Bitzec Protocol Specification

Version 2018 [Overwinter+Sapling]

November 2018

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

This specification the **bitzec** consensus protocol at launch; **Sapling ENDT**. It is a work in progress. Protocol differences from **Zerocash** and **Bitcoin** are also explained.

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

Contents

1Introduction

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

Changes from the original **Zerocash** are explained in § 8 *'Differences from the Zerocash paper'* on p. 90, and highlighted inmagentathroughout the document. Changes speciZc to the **Overwinter** upgrade (which are also changes from **Zerocash**) are highlighted inblue. Changes speciZc to the **Sapling** upgrade following **Overwinter** (which are also changes from **Zerocash**) are highlighted ingreen. The name **Sprout** is used for the **bitzec** protocol prior to **Sapling** (both before and after **Overwinter**).

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

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

This speciZcation is structured as follows:

- Notation — deZnitions 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 to **Overwinter** and then **Sapling**;
- Consensus Changes from **Bitcoin** — how **bitzec** 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** circuit is deZned as a *quadratic constraint program*.
- Appendix: Batching Optimizations — improvements to the efZciency of verifying multiple signatures and proofs.

1.1Caution

bitzec security depends on consensus. Should a program interacting with the **bitzec** 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 speciZc cause will not matter to the users of your software whose wealth is lost.

Having said that, a speciZcation of *intended* behaviour is essential for security analysis, understanding of the protocol, and maintenance of **bitzec** and related software. If you Znd any mistake in this speciZcation, please Zle an issue at https://github.com/bitzec/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 speciZcation. This overview applies to both **Sprout** and **Sapling**, differences in the cryptographic constructions usednotwithstanding.

Value in **bitzec** 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 **bitzec** this is called a *spending key*.

To each *note* there is cryptographically associated a *note commitment* . Once the *transaction* creating the *note* has been mined, it is associated with a Zxed *note position* in a tree of *note commitments*, and with a *nulliber* [2] unique to that *note*. Computing the *nulliber* requires the associated private *spending key* (or the *nulliber deriving key* for **Sapling** *notes*). It is infeasible to correlate the *note commitment* or *note position* with the corresponding *nulliber* 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 *nulliber* has not.

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

The *nullibers* 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 insigniZcant probability:

For each *shielded input* ,

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

and for each *shielded output* ,

- [**Sapling** onward] there is a revealed *value commitment* to the same value as the output *note*;
- the *note commitment* is computed correctly;
- it is infeasible to cause the *nulliber* of the output *note* to collide with the *nulliber* of any other *note*.

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

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

Outside the *zk-SNARK* , it is checked that the *nullibers* 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**, 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*.

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

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

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

2Notation

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

N means the type of nonnegative integers. N^{+} means the type of positive integers. Z means the type of integers. 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 $S \xrightarrow{R} T$. The domain of a randomized algorithm may be (), indicating that it requires no arguments. Given f ◦ $S \xrightarrow{R} T$ and s ◦ S, sampling a variable x ◦ T from the output of f applied to s is denoted by $x \xleftarrow{R} f(s)$.

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

{x ◦ T | 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 | $x \in T$}.

$S \setminus T$ means the set difference obtained by removing elements in T from S, i.e. {x ◦ S | x g T }.

x ◦ $T \rightarrowtail e_x$ ◦ 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 ◦ $T \rightarrowtail_{\mathsf{C}y} e_x$ ◦ U means x ◦ $T \rightarrowtail e_x$ ◦ $U \cup \{y\}$ restricted to the domain {x ◦ T | e_x Ç y} and range U .

┌· T means the powerset of T .

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

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

length(S) means the length of (number of elements in) S.

truncate$_k$ (S) means the sequence formed from the Zrst k elements of S.

0x followed by a string of monospace hexadecimal digits means the corresponding integer converted from hexadecimal. $[0x00]^A$ means the sequence of A zero bytes.

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

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

$[0]^A$ means the sequence of A zero bits. $[1]^A$ means the sequence of A 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 $[\mathsf{a}^{\mathsf{new}}_{\mathsf{pk},1}, \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},2}, ... \, \mathsf{a}^{\mathsf{new}}_{\mathsf{pk},\mathrm{N}^{\mathsf{new}}}]$. (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 $a \in \mathbb{F}_n$ in the range $\{0 \,..\, n-1\}$ (or the unique representative of $a \in \mathbb{F}^*_n$ in the range $\{1 \,..\, n-1\}$) as $a \bmod n$. Conversely, we denote the element of $\mathbb{F}_n$ corresponding to an integer $k \in \mathbb{Z}$ as $k \pmod{n}$. We also use the latter notation in the context of an equality $k = k' \pmod{n}$ as shorthand for $k \bmod n = k' \bmod n$, and similarly $k \neq k' \pmod{n}$ as shorthand for $k \bmod n \neq k' \bmod n$. (When referring to constants such as 0 and 1 it is usually not necessary to make the distinction between field elements and their representatives, since the meaning is normally clear from context.)

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

$a + b$ means the sum of a and b. This may refer to addition of integers, rationals, finite field elements, or group elements (see §4.1.8 *'Represented Group'* on p. 24) 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 $a \in \mathbb{N}$ and $q \in \mathbb{N}^+$, 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 $a \wedge b$ means the bitwise-and of a and b. These are defined on integers or (equal-length) bit sequences according to context.

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

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

sequence of the appropriate length given by the type of a.

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

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

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

$$a^b := \prod_{i=1}^{b} a, \text{ if } b \geq 0$$

$\square\ Q^{i=1\ -}_{b\ 1}\ {}_{i=1}\ a$, otherwise.

The $[k]\,P$ notation for scalar multiplication in a group is defined in §4.1.8 *'Represented Group'* on p. 24.

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 $x \in \mathbb{N}$, means the smallest integer A such that $2^A > x$.

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

The following integer constants will be instantiated in §5.3 *'Constants'* on p. 49:

> $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, A_{value}, $A_{\mathsf{MerkleSprout}}$, $A_{\mathsf{MerkleSapling}}$, A_{hSig}, $A_{\mathsf{PRFSprout}}$, $A_{\mathsf{PRFexpand}}$, $A_{\mathsf{PRFnfSapling}}$, A_{rcm}, A_{Seed}, $A_{\mathsf{a_{sk}}}$, A_{ϕ}, A_{sk}, A_{d}, A_{ivk}, A_{ovk}, A_{scalar}, $\mathsf{MAX\_MONEY}$, $\mathsf{SlowStartInterval}$, $\mathsf{HalvingInterval}$, $\mathsf{MaxBlockSubsidy}$, $\mathsf{NumFounderAddresses}$, $\mathsf{PoWLimit}$, $\mathsf{PoWAveragingWindow}$, $\mathsf{PoWMedianBlockSpan}$, $\mathsf{PoWDampingFactor}$, and $\mathsf{PoWTargetSpacing}$.

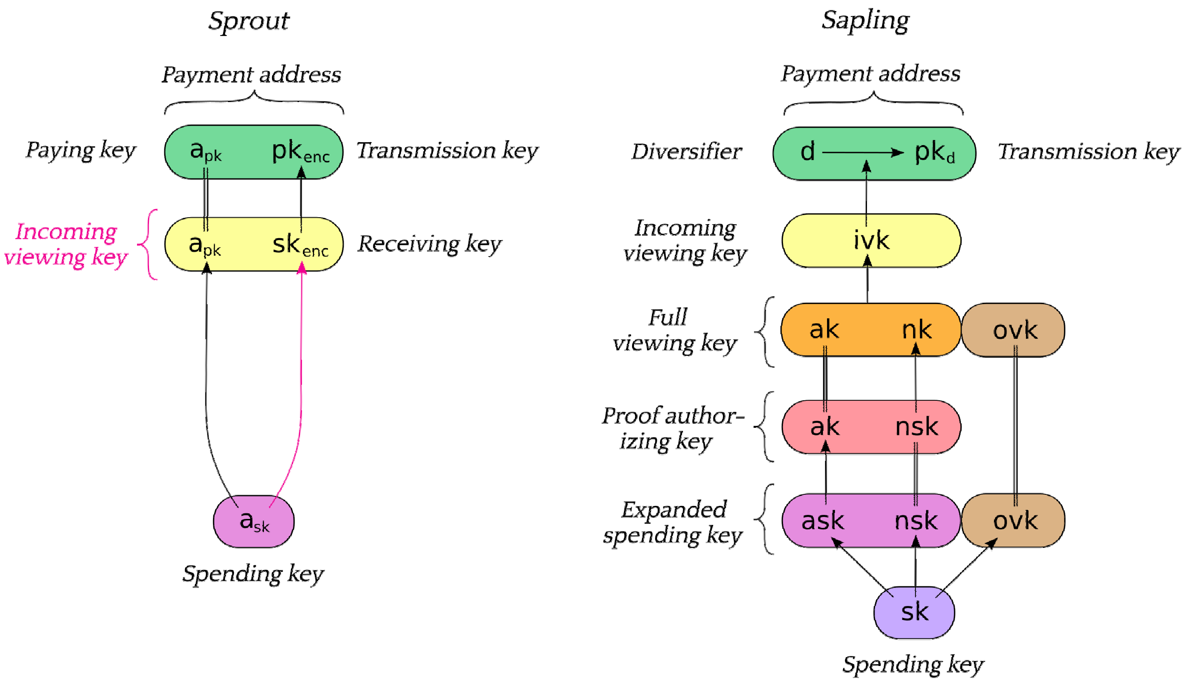
The bit sequence constants $\mathsf{Uncommitted}^{\mathsf{Sprout}} \in \mathbb{B}^{[A_{\mathsf{MerkleSprout}}]}$ and $\mathsf{Uncommitted}^{\mathsf{Sapling}} \in \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$, and rational constants $\mathsf{FoundersFraction}$, $\mathsf{PoWMaxAdjustDown}$, and $\mathsf{PoWMaxAdjustUp}$ will also be defined in that section.

