
- Fix the description of the *Sapling balancing value* in § 4.13 ***'Balance and Binding Signature (Sapling)'*** on p. 47.
- Correct a type error in § 5.4.9.5 ***'Group Hash into** Jubjub'* on p. 96.
- Correct a type error in RedDSA.Sign in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85.
- Ensure $\mathcal{G}^{\mathsf{Sapling}}$ is defined in § 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87.
- Make the *validating key* prefix part of the input to the *hash function* in RedDSA, not part of the message.
- Correct the statement about $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ never returning $\bot$.
- Correct an error in the computation of generators for *Pedersen hashes*.
- Change the order in which $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ commits to its inputs, to match the sapling-crypto implementation.
- Fail **Sapling** key generation if $\mathsf{ivk} = 0$. (This has negligible probability.)
- Change the notation $\mathsf{H}^{\star}$ to $\mathsf{H}^{\circledast}$ in § 5.4.7 'RedDSA, RedJubjub, *and* RedPallas' on p. 85, to avoid confusion with the $^{\star}$ convention for representations of group elements.
- $\mathtt{cmu}$ encodes only the $u$-coordinate of the *note commitment*, not the full curve point.
- rk is checked to be not of small order outside the *Spend statement*, not in the *Spend statement*.

- Change terminology describing constraint systems.

**2018.0-beta-19** 2018-04-23

- Minor clarifications.

**2018.0-beta-18** 2018-04-23

- Clarify the security argument for balance in **Sapling**.
- Correct a subtle problem with the type of the value input to $\mathsf{ValueCommit}^{\mathsf{Sapling}}$: although it is only directly used to commit to values in $\{0\,..\,2^{\ell_{\mathsf{value}}}-1\}$, the security argument depends on a sum of commitments being binding on $\left\{-\frac{r_{\mathbb{J}}-1}{2}\,..\,\frac{r_{\mathbb{J}}-1}{2}\right\}$.
- Fix the loss of tightness in the use of $\mathsf{PRF}^{\mathsf{nfSapling}}$ by specifying the keyspace more precisely.
- Correct type ambiguities for ρ.
- Specify the representation of $i$ in group $\mathbb{G}_2$ of BLS12-381.

**2018.0-beta-17** 2018-04-21

- Correct an error in the definition of DefaultDiversifier.

**2018.0-beta-16** 2018-04-21

- Explicitly note that outputs from *coinbase transactions* include *Founders' Reward* outputs.
- The point represented by $\underline{R}$ in an Ed25519 signature is checked to not be of small order; this is not the same as checking that it is of prime order $\ell$.
- 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 § 1.2 *'High-level Overview'* on p. 8 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.
- Correct the explanation in § 1.2 *'High-level Overview'* on p. 8 to apply to **Sapling**.
- Add the definition of a *signing key* to *validating key* homomorphism for *signature schemes*.
- Remove the output index as an input to $\mathsf{KDF}^{\mathsf{Sapling}}$.
- Allow dummy **Sapling** input *notes*.
- Specify RedDSA and RedJubjub.
- Specify *Sapling binding signatures* and *spend authorization signatures*.
- Specify the randomness beacon.
- Add *Output ciphertexts* and ock.
- Define DefaultDiversifier.
- Change the *Spend circuit* and *Output circuit* specifications to remove unintended differences from sapling-crypto.

- Use $h_{\mathbb{J}}$ to refer to the Jubjub curve cofactor, rather than 8.
- Correct an error in the $y$-coordinate formula for addition in § A.3.3.4 ***'Affine-Montgomery arithmetic'*** on p. 179 (the constraints were correct).
- Add acknowledgements for Brian Warner, Mary Maller, and the Least Authority audit.
- `Makefile` improvements.

**2018.0-beta-15** 2018-03-19

- Clarify the bit ordering of SHA-256.
- Drop `_t` from the names of representation types.
- Remove functions from the **Sprout** specification that it does not use.
- Updates to transaction format and consensus rules for Overwinter and Sapling.
- Add specification of the *Output statement*.
- Change $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ from 29 to 32.
- Updates to **Sapling** construction, changing how the *nullifier* is computed and separating it from the *randomized Spend validating key* ($\mathsf{rk}$).
- Clarify conversions between bit and byte sequences for $\mathsf{sk}$, $\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})$, and $\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$.
- Change the `Makefile` to avoid multiple reloads in PDF readers while rebuilding the PDF.
- Spacing and pagination improvements.

**2018.0-beta-14** 2018-03-11

- Only cosmetic changes to **Sprout**.
- Simplify $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ to use a single-byte index.
- Changes to diversification for *Pedersen hashes* and *Pedersen commitments*.
- Improve security definitions for signatures.

**2018.0-beta-13** 2018-03-11

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

**2018.0-beta-12** 2018-03-06

- Add references to **Overwinter** ZIPs and update the section on **Overwinter**/**Sapling** transitions.
- Add a section on re-randomizable signatures.
- Add definition of $\mathsf{PRF}^{\mathsf{nr}}$.
- Work-in-progress on **Sapling** *statements*.
- Rename *"raw"* to *"homomorphic" Pedersen commitments*.
- Add packing modulo the field size and range checks to Appendix A.
- Update the algorithm for variable-base scalar multiplication to what is implemented by sapling-crypto.

**2018.0-beta-11** 2018-02-26

- Add sections on *Spend descriptions* and *Output descriptions*.
- Swap order of $\mathsf{cv}$ and $\mathsf{rt}$ in a *Spend description* for consistency.
- Fix off-by-one error in the range of $\mathsf{ivk}$.

**2018.0-beta-10** 2018-02-26

- Split the descriptions of SHA-256 and SHA256Compress, and of BLAKE2, into their own sections. Specify SHA256Compress more precisely.
- Add Tracy Hu to acknowledgements (for the idea of explicitly encoding the root of the **Sapling** *note commitment tree* in *block headers*).
- Move bit/byte/integer conversion primitives into § 5.2 ***'Integers, Bit Sequences, and Endianness'*** on p. 66.
- Refer to **Overwinter** and **Sapling** just as "upgrades" in the abstract, not as the next "minor version" and "major version".
- $\mathsf{PRF}^{\mathsf{nr}}$ must be *collision-resistant* .
- Correct an error in the *Pedersen hash* specification.
- Use a named variable, $c$, for chunks per segment in the *Pedersen hash* specification, and change its value from $61$ to $63$. Add a proof justifying this value of $c$.
- Specify *Pedersen commitments*.
- Notation changes.
- Generalize the *distinct-$x$ criterion* (Theorem A.3.4 on p. 179) to allow negative indices.

**2018.0-beta-9** 2018-02-10

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

**2018.0-beta-8** 2018-02-08

- Add instantiation of $\mathsf{CRH}^{\mathsf{ivk}}$.
- Add instantiation of a hash extractor (later renamed to *coordinate extractor*) for Jubjub.
- Make the background lighter and the **Sapling** green darker, for contrast.

**2018.0-beta-7** 2018-02-07

- Specify the 100000-byte limit on *transaction* size. (The implementation in zcashd was as intended.)
- Specify that 0xF6 followed by 511 zero bytes encodes an empty *memo field*.
- Reference security definitions for *Pseudo Random Functions* and *Pseudo Random Generators*.
- 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.
- Add an explanation of how **Sapling** prevents Faerie Gold and roadblock attacks.
- **Sapling** work in progress.

**2018.0-beta-6** 2018-01-31

- **Sapling** work in progress, mainly on Appendix § A ***'Circuit Design'*** on p. 173.

**2018.0-beta-5** 2018-01-30

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

**2018.0-beta-4** 2018-01-25

- Update key components diagram for **Sapling**.

**2018.0-beta-3** 2018-01-22

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

**2017.0-beta-2.9** 2017-12-17

- Refer to $\mathsf{sk}_{\mathsf{enc}}$ as a *receiving key* rather than as a viewing key.
- Updates for *incoming viewing key* support.
- Refer to Network Upgrade 0 as **Overwinter**.

**2017.0-beta-2.8** 2017-12-02

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

**2017.0-beta-2.7** 2017-07-10

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

**2017.0-beta-2.6** 2017-05-09

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

**2017.0-beta-2.5** 2017-03-07

- Clarify the consensus rule preventing double-spends.
- Clarify what a *note commitment* opens to in § 8.8 ***'Omission in Zerocash security proof'*** on p. 137.
- Correct the order of arguments to COMM in § 5.4.8.1 ***'Sprout Note Commitments'*** on p. 88.
- Correct a statement about indistinguishability of *JoinSplit descriptions*.
- Change the *Founders' Reward* addresses, for *Testnet* only, to reflect the hard-fork upgrade described in [Zcash-Issue2113].

**2017.0-beta-2.4** 2017-02-25

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

**2017.0-beta-2.3** 2017-02-12

- Specify the security requirements on the SHA256Compress function in order for the scheme in § 5.4.8.1 ***'Sprout Note Commitments'*** on p. 88 to be a secure commitment.
- Specify $\mathbb{G}_2$ more precisely.
- Explain the use of interstitial *treestates* in chained *JoinSplit transfers*.

**2017.0-beta-2.2** 2017-02-11

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

**2017.0-beta-2.1** 2017-02-06

- $\ell_{\mathsf{Merkle}}$ is a bit length, not a byte length.
- Specify the maximum *block* size.

**2017.0-beta-2** 2017-02-04

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

**2016.0-beta-1.13** 2017-01-20

- Specify the difficulty adjustment algorithm.
- Clarify some definitions of fields in a *block header*.
- Define $\mathsf{PRF}^{\mathsf{addr}}$ in § 4.2.1 ***‘Sprout Key Components’*** on p. 32.

**2016.0-beta-1.12** 2017-01-09

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

**2016.0-beta-1.11** 2016-12-19

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

**2016.0-beta-1.10** 2016-10-30

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

**2016.0-beta-1.9** 2016-10-28

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

**2016.0-beta-1.8** 2016-10-04

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

**2016.0-beta-1.7** 2016-10-02

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

**2016.0-beta-1.6** 2016-09-26

- Fix an error in the definition of the sortedness condition for *Equihash*: it is the sequences of indices that are sorted, not the sequences of hashes.
- Correct the number of bytes in the encoding of `solutionSize`.
- Update the section on encoding of *transparent* addresses. (The precise prefixes are not decided yet.)
- Clarify why $\mathsf{BLAKE2b}\text{-}\ell$ is different from truncated $\mathsf{BLAKE2b}\text{-}512$.
- Clarify a note about SU-CMA security for signatures.
- Add a note about $\mathsf{PRF}^{\mathsf{nfSprout}}$ corresponding to $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash**.
- Add a paragraph about key length in § 8.7 ***'In-band secret distribution'*** on p. 136.
- Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

**2016.0-beta-1.5** 2016-09-22

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

**2016.0-beta-1.4** 2016-09-19

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

**2016.0-beta-1.3** 2016-09-16

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

**2016.0-beta-1.2** 2016-09-11

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

**2016.0-beta-1.1** 2016-09-05

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

**2016.0-beta-1** 2016-09-04

- Major reorganization to separate the abstract cryptographic protocol from the algorithm instantiations.
- Add type declarations.
- Add a "High-level Overview" section.
- Add a section specifying the *zero-knowledge proving system* and the encoding of proofs. Change the encoding of points in proofs to follow IEEE Std 1363[a].
- Add a section on consensus changes from **Bitcoin**, and the specification of *Equihash*.
- Complete the "Differences from the **Zerocash** paper" section.
- Correct the Merkle tree depth to 29.
- Change the length of *memo fields* to 512 bytes.
- Switch the *JoinSplit signature* scheme to Ed25519, with consequent changes to the computation of $\mathsf{h}_{\mathsf{Sig}}$.
- Fix the lead bytes in *shielded payment address* and *spending key* encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $\mathsf{v}_{\mathsf{pub}}^{\mathsf{old}}$ 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

- Change main font to Quattrocento.

**2016.0-alpha-3** 2016-05-09

- Change version numbering convention (no other changes).

**2.0-alpha-3** 2016-05-06

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

**2.0-alpha-2** 2016-04-21

- Change from truncated BLAKE2b-512 to BLAKE2b-256.
- Clarify endianness, and that uses of BLAKE2b are unkeyed.
- Minor correction to what *SIGHASH types* cover.
- Add "as intended for the **Zcash** release of summer 2016" to title page.
- Require $\mathsf{PRF}^{\mathsf{addr}}$ to be *collision-resistant* (see § 8.8 ***'Omission in Zerocash security proof'*** on p. 137).
- Add specification of path computation for the *incremental Merkle tree*.
- Add a note in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54 about how this condition corresponds to conditions in the **Zerocash** paper.
- Changes to terminology around keys.

**2.0-alpha-1** 2016-03-30

- First version intended for public review.

# 11 References

[ABR1999] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. *DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem*. Cryptology ePrint Archive: Report 1999/007. Received March 17, 1999. September 1998. URL: `https://eprint.iacr.org/1999/007` (visited on 2016-08-21) (↑ p23, 136, 137).

[ADMA2015] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. *Security of Keyed Sponge Constructions Using a Modular Proof Approach*. Team Keccak web page, `https://keccak.team/papers.html`. URL: `https://keccak.team/files/ModularKeyedSponge.pdf` (visited on 2021-03-01). Originally published in *Fast Software Encryption - Proceeedings of the 22nd International Workshop (Istanbul, Turkey, March 8–11, 2015)*, pages 364–384; Springer, 2015. Note that the pre-proceedings version contained an oversight in the analysis of the outer-keyed sponge. (↑ p80).

[AGRRT2017] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity*. Cryptology ePrint Archive: Report 2016/492. Received May 21, 2016. January 5, 2017. URL: `https://eprint.iacr.org/2016/492` (visited on 2018-01-12) (↑ p189).

[AKLGL2010] Diego Aranha, Koray Karabina, Patrick Longa, Catherine Gebotys, and Julio López. *Faster Explicit Formulas for Computing Pairings over Ordinary Curves*. Cryptology ePrint Archive: Report 2010/526. Last revised September 12, 2011. URL: `https://eprint.iacr.org/2010/526` (visited on 2018-04-03) (↑ p92, 152).

[ANWW2013] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. *BLAKE2: simpler, smaller, fast as MD5*. January 29, 2013. URL: `https://blake2.net/#sp` (visited on 2016-08-14) (↑ p69, 187).

[AR2017] Leo Alcock and Ling Ren. "A Note on the Security of Equihash". In: *CCSW '17. Proceedings of the 2017 Cloud Computing Security Workshop (Dallas, TX, USA, November 3, 2017); post-workshop of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. URL: `http://sci-hub.tw/10.1145/3140649.3140652` (visited on 2019-01-09) (↑ p124, 146).

[BBDP2001] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. *Key-Privacy in Public-Key Encryption*. September 2001. URL: `https://cseweb.ucsd.edu/~mihir/papers/anonenc.html` (visited on 2016-08-14). Full version. (↑ p24, 72, 136, 148).

[BBJLP2008] Daniel Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. *Twisted Edwards Curves*. Cryptology ePrint Archive: Report 2008/013. Received January 8, 2008. March 13, 2008. URL: `https://eprint.iacr.org/2008/013` (visited on 2018-01-12) (↑ p179, 180).

[BCCGLRT2014] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. *The Hunting of the SNARK*. Cryptology ePrint Archive: Report 2014/580. Received July 24, 2014. URL: `https://eprint.iacr.org/2014/580` (visited on 2020-08-01) (↑ p31, 142).

[BCD+2020] Tim Beyne, Anne Canteaut, Itai Dinur, Maria Eichlseder, Gregor Leander, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Yu Sasaki, Yosuke Todo, and Friedrich Wiemer. *Out of Oddity — New Cryptanalytic Techniques against Symmetric Primitives Optimized for Integrity Proof Systems*. Cryptology ePrint Archive: Report 2020/188. Last revised November 11, 2020. URL: `https://eprint.iacr.org/2020/188` (visited on 2021-03-01). Originally published (with major differences) in *Advances in Cryptology - CRYPTO 2020*, Vol. 12172 pages 299–328; Lecture Notes in Computer Science; Springer, 2020. (↑ p78).

[BCGGMTV2014] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin (extended version)*. URL: `http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf` (visited on 2016-08-06). A condensed version appeared in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland) 2014*, pages 459–474; IEEE, 2014. (↑ p7, 8, 10, 22, 47, 54, 60, 133, 135, 137).

[BCGTV2013] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. *SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge*. Cryptology ePrint Archive: Report 2013/507. Last revised October 7, 2013. URL: `https://eprint.iacr.org/2013/507` (visited on 2016-08-31). An earlier version appeared in *Proceedings of the 33rd Annual International Cryptology Conference, CRYPTO 2013*, pages 90–108; IACR, 2013. (↑ p102).

[BCIMRT2010] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. "Efficient Indifferentiable Hashing into Ordinary Elliptic Curves". In: *Advances in Cryptology - CRYPTO 2010. Proceedings of the 30th Annual International Cryptology Conference (Santa Barbara, California, USA, August 15–19, 2010)*. Ed. by Tal Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, pages 237–254. ISBN: 978-3-642-14623-7. DOI: `10.1007/978-3-642-14623-7_13`. URL: `https://www.iacr.org/archive/crypto2010/62230238/62230238.pdf` (visited on 2021-01-27) (↑ p99).

[BCP1988] Jurgen Bos, David Chaum, and George Purdy. "A Voting Scheme". Unpublished. Presented at the rump session of CRYPTO '88 (Santa Barbara, California, USA, August 21–25, 1988); does not appear in the proceedings. (↑ p73).

[BCTV2014a] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Cryptology ePrint Archive: Report 2013/879. Last revised February 5, 2019. URL: `https://eprint.iacr.org/2013/879` (visited on 2019-02-08) (↑ p102, 147, 173).

[BCTV2014a-old] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture (May 19, 2015 version)*. Cryptology ePrint Archive: Report 2013/879. Version: 20150519:172604. URL: `https://eprint.iacr.org/2013/879/20150519:172604` (visited on 2019-02-08) (↑ p102).

[BCTV2014b] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Scalable Zero Knowledge via Cycles of Elliptic Curves (extended version)". In: *Advances in Cryptology - CRYPTO 2014*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pages 276–294. URL: `https://www.cs.tau.ac.il/~tromer/papers/scalablezk-20140803.pdf` (visited on 2016-09-01) (↑ p31, 147).

[BDEHR2011] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. *On the Security of the Winternitz One-Time Signature Scheme (full version)*. Cryptology ePrint Archive: Report 2011/191. Received April 13, 2011. URL: `https://eprint.iacr.org/2011/191` (visited on 2016-09-05) (↑ p25).

[BDJR2000] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. *A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation*. September 2000. URL: `https://cseweb.ucsd.edu/~mihir/papers/sym-enc.html` (visited on 2018-02-07). An extended abstract appeared in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (Miami Beach, Florida, USA, October 20–22, 1997)*, pages 394–403; IEEE Computer Society Press, 1997; ISBN 0-8186-8197-7. (↑ p22).

[BDLSY2012] Daniel Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. “High-speed high-security signatures”. In: *Journal of Cryptographic Engineering* 2 (September 26, 2011), pages 77–89. URL: `http://cr.yp.to/papers.html#ed25519` (visited on 2016-08-14). Document ID: a1a62a2f76d23f65d622484ddd09caf8. (↑ p83, 84, 194).

[BDPA2007] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Sponge functions*. ECRYPT Hash Workshop (May 2007), also available as a public comment to NIST as part of the Hash Algorithm Requirements and Evaluation Criteria for the SHA-3 competition. URL: `https://www.researchgate.net/publication/242285874_Sponge_Functions` (visited on 2021-03-01) (↑ p77).

[BDPA2011] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Cryptographic sponge functions*. Team Keccak web page, `https://keccak.team/sponge_duplex.html`. Version 0.1, January 14, 2011. URL: `https://keccak.team/files/CSF-0.1.pdf` (visited on 2021-03-01) (↑ p77, 80).

[Bernstein2001] Daniel Bernstein. *Pippenger's exponentiation algorithm*. December 18, 2001. URL: `https://cr.yp.to/papers.html#pippenger` (visited on 2018-07-27). Draft. To be incorporated into the author's *High-speed cryptography* book. Error pointed out by Sam Hocevar: the example in Figure 4 needs 2 and is thus of length 18. (↑ p194, 195).

[Bernstein2005] Daniel Bernstein. “Understanding brute force”. In: *ECRYPT STVL Workshop on Symmetric Key Encryption, eSTREAM report 2005/036*. April 25, 2005. URL: `https://cr.yp.to/papers.html#bruteforce` (visited on 2016-09-24). Document ID: 73e92f5b71793b498288efe81fe55dee. (↑ p137).

[Bernstein2006] Daniel Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *Public Key Cryptography – PKC 2006. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography (New York, NY, USA, April 24–26, 2006)*. Springer-Verlag, February 9, 2006. URL: `http://cr.yp.to/papers.html#curve25519` (visited on 2016-08-14). Document ID: 4230efdfa673480fc079449d90f322c0. (↑ p23, 81, 106, 136).

[BFIJSV2010] Olivier Blazy, Georg Fuchsbauer, Malika Izabachène, Amandine Jambert, Hervé Sibert, and Damien Vergnaud. *Batch Groth–Sahai*. Cryptology ePrint Archive: Report 2010/040. Last revised February 3, 2010. URL: `https://eprint.iacr.org/2010/040` (visited on 2020-10-17) (↑ p139, 141, 194).

[BGG-mpc] Sean Bowe, Ariel Gabizon, and Matthew Green. *GitHub repository 'zcash/mpc': zk-SNARK parameter multi-party computation protocol*. URL: `https://github.com/zcash/mpc` (visited on 2017-01-06) (↑ p111).

[BGG1995] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. “Incremental Cryptography: The Case of Hashing and Signing”. In: *Advances in Cryptology - CRYPTO '94. Proceedings of the 14th Annual International Cryptology Conference (Santa Barbara, California, USA, August 21–25, 1994)*. Ed. by Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer, October 20, 1995, pages 216–233. ISBN: 978-3-540-48658-9. DOI: `10.1007/3-540-48658-5_22`. URL: `https://cseweb.ucsd.edu/~mihir/papers/inc1.pdf` (visited on 2018-02-09) (↑ p73, 74, 76, 183).

[BGG2017] Sean Bowe, Ariel Gabizon, and Matthew Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive: Report 2017/602. Last revised June 25, 2017. URL: `https://eprint.iacr.org/2017/602` (visited on 2019-02-10) (↑ p102, 111, 146, 156).

[BGHOZ2013] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Frederico Olmedo, and Santiago Zanella-Béguelin. “Verified indifferentiable hashing into elliptic curves”. In: *Journal of Computer Security, Security and Trust Principles* 21.6 (2013), pages 881–917. URL: `https://software.imdea.org/~szanella/Zanella.2012.POST.pdf` (visited on 2021-01-28) (↑ p101).

[BGM2017] Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive: Report 2017/1050. Last revised November 5, 2017. URL: `https://eprint.iacr.org/2017/1050` (visited on 2018-08-31) (↑ p103, 112, 147, 148).

[BIP-11] Gavin Andresen. *M-of-N Standard Transactions*. Bitcoin Improvement Proposal 11. Created October 18, 2011. URL: `https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-13] Gavin Andresen. *Address Format for pay-to-script-hash*. Bitcoin Improvement Proposal 13. Created October 18, 2011. URL: `https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki` (visited on 2020-07-13) (↑ p105, 132).

[BIP-14] Amir Taaki and Patrick Strateman. *Protocol Version and User Agent*. Bitcoin Improvement Proposal 14. Created November 10, 2011. URL: `https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-16] Gavin Andresen. *Pay to Script Hash*. Bitcoin Improvement Proposal 16. Created January 3, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-30] Pieter Wuille. *Duplicate transactions*. Bitcoin Improvement Proposal 30. Created February 22, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-31] Mike Hearn. *Pong message*. Bitcoin Improvement Proposal 31. Created April 11, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-32] Pieter Wuille. *Hierarchical Deterministic Wallets*. Bitcoin Improvement Proposal 32. Created February 11, 2012. Last updated January 15, 2014. URL: `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki` (visited on 2020-07-13) (↑ p105, 143).

[BIP-34] Gavin Andresen. *Block v2, Height in Coinbase*. Bitcoin Improvement Proposal 34. Created July 6, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki` (visited on 2020-07-13) (↑ p117, 132, 157).

[BIP-35] Jeff Garzik. *mempool message*. Bitcoin Improvement Proposal 35. Created August 16, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-37] Mike Hearn and Matt Corallo. *Connection Bloom filtering*. Bitcoin Improvement Proposal 37. Created October 24, 2012. URL: `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-61] Gavin Andresen. *Reject P2P message*. Bitcoin Improvement Proposal 61. Created June 18, 2014. URL: `https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-62] Pieter Wuille. *Dealing with malleability*. Bitcoin Improvement Proposal 62. Withdrawn November 17, 2015. URL: `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki` (visited on 2020-07-13) (↑ p25).

[BIP-65] Peter Todd. *OP_CHECKLOCKTIMEVERIFY*. Bitcoin Improvement Proposal 65. Created October 10, 2014. URL: `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki` (visited on 2020-07-13) (↑ p132).

[BIP-66] Pieter Wuille. *Strict DER signatures*. Bitcoin Improvement Proposal 66. Created January 10, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki (visited on 2020-07-13) (↑ p132).

[BIP-68] Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. *Relative lock-time using consensus-enforced sequence numbers*. Bitcoin Improvement Proposal 68. Last revised November 21, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki (visited on 2020-07-13) (↑ p118).

[BIP-111] Matt Corallo and Peter Todd. *NODE_BLOOM service bit*. Bitcoin Improvement Proposal 111. Created August 20, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0111.mediawiki (visited on 2020-07-13) (↑ p132, 152).

[BIP-350] Pieter Wuille. *Bech32m format for v1+ witness addresses*. Bitcoin Improvement Proposal 350. Created December 16, 2020. URL: https://github.com/bitcoin/bips/blob/master/bip-0350.mediawiki (visited on 2021-03-17) (↑ p105, 109).

[Bitcoin-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Base58Check_encoding (visited on 2020-07-13) (↑ p104, 105).

[Bitcoin-Block] *Block Headers — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/block_chain.html#block-headers (visited on 2020-07-13) (↑ p123, 124).

[Bitcoin-CoinJoin] *CoinJoin — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/CoinJoin (visited on 2020-07-13) (↑ p9).

[Bitcoin-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/transactions.html#raw-transaction-format (visited on 2020-07-13) (↑ p118).

[Bitcoin-Multisig] *Transactions: Multisig — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#multisig (visited on 2020-07-13) (↑ p129, 131).

[Bitcoin-nBits] *Target nBits — Bitcoin Developer Reference*. URL: https://developer.bitcoin.org/reference/block_chain.html#target-nbits (visited on 2020-07-13) (↑ p122, 127).

[Bitcoin-P2PKH] *Transactions: P2PKH Script Validation — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#p2pkh-script-validation (visited on 2020-07-13) (↑ p105, 109).

[Bitcoin-P2SH] *Transactions: P2SH Scripts — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#pay-to-script-hash-p2sh (visited on 2020-07-13) (↑ p105, 109, 131).

[Bitcoin-Protocol] *Protocol documentation — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 2020-07-13) (↑ p8).

[Bitcoin-SigHash] *Signature Hash Types — Bitcoin Developer Guide*. URL: https://developer.bitcoin.org/devguide/transactions.html#signature-hash-types (visited on 2020-07-13) (↑ p46).

[BJLSY2015] Daniel Bernstein, Simon Josefsson, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. *EdDSA for more curves*. Technical Report. July 4, 2015. URL: https://cr.yp.to/papers.html#eddsa (visited on 2018-01-22) (↑ p85, 95).

[BK2016] Alex Biryukov and Dmitry Khovratovich. *Equihash: Asymmetric Proof-of-Work Based on the Generalized Birthday Problem (full version)*. Cryptology ePrint Archive: Report 2015/946. Last revised October 27, 2016. URL: https://eprint.iacr.org/2015/946 (visited on 2016-10-30) (↑ p10, 124, 157).

[BKR2001] Mihir Bellare, Joe Kilian, and Phillip Rogaway. “The Security of the Cipher Block Chaining Message Authentication Code”. In: *Journal of Computer and System Sciences* 61.3 (December 2000), pages 362–399. DOI: https://doi.org/10.1006/jcss.1999.1694. URL: https://cseweb.ucsd.edu/~mihir/papers/cbc.pdf (visited on 2021-03-08). Updated September 12, 2001. (↑ p22).