3 Concepts

3.1 Payment Addresses and Keys

Users who wish to receive payments under this scheme first generate a random *spending key* . In **Sprout** this is called $\mathsf{a_{sk}}$ and in **Sapling** it is called sk.

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



[**Sprout**] The *receiving key* $\mathsf{sk_{enc}}$, the *incoming viewing key* $\mathsf{ivk} = (\mathsf{a_{pk}}, \mathsf{sk_{enc}})$, and the *shielded payment address* $\mathsf{addr_{pk}} = (\mathsf{a_{pk}}, \mathsf{pk_{enc}})$ are derived from $\mathsf{a_{sk}}$, as described in §4.2.1 *'**Sprout** Key Components'* on p. 27.

[**Sapling** onward] The *spend authorizing key* ask, *proof authorizing key* $(\mathsf{ak}, \mathsf{nsk})$, *full viewing key* $(\mathsf{ak}, \mathsf{nk}, \mathsf{ovk})$, *incoming viewing key* ivk, and each *diversified payment address* $\mathsf{addr_d} = (\mathsf{d}, \mathsf{pk_d})$ are derived from sk, as described in §4.2.2 *'**Sapling** Key Components'* on p. 27.

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

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

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

Note: It is conventional in cryptography to refer to the key used to encrypt a message in an asymmetric encryption scheme as the "*public key* ". However, the public key used as the *transmission key* component of an address ($\mathsf{pk_{enc}}$ or $\mathsf{pk_d}$) need not be publically distributed; it has the same distribution as the *shielded payment address* itself. As mentioned above, limiting the distribution of the *shielded payment address* is important for some use cases. This also helps to reduce reliance of the overall protocol on the security of the cryptosystem used for *note* encryption (see §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43and §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44), since an adversary would have to know $\mathsf{pk_{enc}}$ or some $\mathsf{pk_d}$ in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

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

Let $\mathsf{MAX\_MONEY}$, $A_{\mathsf{PRFSprout}}$, $A_{\mathsf{PRFnfSapling}}$, and A_{d} be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{NoteCommit^{Sprout}}$ be as defined in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 62.

Let $\mathsf{NoteCommit^{Sapling}}$ be as defined in §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63.

Let $\mathsf{KA^{Sapling}}$ be as defined in §5.4.4.3 *'**Sapling** Key Agreement'* on p. 58.

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

- $\mathsf{a_{pk}} ⦂ \mathbb{B}^{[A_{\mathsf{PRFSprout}}]}$ is the *paying key* of the recipient's *shielded payment address*;
- $\mathsf{v} ⦂ \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi* (1 **ZEC** = 10^8 *zatoshi*);
- $\rho ⦂ \mathbb{B}^{[A_{\mathsf{PRFSprout}}]}$ is used as input to $\mathsf{PRF^{nf}_{a_{sk}}}$ to derive the *nullifier* of the *note*;
- $\mathsf{rcm} ⦂ \mathsf{NoteCommit^{Sprout}.Trapdoor}$ is a random *commitment trapdoor* as defined in §4.1.7 *'Commitment'* on p. 23.

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

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

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

- $\mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the *diversiber* of the recipient's *shielded payment address*;
- $\mathsf{pk_d} ⦂ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$ is the *diversibed transmission key* of the recipient's *shielded payment ad- dress*;
- $\mathsf{v} ⦂ \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi*;
- $\mathsf{rcm} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as deZned in §4.1.7 *'Commitment'* on p. 23.

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

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

Creation of new *notes* is described in §4.6 *'Sending Notes'* on p. 31. When *notes* are sent, only a commitment (see §4.1.7 *'Commitment'* on p. 23) to the above values is disclosed publically, and added to a data structure called the *note commitment tree*. This allows the value and recipient to be kept private, while the commitment is used by the *zero-knowledge proof* when the *note* is spent, to check that it exists on the *block chain*.

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

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

where $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ is instantiated in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 62.

Let $\mathsf{DiversifyHash}$ be as deZned in §5.4.1.6 '$\mathsf{DiversifyHash}$ *Hash Function'* on p. 52.

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

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

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

where $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ is instantiated in §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63.

Notice that the above deZnition of a **Sapling** *note* does not have a ρ Zeld. There is in fact a ρ value associated with each **Sapling** *note*, but this only be computed once its position in the *note commitment tree* is known (see §3.4 *'Transactions and Treestates'* on p. 14 and §3.7 *'Note Commitment Trees'* on p. 16). We refer to the combination of a *note* and its *note position* pos, as a *positioned note*.

For a *positioned note*, we can compute the value ρ as described in §4.14 *'Note Commitments and Nullifiers'* on p. 39.

A *nulliber* (denoted nf) is derived from the ρ value of a *note* and the recipient's *spending key* $\mathsf{a_{sk}}$ or *nulliber deriving key* nk. This computation uses a *Pseudo Random Function* (see §4.1.2 *'Pseudo Random Functions'* on p. 18), as described in §4.14 *'Note Commitments and Nullifiers'* on p. 39.

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

3.2.1Note Plaintexts and Memo Fields

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

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

Each **Sprout** *note plaintext* (denoted **np**) consists of

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

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

Each **Sapling** *note plaintext* (denoted **np**) consists of

$(\mathsf{d} ⦂ \mathbb{B}^{[A\mathsf{d}]}, \mathsf{v} ⦂ \{0 \,..\, 2^{A\mathsf{value}}-1\}, \underline{\mathsf{rcm}} ⦂ \mathbb{B}^{\mathbb{Y}[32]}, \mathsf{memo} ⦂ \mathbb{B}^{\mathbb{Y}[512]})$.

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

Other fields are as defined in §3.2 *'Notes'* on p. 12.

Encodings are given in §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 71. The result of encryption forms part of a *transmitted note(s) ciphertext* . For further details, see §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43and §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44.

3.3 The Block Chain

At a given point in time, each *full validator* is aware of a set of candidate *blocks*. These form a tree rooted at the *genesis block* , where each node in the tree refers to its parent via the hashPrevBlock *block header* field (see §7.5 *'Block Header'* on p. 83).

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 §7.6.5 *'Definition of Work'* on p. 87, 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 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 value pool* **MUST** be nonnegative.

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

- a *note commitment tree* (§3.7 *'Note Commitment Trees'* on p. 16);
- a *nullifier set* (§3.8 *'Nullifier Sets'* on p. 17).

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

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

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

- The input *treestate* of the Zrst *block* is the empty *treestate*.
- The input *treestate* of the Zrst *transaction* of a *block* is the Znal *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 Znal *treestate* of a *block* is the output *treestate* of its last *transaction*.

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

3.5 JoinSplit Transfers and Descriptions

A *JoinSplit description* is data included in a *transaction* that describes a *JoinSplit transfer* , i.e. a *shielded* value transfer. In **Sprout**, this kind of value transfer was the primary **bitzec**-speciZc 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 (§4.15.1 *'JoinSplit Statement (**Sprout**)'* on p. 40), for which it provides a *zk-SNARK proof* .

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

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

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

For each of the $\mathrm{N}^{\mathsf{old}}$ *shielded inputs*, a *nulliber* is revealed. This allows detection of double-spends as described in §3.8 *'Nullifier Sets'* on p. 17.

For each *JoinSplit description* in a *transaction*, an interstitial output *treestate* is constructed which adds the *note commitments* and *nullibers* speciZed 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 Zrst *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 Znal **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* (§4.15.2 *'Spend Statement (**Sapling**)'* on p. 41) 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* (§4.15.3 *'Output Statement (**Sapling**)'* on p. 42) 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 (§5.4.7.3 *'Homomorphic Pedersen commitments'* on p. 63). The result of adding two *Pedersen value commitments*, committing to values v_1 and v_2, is a new *Pedersen value commitment* that commits to $\mathsf{v}_1 + \mathsf{v}_2$. Subtraction works similarly.

Therefore, balance can be enforced by adding all of the *value commitments* for *shielded inputs*, subtracting all of the *value commitments* for *shielded outputs*, and proving by use of a *binding signature* (as described in §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36) 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 forprecomputation.

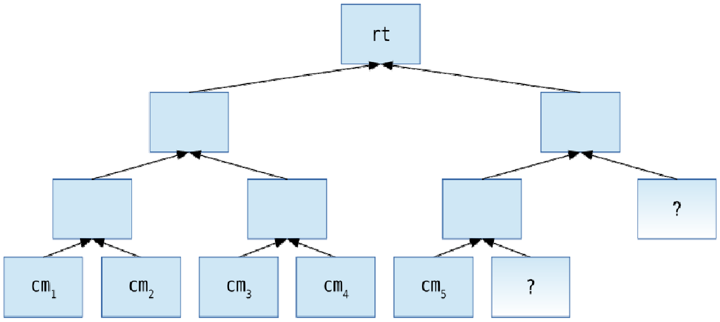
A *Spend description* includes an *anchor* , which refers to the output **Sapling** *treestate* of a previous *block* . It also reveals a *nulliber* , which allows detection of double-spends as described in §3.8 *'Nullifier Sets'* on p. 17.

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 *transaction* **MUST** balance as speciZed in §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36.
- The *anchor* of each *Spend description* **MUST** refer to some earlier *block*'s Znal **Sapling** *treestate*.

3.7Note Commitment Trees



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

A *root* of a *note commitment tree* is associated with each *treestate* (§3.4 *'Transactions and Treestates'* on p. 14).

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

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

3.8 Nulliber Sets

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

Consensus rule: A *nulliber* **MUST NOT** repeat either within a *transaction*, or across *transactions* in a *valid block chain*. **Sprout** and **Sapling** *nullibers* are considered disjoint, even if they have the same bitpattern.