[BL-SafeCurves] Daniel Bernstein and Tanja Lange. *SafeCurves: choosing safe curves for elliptic-curve cryptography*. URL: `https://safecurves.cr.yp.to` (visited on 2018-01-29) (↑ p136, 167).

[BL2017] Daniel Bernstein and Tanja Lange. *Montgomery curves and the Montgomery ladder*. Cryptology ePrint Archive: Report 2017/293. Received March 30, 2017. URL: `https://eprint.iacr.org/2017/293` (visited on 2017-11-26) (↑ p94, 173, 179, 180).

[BLS2002] Paulo Barreto, Ben Lynn, and Michael Scott. *Constructing Elliptic Curves with Prescribed Embedding Degrees*. Cryptology ePrint Archive: Report 2002/088. Last revised February 22, 2005. URL: `https://eprint.iacr.org/2002/088` (visited on 2018-04-20) (↑ p93, 152).

[BN2005] Paulo Barreto and Michael Naehrig. *Pairing-Friendly Elliptic Curves of Prime Order*. Cryptology ePrint Archive: Report 2005/133. Last revised February 28, 2006. URL: `https://eprint.iacr.org/2005/133` (visited on 2018-04-20) (↑ p91, 152).

[BN2007] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm*. Cryptology ePrint Archive: Report 2000/025. Last revised July 14, 2007. URL: `https://eprint.iacr.org/2000/025` (visited on 2016-09-02) (↑ p23).

[Bowe-bellman] Sean Bowe. *bellman: zk-SNARK library*. URL: `https://github.com/ebfull/bellman` (visited on 2018-04-03) (↑ p103, 112).

[Bowe2017] Sean Bowe. *ebfull/pairing source code, BLS12-381 – README.md as of commit e726600*. URL: `https://github.com/ebfull/pairing/tree/e72660056e00c93d6b054dfb08ff34a1c67cb799/src/bls12_381` (visited on 2017-07-16) (↑ p93).

[Bowe2018] Sean Bowe. *Random Beacon*. March 22, 2018. URL: `https://github.com/ZcashFoundation/powersoftau-attestations/tree/master/0088` (visited on 2018-04-08) (↑ p112).

[Carroll1876] Lewis Carroll. *The Hunting of the Snark*. With illustrations by Henry Holiday. MacMillan and Co. London. March 29, 1876. URL: `https://www.gutenberg.org/files/29888/29888-h/29888-h.htm` (visited on 2018-05-23) (↑ p94, 145).

[Carroll1902] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There (1902 edition)*. Illustrated by Peter Newell and Robert Murray Wright. Harper and Brothers Publishers. New York. October 1902. URL: `https://archive.org/details/throughlookinggl00carr4` (visited on 2018-06-20) (↑ p139, 151).

[CDvdG1987] David Chaum, Ivan Damgård, and Jeroen van de Graaf. “Multiparty computations ensuring privacy of each party’s input and correctness of the result”. In: *Advances in Cryptology - CRYPTO ’87. Proceedings of the 14th Annual International Cryptology Conference (Santa Barbara, California, USA, August 16–20, 1987)*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, January 1988, pages 87–119. ISBN: 978-3-540-48184-3. DOI: `10.1007/3-540-48184-2_7`. URL: `https://www.researchgate.net/profile/Jeroen_Van_de_Graaf/publication/242379939_Multiparty_computations_ensuring_secrecy_of_each_party%27s_input_and_correctness_of_the_output` (visited on 2018-03-01) (↑ p73).

[Cook2019] John D. Cook. *What is an isogeny?* Blog post. April 21, 2019. URL: `https://www.johndcook.com/blog/2019/04/21/what-is-an-isogeny/` (visited on 2021-02-10) (↑ p99).

[CVE-2019-7167] Common Vulnerabilities and Exposures. *CVE-2019-7167*. URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-7167` (visited on 2019-02-05) (↑ p102).

[CvHP1991] David Chaum, Eugène van Heijst, and Birgit Pfitzmann. *Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer*. February 1991. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.8570` (visited on 2018-02-17). An extended abstract appeared in *Advances in Cryptology - CRYPTO ’91: Proceedings of the 11th Annual International Cryptology Conference (Santa Barbara, California, USA, August 11–15, 1991)*; Ed. by Joan Feigenbaum; Vol. 576, Lecture Notes in Computer Science, pages 470–484; Springer, 1992; ISBN 978-3-540-55188-1. (↑ p73, 183).

[Dalek-notes] Cathie Yun, Henry de Valence, Oleg Andreev, and Dimitris Apostolou. *ristretto_bulletproofs notes*. URL: https://doc-internal.dalek.rs/ristretto_bulletproofs/notes/index.html (visited on 2018-08-17) (↑ p49, 148).

[deRooij1995] Peter de Rooij. "Efficient exponentiation using precomputation and vector addition chains". In: *Advances in Cryptology - EUROCRYPT '94. Proceedings, Workshop on the Theory and Application of Cryptographic Techniques (Perugia, Italy, May 9–12, 1994)*. Ed. by Alfredo De Santis. Vol. 950. Lecture Notes in Computer Science. Springer, pages 389–399. ISBN: 978-3-540-60176-0. DOI: 10.1007/BFb0053453. URL: https://link.springer.com/chapter/10.1007/BFb0053453 (visited on 2018-07-27) (↑ p194, 195).

[DGKM2011] Dana Dachman-Soled, Rosario Gennaro, Hugo Krawczyk, and Tal Malkin. *Computational Extractors and Pseudorandomness*. Cryptology ePrint Archive: Report 2011/708. December 28, 2011. URL: https://eprint.iacr.org/2011/708 (visited on 2016-09-02) (↑ p137).

[DigiByte-PoW] DigiByte Core Developers. *DigiSpeed 4.0.0 source code, functions GetNextWorkRequiredV3/4 in src/main.cpp as of commit 178e134*. URL: https://github.com/digibyte/digibyte/blob/178e1348a67d9624db328062397fde0de03fe388/src/main.cpp#L1587 (visited on 2017-01-20) (↑ p125).

[DS2016] David Derler and Daniel Slamanig. *Key-Homomorphic Signatures and Applications to Multiparty Signatures and Non-Interactive Zero-Knowledge*. Cryptology ePrint Archive: Report 2016/792. Last revised February 6, 2017. URL: https://eprint.iacr.org/2016/792 (visited on 2018-04-09) (↑ p27).

[DSDCOPS2001] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Guiseppe Persiano, and Amit Sahai. "Robust Non-Interactive Zero Knowledge". In: *Advances in Cryptology - CRYPTO 2001. Proceedings of the 21st Annual International Cryptology Conference (Santa Barbara, California, USA, August 19–23, 2001)*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pages 566–598. ISBN: 978-3-540-42456-7. DOI: 10.1007/3-540-44647-8_33. URL: https://www.iacr.org/archive/crypto2001/21390566.pdf (visited on 2018-05-28) (↑ p31, 45).

[ECCZF2019] Electric Coin Company and Zcash Foundation. *Zcash Trademark Donation and License Agreement*. November 6, 2019. URL: https://www.zfnd.org/about/contracts/2019_ECC_ZFND_TM_agreement.pdf (visited on 2020-07-05) (↑ p20).

[ElGamal1985] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31.4 (July 1985), pages 469–472. ISSN: 0018-9448. DOI: 10.1109/TIT.1985.1057074. URL: https://people.csail.mit.edu/alinush/6.857-spring-2015/papers/elgamal.pdf (visited on 2018-08-17) (↑ p72).

[EWD-831] Edsger W. Dijkstra. *Why numbering should start at zero*. Manuscript. August 11, 1982. URL: https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html (visited on 2016-08-09) (↑ p10).

[FFSTV2013] Reza Farashahi, Pierre-Alain Fouque, Igor Shparlinski, Mehdi Tibouchi, and J. Felipe Voloch. "Indifferentiable deterministic hashing to elliptic and hyperelliptic curves". In: *Mathematics of Computation* 82 (2013), pages 491–512. DOI: 10.1090/S0025-5718-2012-02606-8. URL: https://www.ams.org/journals/mcom/2013-82-281/S0025-5718-2012-02606-8/ (visited on 2021-01-27) (↑ p101).

[FKMSSS2016] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. *Efficient Unlinkable Sanitizable Signatures from Signatures with Re-Randomizable Keys*. Cryptology ePrint Archive: Report 2012/159. Last revised February 11, 2016. URL: https://eprint.iacr.org/2015/395 (visited on 2018-03-03). An extended abstract appeared in *Public Key Cryptography – PKC 2016: 19th IACR International Conference on Practice and Theory in Public-Key Cryptography (Taipei, Taiwan, March 6–9, 2016), Proceedings, Part 1*; Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang; Vol. 9614, Lecture Notes in Computer Science, pages 301–330; Springer, 2016; ISBN 978-3-662-49384-7. (↑ p26, 85).

[Gabizon2019] Ariel Gabizon. *On the security of the BCTV Pinocchio zk-SNARK variant*. Draft. February 5, 2019. URL: `https://github.com/arielgabizon/bctv/blob/master/bctv.pdf` (visited on 2019-02-07) (↑ p102, 139, 147).

[GG2015] Shoni Gilboa and Shay Gueron. *Distinguishing a truncated random permutation from a random function*. Cryptology ePrint Archive: Report 2015/773. Received August 3, 2015. URL: `https://eprint.iacr.org/2015/773` (visited on 2021-03-01) (↑ p80).

[GGM2016] Christina Garman, Matthew Green, and Ian Miers. *Accountable Privacy for Decentralized Anonymous Payments*. Cryptology ePrint Archive: Report 2016/061. Last revised January 24, 2016. URL: `https://eprint.iacr.org/2016/061` (visited on 2016-09-02) (↑ p133).

[GKRRS2019] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. Cryptology ePrint Archive: Report 2019/458. Last updated December 16, 2020. URL: `https://eprint.iacr.org/2019/458` (visited on 2021-02-28) (↑ p77, 78, 80).

[GPT2015] Peter Gazi, Krzysztof Pietrzak, and Stefano Tessaro. "The Exact PRF Security of Truncation: Tight Bounds for Keyed Sponges and Truncated CBC". In: *Advances in Cryptology - CRYPTO 2015. Proceedings of the 35th Annual International Cryptology Conference (Santa Barbara, California, USA, August 16–20, 2015), Part I*. Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9215. Lecture Notes in Computer Science. Springer, August 1, 2015, pages 368–387. ISBN: 978-3-662-47989-6. DOI: `10.1007/978-3-662-47989-6\_18`. URL: `https://iacr.org/cryptodb/data/paper.php?pubkey=27279` (visited on 2021-03-01) (↑ p80).

[Groth2016] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive: Report 2016/260. Last revised May 31, 2016. URL: `https://eprint.iacr.org/2016/260` (visited on 2017-08-03) (↑ p103, 146, 194).

[GWC2019] Ariel Gabizon, Zachary Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive: Report 2019/953. Last revised September 3, 2020. URL: `https://eprint.iacr.org/2019/953` (visited on 2021-01-28) (↑ p139).

[Hamdon2018] Elise Hamdon. *Sapling Activation Complete*. Electric Coin Company blog. June 28, 2018. URL: `https://electriccoin.co/blog/sapling-activation-complete/` (visited on 2021-01-10) (↑ p112).

[Hışıl2010] Hüseyin Hışıl. "Elliptic Curves, Group Law, and Efficient Computation". PhD thesis. Queensland University of Technology, 2010. URL: `https://eprints.qut.edu.au/33233/` (visited on 2021-01-26) (↑ p97).

[Hopwood2018] Daira Hopwood. *GitHub repository 'daira/jubjub': Supporting evidence for security of the Jubjub curve to be used in Zcash*. URL: `https://github.com/daira/jubjub` (visited on 2018-02-18). Based on code written for SafeCurves [BL-SafeCurves] by Daniel Bernstein and Tanja Lange. (↑ p136).

[HW2016] Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the Zcash Protocol*. Electric Coin Company blog. April 26, 2016. URL: `https://electriccoin.co/blog/fixing-zcash-vulns/` (visited on 2019-08-27). Updated December 26, 2017. (↑ p135).

[ID-hashtocurve] Armando Faz-Hernández, Sam Scott, Nick Sullivan, Riad Wahby, and Christopher Wood. *Internet Draft: Hashing to Elliptic Curves, version 10*. Internet Research Task Force (IRTF) Crypto Forum Research Group (CFRG). Work in progress. Last revised December 22, 2020. URL: `https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-10.html` (visited on 2021-01-27) (↑ p30, 98, 100, 101).

[IEEE2000] IEEE Computer Society. *IEEE Std 1363-2000: Standard Specifications for Public-Key Cryptography*. IEEE, August 29, 2000. DOI: `10.1109/IEEESTD.2000.92292`. URL: `http://ieeexplore.ieee.org/servlet/opac?punumber=7168` (visited on 2016-08-03) (↑ p93).

[IEEE2004] IEEE Computer Society. *IEEE Std 1363a-2004: Standard Specifications for Public-Key Cryptography – Amendment 1: Additional Techniques*. IEEE, September 2, 2004. DOI: `10.1109/IEEESTD.2004.94612`. URL: `http://ieeexplore.ieee.org/servlet/opac?punumber=9276` (visited on 2016-08-03) (↑ p92, 93, 136, 138).

[Jedusor2016] Tom Elvis Jedusor. *Mimblewimble*. July 19, 2016. URL: `http://diyhpl.us/~bryan/papers2/bitcoin/mimblewimble.txt` (visited on 2018-04-03) (↑ p49).

[KR2020] Nathan Keller and Asaf Rosemarin. *Mind the Middle Layer: The HADES Design Strategy Revisited*. Cryptology ePrint Archive: Report 2020/179. Received February 13, 2020. URL: `https://eprint.iacr.org/2020/179` (visited on 2021-03-01) (↑ p78).

[KT2015] Taechan Kim and Mehdi Tibouchi. “Improved Elliptic Curve Hashing and Point Representation”. In: *Proceedings of WCC2015 - 9th International Workshop on Coding and Cryptography (Paris, France, April 2015)*. Ed. by Anne Canteaut, Gaëtan Leurent, and Maria Naya-Plasencia. URL: `https://hal.inria.fr/hal-01275711` (visited on 2021-01-28) (↑ p101).

[KvE2013] Kaa1el and Hagen von Eitzen. *If a group G has odd order, then the square function is injective (answer)*. Mathematics Stack Exchange. URL: `https://math.stackexchange.com/a/522277/185422` (visited on 2018-02-08). Version: 2013-10-11. (↑ p96).

[KYMM2018] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. *An Empirical Analysis of Anonymity in Zcash*. Preprint, to be presented at the 27th Usenix Security Syposium (Baltimore, Maryland, USA, August 15–17, 2018). May 8, 2018. URL: `https://smeiklej.com/files/usenix18.pdf` (visited on 2018-06-05) (↑ p9).

[LG2004] Eddie Lenihan and Carolyn Eve Green. *Meeting the Other Crowd: The Fairy Stories of Hidden Ireland*. TarcherPerigee, February 2004, pages 109–110. ISBN: 1-58542-206-1 (↑ p133).

[libsodium] *libsodium documentation*. URL: `https://libsodium.org/` (visited on 2020-03-02) (↑ p84).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: `https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html` (visited on 2016-02-01) (↑ p136).

[LM2017] Philip Lafrance and Alfred Menezes. *On the security of the WOTS-PRF signature scheme*. Cryptology ePrint Archive: Report 2017/938. Last revised February 5, 2018. URL: `https://eprint.iacr.org/2017/938` (visited on 2018-04-16) (↑ p25).

[MAEÁ2010] V. Gayoso Martínez, F. Hernández Alvarez, L. Hernández Encinas, and C. Sánchez Ávila. “A Comparison of the Standardized Versions of ECIES”. In: *Proceedings of Sixth International Conference on Information Assurance and Security (Atlanta, Georgia, USA, August 23–25, 2010)*. IEEE, 2010, pages 1–4. ISBN: 978-1-4244-7407-3. DOI: `10.1109/ISIAS.2010.5604194`. URL: `https://digital.csic.es/bitstream/10261/32674/1/Gayoso_A%20Comparison%20of%20the%20Standardized%20Versions%20of%20ECIES.pdf` (visited on 2016-08-14) (↑ p136).

[Maller2018] Mary Maller. *A Proof of Security for the Sapling Generation of zk-SNARK Parameters in the Generic Group Model*. November 16, 2018. URL: `https://github.com/zcash/sapling-security-analysis/blob/master/MaryMallerUpdated.pdf` (visited on 2018-02-10) (↑ p103, 147).

[ISO2015] ISO/IEC. *International Standard ISO/IEC 18004:2015(E): Information Technology – Automatic identification and data capture techniques – QR Code bar code symbology specification*. February 1, 2015. URL: `https://raw.githubusercontent.com/yansikeim/QR-Code/master/ISO%20IEC%2018004%202015%20Standard.pdf` (visited on 2021-03-22). Third edition. (↑ p105).

[MRH2003] Ueli Maurer, Renato Renner, and Clemens Holenstein. *Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology*. Cryptology ePrint Archive: Report 2003/161. Received August 8, 2003. September 2003. URL: `https://eprint.iacr.org/2003/161` (visited on 2021-02-10) (↑ p101).

[Nakamoto2008] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. October 31, 2008. URL: `https://bitcoin.org/en/bitcoin-paper` (visited on 2016-08-14) (↑ p7).

[NIST2015] NIST. *FIPS 180-4: Secure Hash Standard (SHS)*. August 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: https://csrc.nist.gov/publications/detail/fips/180/4/final (visited on 2021-03-08) (↑ p68, 69, 105).

[NIST2016] NIST. *NIST SP 800-38G — Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption*. March 2016. DOI: 10.6028/NIST.SP.800-38G. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf (visited on 2021-03-08) (↑ p81).

[Parno2015] Bryan Parno. *A Note on the Unsoundness of vnTinyRAM's SNARK*. Cryptology ePrint Archive: Report 2015/437. Received May 6, 2015. URL: https://eprint.iacr.org/2015/437 (visited on 2019-02-08) (↑ p102, 139, 147).

[Peterson2017] Paige Peterson. *Transaction Linkability*. Electric Coin Company blog. January 25, 2017. URL: https://electriccoin.co/blog/transaction-linkability/ (visited on 2019-08-27) (↑ p9, 152).

[PHGR2013] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. *Pinocchio: Nearly Practical Verifiable Computation*. Cryptology ePrint Archive: Report 2013/279. Last revised May 13, 2013. URL: https://eprint.iacr.org/2013/279 (visited on 2016-08-31) (↑ p102).

[Quesnelle2017] Jeffrey Quesnelle. *On the linkability of Zcash transactions*. arXiv:1712.01210 [cs.CR]. December 4, 2017. URL: https://arxiv.org/abs/1712.01210 (visited on 2018-04-15) (↑ p9, 152).

[RFC-2119] Scott Bradner. *Request for Comments 7693: Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF). March 1997. URL: https://www.rfc-editor.org/rfc/rfc2119.html (visited on 2016-09-14) (↑ p7).

[RFC-7539] Yoav Nir and Adam Langley. *Request for Comments 7539: ChaCha20 and Poly1305 for IETF Protocols*. Internet Research Task Force (IRTF). May 2015. URL: https://www.rfc-editor.org/rfc/rfc7539.html (visited on 2016-09-02). As modified by verified errata at https://www.rfc-editor.org/errata_search.php?rfc=7539 (visited on 2016-09-02). (↑ p81).

[RFC-8032] Simon Josefsson and Ilari Liusvaara. *Request for Comments 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)*. Internet Engineering Task Force (IETF). January 2017. URL: https://www.rfc-editor.org/rfc/rfc8032.html (visited on 2020-07-06). As modified by errata at https://www.rfc-editor.org/errata_search.php?rfc=8032 (visited on 2020-07-06). (↑ p84).

[RIPEMD160] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. *RIPEMD-160, a strengthened version of RIPEMD*. URL: http://homes.esat.kuleuven.be/~bosselae/ripemd160.html (visited on 2016-09-24) (↑ p105).

[ST1999] Tomas Sander and Amnon Ta–Shma. "Auditable, Anonymous Electronic Cash". In: *Advances in Cryptology - CRYPTO '99. Proceedings of the 19th Annual International Cryptology Conference (Santa Barbara, California, USA, August 15–19, 1999)*. Ed. by Michael Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pages 555–572. ISBN: 978-3-540-66347-8. DOI: 10.1007/3-540-48405-1_35. URL: https://link.springer.com/content/pdf/10.1007/3-540-48405-1_35.pdf (visited on 2018-06-05) (↑ p139, 150).

[Sutherland2019] Andrew Sutherland. *MIT Open Courseware, Mathematics 18.783 Elliptic Curves, Lecture Notes*. Massachusetts Institute of Technology. Spring 2019. April 21, 2019. URL: https://ocw.mit.edu/courses/mathematics/18-783-elliptic-curves-spring-2019/lecture-notes/index.htm (visited on 2021-02-10) (↑ p99).

[SvdW2006] Andrew Shallue and Christiaan E. van de Woestijne. "Construction of Rational Points on Elliptic Curves over Finite Fields". In: *Algorithmic Number Theory: 7th International Symposium, ANTS-VII (Berlin, Germany, July 23–28, 2006)*. Ed. by F. Hess, S. Pauli, and M. Pohst. Vol. 4076. Lecture Notes in Computer Science. Springer, 2006, pages 510–524. ISBN: 978-3-540-36076-6. DOI: 10.1007/11792086_36. URL: https://digitalcommons.iwu.edu/math_scholarship/72/ (visited on 2021-01-28) (↑ p99).

[SVPBABW2012] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Muqeet Ali, Andrew J. Blumberg, and Michael Walfish. *Taking proof-based verified computation a few steps closer to practicality (extended version)*. Cryptology ePrint Archive: Report 2012/598. Last revised February 28, 2013. URL: `https://eprint.iacr.org/2012/598.pdf` (visited on 2018-04-25) (↑ p175).

[SWB2019] Josh Swihart, Benjamin Winston, and Sean Bowe. *Zcash Counterfeiting Vulnerability Successfully Remediated*. February 5, 2019. URL: `https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/` (visited on 2019-08-27) (↑ p102, 147).

[Swihart2018] Josh Swihart. *Overwinter Activated Successfully*. Electric Coin Company blog. June 26, 2018. URL: `https://electriccoin.co/blog/overwinter-activated-successfully/` (visited on 2021-01-10) (↑ p112).

[Ulas2007] Maciej Ulas. "Rational Points on Certain Hyperelliptic Curves over Finite Fields". In: Bulletin of the Polish Academy of Sciences - Mathematics 55.2 (2007), pages 97–104. DOI: `10.4064/ba55-2-1`. URL: `https://www.impan.pl/shop/publication/transaction/download/product/85475` (visited on 2021-01-27) (↑ p99).

[vanSaberh2014] Nicolas van Saberhagen. *CryptoNote v 2.0*. Date disputed. URL: `https://cryptonote.org/whitepaper.pdf` (visited on 2016-08-17) (↑ p9).

[Vercauter2009] Frederik Vercauteren. *Optimal pairings*. Cryptology ePrint Archive: Report 2008/096. Last revised March 7, 2008. URL: `https://eprint.iacr.org/2008/096` (visited on 2018-04-06). A version of this paper appeared in *IEEE Transactions of Information Theory*, Vol. 56, pages 455–461; IEEE, 2009. (↑ p92, 152).

[WB2019] Riad Wahby and Dan Boneh. *Fast and simple constant-time hashing to the BLS12-381 elliptic curve*. Cryptology ePrint Archive: Report 2018/403. Last revised September 30, 2019. URL: `https://eprint.iacr.org/2019/403` (visited on 2021-01-27) (↑ p98, 101).

[WCBTV2015] Zooko Wilcox, Alessandro Chiesa, Eli Ben-Sasson, Eran Tromer, and Madars Virza. *A Bug in libsnark*. Least Authority blog. May 16, 2015. URL: `https://leastauthority.com/blog/a_bug_in_libsnark/` (visited on 2019-08-27) (↑ p102, 173).

[WG2016] Zooko Wilcox and Jack Grigg. *Why Equihash?* Electric Coin Company blog. April 15, 2016. URL: `https://electriccoin.co/blog/why-equihash/` (visited on 2019-08-27). Updated August 21, 2019. (↑ p124).

[Zaverucha2012] Gregory M. Zaverucha. *Hybrid Encryption in the Multi-User Setting*. Cryptology ePrint Archive: Report 2012/159. Received March 20, 2012. URL: `https://eprint.iacr.org/2012/159` (visited on 2016-09-24) (↑ p137).

[Zcash-Blossom] Electric Coin Company. *Blossom*. December 11, 2019. URL: `https://z.cash/upgrade/blossom/` (visited on 2021-01-10) (↑ p112).

[Zcash-Canopy] Electric Coin Company. *Canopy*. November 18, 2020. URL: `https://z.cash/upgrade/canopy/` (visited on 2021-01-10) (↑ p112).

[Zcash-Heartwd] Electric Coin Company. *Heartwood*. July 16, 2020. URL: `https://z.cash/upgrade/heartwood/` (visited on 2021-01-10) (↑ p112).

[Zcash-Issue2113] Simon Liu. *GitHub repository 'zcash/zcash': Issue 2113*. URL: `https://github.com/zcash/zcash/issues/2113` (visited on 2017-02-20) (↑ p129, 156).

[Zcash-libsnark] *libsnark: C++ library for zkSNARK proofs (Zcash fork)*. URL: `https://github.com/zcash/zcash/tree/master/src/snark` (visited on 2018-02-04) (↑ p102).

[Zcash-Orchard] Daira Hopwood, Sean Bowe, Jack Grigg, Kris Nuttycombe, Ying Tong Lai, and Steven Smith. *The Orchard Book*. URL: `https://zcash.github.io/orchard/` (visited on 2021-03-02) (↑ p75, 90).

[ZIP-32] Jack Grigg and Daira Hopwood. *Shielded Hierarchical Deterministic Wallets*. Zcash Improvement Proposal 32. URL: `https://zips.z.cash/zip-0032` (visited on 2019-08-28) (↑ p12, 22, 34, 41, 61, 72, 81, 86, 143, 145, 151).

[ZIP-76] Jack Grigg and Daira Hopwood. *Transaction Signature Validation before Overwinter*. Zcash Improvement Proposal 76 (in progress). (↑ p46, 132).

[ZIP-143] Jack Grigg and Daira Hopwood. *Transaction Signature Validation for Overwinter*. Zcash Improvement Proposal 143. Created December 27, 2017. URL: `https://zips.z.cash/zip-0143` (visited on 2019-08-28) (↑ p46, 69, 112).

[ZIP-173] Daira Hopwood. *Bech32 Format*. Zcash Improvement Proposal 173. Created June 13, 2018. URL: `https://zips.z.cash/zip-0173` (visited on 2020-06-01) (↑ p104, 107, 109, 143).

[ZIP-200] Jack Grigg. *Network Upgrade Mechanism*. Zcash Improvement Proposal 200. Created January 8, 2018. URL: `https://zips.z.cash/zip-0200` (visited on 2019-08-28) (↑ p112, 118).

[ZIP-201] Simon Liu. *Network Peer Management for Overwinter*. Zcash Improvement Proposal 201. Created January 15, 2018. URL: `https://zips.z.cash/zip-0201` (visited on 2019-08-28) (↑ p112, 113).

[ZIP-202] Simon Liu. *Version 3 Transaction Format for Overwinter*. Zcash Improvement Proposal 202. Created January 10, 2018. URL: `https://zips.z.cash/zip-0202` (visited on 2019-08-28) (↑ p112).

[ZIP-203] Jay Graber. *Transaction Expiry*. Zcash Improvement Proposal 203. Created January 9, 2018. URL: `https://zips.z.cash/zip-0203` (visited on 2019-08-28) (↑ p112, 114, 115).

[ZIP-205] Daira Hopwood. *Deployment of the Sapling Network Upgrade*. Zcash Improvement Proposal 205. Created October 8, 2018. URL: `https://zips.z.cash/zip-0205` (visited on 2019-08-28) (↑ p112, 126).