3.9 Block Subsidy and Founders' Reward

Like **Bitcoin**, **bitzec** creates currency when *blocks* are mined. The value created on mining a *block* is called the *block subsidy* . It is composed of a *miner subsidy* and a *Founders' Reward* . As in **Bitcoin**, the miner of a *block* also receives *transaction fees*.

The calculations of the *block subsidy* , *miner subsidy* , and *Founders' Reward* depend on the *block height* , as deZned in §3.3 *'The Block Chain'* on p. 14.

These calculations are described in §7.7 *'Calculation of Block Subsidy and Founders' Reward'* on p. 87.

3.10 Coinbase Transactions

The Zrst (and only the Zrst) *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* . The *coinbase transaction* **MUST** also pay the *Founders' Reward* as described in §7.8 *'Payment of Founders' Reward'* on p. 88.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

Let $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, $\ell_{\mathsf{MerkleSprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, $\ell_{\mathsf{MerkleSapling}}$, ℓ_{ivk}, ℓ_{d}, ℓ_{Seed}, $\ell_{\mathsf{PRFSprout}}$, ℓ_{hSig}, and $\mathrm{N}^{\mathsf{old}}$ be as deZned in §5.3 *'Constants'* on p. 49.

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, $r_{\mathbb{J}}$, and $\ell_{\mathbb{J}}$ be as deZned in §5.4.8.3 *'Jubjub'* on p. 67.

The functions $\mathsf{MerkleCRH}^{\mathsf{Sprout}} : \{0\,..\,\mathsf{MerkleDepth}^{\mathsf{Sprout}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSprout}}]}$ and (for **Sapling**), $\mathsf{MerkleCRH}^{\mathsf{Sapling}} : \{0\,..\,\mathsf{MerkleDepth}^{\mathsf{Sapling}} - 1\} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \times \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ are *hash functions* used in §4.8 *'Merkle path validity'* on p. 34. $\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ is collision-resistant on all its arguments, and $\mathsf{MerkleCRH}^{\mathsf{Sprout}}$ is collision-resistant except on its Zrst argument. Both of these functions are instantiated in §5.4.1.3 *'Merkle Tree Hash Function'* on p. 51.

$\mathsf{hSigCRH} : \mathbb{B}^{[\ell_{\mathsf{Seed}}]} \times \mathbb{B}^{[\ell_{\mathsf{PRFSprout}}][\mathrm{N}^{\mathsf{old}}]} \times \mathsf{JoinSplitSig.Public} \to \mathbb{B}^{[\ell_{\mathsf{hSig}}]}$ is a collision-resistant *hash function* used in §4.3 *'JoinSplit Descriptions'* on p. 29. It is instantiated in §5.4.1.4 *'$\mathsf{h}_{\mathsf{Sig}}$ Hash Function'* on p. 52.

$\mathsf{EquihashGen} : (n : \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{BY}^{[\mathbb{N}]} \times \mathbb{N}^+ \to \mathbb{B}^{[n]}$ is another *hash function*, used in §7.6.1 *'Equihash'* on p. 85 to generate input to the Equihash solver. The Zrst two arguments, representing the Equihash parameters n and k, are written subscripted. It is instantiated in §5.4.1.9 *'Equihash Generator'* on p. 56.

$\mathsf{CRH}^{\mathsf{ivk}} : \mathbb{B}^{[\ell_{\mathbb{J}}]} \times \mathbb{B}^{[\ell_{\mathbb{J}}]} \to \{0\,..\,2^{\ell_{\mathsf{ivk}}}-1\}$ is a collision-resistant *hash function* used in §4.2.2 *'**Sapling** Key Components'* on p. 27 to derive an *incoming viewing key* for a **Sapling** *shielded payment address*. It is also used in the *Spend statement* (§4.15.2 *'Spend Statement (**Sapling**)'* on p. 41) to conZrm 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. 52.

$\mathsf{MixingPedersenHash} : \mathbb{J} \times \{0\,..\,r_{\mathbb{J}} - 1\} \to \mathbb{J}$ is a *hash function* used in §4.14 *'Note Commitments and Nullifiers'* on p. 39 to derive the unique ρ value for a **Sapling** *note*. It is also used in the *Spend statement* to confirm use of the correct ρ value as an input to *nullifier* derivation. It is instantiated in §5.4.1.8 *'Mixing Pedersen Hash Function'* on p. 55.

$\mathsf{DiversifyHash} : \mathbb{B}^{[\ell_{\mathsf{d}}]} \to \mathbb{J}^{(r)*}$ is a *hash function* instantiated in §5.4.1.6 'DiversifyHash *Hash Function'* on p. 52, and satisfying the Unlinkability security property described in that section. It is used to derive a *diversified base* from a *diversifier* in §4.2.2 *'**Sapling** Key Components'* on p.27.

4.1.2 Pseudo Random Functions

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

Let $\ell_{a_{\mathsf{sk}}}$, ℓ_{ϕ}, ℓ_{hSig}, $\ell_{\mathsf{PRFSprout}}$, ℓ_{sk}, ℓ_{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. 49.

Let $\ell_{\mathbb{J}}$ and $\mathbb{J}^{(r)}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

Let Sym be as defined in §5.4.3 *'Authenticated One-Time Symmetric Encryption'* on p. 57.

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

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

These are used in §4.15.1 *'JoinSplit Statement (**Sprout**)'* on p. 40; $\mathsf{PRF}^{\mathsf{addr}}$ is also used to derive a *shielded payment address* from a *spending key* in §4.2.1 *'**Sprout** Key Components'* on p. 27.

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

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

$\mathsf{PRF}^{\mathsf{expand}}$ is used in §4.2.2 *'**Sapling** Key Components'* on p. 27.

$\mathsf{PRF}^{\mathsf{ock}}$ is used in §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used in §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41.

All of these *Pseudo Random Functions* are instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 56.

Security requirements:

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

Non-normative note: $\mathsf{PRF}^{\mathsf{nf}}$ was called $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash** [BCGGMTV2014].

4.1.3 Authenticated One-Time Symmetric Encryption

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

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

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

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

4.1.4 Key Agreement

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

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

Optional: Let $\mathsf{KA.FormatPrivate} : \mathbb{B}^{[\ell_{\mathsf{PRFSprout}}]} \to \mathsf{KA.Private}$ be a function to convert a bit string of length $\ell_{\mathsf{PRFSprout}}$ 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 $\mathsf{KA.DerivePublic}$ may be a strict subset of $\mathsf{KA.Public}$.

Security requirements:

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

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

4.1.5 Key Derivation

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

The inputs to the *Key Derivation Function* differ between the **Sprout** and **Sapling** 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 $\mathsf{sharedSecret}$, 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 $\mathsf{Sym.Encrypt}$.

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

$\mathsf{KDF}^{\mathsf{Sapling}}$ takes as input the shared Diffie-Hellman secret $\mathsf{sharedSecret}$ and the ephemeral public key 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 $\mathsf{Sym.Encrypt}$.

$$\mathsf{KDF}^{\mathsf{Sapling}} : \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{SharedSecret} \times \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public} \to \mathsf{Sym}.\mathbf{K}$$

Security requirements:

- The asymmetric encryption scheme in §4.16 *'In-band secret distribution (**Sprout**)'* on p.43, constructed from $\mathsf{KA}^{\mathsf{Sprout}}$, $\mathsf{KDF}^{\mathsf{Sprout}}$ and Sym, is required to be IND-CCA2-secure and *key-private*.
- The asymmetric encryption scheme in §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44, constructed from $\mathsf{KA}^{\mathsf{Sapling}}$, $\mathsf{KDF}^{\mathsf{Sapling}}$ and Sym, is required to be IND-CCA2-secure and *key-private*.

Key privacy is defined in [BBDP2001].

4.1.6 Signature

A signature scheme Sig defines:

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

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

bitzec uses four signature schemes:

- one used for signatures that can be verified by script operations such as OP_CHECKSIG and OP_CHECKMULTISIG as in **Bitcoin**;
- one called $\mathsf{JoinSplitSig}$ (instantiated in §5.4.5 *'JoinSplit Signature'* on p. 59), which is used to sign *transactions* that contain at least one *JoinSplit description*;
- [**Sapling** onward] one called $\mathsf{SpendAuthSig}$ (instantiated in §5.4.6.1 *'Spend Authorization Signature'* on p. 62) which is used to sign authorizations of *Spend transfers*;
- [**Sapling** onward] one called $\mathsf{BindingSig}$ (instantiated in §5.4.6.2 *'Binding Signature'* on p. 62), which is used to enforce balance of *Spend transfers* and *Output transfers*, and to prevent their replay across *transactions*.

The following security property is needed for $\mathsf{JoinSplitSig}$ and $\mathsf{BindingSig}$.Security requirements for $\mathsf{SpendAuthSig}$ are defined in the next section, §4.1.6.1 *'Signature with Re-Randomizable Keys'* on p. 21. An additional requirement for $\mathsf{BindingSig}$ is defined in §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22.

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

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

Non-normative notes:

- We need separate signing key generation and verifying key derivation algorithms, rather than the more conventional combined key pair generation algorithm $\mathsf{Sig.Gen} : () \xrightarrow{R} \mathsf{Sig.Private} \times \mathsf{Sig.Public}$, to support the key derivation in §4.2.2 *'**Sapling** Key Components'* on p. 27. This also simplifies some aspects of the definitions of *signature schemes* with additional features in §4.1.6.1 *'Signature with Re-Randomizable Keys'* on p. 21 and §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22.
- 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 §4.10 *'Non-malleability (**Sprout**)'* on p. 35), a one-time signature scheme would suffice for $\mathsf{JoinSplitSig}$. This is also the reason why only security against *non-adaptive* chosen message attack is needed. In fact the instantiation of $\mathsf{JoinSplitSig}$ 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 $\mathsf{BindingSig}$, 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*.
- 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 *binding signatures* are intended to be nonmalleable in the sense of [BIP-62].