[ZIP-206] Daira Hopwood. *Deployment of the Blossom Network Upgrade*. Zcash Improvement Proposal 206. Created July 29, 2019. URL: `https://zips.z.cash/zip-0206` (visited on 2019-08-28) (↑ p46, 112, 145).

[ZIP-207] Jack Grigg. *Funding Streams*. Zcash Improvement Proposal 207. Created January 4, 2019. URL: `https://zips.z.cash/zip-0207` (visited on 2019-08-28) (↑ p113, 142, 144).

[ZIP-208] Simon Liu and Daira Hopwood. *Shorter Block Target Spacing*. Zcash Improvement Proposal 208. Created January 10, 2019. URL: `https://zips.z.cash/zip-0208` (visited on 2019-08-28) (↑ p112, 126, 146).

[ZIP-209] Sean Bowe. *Prohibit Negative Shielded Value Pool Balances*. Zcash Improvement Proposal 209. Created February 25, 2019. URL: `https://zips.z.cash/zip-0209` (visited on 2020-11-05) (↑ p47, 50, 141).

[ZIP-211] Daira Hopwood. *Disabling Addition of New Value to the Sprout Value Pool*. Zcash Improvement Proposal 211. Created March 29, 2019. URL: `https://zips.z.cash/zip-0211` (visited on 2020-06-01) (↑ p40, 109, 113, 142, 143).

[ZIP-212] Sean Bowe. *Allow Recipient to Derive Sapling Ephemeral Secret from Note Plaintext*. Zcash Improvement Proposal 212. Created March 31, 2019. URL: `https://zips.z.cash/zip-0212` (visited on 2020-06-01) (↑ p58, 113, 117, 143).

[ZIP-213] Jack Grigg. *Shielded Coinbase*. Zcash Improvement Proposal 213. Created March 30, 2019. URL: `https://zips.z.cash/zip-0213` (visited on 2020-03-20) (↑ p112, 118, 131).

[ZIP-214] Daira Hopwood. *Consensus rules for a Zcash Development Fund*. Zcash Improvement Proposal 214. Created February 28, 2020. URL: `https://zips.z.cash/zip-0214` (visited on 2020-03-24) (↑ p113, 131, 142, 144).

[ZIP-215] Henry de Valance. *Explicitly Defining and Modifying Ed25519 Validation Rules*. Zcash Improvement Proposal 215. Created April 27, 2020. URL: `https://zips.z.cash/zip-0215` (visited on 2020-05-27) (↑ p113, 143, 196).

[ZIP-216] Jack Grigg and Daira Hopwood. *Require Canonical Point Encodings*. Zcash Improvement Proposal 216. Created February 11, 2021. URL: `https://zips.z.cash/zip-0216` (visited on 2021-02-25) (↑ p37, 38, 64, 86, 107, 113).

[ZIP-221] Jack Grigg. *FlyClient - Consensus-Layer Changes*. Zcash Improvement Proposal 221. Created March 30, 2019. URL: `https://zips.z.cash/zip-0221` (visited on 2020-03-19) (↑ p112, 122, 123, 124).

[ZIP-224] Daira Hopwood, Jack Grigg, Sean Bowe, Kris Nuttycombe, and Ying Tong Lai. *Orchard Shielded Protocol*. Zcash Improvement Proposal 224. Created February 27, 2021. URL: `https://zips.z.cash/zip-0225` (visited on 2021-03-21) (↑ p113).

[ZIP-225] Daira Hopwood, Jack Grigg, Sean Bowe, Kris Nuttycombe, and Ying Tong Lai. *Version 5 Transaction Format*. Zcash Improvement Proposal 225. Created February 28, 2021. URL: `https://zips.z.cash/zip-0225` (visited on 2021-03-21) (↑ p46, 48, 51, 113).

[ZIP-243] Jack Grigg and Daira Hopwood. *Transaction Signature Validation for Sapling*. Zcash Improvement Proposal 243. Created April 10, 2018. URL: `https://zips.z.cash/zip-0243` (visited on 2019-08-28) (↑ p46, 48, 52, 69, 112).

[ZIP-244] Kris Nuttycombe and Daira Hopwood. *Transaction Identifier Non-Malleability*. Zcash Improvement Proposal 244. Created January 6, 2021. URL: `https://zips.z.cash/zip-0244` (visited on 2021-01-10) (↑ p46, 48, 51, 113, 122, 124).

[ZIP-250] Daira Hopwood. *Deployment of the Heartwood Network Upgrade*. Zcash Improvement Proposal 250. Created February 28, 2020. URL: `https://zips.z.cash/zip-0250` (visited on 2020-03-20) (↑ p46, 112).

[ZIP-251] Daira Hopwood. *Deployment of the Canopy Network Upgrade*. Zcash Improvement Proposal 251. Created February 28, 2020. URL: `https://zips.z.cash/zip-0251` (visited on 2020-03-24) (↑ p46, 112, 113, 144).

[ZIP-252] Daira Hopwood. *Deployment of the NU5 Network Upgrade*. Zcash Improvement Proposal 252. Reserved. URL: `https://zips.z.cash/zip-0252` (visited on 2021-01-10) (↑ p46).

[ZIP-302] Jay Graber and Jack Grigg. *Standardized Memo Field Format*. Zcash Improvement Proposal 302. Reserved. URL: `https://github.com/zcash/zips/pull/105` (visited on 2020-02-13) (↑ p104, 144).

# Appendices

# A Circuit Design

## A.1 Quadratic Constraint Programs

**Sapling** defines two circuits, Spend and Output, each implementing an abstract *statement* described in § 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55 and § 4.17.3 ***'Output Statement (Sapling)'*** on p. 56 respectively. It also adds a Groth16 circuit for the *JoinSplit statement* described in § 4.17.1 ***'JoinSplit Statement (Sprout)'*** on p. 54.

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, section 2.3] [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 § 5.4.9.3 'Jubjub' on p. 94.

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 § 2 ***'Notation'*** on p. 9.

## A.2 Elliptic curve background

The **Sapling** circuits make use of a *complete twisted Edwards elliptic curve* ("*ctEdwards curve*") Jubjub, defined in § 5.4.9.3 'Jubjub' on p. 94, 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 § 5.4.9.3 'Jubjub' on p. 94 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.*

*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

Each of the following sections describes how to implement a particular component of the circuit, and counts the number of constraints required. Some components make use of others; the order of presentation is "bottom-up".

It is important for security to ensure that variables intended to be of boolean type are boolean-constrained; and for efficiency that they are boolean-constrained only once. We explicitly state for the boolean inputs and outputs of each component whether they are boolean-constrained by the component, or are assumed to have been boolean-constrained separately.

Affine coordinates for elliptic curve points are assumed to represent points on the relevant curve, unless otherwise specified.

In this section, variables have type $\mathbb{F}_{r_{\mathbb{S}}}$ unless otherwise specified. In contrast to most of this document, we use zero-based indexing in order to more closely match the implementation.

### A.3.1 Operations on individual bits

#### A.3.1.1 Boolean constraints

A boolean constraint $b \in \mathbb{B}$ can be implemented as:

$(1 - b) \times (b) = (0)$

#### A.3.1.2 Conditional equality

The constraint "either $a = 0$ or $b = c$" can be implemented as:

$$\big(a\big) \times \big(b - c\big) = \big(0\big)$$

#### A.3.1.3 Selection constraints

A selection constraint $(b \mathbin{?} x : y) = z$, where $b \mathrel{⦂} \mathbb{B}$ has been boolean-constrained, can be implemented as:

$$\big(b\big) \times \big(y - x\big) = \big(y - z\big)$$

#### A.3.1.4 Nonzero constraints

Since only nonzero elements of $\mathbb{F}_{r_\mathbb{S}}$ have a multiplicative inverse, the assertion $a \neq 0$ can be implemented by witnessing the inverse, $a_{\mathsf{inv}} = a^{-1} \pmod{r_\mathbb{S}}$:

$$\big(a_{\mathsf{inv}}\big) \times \big(a\big) = \big(1\big)$$

This technique comes from [SVPBABW2012, Appendix D.1].

**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

An exclusive-or operation $a \oplus b = c$, where $a, b \mathrel{⦂} \mathbb{B}$ are already boolean-constrained, can be implemented in one constraint as:

$$\big(2 \cdot a\big) \times \big(b\big) = \big(a + b - c\big)$$

This automatically boolean-constrains $c$. Its correctness can be seen by checking the truth table of $(a, b)$.

### A.3.2 Operations on multiple bits

#### A.3.2.1 [Un]packing modulo $r_\mathbb{S}$

Let $n \mathrel{⦂} \mathbb{N}^+$ be a constant. The operation of converting a field element, $a \mathrel{⦂} \mathbb{F}_{r_\mathbb{S}}$, to a sequence of boolean variables $b_{0\,..\,n-1} \mathrel{⦂} \mathbb{B}^{[n]}$ such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i \pmod{r_\mathbb{S}}$, is called "*unpacking*". The inverse operation is called "*packing*".

In the *quadratic 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:

$$\left(\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_{\mathbb{S}})\right) \times (1) = (a)$$

**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 $a \typecolon \mathbb{F}_{r_{\mathbb{S}}}$ as a sequence of 255 bits, corresponding to $\mathsf{I2LEBSP}_{255}(a)$ and $\mathsf{I2LEBSP}_{255}(a + r_{\mathbb{S}})$. This is a potential hazard, but it may or may not be necessary to force use of the canonical representation $\mathsf{I2LEBSP}_{255}(a)$, depending on the context in which the [un]packing operation is used. We therefore do not consider this to be part of the [un]packing operation itself.

### A.3.2.2 Range check

Let $n \typecolon \mathbb{N}^+$ be a constant, and let $a = \sum_{i=0}^{n-1} a_i \cdot 2^i \typecolon \mathbb{N}$. Suppose we want to constrain $a \leq c$ for some ***constant*** $c = \sum_{i=0}^{n-1} c_i \cdot 2^i \typecolon \mathbb{N}$.

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 $\Pi_m$ 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 $(\Pi_{i+1}) \times (a_i) = (\Pi_i)$.

Then we constrain the $a_i$ as follows:

For $i$ from $n-1$ down to 0,

- if $c_i = 0$, constrain $(1 - \Pi_{i+1} - a_i) \times (a_i) = (0)$;
- if $c_i = 1$, boolean-constrain $a_i$ as in § A.3.1.1 ***'Boolean constraints'*** on p. 174.

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.*

*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* $m \in \{0\,..\,n-1\}$, $A_m \leq C_m$ *iff 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 $\big(1 - \Pi_{i+1} - a_i\big) \times \big(a_i\big) = \big(0\big)$ 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 $A_m \leq C_m$ iff $a_m \leq c_m$.
    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 § A.3.3.2 ***'ctEdwards [de]compression and validation'*** on p. 178 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 $\Pi_m$ 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 $\Pi_{m+N-1}$: $R = \prod_{i=0}^{N-1} X_i$. This can be computed in 3 constraints for any $N$; boolean-constrain the output $R$, and then add constraints

$$\Big(N - \sum_{i=0}^{N-1} X_i\Big) \times \big(\mathsf{inv}\big) = \big(1 - R\big) \quad \text{to enforce that } \sum_{i=0}^{N-1} X_i \neq N \text{ when } R = 0;$$

$$\Big(N - \sum_{i=0}^{N-1} X_i\Big) \times \big(R\big) = \big(0\big) \quad \text{to enforce that } \sum_{i=0}^{N-1} X_i = N \text{ when } R = 1.$$

where $\mathsf{inv}$ is witnessed as $\Big(N - \sum_{i=0}^{N-1} X_i\Big)^{-1}$ 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 $\Pi_m$ (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 $\Pi$ 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

#### A.3.3.1 Checking that Affine-ctEdwards coordinates are on the curve

To check that $(u, \mathsf{v})$ is a point on the *ctEdwards curve*, the **Sapling** circuit uses $4$ constraints:

$$\begin{aligned}
&\big(u\big) \times \big(u\big) = \big(uu\big)\\
&\big(\mathsf{v}\big) \times \big(\mathsf{v}\big) = \big(\mathsf{vv}\big)\\
&\big(uu\big) \times \big(\mathsf{vv}\big) = \big(uu\mathsf{vv}\big)\\
&\big(a_{\mathbb{J}}\cdot uu + \mathsf{vv}\big) \times \big(1\big) = \big(1 + d_{\mathbb{J}}\cdot uu\mathsf{vv}\big)
\end{aligned}$$

**Non-normative note:** The last two constraints can be combined into $\big(d_{\mathbb{J}}\cdot uu\big) \times \big(\mathsf{vv}\big) = \big(a_{\mathbb{J}}\cdot uu + \mathsf{vv} - 1\big)$. The **Sapling** circuit does not use this optimization.

#### A.3.3.2 ctEdwards [de]compression and validation

Define DecompressValidate ⦂ CompressedCtEdwardsJubjub → AffineCtEdwardsJubjub as follows:

```
DecompressValidate($\tilde{u}, \mathsf{v}$) :
    // Prover supplies the $u$-coordinate.
    Let $u$ ⦂ $\mathbb{F}_{r_{\mathbb{S}}}$.
    // § A.3.3.1 ‘Checking that Affine-ctEdwards coordinates are on the curve’ on p. 178.
    Check that $(u, \mathsf{v})$ is a point on the ctEdwards curve.
    // § A.3.2.1 ‘[Un]packing modulo $r_{\mathbb{S}}$’ on p. 175.
    Unpack $u$ to $\sum_{i=0}^{254} u_i \cdot 2^i$, equating $\tilde{u}$ with $u_0$.
    // § A.3.2.2 ‘Range check’ on p. 176.
    Check that $\sum_{i=0}^{254} u_i \cdot 2^i \leq r_{\mathbb{S}} - 1$.
    Return $(u, \mathsf{v})$.
```

This costs $4$ constraints for the curve equation check, $1$ constraint for the unpacking, and $387$ constraints for the range check (as computed in § A.3.2.2 ***‘Range check’*** on p. 176) 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 $(u, \mathsf{v})$ 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 ↔ Montgomery conversion

Define CtEdwardsToMont ⦂ AffineCtEdwardsJubjub → AffineMontJubjub as follows:

$$\mathsf{CtEdwardsToMont}(u, \mathsf{v}) = \left(\frac{1 + \mathsf{v}}{1 - \mathsf{v}},\ \sqrt[+]{-40964} \cdot \frac{1 + \mathsf{v}}{(1 - \mathsf{v}) \cdot u}\right) \qquad [1 - \mathsf{v} \neq 0 \text{ and } u \neq 0]$$

Define MontToCtEdwards ⦂ AffineMontJubjub → 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*:

$$\begin{aligned}\big(y\big) &\times \big(u\big) = \big(\sqrt[+]{-40964}\cdot x\big)\\ \big(x+1\big) &\times \big(v\big) = \big(x-1\big)\end{aligned}$$

The above conversions should only be used if the input is guaranteed to be a point on the relevant curve. If that is the case, the theorems below enumerate all exceptional inputs that may violate the side-conditions.