4.1.6.1 Signature with Re-Randomizable Keys

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

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

such that:

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

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

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

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

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

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

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

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

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

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 $\mathsf{O}_{\mathsf{Sig.Random}}$ simpliZes the deZnition of SURK-CMA by removing the need for two oracles (because the oracle for original keys, called O_1 in [FKMSSS2016], is a special case of the oracle for randomized keys).
- Since $\mathsf{Sig.RandomizePrivate}(\alpha, \mathsf{sk}) : \alpha \xleftarrow{R} \mathsf{Sig.Random}$ has an identical distribution to $\mathsf{Sig.GenPrivate}()$, and since $\mathsf{Sig.DerivePublic}$ is a deterministic function, the combination of a re-randomized public 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* α implies knowledge of sk.

4.1.6.2 Signature with Private Key to Public Key Homomorphism

A *signature scheme with private key to public key homomorphism* Sig is a *signature scheme* that additionally deZnes:

- an abelian group on private keys, with operation $\boxplus ⦂ \mathsf{Sig.Private} \times \mathsf{Sig.Private} \to \mathsf{Sig.Private}$ and identity $\mathsf{O}_{\boxplus}$;
- an abelian group on public keys, with operation $\oplus ⦂ \mathsf{Sig.Public} \times \mathsf{Sig.Public} \to \mathsf{Sig.Public}$ and identity $\mathsf{O}_{\oplus}$.

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

In other words, $\mathsf{Sig.DerivePublic}$ is an injective homomorphism from the private key group to the public key group.

For $\mathrm{N} ⦂ \mathbb{N}^+$,

- $\mathop{\boxplus}\limits_{i=1}^{\mathrm{N}} \mathsf{sk}_i$ means $\mathsf{sk}_1 \boxplus \mathsf{sk}_2 \boxplus \cdots \boxplus \mathsf{sk}_{\mathrm{N}}$;
- $\mathop{\oplus}\limits_{i=1}^{\mathrm{N}} \mathsf{vk}_i$ means $\mathsf{vk}_1 \oplus \mathsf{vk}_2 \oplus \cdots \oplus \mathsf{vk}_{\mathrm{N}}$.

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

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

$\ominus \mathsf{vk}$ means the public key such that $(\ominus \mathsf{vk}) \oplus \mathsf{vk} = \mathsf{O}_{\oplus}$, and $\mathsf{vk}_1 \ominus \mathsf{vk}_2$ means $\mathsf{vk}_1 \oplus (\ominus \mathsf{vk}_2)$.

With a change of notation from μ to $\mathsf{Sig.DerivePublic}$, $+ \boxplus$o , and to$\oplus$, this is similar to the deZnition of a "*Signature with Secret Key to Public Key Homomorphism*" in [DS2016, DeZnition 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 $\mathsf{sk}_2 \xleftarrow{R} \mathsf{Sig.GenPrivate}()$ chosen independently of 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 $n ⦂ \mathbb{N}^+$, $\boxplus_{i=1}^{n} \mathsf{sk}_i$ is computationally indistinguishable from $\mathsf{Sig.GenPrivate}()$ when at least one of $\mathsf{sk}_{1..n}$ is unknown.)

4.1.7 Commitment

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

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

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

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

Security requirements:

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

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 $\mathsf{ValueCommit}$ because trapdoors are equivalent modulo $r_{\mathbb{J}}$, and the range of a trapdoor for those algorithms is $\{0 \,..\, 2^{\ell_{\mathsf{scalar}}} - 1\}$ where $2^{\ell_{\mathsf{scalar}}} > r_{\mathbb{J}}$.)

Let ℓ_{rcm}, $\ell_{\mathsf{MerkleSprout}}$, $\ell_{\mathsf{PRFSprout}}$, and ℓ_{value} be as defined in §5.3 *'Constants'* on p. 49.

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

Sprout uses a *note commitment scheme*

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

instantiated in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 62.

Let ℓ_{scalar} be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathbb{J}^{(r)}$ and $r_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

Define:

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

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

Sapling uses two additional commitment schemes:

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

$\mathsf{NoteCommit^{Sapling}}$ is instantiated in §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63, and $\mathsf{ValueCommit}$ is instantiated in §5.4.7.3 *'Homomorphic Pedersen commitments'* on p. 63.

Non-normative note: $\mathsf{NoteCommit^{Sapling}}$ and $\mathsf{ValueCommit}$ 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 $\mathsf{ValueCommit}$ outputs appear in *Spend descriptions* and *Output descriptions*, or when the cmu Zeld derived from a $\mathsf{NoteCommit^{Sapling}}$ appears in an *Output description*.

4.1.8 Represented Group

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

- a subgroup order parameter $r_{\mathbb{G}} : \mathbb{N}^+$, which must be prime;
- a cofactor parameter $h_{\mathbb{G}} : \mathbb{N}^+$;
- a group $\mathbb{G}$ of order $h_{\mathbb{G}} \cdot r_{\mathbb{G}}$, written additively with operation $+ : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, and additive identity $\mathcal{O}_{\mathbb{G}}$;
- a bit-length parameter $\ell_{\mathbb{G}} : \mathbb{N}$;
- a representation function $\mathsf{repr}_{\mathbb{G}} : \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]}$ and an abstraction function $\mathsf{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\}$, such that $\mathsf{abst}_{\mathbb{G}}$ is the left inverse of $\mathsf{repr}_{\mathbb{G}}$, i.e. for all $P \in \mathbb{G}$, $\mathsf{abst}_{\mathbb{G}}(\mathsf{repr}_{\mathbb{G}}(P)) = P$, and for all S not in the image of $\mathsf{repr}_{\mathbb{G}}$, $\mathsf{abst}_{\mathbb{G}}(S) = \bot$.

DeZne $\mathbb{G}^{(r)}$ 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 $\mathbb{G}^{(r)*}$.

DeZne $\mathbb{G}^{(r)}_{\star} := \{\mathsf{repr}_{\mathbb{G}}(P) : \mathbb{B}^{[\ell_{\mathbb{G}}]} \mid P \in \mathbb{G}^{(r)}\}$.

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

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

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

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

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

4.1.9 Hash Extractor

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

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

4.1.10 Group Hash

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

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

In §5.4.8.5 *'Group Hash into Jubjub'* on p. 69, we instantiate a family of group hashes into the *Jubjub curve* defined by §5.4.8.3 *'Jubjub'* on p. 67.

Security requirement: For a randomly selected $\mathsf{URS} : \mathsf{GroupHash.URSType}$, 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.

Non-normative notes:

- $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ is used to obtain generators of the *Jubjub curve* for various purposes: the bases $\mathcal{G}$ and $\mathcal{H}$ used in **Sapling** key generation, the *Pedersen hash* defined in §5.4.1.7 *'Pedersen Hash Function'* on p. 53, and the commitment schemes defined in §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63 and in §5.4.7.3 *'Homomorphic Pedersen commitments'* on p. 63.

  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 $m_{1..n} : \mathsf{GroupHash.Input}^{[n]}$ and a sequence of nonzero $x_{1..n} : \mathbb{F}^{*[n]}_{r_{\mathbb{G}}}$ such that $\sum_{i=1}^{n} \left([x_i]\, \mathsf{GroupHash}^{\mathbb{G}^{(r)}}_{\mathsf{URS}}(m_i) \right) = \mathcal{O}_{\mathbb{G}}$.

- 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 $x_2 = -1$. It is in fact stronger than collision resistance.
- $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ is also used to instantiate $\mathsf{DiversifyHash}$ in §5.4.1.6 '$\mathsf{DiversifyHash}$ *Hash Function'* on p. 52. 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 $\mathbb{J}^{(r)}$.
- URS is a *Uniform Random String*; we choose it verifiably at random (see §5.9 *'Randomness Beacon'* on p. 76), *after* fixing the concrete group hash algorithm to be used. This mitigates the possibility that the group hash algorithm could have been backdoored.

4.1.11 Represented Pairing

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

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

4.1.12 Zero-Knowledge Proving System

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

A *preprocessing zk-SNARK* instance ZK defines:

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

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

Security requirements:

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

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

These definitions are derived from those in [BCTV2014, Appendix C], adapted to state concrete security for a fixed circuit, rather than asymptotic security for arbitrary circuits. (ZK.Prove corresponds to P , ZK.Verify corresponds to V , and ZK.SatisfyingInputs corresponds to $\mathcal{R}_C$ in the notation of that appendix.)

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

Non-normative note: 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.

bitzec uses two *proving systems*:

- PHGR13 (§5.4.9.1 *'PHGR13'* on p. 69) is used with the BN-254 pairing (§5.4.8.1 *'BN-254'* on p. 64), to prove and verify the **Sprout** *JoinSplit statement* (§4.15.1 *'JoinSplit Statement (**Sprout**)'* on p. 40) before **Sapling** activation.
- Groth16 (§5.4.9.2 *'Groth16'* on p. 70) is used with the BLS12-381 pairing (§5.4.8.2 *'BLS12-381'* on p. 66), to prove and verify the **Sapling** *Spend statement* (§4.15.2 *'Spend Statement (**Sapling**)'* on p. 41) and *Output statement* (§4.15.3 *'Output Statement (**Sapling**)'* on p. 42). It is also used to prove and verify the *JoinSplit statement* after **Sapling** activation.

These specializations are: ZKJoinSplit for the **Sprout** *JoinSplit statement* (with PHGR13 and BN-254, or Groth16 and BLS12-381); ZKSpend for the **Sapling** *Spend statement* ; and ZKOutput for the **Sapling** *Output statement* .

We omit the key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be either the PHGR13 *proving key* and *verifying key* deZned in §5.7 *'**Sprout** zk-SNARK Parameters'* on p. 76, or the sprout-groth16.params Groth16 *proving key* and *verifying key* deZned in §5.8 *'**Sapling** zk-SNARK Parameters'* on p. 76, according to whether the proof appears in a *block* before or after **Sapling** activation.

We also omit subscripts on ZKSpend.Prove, ZKSpend.Verify, ZKOutput.Prove, and ZKOutput.Verify, taking them to be the relevant Groth16 *proving keys* and *verifying keys* deZned in §5.8 *'**Sapling** zk-SNARK Parameters'* on p. 76.

4.2 Key Components

4.2.1 Sprout Key Components

Let $\ell_{\mathsf{a_{sk}}}$ be as deZned in §5.3 *'Constants'* on p. 49.

Let $\mathsf{PRF}^{\mathsf{addr}}$ be a *Pseudo Random Function*, instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 56.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be a *key agreement scheme*, instantiated in §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58.

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

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

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

4.2.2 Sapling Key Components

Let $\ell_{\mathsf{PRFexpand}}$, ℓ_{sk}, ℓ_{ovk}, and ℓ_{d} be as deZned in §5.3 *'Constants'* on p. 49.

Let $\mathsf{PRF}^{\mathsf{expand}}$ and $\mathsf{PRF}^{\mathsf{ock}}$ be *Pseudo Random Functions* instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 56.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be a *key agreement scheme*, instantiated in §5.4.4.3 *'**Sapling** Key Agreement'* on p. 58.

Let $\mathsf{CRH}^{\mathsf{ivk}}$ be a *hash function*, instantiated in §5.4.1.5 *'$\mathsf{CRH}^{\mathsf{ivk}}$ Hash Function'* on p. 52.

Let DiversifyHash be a *hash function*, instantiated in §5.4.1.6 *'DiversifyHash Hash Function'* on p. 52.

Let SpendAuthSig, instantiated in §5.4.6.1 *'Spend Authorization Signature'* on p. 62, be a *signature scheme with re-randomizable keys*.
Let $\mathsf{repr}_{\mathbb{J}}$, $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $\mathbb{J}^{(r)}_{>}$ be as deZned in §5.4.8.3 *'Jubjub'* on p. 67, and let $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ be as deZned in §5.4.8.5 *'Group Hash into Jubjub'* on p. 69.

Let $\mathsf{LEBS2OSP} \circ (\ell \circ \mathbb{N}) \times \mathbb{B}^{[\ell]} \rightarrow \mathbb{B}^{\mathbb{Y}[\mathrm{ceiling}(\ell/8)]}$ and $\mathsf{LEOS2IP} \circ (\ell \circ \mathbb{N} \mid \circ \ell \bmod 8 = 0) \times \mathbb{B}^{\mathbb{Y}[\ell/8]} \rightarrow \{0\,..\,2^{\ell}-1\}$ be as deZned in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

DeZne $\mathcal{H} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\textbf{“bitzec\_H\_”}, \text{“”})$.

DeZne $\mathsf{ToScalar}(x \circ \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{PRFexpand}}/8]}}) := \mathsf{LEOS2IP}_{\ell_{\mathsf{PRFexpand}}}(x) \pmod{r_{\mathbb{J}}}$.

A new **Sapling** *spending key* sk is generated by choosing a bit sequence uniformly at random from $\mathbb{B}^{[\ell_{\mathsf{sk}}]}$. From this *spending key* , the *spend authorizing key* $\mathsf{ask} \circ \mathbb{F}^{*}_{r_{\mathbb{J}}}$, the *proof authorizing key* $\mathsf{nsk} \circ \mathbb{F}_{r_{\mathbb{J}}}$, and the *outgoing viewing key* $\mathsf{ovk} \circ \mathbb{B}^{\mathbb{Y}^{[\ell_{\mathsf{ovk}}/8]}}$ are derived as follows:

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

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

$$\mathsf{ovk} := \mathsf{truncate}_{(\ell_{\mathsf{ovk}}/8)}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([2]))$$

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

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

$$\mathsf{ak} := \mathsf{SpendAuthSig.DerivePublic}(\mathsf{ask})$$

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

$$\mathsf{ivk} := \mathsf{CRH}^{\mathsf{ivk}}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak}), \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big).$$

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

As explained in §3.1 *‘Payment Addresses and Keys’* on p. 11, **Sapling** allows the efZcient creation of multiple *diversibed 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 *diversibed payment address* given an *incoming viewing key* ivk, repeatedly pick a *diversiber* d uniformly at random from $\mathbb{B}^{[\ell_{\mathsf{d}}]}$ until $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$ is not $\bot$. Then calculate:

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

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

For each *spending key* , there is also a *default diversibed payment address* with a “random-looking” *diversiber* . This allows an implementation that does not expose diversiZed 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 *diversiber* as above.

Let $\mathsf{first} \circ (\mathbb{B}^{\mathbb{Y}} \to T \cup \{\bot\}) \to T \cup \{\bot\}$ be as deZned in §5.4.8.5 *‘Group Hash into Jubjub’* on p. 69. DeZne:

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

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

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 sk.

Notes:

- The protocol does not prevent using the *diversiber* d to produce “*vanity* ” addresses that start with a meaningful string when encoded in Bech32 (see §5.6.4 *‘**Sapling** Shielded Payment Addresses’* on p. 73). Users and writers of software that generates addresses should be aware that this provides weaker privacy properties than a randomly chosen *diversiber* , 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 *diversiber* that can be recovered by the recipient of a payment to determine which *diversibed payment address* was used. It is **RECOMMENDED** that such *diversibers* 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 $\mathsf{B}^{\mathsf{Y}[A_{\mathsf{PRFexpand}}/8]}$, where $2^{A_{\mathsf{PRFexpand}}}$ is large compared to $r_{\mathbb{J}}$.

  DeZne $f \circ \mathsf{B}^{[A_{\mathsf{sk}}]} \times \mathsf{B}^{\mathsf{Y}[\mathbb{N}]} \rightarrow \mathbb{F}_{r_{\mathbb{J}}}$ by $f_{\mathsf{sk}}(t) := \mathsf{ToScalar}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(t))$.

  Then f is also a PRF, since $\mathsf{LEOS2IP}_{A_{\mathsf{PRFexpand}}} \circ \mathsf{B}^{\mathsf{Y}[A_{\mathsf{PRFexpand}}/8]} \rightarrow \{0\,..\,2^{A_{\mathsf{PRFexpand}}}-1\}$ is injective, and the bias introduced by the reduction modulo $r_{\mathbb{J}}$ is small because §5.3 *'Constants'* on p. 49 deZnes $A_{\mathsf{PRFexpand}}$ as 512, while $r_{\mathbb{J}}$ has length 252 bits.

  It follows that the distribution of ask, i.e. $\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([0]) : \mathsf{sk} \ ^{\mathrm{R}}\ \mathsf{B}^{[A_{\mathsf{sk}}]}$, is computationally indistinguishable from that of $\mathsf{SpendAuthSig.GenPrivate}()$ (deZned in §5.4.6.1 *'Spend Authorization Signature'* on p. 62).

- Similarly, the distribution of nsk, i.e. $\mathsf{ToScalar}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([1])) : \mathsf{sk} \xleftarrow{\mathrm{R}} \mathsf{B}^{[A_{\mathsf{sk}}]}$, is computationally indistinguishable from the uniform distribution on $\mathbb{F}_{r_{\mathbb{J}}}$. Since $\mathsf{nsk} \circ \mathbb{F}_{r_{\mathbb{J}}} \rightarrowtail \mathsf{repr}_{\mathbb{J}}([\mathsf{nsk}]\,\mathcal{H}) \circ \mathbb{J}^{(r)}$ is bijective, the distribution of $\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$ will be computationally indistinguishable from the uniform distribution on $\mathbb{J}^{(r)}$ which is the keyspace of $\mathsf{PRF}^{\mathsf{nfSapling}}$.

- The bitzecd wallet generates *diversibers* according to [ZIP-32] rather than using the default *diversiber* spec- iZed above.

4.3 JoinSplit Descriptions

A *JoinSplit transfer* , as speciZed in §3.5 *'JoinSplit Transfers and Descriptions'* on p. 15, is encoded in *transactions* as a *JoinSplit description*.

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

Let $A_{\mathsf{MerkleSprout}}$, $A_{\mathsf{PRFSprout}}$, A_{Seed}, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$, and $\mathsf{MAX\_MONEY}$ be as deZned in §5.3 *'Constants'* on p. 49.

Let $\mathsf{hSigCRH}$ be as deZned in §4.1.1 *'Hash Functions'* on p. 17.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as deZned in §4.1.7 *'Commitment'* on p. 23.

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be as deZned in §4.1.4 *'Key Agreement'* on p. 19.

Let Sym be as deZned in §4.1.3 *'Authenticated One-Time Symmetric Encryption'* on p. 19.

Let $\mathsf{ZKJoinSplit}$ be as deZned in §4.1.12 *'Zero-Knowledge Proving System'* on p. 26.