**Theorem A.3.2.** *Exceptional points (ctEdwards $\to$ Montgomery).*

*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, Definition 2.1]). 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 $\to$ ctEdwards).*

*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 A.2.1 on p. 174.

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, section 4.3.5]). 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 § A.2 ***'Elliptic curve background'*** on p. 173, the *ctEdwards curve* referred to in the proof is an isomorphic rescaling of the Jubjub curve.)

### A.3.3.4 Affine-Montgomery arithmetic

The incomplete *affine-Montgomery* addition formulae given in [BL2017, section 4.3.2] are:

$$\begin{aligned}&x_3 = B_{\mathbb{M}}\cdot\lambda^2 - A_{\mathbb{M}} - x_1 - x_2\\ &y_3 = (x_1 - x_3)\cdot\lambda - y_1\\ &\text{where } \lambda = \begin{cases}\dfrac{3\cdot x_1^2 + 2\cdot A_{\mathbb{M}}\cdot x_1 + 1}{2\cdot B_{\mathbb{M}}\cdot y_1}, & \text{if } x_1 = x_2\\ \dfrac{y_2 - y_1}{x_2 - x_1}, & \text{otherwise.}\end{cases}\end{aligned}$$

The following theorem helps to determine when these incomplete addition formulae can be safely used:

**Theorem A.3.4.** *Distinct-$x$ theorem.*

*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 $\big\{-\frac{s-1}{2}\,..\,\frac{s-1}{2}\big\} \setminus \{0\}$. 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*

$$\begin{aligned}\big(x_2 - x_1\big) &\times \big(\lambda\big) = \big(y_2 - y_1\big)\\ \big(B_{\mathbb{M}}\cdot\lambda\big) &\times \big(\lambda\big) = \big(A_{\mathbb{M}} + x_1 + x_2 + x_3\big)\\ \big(x_1 - x_3\big) &\times \big(\lambda\big) = \big(y_3 + y_1\big)\end{aligned}$$

*implement the affine-Montgomery addition $P_1 + P_2 = (x_3, y_3)$ for all such $P_{1\,..\,2}$.*

*Proof.* The given constraints are equivalent to the Montgomery addition formulae under the side condition 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:

$$
\begin{aligned}
&(x) \times (x) = (xx) \\
&(2 \cdot B_{\mathbb{M}} \cdot y) \times (\lambda) = (3 \cdot xx + 2 \cdot A_{\mathbb{M}} \cdot x + 1) \\
&(B_{\mathbb{M}} \cdot \lambda) \times (\lambda) = (A_{\mathbb{M}} + 2 \cdot x + x_3) \\
&(x - x_3) \times (\lambda) = (y_3 + y)
\end{aligned}
$$

This doubling formula is valid when $y \neq 0$, which is the case when $(x, y)$ is not the point $(0, 0)$ (the only point of order 2), as proven in Theorem A.2.1 on p. 174.

#### A.3.3.5 Affine-ctEdwards arithmetic

Formulae for *affine-ctEdwards* addition are given in [BBJLP2008, section 6]. With a change of variable names to match our convention, the formulae for $(u_1, v_1) + (u_2, v_2) = (u_3, v_3)$ are:

$$
\begin{aligned}
u_3 &= \frac{u_1 \cdot v_2 + v_1 \cdot u_2}{1 + d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2} \\
v_3 &= \frac{v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2}{1 - d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2}
\end{aligned}
$$

We use an optimized implementation found by Daira Hopwood making use of an observation by Bernstein and Lange in [BL2017, last paragraph of section 4.5.2]:

$$
\begin{aligned}
&(u_1 + v_1) \times (v_2 - a_{\mathbb{J}} \cdot u_2) = (T) \\
&(u_1) \times (v_2) = (A) \\
&(v_1) \times (u_2) = (B) \\
&(d_{\mathbb{J}} \cdot A) \times (B) = (C) \\
&(1 + C) \times (u_3) = (A + B) \\
&(1 - C) \times (v_3) = (T - A + a_{\mathbb{J}} \cdot B)
\end{aligned}
$$

The correctness of this implementation can be seen by expanding $T - A + a_{\mathbb{J}} \cdot B$:

$$
\begin{aligned}
T - A + a_{\mathbb{J}} \cdot B &= (u_1 + v_1) \cdot (v_2 - a_{\mathbb{J}} \cdot u_2) - u_1 \cdot v_2 + a_{\mathbb{J}} \cdot v_1 \cdot u_2 \\
&= v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2 + u_1 \cdot v_2 - a_{\mathbb{J}} \cdot v_1 \cdot u_2 - u_1 \cdot v_2 + a_{\mathbb{J}} \cdot v_1 \cdot u_2 \\
&= v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2
\end{aligned}
$$

The above addition formulae are "unified", that is, they can also be used for doubling. *Affine-ctEdwards* doubling $[2]\,(u, v) = (u_3, v_3)$ can also be implemented slightly more efficiently as:

$$
\begin{aligned}
&\big(u + v\big) \times \big(v - a_{\mathbb{J}}{\cdot}u\big) = \big(T\big)\\
&\big(u\big) \times \big(v\big) = \big(A\big)\\
&\big(d_{\mathbb{J}}{\cdot}A\big) \times \big(A\big) = \big(C\big)\\
&\big(1 + C\big) \times \big(u_3\big) = \big(2{\cdot}A\big)\\
&\big(1 - C\big) \times \big(v_3\big) = \big(T + (a_{\mathbb{J}} - 1){\cdot}A\big)
\end{aligned}
$$

This implementation is obtained by specializing the addition formulae to $(u, v) = (u_1, v_1) = (u_2, v_2)$ and observing that $u \cdot v = A = B$.

#### A.3.3.6 Affine-ctEdwards nonsmall-order check

In order to avoid small-subgroup attacks, we check that certain points used in the circuit are not of small order. In practice the **Sapling** circuit uses this in combination with a check that the coordinates are on the curve (§ A.3.3.1 ***'Checking that Affine-ctEdwards coordinates are on the curve'*** on p. 178), 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 § A.3.3.5 ***'Affine-ctEdwards arithmetic'*** on p. 180) and checks that the resulting $u$-coordinate is not 0 (as in § A.3.1.4 ***'Nonzero constraints'*** on p. 175).

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 § A.3.3.1 ***'Checking that Affine-ctEdwards coordinates are on the curve'*** on p. 178); 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

If the base point $B$ is fixed for a given scalar multiplication $[k]\,B$, we can fully precompute window tables for each window position.

It is most efficient to use 3-bit fixed windows. Since the length of $r_{\mathbb{J}}$ is 252 bits, we need 84 windows.

Express $k$ in base 8, i.e. $k = \sum_{i=0}^{83} k_i {\cdot} 8^i$.

Then $[k]\,B = \sum_{i=0}^{83} w_{(B,\,i,\,k_i)}$, where $w_{(B,\,i,\,k_i)} = [k_i {\cdot} 8^i]\,B$.

We precompute all of $w_{(B,\,i,\,s)}$ for $i \in \{0\,..\,83\}, s \in \{0\,..\,7\}$.

To look up a given window entry $w_{(B,\,i,\,s)} = (u_s, v_s)$, where $s = 4 \cdot s_2 + 2 \cdot s_1 + s_0$, we use:

$$
\begin{aligned}
&\big(s_1\big) \times \big(s_2\big) = \big(s_\&\big)\\
&\big(s_0\big) \times \big(- u_0 \cdot s_\& + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_\& - u_2 \cdot s_1 + u_4 \cdot s_\& - u_4 \cdot s_2 - u_6 \cdot s_\&\\
&\qquad\quad + u_1 \cdot s_\& - u_1 \cdot s_2 - u_1 \cdot s_1 + u_1 - u_3 \cdot s_\& + u_3 \cdot s_1 - u_5 \cdot s_\& + u_5 \cdot s_2 + u_7 \cdot s_\&\big) =\\
&\quad\big(u_s - u_0 \cdot s_\& + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_\& - u_2 \cdot s_1 + u_4 \cdot s_\& - u_4 \cdot s_2 - u_6 \cdot s_\&\big)\\
&\big(s_0\big) \times \big(- v_0 \cdot s_\& + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_\& - v_2 \cdot s_1 + v_4 \cdot s_\& - v_4 \cdot s_2 - v_6 \cdot s_\&\\
&\qquad\quad + v_1 \cdot s_\& - v_1 \cdot s_2 - v_1 \cdot s_1 + v_1 - v_3 \cdot s_\& + v_3 \cdot s_1 - v_5 \cdot s_\& + v_5 \cdot s_2 + v_7 \cdot s_\&\big) =\\
&\quad\big(v_s - v_0 \cdot s_\& + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_\& - v_2 \cdot s_1 + v_4 \cdot s_\& - v_4 \cdot s_2 - v_6 \cdot s_\&\big)
\end{aligned}
$$

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 § A.3.3.5 ***'Affine-ctEdwards arithmetic'*** on p. 180), for a total of 750 constraints.

Fixed-base scalar multiplication is also used in two places with shorter scalars:

- § A.3.6 ***'Homomorphic Pedersen Commitment'*** on p. 186 uses a 64-bit scalar for the v input to $\mathsf{ValueCommit}^{\mathsf{Sapling}}$[redacted] requiring 22 windows at a cost of $3 \cdot 22 - 1 + 6 \cdot 21 = 191$ constraints;
- § A.3.3.10 ***'Mixing Pedersen hash'*** on p. 185 uses a 32-bit scalar 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 § A.3.3.9 ***'Pedersen hash'*** on p. 183. However since there are only three instances of fixed-base scalar multiplication in the *Spend circuit* and two in the *Output circuit*[11], 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

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 $\mathsf{Base}_i$ 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 $\mathsf{Acc}_i = \mathsf{Acc}_{i-1} + \mathsf{Addend}_i$
```

[11] A *Pedersen commitment* uses fixed-base scalar multiplication as a subcomponent.

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.

**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 § A.3.3.9 ***'Pedersen hash'*** on p. 183. However since there are only two instances of variable-base scalar multiplication in the *Spend circuit* and one in the *Output circuit*, the additional complexity was not considered justified for **Sapling**.

### A.3.3.9 Pedersen hash

The specification of the *Pedersen hashes* used in **Sapling** is given in § 5.4.1.7 ***'Pedersen Hash Function'*** on p. 72. It is based on the scheme from [CvHP1991, section 5.2] –for which a tighter security reduction to the Discrete Logarithm Problem was given in [BGG1995]– but tailored to allow several optimizations in the circuit implementation.

*Pedersen hashes* are the single most commonly used primitive in the **Sapling** circuits. $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$ *Pedersen hash* instances are used in the *Spend circuit* to check a *Merkle path* to the *note commitment* of the *note* being spent. We also reuse the *Pedersen hash* implementation to construct the *commitment scheme* $\mathsf{NoteCommit}^{\mathsf{Sapling}}$.

This motivates considerable attention to optimizing this circuit implementation of this primitive, even at the cost of complexity.

First, we use a windowed scalar multiplication algorithm with signed digits. Each 3-bit message chunk corresponds to a window; the chunk is encoded as an integer from the set $\mathsf{Digits} = \{-4 \,..\, 4\} \setminus \{0\}$. This allows a more efficient lookup of the window entry for each chunk than if the set $\{1 \,..\, 8\}$ had been used, because a point can be conditionally negated using only a single constraint.

Next, we optimize the cost of point addition by allowing as many additions as possible to be performed on the *Montgomery curve*. An incomplete Montgomery addition costs 3 constraints, in comparison with 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 § A.3.3.4 ***'Affine-Montgomery arithmetic'*** on p. 179 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:

$$\mathsf{PedersenHashToPoint}(D, M) = \sum_{j=1}^{N} [\langle M_j \rangle]\, \mathcal{I}_j^{\mathsf{D}}$$

where $\langle\bullet\rangle$ and $\mathcal{I}_j^{\mathsf{D}}$ are defined as in § 5.4.1.7 ***'Pedersen Hash Function'*** on p. 72.

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 5.4.1 on p. 74 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 $\mathcal{I}_j^{\mathsf{D}}$ (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 $[\langle M_j \rangle]\, \mathcal{I}_j^{\mathsf{D}}$ to be converted to ctEdwards form are of prime order. From Theorem A.3.3 on p. 179, 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.*

*For all disjoint nonempty subsets $S$ and $S'$ of $\{1 \mathinner{..} c\}$, 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 $S$, $S'$, $m$, $\Theta$ is a counterexample. Taking the multiplication by $\Theta$ 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 $\mathsf{enc}' : \{-1, 1\} \times \mathbb{B}^{[3]} \to \{0 \mathinner{..} 8\} \setminus \{4\}$ 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 5.4.1 on p. 74. By adding $\Delta$ to both sides, we get

$$\sum_{j \in S} \mathsf{enc}'_1(m_j) \cdot 2^{4 \cdot (j-1)} + \sum_{j \in \{1 \mathinner{..} c\} \setminus S} 4 \cdot 2^{4 \cdot (j-1)} = \sum_{j' \in S'} \mathsf{enc}'_\Theta(m_{j'}) \cdot 2^{4 \cdot (j'-1)} + \sum_{j' \in \{1 \mathinner{..} c\} \setminus S'} 4 \cdot 2^{4 \cdot (j'-1)}$$

where all of the $\mathsf{enc}'_1(m_j)$ and $\mathsf{enc}'_\Theta(m_{j'})$ are in $\{0 \mathinner{..} 8\} \setminus \{4\}$.

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 $\{1 \mathinner{..} c\}$ and $S$ is nonempty, $S \cap (\{1 \mathinner{..} c\} \setminus S')$ 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 A.3.4 on p. 179. (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 $s_\& = s_0 \mathbin{\&} s_1$:

$$\big(s_0\big) \times \big(s_1\big) = \big(s_\&\big)$$

Let $(x_k, y_k) = [k]\, P$ for $k \in \{1 \mathinner{..} 4\}$. 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 $s_\&$:

$$\text{let } x_S = x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_\&$$
$$\text{let } y_R = y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\&$$

We implement the conditional negation as $\big(2 \cdot y_R\big) \times \big(s_2\big) = \big(y_R - y_S\big)$. After substitution of $y_R$ this becomes:

$$\begin{aligned} &\big(2 \cdot (y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\&)\big) \times \big(s_2\big) = \\ &\qquad \big(y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\& - y_S\big) \end{aligned}$$

Then we substitute $x_S$ into the Montgomery addition constraints from § A.3.3.4 ***'Affine-Montgomery arithmetic'*** on p. 179, as follows:

$$\begin{aligned}
&\big(x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_{\&} - x_A\big) \times \big(\lambda\big) \;=\; \big(y_S - y_A\big) \\
&\big(B_{\mathbb{M}} \cdot \lambda\big) \times \big(\lambda\big) \;=\; \big(A_{\mathbb{M}} + x_A + x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_{\&} + x_C\big) \\
&\big(x_A - x_C\big) \times \big(\lambda\big) \;=\; \big(y_C + y_A\big)
\end{aligned}$$

(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 $\ell$ bits (where $\ell$ 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 $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ (§ 5.4.8.2 ***'Windowed Pedersen commitments'*** on p. 88), $\ell = 6 + \ell_{\mathsf{value}} + 2 \cdot \ell_{\mathbb{J}} = 582$, $c = 194$, and $n = 4$, so the cost of the hash alone is $984$ constraints.

### A.3.3.10 Mixing Pedersen hash

A mixing *Pedersen hash* is used to compute $\mathsf{\rho}$ from $\mathsf{cm}$ and $\mathsf{pos}$ in § 4.16 ***'Note Commitments and Nullifiers'*** on p. 53. It takes as input a *Pedersen commitment* $P$, and hashes it with another input $x$.

Let $\mathcal{J}^{\mathsf{Sapling}}$ be as defined in § 5.4.1.8 ***'Mixing Pedersen Hash Function'*** on p. 74.

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 (§ A.3.3.7 ***'Fixed-base Affine-ctEdwards scalar multiplication'*** on p. 181), and 6 constraints for a ctEdwards addition (§ A.3.3.5 ***'Affine-ctEdwards arithmetic'*** on p. 180), for a total of 98 constraints.

### A.3.4 Merkle path check

Checking each layer of a Merkle authentication path, as described in § 4.9 ***'Merkle Path Validity'*** on p. 45, 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 § A.3.2.1 ***'[Un]packing modulo $r_{\mathbb{S}}$'*** on p. 175; that is, it is ***not*** necessary to ensure that the left or right inputs to the hash represent integers in the range $\{0 \mathrel{..} r_{\mathbb{S}} - 1\}$. 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 (§ A.3.3.9 ***'Pedersen hash'*** on p. 183), 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

We construct *windowed Pedersen commitments* by reusing the Pedersen hash implementation described in § A.3.3.9 ***'Pedersen hash'*** on p. 183, and adding a randomized point:

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

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 § A.3.3.9 ***'Pedersen hash'*** on p. 183, 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

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:

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

In the case that we need for ValueCommit, v has $64$ bits[12]. This value is given as a bit representation, which does not need to be constrained equal to an integer.

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

BLAKE2s is defined in [ANWW2013]. Its main subcomponent is a “$G$ function”, defined as follows:

$$\begin{aligned}
&G ⦂ \{0\,..\,9\} \times \{0\,..\,2^{32}{-}1\}^{[4]} \to \{0\,..\,2^{32}{-}1\}^{[4]} \\
&G(a, b, c, d, x, y) = (a'', b'', c'', d'') \text{ where} \\
&\quad a' = (a + b + x) \bmod 2^{32} \\
&\quad d' = (d \oplus a') \ggg 16 \\
&\quad c' = (c + d') \bmod 2^{32} \\
&\quad b' = (b \oplus c') \ggg 12 \\
&\quad a'' = (a' + b' + y) \bmod 2^{32} \\
&\quad d'' = (d' \oplus a'') \ggg 8 \\
&\quad c'' = (c' + d'') \bmod 2^{32} \\
&\quad b'' = (b' \oplus c'') \ggg 7
\end{aligned}$$

The following table is used to determine which message words the $x$ and $y$ arguments to $G$ are selected from:

$$\begin{aligned}
\sigma_0 &= [\ 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11,\ 12,\ 13,\ 14,\ 15\ ] \\
\sigma_1 &= [\ 14,\ 10,\ 4,\ 8,\ 9,\ 15,\ 13,\ 6,\ 1,\ 12,\ 0,\ 2,\ 11,\ 7,\ 5,\ 3\ ] \\
\sigma_2 &= [\ 11,\ 8,\ 12,\ 0,\ 5,\ 2,\ 15,\ 13,\ 10,\ 14,\ 3,\ 6,\ 7,\ 1,\ 9,\ 4\ ] \\
\sigma_3 &= [\ 7,\ 9,\ 3,\ 1,\ 13,\ 12,\ 11,\ 14,\ 2,\ 6,\ 5,\ 10,\ 4,\ 0,\ 15,\ 8\ ] \\
\sigma_4 &= [\ 9,\ 0,\ 5,\ 7,\ 2,\ 4,\ 10,\ 15,\ 14,\ 1,\ 11,\ 12,\ 6,\ 8,\ 3,\ 13\ ] \\
\sigma_5 &= [\ 2,\ 12,\ 6,\ 10,\ 0,\ 11,\ 8,\ 3,\ 4,\ 13,\ 7,\ 5,\ 15,\ 14,\ 1,\ 9\ ] \\
\sigma_6 &= [\ 12,\ 5,\ 1,\ 15,\ 14,\ 13,\ 4,\ 10,\ 0,\ 7,\ 6,\ 3,\ 9,\ 2,\ 8,\ 11\ ] \\
\sigma_7 &= [\ 13,\ 11,\ 7,\ 14,\ 12,\ 1,\ 3,\ 9,\ 5,\ 0,\ 15,\ 4,\ 8,\ 6,\ 2,\ 10\ ] \\
\sigma_8 &= [\ 6,\ 15,\ 14,\ 9,\ 11,\ 3,\ 0,\ 8,\ 12,\ 2,\ 13,\ 7,\ 1,\ 4,\ 10,\ 5\ ] \\
\sigma_9 &= [\ 10,\ 2,\ 8,\ 4,\ 7,\ 6,\ 1,\ 5,\ 15,\ 11,\ 9,\ 14,\ 3,\ 12,\ 13,\ 0\ ]
\end{aligned}$$

[12] It would be sufficient to use $51$ bits, which accomodates the range $\{0\,..\,\mathsf{MAX\_MONEY}\}$, but the **Sapling** circuit uses $64$.

The Initialization Vector is defined as:

$$\mathsf{IV} ⦂ \{0\,..\,2^{32}{-}1\}^{[8]} := [\,\texttt{0x6A09E667},\ \texttt{0xBB67AE85},\ \texttt{0x3C6EF372},\ \texttt{0xA54FF53A} \\ \texttt{0x510E527F},\ \texttt{0x9B05688C},\ \texttt{0x1F83D9AB},\ \texttt{0x5BE0CD19}\,]$$

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 $\mathsf{BLAKE2s\text{-}256} ⦂ (p ⦂ \mathbb{BY}^{[8]}) \times (x ⦂ \mathbb{BY}^{[64]}) \to \mathbb{BY}^{[32]}$ as:
  let $\mathsf{PB} ⦂ \mathbb{BY}^{[32]} = [32, 0, 1, 1] \,||\, [\texttt{0x00}]^{20} \,||\, p$
  let $[\,t_0, t_1, f_0, f_1\,] ⦂ \{0\,..\,2^{32}{-}1\}^{[4]} = [\,0, 0, 0, \texttt{0xFFFFFFFF}, 0\,]$
  let $h ⦂ \{0\,..\,2^{32}{-}1\}^{[8]} = \llbracket\, \mathsf{LEOS2IP}_{32}(\mathsf{PB}_{4\cdot i\,..\,4\cdot i\,+\,3}) \oplus \mathsf{IV}_i$ for $i$ from 0 up to 7 $\rrbracket$
  let $m ⦂ \{0\,..\,2^{32}{-}1\}^{[16]} = \llbracket\, \mathsf{LEOS2IP}_{32}(x_{4\cdot i\,..\,4\cdot i\,+\,3})$ for $i$ from 0 up to 15 $\rrbracket$
  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 $\mathsf{LEBS2OSP}_{256}(\mathsf{concat}_{\mathbb{B}}(\llbracket\, \mathsf{I2LEBSP}_{32}(h_i \oplus v_i \oplus v_{i+8})$ for $i$ from 0 up to 7 $\rrbracket))$
```

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 § A.3.1.5 ***'Exclusive-or constraints'*** on p. 175.

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 $G$);
- $\mathsf{ceiling}\left(\frac{10 \cdot 8 \cdot 4}{7}\right) = 46$ 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

The **Sapling** Spend *statement* is defined in § 4.17.2 ***‘Spend Statement (Sapling)’*** on p. 55.

The primary input is

$$\begin{aligned}&\big(\mathsf{rt}^{\mathsf{Sapling}} ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]},\\ &\ \mathsf{cv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit}’\mathsf{.Output}\\ &\ \mathsf{nf}^{\mathsf{old}} ⦂ \mathbb{BY}^{[\ell_{\mathsf{PRFnfSapling}}/8]},\\ &\ \mathsf{rk} ⦂ \mathsf{SpendAuthSig}\big)\mathsf{.Public},\end{aligned}$$

which is encoded as $8\ \mathbb{F}_{r_{\mathbb{S}}}$ elements (starting with the fixed element 1 required by Groth16):

$$[1, \mathcal{U}(\mathsf{rk}), \mathcal{V}(\mathsf{rk}), \mathcal{U}(\mathsf{cv}^{\mathsf{old}}), \mathcal{V}(\mathsf{cv}^{\mathsf{old}}), \mathsf{LEBS2IP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}\left(\mathsf{rt}^{\mathsf{Sapling}}\right), \mathsf{LEBS2IP}_{254}\big(\mathsf{nf}^{\mathsf{old}}_{\star}\,{}_{0\,..\,253}\big), \mathsf{LEBS2IP}_{2}\big(\mathsf{nf}^{\mathsf{old}}_{\star}\,{}_{254\,..\,255}\big)]$$

where $\mathsf{nf}^{\mathsf{old}}_{\star} = \mathsf{LEOS2BSP}_{\ell_{\mathsf{PRFnfSapling}}}(\mathsf{nf}^{\mathsf{old}})$.

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 $\mathsf{\rho}\star$ used in **Nullifier integrity** are compressed representations of Jubjub curve points. TODO: explain why these are implemented as § A.3.3.2 ***‘ctEdwards [de]compression and validation’*** on p. 178 even though the statement spec doesn’t explicitly say to do validation.

Therefore we have $\mathsf{g_d}$, ak, nk, and $\mathsf{\rho}$ that need to be constrained to valid Jubjub curve points as described in § A.3.3.2 ***‘ctEdwards [de]compression and validation’*** on p. 178.

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 |
|---|---|---|---|
| $\mathsf{ak}$ is on the curve TODO: FIXME also decompressed below | $\mathsf{ak} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$ | 4 | § A.3.3.1 on p. 178 |
| $\mathsf{ak}$ is not small order | **Small order checks** | 16 | § A.3.3.6 on p. 181 |
| $\alpha\star ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\alpha ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\alpha' = [\alpha\star]\,\mathcal{G}^{\mathsf{Sapling}}$ | **Spend authority** | 750 | § A.3.3.7 on p. 181 |
| $\mathsf{rk} = \alpha' + \mathsf{ak}$ | | 6 | § A.3.3.5 on p. 180 |
| inputize $\mathsf{rk}$ TODO: not ccteddecompress-validate => wrong count | $\mathsf{rk} ⦂ \mathsf{SpendAuthSig}^{\mathsf{Sapling}}.\mathsf{Public}$ | 392? | § A.3.3.2 on p. 178 |
| $\mathsf{nsk}\star ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{nsk} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{nk} = [\mathsf{nsk}\star]\,\mathcal{H}^{\mathsf{Sapling}}$ | **Nullifier integrity** | 750 | § A.3.3.7 on p. 181 |
| $\mathsf{ak}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{ak} ⦂ \mathbb{J})$ | **Diversified address integrity** | 392 | § A.3.3.2 on p. 178 |
| $\mathsf{nk}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$ TODO: spec doesn't say to validate $\mathsf{nk}$ since it's calculated | **Nullifier integrity** | 392 | § A.3.3.2 on p. 178 |
| $\mathsf{ivk}\star = \mathsf{I2LEBSP}_{251}(\mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}, \mathsf{nk}))$ † | **Diversified address integrity** | 21006 | § A.3.7 on p. 187 |
| $\mathsf{g_d}$ is on the curve | $\mathsf{g_d} ⦂ \mathbb{J}$ | 4 | § A.3.3.1 on p. 178 |
| $\mathsf{g_d}$ is not small order | **Small order checks** | 16 | § A.3.3.6 on p. 181 |
| $\mathsf{pk_d} = [\mathsf{ivk}\star]\,\mathsf{g_d}$ | **Diversified address integrity** | 3252 | § A.3.3.8 on p. 182 |
| $\mathsf{v}^{\mathsf{old}}_{\star} ⦂ \mathbb{B}^{[64]}$ | $\mathsf{v}^{\mathsf{old}} ⦂ \{0\,..\,2^{64}-1\}$ | 64 | § A.3.1.1 on p. 174 |
| $\mathsf{rcv}\star ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{rcv} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{rcv}}_{(}\mathsf{v}^{\mathsf{old}})$ | **Value commitment integrity** | 947 | § A.3.6 on p. 186 |
| inputize $\mathsf{cv}$ | | ? | |
| $\mathsf{rcm}\star ⦂ \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{rcm} ⦂ \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g_d}, \mathsf{pk_d}, \mathsf{v}^{\mathsf{old}})$ | **Note commitment integrity** | 1740 | § A.3.5 on p. 186 |
| $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm})$ | **Merkle path validity** | 0 | |
| $\mathsf{rt}'$ is the root of a Merkle tree with leaf $\mathsf{cm}_u$, and authentication path $(\mathsf{path}, \mathsf{pos}\star)$ | | $32 \cdot 1380$ | § A.3.4 on p. 186 |
| $\mathsf{pos}\star = \mathsf{I2LEBSP}_{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}(\mathsf{pos})$ | | 1 | § A.3.2.1 on p. 175 |
| if $\mathsf{v}^{\mathsf{old}} \neq 0$ then $\mathsf{rt}' = \mathsf{rt}^{\mathsf{Sapling}}$ | | 1 | § A.3.1.2 on p. 175 |
| inputize $\mathsf{rt}^{\mathsf{Sapling}}$ | | ? | |
| $\rho = \mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})$ | **Nullifier integrity** | 98 | § A.3.3.10 on p. 185 |
| $\rho\star = \mathsf{repr}_{\mathbb{J}}(\rho)$ TODO: spec doesn't say to validate $\rho$ since it's calculated | | 392 | § A.3.3.2 on p. 178 |
| $\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$ | | 21006 | § A.3.7 on p. 187 |
| pack $\mathsf{nf}^{\mathsf{old}}_{0\,..\,253}$ and $\mathsf{nf}^{\mathsf{old}}_{254\,..\,255}$ into two $\mathbb{F}_{r_{\mathbb{S}}}$ inputs | input encoding | 2 | § A.3.2.1 on p. 175 |

† 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 $\mathsf{v}^{\mathsf{old}}_{\star}$ as bit sequences rather than integers. It represents $\mathsf{nf}$ as a bit sequence rather than a byte sequence.