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

- $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}} \circ \{0\,..\,\mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* removes from the *transparent value pool* ;
- $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}} \circ \{0\,..\,\mathsf{MAX\_MONEY}\}$ is the value that the *JoinSplit transfer* inserts into the *transparent value pool* ;
- $\mathsf{rt} \circ \mathsf{B}^{[A_{\mathsf{MerkleSprout}}]}$ is an *anchor*, as deZned in §3.3 *'The Block Chain'* on p. 14, for the output *treestate* of either a previous *block*, or a previous *JoinSplit transfer* in this *transaction*.
- $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}} \circ \mathsf{B}^{[A_{\mathsf{PRFSprout}}][\mathrm{N}^{\mathsf{old}}]}$ is the sequence of *nullibers* for the input *notes*;
- $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}} \circ \mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Output}^{[\mathrm{N}^{\mathsf{new}}]}$ is the sequence of *note commitments* for the output *notes*;
- $\mathsf{epk} \circ \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Public}$ is a key agreement public key, used to derive the key for encryption of the *transmitted notes ciphertext* (§4.16 *'In-band secret distribution (**Sprout**)'* on p.43);
- $\mathsf{randomSeed} \circ \mathsf{B}^{[A_{\mathsf{Seed}}]}$ is a seed that must be chosen independently at random for each *JoinSplit description*;
- $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}} \circ \mathsf{B}^{[A_{\mathsf{PRFSprout}}][\mathrm{N}^{\mathsf{old}}]}$ is a sequence of tags that bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the input *notes*;
- $\pi_{\mathsf{ZKJoinSplit}} \circ \mathsf{ZKJoinSplit.Proof}$ is a *zk proof* with *primary input* $(\mathsf{rt}, \mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}, \mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}, \mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}, \mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}, \mathsf{h}_{\mathsf{Sig}}, \mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}})$ for the *JoinSplit statement* deZned in §4.15.1 *'JoinSplit Statement (**Sprout**)'* on p. 40(this is a $\mathsf{PHGR13}$ proof before **Sapling** activation, and a $\mathsf{Groth16}$ proof after **Sapling** activation);

- $\mathsf{C}^{\mathsf{enc}}_{1..N^{\mathsf{new}}} ⦂ \mathsf{Sym}.\mathbf{C}^{[N^{\mathsf{new}}]}$ is a sequence of ciphertext components for the encrypted output *notes*.

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

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

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

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

4.4 Spend Descriptions

A *Spend transfer*, as specified in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 15, 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 $A_{\mathsf{MerkleSapling}}$ and $A_{\mathsf{PRFnfSapling}}$ be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{ValueCommit.Output}$ be as defined in §4.1.7 *'Commitment'* on p. 23.

Let $\mathsf{SpendAuthSig}$ be as defined in §4.13 *'Spend Authorization Signature'* on p. 38.

Let $\mathsf{ZKSpend}$ be as defined in §4.1.12 *'Zero-Knowledge Proving System'* on p. 26.

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

- $\mathsf{cv} ⦂ \mathsf{ValueCommit.Output}$ is the *value commitment* to the value of the input *note*;
- $\mathsf{rt} ⦂ \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$ is an *anchor* , as defined in §3.3 *'The Block Chain'* on p. 14, for the output *treestate* of a previous *block*;
- $\mathsf{nf} ⦂ \mathbb{B}^{[A_{\mathsf{PRFnfSapling}}]}$ is the *nullifier* for the input *note*;
- $\mathsf{rk} ⦂ \mathsf{SpendAuthSig.Public}$ is a randomized public key that should be used to verify spendAuthSig;
- $\pi_{\mathsf{ZKSpend}} ⦂ \mathsf{ZKSpend.Proof}$ is a *zero-knowledge proof* with *primary input* $(\mathsf{cv}, \mathsf{rt}, \mathsf{nf}, \mathsf{rk})$ for the *Spend statement* defined in §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41;
- spendAuthSig $⦂ \mathsf{SpendAuthSig.Signature}$ is as specified in §4.13 *'Spend Authorization Signature'* on p. 38.

Consensus rules:

- Elements of a *Spend description* **MUST** be canonical encodings of the types given above.
- cv and 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 π_{ZKSpend} **MUST** be valid given a *primary input* formed from the other fields except spendAuthSig — i.e. $\mathsf{ZKSpend.Verify}((\mathsf{cv}, \mathsf{rt}, \mathsf{nf}, \mathsf{rk}), \pi_{\mathsf{ZKSpend}}) = 1$.
- Let $\mathsf{SigHash}$ be the *SIGHASH transaction hash* of this *transaction*, not associated with an input, as defined in §4.9 *'SIGHASH Transaction Hashing'* on p. 35 using $\mathsf{SIGHASH\_ALL}$.

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

Non-normative note: The check that 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.

4.5 Output Descriptions

An *Output transfer* , as specified in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 15, 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 $\mathsf{ValueCommit.Output}$ be as defined in §4.1.7 *'Commitment'* on p. 23.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in §4.1.4 *'Key Agreement'* on p. 19.

Let Sym be as defined in §4.1.3 *'Authenticated One-Time Symmetric Encryption'* on p. 19.

Let $\mathsf{ZKSpend}$ be as defined in §4.1.12 *'Zero-Knowledge Proving System'* on p. 26.

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

- $\mathsf{cv} ⦂ \mathsf{ValueCommit.Output}$ is the *value commitment* to the value of the output *note*;
- $\mathsf{cm}_u ⦂ \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$ is the result of applying $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ (defined in §5.4.8.4 *'Hash Extractor for Jubjub'* on p.68) 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* (§4.17 *'In-band secret distribution (**Sapling**)'* on p.44);
- $\mathsf{C}^{\mathsf{enc}} ⦂ \mathsf{Sym}.\mathbf{C}$ 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 *diversibed transmission key* $\mathsf{pk_d}$ and the ephemeral private key esk (and therefore the entire *note plaintext*);
- $\pi_{\mathsf{ZKOutput}} ⦂ \mathsf{ZKOutput.Proof}$ is a *zero-knowledge proof* with *primary input* $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk})$ for the *Output statement* defined in §4.15.3 *'Output Statement (**Sapling**)'* on p. 42.

Consensus rules:

- Elements of an *Output description* **MUST** be canonical encodings of the types given above.
- cv and 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 π_{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. $\mathsf{ZKSpend.Verify}((\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk}), \pi_{\mathsf{ZKOutput}}) = 1$.

4.6 Sending Notes

4.6.1 Sending Notes (Sprout)

In order to send **Sprout** *shielded* value, the sender constructs a *transaction* containing one or more *JoinSplit descriptions*. This involves first generating a new $\mathsf{JoinSplitSig}$ key pair:

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

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

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

$\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ are then encrypted to the recipient *transmission keys* $\mathsf{pk}^{\mathsf{new}}_{\mathsf{enc},1..\mathrm{N}^{\mathsf{new}}}$, giving the *transmitted notes ciphertext* $(\mathsf{epk}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$, as described in §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43.

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 $\mathsf{dataToBeSigned} \in \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ as described in §4.10 *'Non-malleability (**Sprout**)'* on p. 35, and signs it with the private *JoinSplit signing key* :

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

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

4.6.2 Sending Notes (Sapling)

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

Let $\mathsf{ValueCommit}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in §4.1.7 *'Commitment'* on p. 23.

Let $\mathsf{repr}_{\mathbb{J}}$ and $h_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

Let ovk be an *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.

For each *Output description*, the sender selects a value $\mathsf{v}^{\mathsf{new}}$ 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 $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$, i.e. it is a valid Edwards point on the *Jubjub curve* not equal to $\mathcal{O}_{\mathbb{J}}$, and $[r_{\mathbb{J}}]\,\mathsf{pk_d} = \mathcal{O}_{\mathbb{J}}$.
- Calculate $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$ and check that $\mathsf{g_d} \neq \bot$.
- Choose independent uniformly random commitment trapdoors:

$$\mathsf{rcv}^{\mathsf{new}} \xleftarrow{R} \mathsf{ValueCommit.GenTrapdoor}()$$
$$\mathsf{rcm}^{\mathsf{new}} \xleftarrow{R} \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$$

- Calculate

$$\mathsf{cv}^{\mathsf{new}} := \mathsf{ValueCommit}_{\mathsf{rcv}^{\mathsf{new}}}(\mathsf{v}^{\mathsf{new}})$$
$$\mathsf{cm}^{\mathsf{new}} := \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}^{\mathsf{new}})$$

- Let **np** $= (\mathsf{d}, \mathsf{v}^{\mathsf{new}}, \underline{\mathsf{rcm}}, \mathsf{memo})$, where $\underline{\mathsf{rcm}} = \mathsf{LEBS2OSP}_{256}\dot{}\,\mathsf{I2LEBSP}_{256}(\mathsf{rcm}^{\mathsf{new}})^{\sqcup}$.
- Encrypt **np** to the recipient *diversified transmission key* $\mathsf{pk_d}$ with *diversified transmission base* $\mathsf{g_d}$, and to the *outgoing viewing key* ovk, giving the *transmitted note ciphertext* $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$ as described in §4.17.1 *'Encryption (**Sapling**)'* on p.45. This procedure also uses $\mathsf{cv}^{\mathsf{new}}$ and $\mathsf{cm}^{\mathsf{new}}$ to derive the *outgoing cipher key* .
- Generate a proof π_{ZKOutput} for the *Output statement* in §4.15.3 *'Output Statement (**Sapling**)'* on p. 42.
- Return $(\mathsf{cv}^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi_{\mathsf{ZKOutput}})$.

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 network.

4.7 Dummy Notes

4.7.1 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 $\ell_{\mathsf{PRFSprout}}$ be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{PRF}^{\mathsf{nf}}$ be as defined in §4.1.2 *'Pseudo Random Functions'* on p. 18.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}.\mathsf{Trapdoor}$ be as defined in §4.1.7 *'Commitment'* on p. 23.

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

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

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

4.7.2 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 ℓ_{sk} be as defined in §5.3 *'Constants'* on p. 49.

Let $r_{\mathbb{J}}$ and $\mathsf{repr}_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

Let $\mathcal{H}$ be as defined in §4.2.2 *'**Sapling** Key Components'* on p. 27.

Let $\mathsf{PRF}^{\mathsf{nfSapling}}$ be as defined in §4.1.2 *'Pseudo Random Functions'* on p. 18.

Let $\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}$ be as defined in §4.1.7 *'Commitment'* on p. 23.

A *dummy* **Sapling** input *note* is constructed as follows:

- Choose uniformly random $\mathsf{sk} \xleftarrow{R} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$.
- Generate a new *diversibed payment address* $(\mathsf{d}, \mathsf{pk_d})$ for sk as described in §4.2.2 *'**Sapling** Key Components'* on p. 27.
- Set $\mathsf{v^{old}} = 0$, and set $\mathsf{pos} = 0$.
- Choose uniformly random $\mathsf{rcm} \xleftarrow{R} \mathsf{NoteCommit^{Sapling}.GenTrapdoor}()$. and $\mathsf{nsk} \xleftarrow{R} \mathbb{F}_{r_{\mathbb{J}}}$.
- Compute $\mathsf{nk} = [\mathsf{nsk}]\,\mathcal{H}$ and $\mathsf{nk{>}} = \mathsf{repr_{\mathbb{J}}}(\mathsf{nk})$.
- Compute $\rho = \mathsf{cm^{old}} = \mathsf{NoteCommit^{Sapling}_{rcm}}(\mathsf{repr_{\mathbb{J}}}(\mathsf{g_d}), \mathsf{repr_{\mathbb{J}}}(\mathsf{pk_d}), \mathsf{v^{old}})$.
- Compute $\mathsf{nf^{old}} = \mathsf{PRF^{nfSapling}_{nk\star}}(\mathsf{repr_{\mathbb{J}}}(\rho))$.
- Construct a *dummy Merkle path* path for use in the *auxiliary input* to the *Spend statement* (this will not be checked, because $\mathsf{v^{old}} = 0$).

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 Merkle path validity

Let $\mathsf{MerkleDepth}$ be $\mathsf{MerkleDepth^{Sprout}}$ for the **Sprout** *note commitment tree*, or $\mathsf{MerkleDepth^{Sapling}}$ for the **Sapling** *note commitment tree*. These constants are deZned in §5.3 *'Constants'* on p. 49.

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

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

Each *node* in the *incremental Merkle tree* is associated with a *hash value*, which is a bit sequence.

The *layer* numbered h, counting from *layer* 0 at the *root* , has 2^h *nodes* with *indices* 0 to $2^h - 1$ inclusive.

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

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

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

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

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

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

where

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

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

4.9 SIGHASH Transaction Hashing

Bitcoin and **bitzec** 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*, but (for a given *transaction version*) the same *SIGHASH transaction hash* algorithm is used.

In the case of **bitzec**, the PHGR13 and Groth16 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. **bitzec** has been designed to mitigate malleability attacks, as described in §4.10 *'Non-malleability (**Sprout**)'* on p. 35, §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36, and §4.13 *'Spend Authorization Signature'* on p. 38.

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 **bitzec**-specific signatures, i.e. *JoinSplit signatures*, *spend authorization signatures*, and *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-**bitzec**-specific parts of the *transaction*.

In **bitzec**, all *SIGHASH types* are extended to cover the **bitzec**-specific fields nJoinSplit, vJoinSplit, and if present joinSplitPubKey. These fields are described in §7.1 *'Encoding of Transactions'* on p. 78. 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.

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

[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 bewritten).

[**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].

4.10 Non-malleability (Sprout)

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

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

JoinSplitSig is instantiated in §5.4.5 *'JoinSplit Signature'* on p. 59.

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

Let $\mathsf{h_{Sig}}$ be computed as speciZed in §4.3 *'JoinSplit Descriptions'* on p. 29.

Let $\mathsf{PRF^{pk}}$ be as deZned in §4.1.2 *'Pseudo Random Functions'* on p. 18.

For each $i \in \{1..\mathrm{N^{old}}\}$, 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 $\mathsf{h}_{1..\mathrm{N^{old}}}$ is enforced by the *JoinSplit statement* given in §4.15.1 *'Non-malleability'* on p. 40. This ensures that a holder of all of the $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1..\mathrm{N^{old}}}$ for every *JoinSplit description* in the *transaction* has authorized the use of the private signing key corresponding to joinSplitPubKey to sign this *transaction*.

4.11 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 outputs* minus *transparent inputs* 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*.

bitzec Sprout extends this by adding *JoinSplit transfers*. Each *JoinSplit transfer* can be seen, from the perspective of the *transparent value pool* , as an inputand an output simultaneously. $\mathsf{v^{old}_{pub}}$ takes value from the *transparent value pool* and $\mathsf{v^{new}_{pub}}$ adds value to the *transparent value pool* . As a result, $\mathsf{v^{old}_{pub}}$ is treated like an *output* value, whereas $\mathsf{v^{new}_{pub}}$ is treated like an *input* value.

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

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

As stated in §4.3 *'JoinSplit Descriptions'* on p. 29, either $\mathsf{v^{old}_{pub}}$ or $\mathsf{v^{new}_{pub}}$ **MUST** be zero. No generality is lost because, if a *transaction* in which both $\mathsf{v^{old}_{pub}}$ and $\mathsf{v^{new}_{pub}}$ were nonzero were allowed, it could be replaced by an equivalent one in which $\mathsf{min}(\mathsf{v^{old}_{pub}}, \mathsf{v^{new}_{pub}})$ is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between *transactions* according to client implementation.

4.12Balance 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 *balancing value*, measured in *zatoshi* as a signed integer $\mathsf{v^{balance}}$.

$\mathsf{v^{balance}}$ is encoded explicitly in a *transaction* as the Zeld valueBalance; see §7.1 *'Encoding of Transactions'* on p. 78.

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

Consistency of $\mathsf{v^{balance}}$ with the *value commitments* in *Spend descriptions* and *Output descriptions* is enforced by the *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 $\mathsf{v^{balance}}$ Zeld 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 *balancing value*.

Let $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $r_{\mathbb{J}}$ be as deZned in §5.4.8.3 *'Jubjub'* on p. 67.

Let $\mathsf{ValueCommit}$, $\mathcal{V}$, and $\mathcal{R}$ be as defined in §5.4.7.3 *'Homomorphic Pedersen commitments'* on p. 63:

$$\mathsf{ValueCommit} : \mathsf{ValueCommit.Trapdoor} \times \left\{-\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2}\right\} \to \mathsf{ValueCommit.Output};$$

$\mathcal{V} : \mathbb{J}^{(r)*}$ is the value base in $\mathsf{ValueCommit}$;

$\mathcal{R} : \mathbb{J}^{(r)*}$ is the randomness base in $\mathsf{ValueCommit}$.

$\mathsf{BindingSig}$, $\oplus$, and $\boxplus$ are instantiated in §5.4.6.2 *'Binding Signature'* on p. 62. These and the derived notation $\ominus$, $\bigoplus_{i=1}^{\mathrm{N}}$, $\boxminus$ and $\mathop{\boxplus}_{i=1}^{\mathrm{N}}$ are specified in §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22.

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 $\mathsf{rcv}^{\mathsf{new}}_{1..m}$;
- *balancing value* $\mathsf{v}^{\mathsf{balance}}$.

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

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

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

(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:

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

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

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

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

A validator checks balance by verifying that $\mathsf{BindingSig.Verify}_{\mathsf{bvk}}(\mathsf{SigHash}, \mathsf{bindingSig}) = 1$.

We now explain why this works.

A *binding signature* proves knowledge of the discrete logarithm bsk of bvk with respect to $\mathcal{R}$. That is, $\mathsf{bvk} = [\mathsf{bsk}]\,\mathcal{R}$. So the value 0 and randomness bsk is an opening of the *Pedersen commitment* $\mathsf{bvk} = \mathsf{ValueCommit}_{\mathsf{bsk}}(0)$. 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 more than one opening for any of those commitments, i.e. we may assume that $\mathsf{v}^{\mathsf{old}}_{1..n}$ and $\mathsf{rcv}^{\mathsf{old}}_{1..n}$ are determined by $\mathsf{cv}^{\mathsf{old}}_{1..n}$, and that $\mathsf{v}^{\mathsf{new}}_{1..m}$ and $\mathsf{rcv}^{\mathsf{new}}_{1..m}$ are determined by $\mathsf{cv}^{\mathsf{new}}_{1..m}$.

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

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

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

Suppose that $\mathsf{v}^* = \mathsf{v}^{\mathsf{bad}} \neq 0 \pmod{r_{\mathbb{J}}}$. Then $\mathsf{bvk} = \mathsf{ValueCommit}_{\mathsf{bsk}}(\mathsf{v}^{\mathsf{bad}})$. If the adversary were able to find the discrete logarithm of this bvk with respect to $\mathcal{R}$ say bsk' (as needed to create a valid *binding signature*), then $(\mathsf{v}^{\mathsf{bad}}, \mathsf{bsk})$ and $(0, \mathsf{bsk}')$ would be distinct openings of bvk 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* prove that all of $\mathsf{v}^{\mathsf{old}}_{1..n}$ are in $\{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\}$. Similarly the *Output statements* prove that all of $\mathsf{v}^{\mathsf{new}}_{1..m}$ are in $\{0 \,..\, 2^{\ell_{\mathsf{value}}} - 1\}$. $\mathsf{v}^{\mathsf{balance}}$ is encoded in the *transaction* as a signed two's complement 64-bit integer in the range $\{-2^{63} \,..\, 2^{63} - 1\}$. ℓ_{value} is defined as 64, so v^* is in the range $\{-m \cdot (2^{64} - 1) - 2^{63} + 1 \,..\, n \cdot (2^{64} - 1) + 2^{63}\}$. The maximum *transaction* size of 2 MB limits n to at most $\mathsf{floor}\left(\frac{2000000}{384}\right) = 5208$ and m to at most $\mathsf{floor}\left(\frac{2000000}{948}\right) = 2109$, ensuring $\mathsf{v}^* \in \{-38913406623490299131842 \,..\, 96079866507916199586728\}$ which is a subrange of $\left\{ -\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2} \right\}$.