## A.5 The Sapling Output circuit

The **Sapling** Output *statement* is defined in § 4.17.3 ***'Output Statement (Sapling)'*** on p. 56.

The primary input is

$$\begin{aligned}&\big(\mathsf{cv}^{\mathsf{new}} \mathrel{⦂} \mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output},\\ &\ \mathsf{cm}_u \mathrel{⦂} \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}]},\\ &\ \mathsf{epk} \mathrel{⦂} \mathbb{J}\big),\end{aligned}$$

which is encoded as $6$ $\mathbb{F}_{r_{\mathbb{S}}}$ elements (starting with the fixed element $1$ required by Groth16):

$$[1, \mathcal{U}(\mathsf{cv}^{\mathsf{new}}), \mathcal{V}(\mathsf{cv}^{\mathsf{new}}), \mathcal{U}(\mathsf{epk}), \mathcal{V}(\mathsf{epk}), \mathsf{LEBS2IP}_{\ell^{\mathsf{Sapling}}_{\mathsf{Merkle}}}(\mathsf{cm}_u)]$$

The auxiliary input is

$$\begin{aligned}&(\mathsf{g_d} \mathrel{⦂} \mathbb{J},\\ &\ \mathsf{pk}\star_{\mathsf{d}} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathbb{J}}]},\\ &\ \mathsf{v}^{\mathsf{new}} \mathrel{⦂} \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\ &\ \mathsf{rcv}^{\mathsf{new}} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{rcm}^{\mathsf{new}} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\},\\ &\ \mathsf{esk} \mathrel{⦂} \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\})\end{aligned}$$

$\mathsf{ValueCommit}^{\mathsf{Sapling}}.\mathsf{Output}$ is of type $\mathbb{J}$, so we have $\mathsf{cv}^{\mathsf{new}}$, $\mathsf{epk}$, and $\mathsf{g_d}$ that represent Jubjub curve points. However,

- $\mathsf{cv}^{\mathsf{new}}$ will be constrained to an output of $\mathsf{ValueCommit}^{\mathsf{Sapling}}$;
- $\mathsf{epk}$ will be constrained to $[\mathsf{esk}]\,\mathsf{g_d}$

so $\mathsf{cv}^{\mathsf{new}}$ and $\mathsf{epk}$ 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 § A.3.3.2 ***'ctEdwards [de]compression and validation'*** on p. 178.

**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 | § A.3.1.1 on p. 174 |
| $\mathsf{rcv}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{rcv} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{cv} = \mathsf{ValueCommit}^{\mathsf{Sapling}}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}})$ | **Value commitment integrity** | 947 | § A.3.6 on p. 186 |
| inputize $\mathsf{cv}$ | | ? | |
| $\mathsf{g}\star_{\mathsf{d}} = \mathsf{repr}_{\mathbb{J}}(\mathsf{g}_{\mathsf{d}} : \mathbb{J})$ | **Note commitment integrity** | 392 | § A.3.3.2 on p. 178 |
| $\mathsf{g}_{\mathsf{d}}$ is not small order | **Small order checks** | 16 | § A.3.3.6 on p. 181 |
| $\mathsf{esk}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{esk} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{epk} = [\mathsf{esk}\star]\,\mathsf{g}_{\mathsf{d}}$ | **Ephemeral public key integrity** | 3252 | § A.3.3.8 on p. 182 |
| inputize $\mathsf{epk}$ | | ? | |
| $\mathsf{pk}\star_{\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$ | $\mathsf{pk}\star_{\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$ | 256 | § A.3.1.1 on p. 174 |
| $\mathsf{rcm}\star : \mathbb{B}^{[\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}]}$ | $\mathsf{rcm} : \{0\,..\,2^{\ell^{\mathsf{Sapling}}_{\mathsf{scalar}}}-1\}$ | 252 | § A.3.1.1 on p. 174 |
| $\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g}_{\mathsf{d}}, \mathsf{pk}_{\mathsf{d}}, \mathsf{v}^{\mathsf{old}})$ | **Note commitment integrity** | 1740 | § A.3.5 on p. 186 |
| 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

## B.1 RedDSA batch validation

The reference validation algorithm for RedDSA signatures is defined in § 5.4.7 *'RedDSA, RedJubjub, and RedPallas'* on p. 85.

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 $\mathsf{abst}_{\mathbb{G}}$); $\mathcal{P}_{\mathbb{G}} : \mathbb{G}$; $\ell_{\mathsf{H}} : \mathbb{N}$; $\mathsf{H} : \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{BY}^{[\ell_{\mathsf{H}}/8]}$; and the derived hash function $\mathsf{H}^{\circledast} : \mathbb{BY}^{[\mathbb{N}]} \to \mathbb{F}_{r_{\mathbb{G}}}$ 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 § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

Define $\mathsf{RedDSA.BatchEntry} := \mathsf{RedDSA.Public} \times \mathsf{RedDSA.Message} \times \mathsf{RedDSA.Signature}$.

```
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 $\mathsf{ceiling}\big(\ell_{\mathbb{G}}/8\big)$ bytes of $\sigma_j$, and let $\underline{S_j}$ be the remaining $\mathsf{ceiling}\,(\mathsf{bitlength}(r_{\mathbb{G}})/8)$ bytes.
        Let $R_j = \mathsf{abst}_{\mathbb{G}}\big(\mathsf{LEOS2BSP}_{\ell_{\mathbb{G}}}(\underline{R_j})\big)$, and let $S_j = \mathsf{LEOS2IP}_{8\cdot\mathsf{length}(\underline{S_j})}(\underline{S_j})$.
        Let $\underline{\mathsf{vk}_j} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{vk}_j)\big)$.
        Let $c_j = \mathsf{H}^{\circledast}(\underline{R_j}\,||\,\underline{\mathsf{vk}_j}\,||\,M_j)$.
        Choose random $z_j ⦂ \mathbb{F}^*_{r_{\mathbb{G}}} \xleftarrow{\mathrm{R}} \{1\,..\,2^{128}-1\}$.
    Return 1 if
        · for all $j \in \{0\,..\,N-1\}$, $R_j \neq \bot$ and $S_j < r_{\mathbb{G}}$; and
        · $[h_{\mathbb{G}}]\Big(-\Big[\sum_{j=0}^{N-1}(z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\Big]\mathcal{P}_{\mathbb{G}} + \sum_{j=0}^{N-1}[z_j]\,R_j + \sum_{j=0}^{N-1}[z_j \cdot c_j \pmod{r_{\mathbb{G}}}]\,\mathsf{vk}_j\Big) = \mathcal{O}_{\mathbb{G}}$,
    otherwise 0.
```

The $z_j$ values **MUST** be chosen independently of the *signature batch entries*.

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, section 5].

**Note:** Spend authorization signatures (§ 5.4.7.1 ***'Spend Authorization Signature (Sapling and Orchard)'*** on p. 87) and binding signatures (§ 5.4.7.2 ***'Binding Signature (Sapling and Orchard)'*** on p. 88) use different bases $\mathcal{P}_{\mathbb{G}}$. It is straightforward to adapt the above procedure to handle multiple bases; there will be one $-\Big[\sum_j(z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\Big]\mathcal{P}$ 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

The reference verification algorithm for Groth16 proofs is defined in § 5.4.10.2 'Groth16' on p. 103. The batch verification algorithm in this section applies techniques from [BFIJSV2010, section 4].

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 § 5.4.9.2 'BLS12-381' on p. 93.

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 $\Psi_i = \left[\frac{\beta\cdot u_i(x) + \alpha\cdot v_i(x) + w_i(x)}{\gamma}\right]\mathcal{P}_{\mathbb{S}_1}$ for $i \in \{0\,..\,\ell\}$ are elements of the verification key, as described (with slightly different notation) in [Groth2016, section 3.2].

This can be written as:

$$\hat{e}_{\mathbb{S}}(\pi_A, -\pi_B) \cdot \hat{e}_{\mathbb{S}}(\pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\Big(\sum\nolimits_{i=0}^{\ell} [a_i]\, \Psi_i, \Gamma\Big) \cdot Y = \mathbf{1}_{\mathbb{S}}.$$

Raising to the power of random $z \neq 0$ gives:

$$\hat{e}_{\mathbb{S}}([z]\, \pi_A, -\pi_B) \cdot \hat{e}_{\mathbb{S}}([z]\, \pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\Big(\sum\nolimits_{i=0}^{\ell} [z \cdot a_i]\, \Psi_i, \Gamma\Big) \cdot Y^z = \mathbf{1}_{\mathbb{S}}.$$

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 $z_j \;⦂\; \mathbb{F}^*_{r_{\mathbb{S}}} \xleftarrow{\mathrm{R}} \{1\,..\,2^{128}-1\}$.
    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*.

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 § 4.17.2 ***'Spend Statement (Sapling)'*** on p. 55) and output proofs (of the *statement* in § 4.17.3 ***'Output Statement (Sapling)'*** on p. 56) use different verification keys, with different parameters $\Delta$, $\Gamma$, $Y$, and $\Psi_{0\,..\,\ell}$. It is straightforward to adapt the above procedure to handle multiple verification keys; the accumulator variables $\mathsf{Accum}_{\Delta}$, $\mathsf{Accum}_{\Gamma,i}$, and $\mathsf{Accum}_{Y}$ are duplicated, with one term in the verification equation for each variable, while $\mathsf{Accum}_{AB}$ 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 $\mathsf{Accum}_{\Delta}$; 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.

## B.3 Ed25519 batch validation

The reference validation algorithm for Ed25519 signatures is defined in § 5.4.6 'Ed25519' on p. 83.

[**Canopy** onward] 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. The correctness of this procedure is dependent on the Ed25519 validation changes made for the **Canopy** *network upgrade* in [ZIP-215] (in particular the change to use the cofactor variant of the validation equation). The input is a sequence of $N$ *signature batch entries*, each of which is a (*validating key*, message, signature) triple.

Let $\ell$, $B$, $\mathsf{abstBytes}_{\mathsf{Ed25519}}$, and $\mathsf{reprBytes}_{\mathsf{Ed25519}}$ be as defined in § 5.4.6 'Ed25519' on p. 83.

Let LEOS2IP be as defined in § 5.2 *'Integers, Bit Sequences, and Endianness'* on p. 66.

SHA-512 is defined in § 5.4.1.1 'SHA-256, SHA-256d, SHA256Compress, *and* SHA-512 *Hash Functions*' on p. 68.

Define $\mathsf{Ed25519.BatchEntry} := \mathsf{Ed25519.Public} \times \mathsf{Ed25519.Message} \times \mathsf{Ed25519.Signature}$.

Define $\mathsf{Ed25519.BatchValidate} \;⦂\; (\mathsf{entry}_{0\,..\,N-1} \;⦂\; \mathsf{Ed25519.BatchEntry}^{[N]}) \to \mathbb{B}$ as:

```
For each $j \in \{0\,..\,N-1\}$:
    Let $(A_j, M_j, \sigma_j) = \mathsf{entry}_j$.
    Let $\underline{R_j}$ be the first 32 bytes of $\sigma_j$, and let $\underline{S_j}$ be the remaining 32 bytes.
    Let $R_j = \mathsf{abstBytes}_{\mathsf{Ed25519}}(\underline{R_j})$, and let $S_j = \mathsf{LEOS2IP}_{256}(\underline{S_j})$.
    Let $\underline{A_j} = \mathsf{reprBytes}_{\mathsf{Ed25519}}(A_j)$.
    Let $c_j = \mathsf{LEOS2IP}_{512}\big(\mathsf{SHA\text{-}512}(\underline{R_j} \,||\, \underline{A_j} \,||\, M_j)\big)$.
    Choose random $z_j \;⦂\; \mathbb{F}_\ell^* \overset{\mathrm{R}}{\leftarrow} \{1\,..\,2^{128}-1\}$.
Return 1 if
    · for all $j \in \{0\,..\,N-1\}$, $R_j \neq \bot$ and $S_j < \ell$; and
    · $[8]\Big(-\Big[\sum_{j=0}^{N-1} (z_j \cdot S_j) \pmod{\ell}\Big] B + \sum_{j=0}^{N-1} [z_j]\, R_j + \sum_{j=0}^{N-1} [z_j \cdot c_j \pmod{\ell}]\, A_j\Big) = \mathcal{O}_{\mathsf{Ed25519}}$,
otherwise 0.
```

The $z_j$ values **MUST** be chosen independently of the *signature batch entries*.

The performance benefits of this approach are the same as for § B.1 'RedDSA *batch validation*' on p. 193.

# List of Theorems and Lemmata

# Index