Thus checking the *binding signature* ensures that the *transaction* balances, without the individual values of the *Spend descriptions* and *Output descriptions* being revealed.

In addition this proves that the signer, knowing the $\boxplus$-sum of the *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 *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 public key derived from a sum of *Pedersen commitments* is also used in the **Mimblewimble** protocol [Jedusor2016]. The private key bsk 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].

4.13 Spend Authorization Signature

$\mathsf{SpendAuthSig}$ is used in **Sapling** to prove knowledge of the *spending key* authorizing spending of an input *note*. It is instantiated in §5.4.6.1 *'Spend Authorization Signature'* on p. 62.

Knowledge of the *spending key* could have been proven directly in the *Spend statement*, similar to the check in §4.15.1 *'Spend authority'* on p. 40 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 $\mathsf{PHGR13}$ or $\mathsf{Groth16}$ proving systems.

The verifying key of the signature must be revealed in the *Spend description* so that the signature can be checked by validators. To ensure that the verifying 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 verifying key is a re-randomization of the *spend authorization address key* 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 ask be the *spend authorization private key* as defined in §4.2.2 *'**Sapling** Key Components'* on p. 27.

For each *Spend description*, the signer chooses a fresh *spend authorization randomizer* α:

1. Choose $\alpha \overset{R}{\leftarrow} \mathsf{SpendAuthSig.GenRandom}()$.
2. Let $\mathsf{rsk} = \mathsf{SpendAuthSig.RandomizePrivate}(\alpha, \mathsf{ask})$.
3. Let $\mathsf{rk} = \mathsf{SpendAuthSig.DerivePublic}(\mathsf{rsk})$.
4. Generate a proof π_{ZKSpend} of the *Spend statement* (§4.15.2 *'Spend Statement (**Sapling**)'* on p. 41), with α in the *auxiliary input* and rk in the *primary input*.
5. Let $\mathrm{spendAuthSig} = \mathsf{SpendAuthSig.Sign}_{\mathsf{rsk}}(\mathsf{SigHash})$.

The resulting spendAuthSig and π_{ZKSpend} are included in the *Spend description*.

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 proof*. In this case privacy will be lost to that party since it needs ak and the *proof authorizing key* nsk; this allows also deriving the nk component of the *full viewing key*. Together ak and 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.

4.14 Note Commitments and Nullifiers

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

All of the constituent *nullifiers* are also entered into the *nullifier set* of the associated *treestate*. A *transaction* is not valid if it would have added a *nullifier* to the *nullifier set* that already exists in the set (see §3.8 *'Nullifier Sets'* on p. 17).

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

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

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

$\mathsf{MixingPedersenHash}$ is defined in §5.4.1.8 *'Mixing Pedersen Hash Function'* on p. 55.

Let $\mathsf{PRF}^{\mathsf{nf}}$ and $\mathsf{PRF}^{\mathsf{nfSapling}}$ be as instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 56.

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

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

4.15 Zk-SNARK Statements

4.15.1 JoinSplit Statement (Sprout)

Let $\ell_{\mathsf{MerkleSprout}}$, $\ell_{\mathsf{PRFSprout}}$, $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$, ℓ_{value}, $\ell_{\mathsf{a_{sk}}}$, ℓ_{ϕ}, ℓ_{hSig}, $\mathrm{N}^{\mathsf{old}}$, $\mathrm{N}^{\mathsf{new}}$ be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nf}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ} be as defined in §4.1.2 *'Pseudo Random Functions'* on p. 18.

Let $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ be as defined in §4.1.7 *'Commitment'* on p. 23, and let $\mathsf{Note}^{\mathsf{Sprout}}$ and $\mathsf{NoteCommitment}^{\mathsf{Sprout}}$ be as defined in §3.2 *'Notes'* on p. 12.

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

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

the prover knows an *auxiliary input* :

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

where:

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

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

such that the following conditions hold:

Merkle path validity for each $i \in \{1..\mathrm{N}^{\mathsf{old}}\} \mid \mathsf{enforceMerklePath}_i = 1$: $(\mathsf{path}_i, \mathsf{pos}_i)$ is a valid *Merkle path* (see §4.8 *'Merkle path validity'* on p. 34) of depth $\mathsf{MerkleDepth}^{\mathsf{Sprout}}$ from $\mathsf{NoteCommitment}^{\mathsf{Sprout}}(\mathbf{n}^{\mathsf{old}}_i)$ to the *anchor* rt.

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

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

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

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

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

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

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

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 §5.4.9.1 *'PHGR13'* on p. 69.

4.15.2 Spend Statement (Sapling)

Let $A_{\mathsf{MerkleSapling}}$, $A_{\mathsf{PRFnfSapling}}$, and A_{scalar} be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{ValueCommit}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in §4.1.7 *'Commitment'* on p. 23.

Let $\mathsf{SpendAuthSig}$ be as defined in §5.4.6.1 *'Spend Authorization Signature'* on p. 62.

Let $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathsf{repr}_{\mathbb{J}}$, $q_{\mathbb{J}}$, $r_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

Let $\mathsf{Extract}_{\mathbb{J}^{(r)}} ⦂ \mathbb{J}^{(r)} \rightarrow \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$ be as defined in §5.4.8.4 *'Hash Extractor for Jubjub'* on p. 68.

Let $\mathcal{H}$ be as defined in §4.2.2 *'**Sapling** Key Components'* on p. 27.

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

$\mathsf{rt} ⦂ \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$,
$\mathsf{cv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit.Output}$,
$\mathsf{nf}^{\mathsf{old}} ⦂ \mathbb{B}^{[A_{\mathsf{PRFnfSapling}}]}$,
$\mathsf{rk} ⦂ \mathsf{SpendAuthSig.Public}$,

the prover knows an *auxiliary input* :

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

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 §4.8 *'Merkle path validity'* on p. 34, from $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm}^{\mathsf{old}})$ to the *anchor* rt.

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

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

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

$\mathsf{nk}> = \mathsf{repr}_{\mathbb{J}}([\mathcal{H}]\, \mathsf{nsk})$

$\rho> = \mathsf{repr}_{\mathbb{J}}(\mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos}))$.

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

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

$\mathsf{ivk} = \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}>, \mathsf{nk}>)$
$\mathsf{ak}> = \mathsf{repr}_{\mathbb{J}}(\mathsf{ak})$.

For details of the form and encoding of *Spend statement* proofs, see §5.4.9.2 *'Groth16'* on p. 70.

Notes:

- Public and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see §A.3.3.2 *'Edwards [de]compression and validation'* on p. 124 for implementation of validity checks on compressed representations of *Jubjub curve* points.

  The $\mathsf{ValueCommit.Output}$ and $\mathsf{SpendAuthSig.Public}$ 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 $\{0 .. r_{\mathbb{S}} - 1\}$) of the integer from the previous *layer*.
- It is *not* checked in the *Spend statement* that rk is not of small order. However, this *is* checked outside the *Spend statement* , as specified in §4.4 *'Spend Descriptions'* on p. 30.
- It is *not* checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.
- $\mathsf{SpendAuthSig.RandomizePublic}(\alpha, \mathsf{ak}) = \mathsf{ak} + [\alpha]\,\mathcal{G}$ (is as defined in §5.4.6.1 *'Spend Authorization Signature'* on p. 62.)

4.15.3 Output Statement (Sapling)

Let $A_{\mathsf{MerkleSapling}}$, $A_{\mathsf{PRFnfSapling}}$, and A_{scalar} be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{ValueCommit}$ and $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as specified in §4.1.7 *'Commitment'* on p. 23.

Let $\mathbb{J}$, $\mathsf{repr}_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

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

$\mathsf{cv}^{\mathsf{new}} ⦂ \mathsf{ValueCommit.Output}$,
$\mathsf{cm}_u ⦂ \mathbb{B}^{[A_{\mathsf{MerkleSapling}}]}$,
$\mathsf{epk} ⦂ \mathbb{J}$,

the prover knows an *auxiliary input* :

$(\mathsf{g_d} ⦂ \mathbb{J}$,
$\mathsf{pk{>}_d} ⦂ \mathbb{B}^{[A_{\mathbb{J}}]}$,
$\mathsf{v}^{\mathsf{new}} ⦂ \{0 .. 2^{A_{\mathsf{value}}} - 1\}$,
$\mathsf{rcv}^{\mathsf{new}} ⦂ \{0 .. 2^{A_{\mathsf{scalar}}} - 1\}$,
$\mathsf{rcm}^{\mathsf{new}} ⦂ \{0 .. 2^{A_{\mathsf{scalar}}} - 1\}$,
$\mathsf{esk} ⦂ \{0 .. 2^{A_{\mathsf{scalar}}} - 1\})$

such that the following conditions hold:

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

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

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 §5.4.9.2 *'Groth16'* on p. 70.

Notes:

- Public and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see §A.3.3.2 *'Edwards [de]compression and validation'* on p. 124 for implementation of validity checks on compressed representations of *Jubjub curve* points.

  The $\mathsf{ValueCommit.Output}$ type also represents points, i.e. $\mathbb{J}$.
- The validity of $\mathsf{pk}>_{\mathsf{d}}$ is *not* checked in this circuit.
- It is *not* checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.

4.16 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 v, $\mathsf{\rho}$, and rcm. A *memo field* (§3.2.1 *'Note Plaintexts and Memo Fields'* on p. 13) is also transmitted.

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

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

For both encryption and decryption,

- let Sym be the scheme instantiated in §5.4.3 *'Authenticated One-Time Symmetric Encryption'* on p. 57;
- let $\mathsf{KDF}^{\mathsf{Sprout}}$ be the *Key Derivation Function* instantiated in §5.4.4.2 *'**Sprout** Key Derivation'* on p. 58;
- let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58;
- let $\mathsf{h_{Sig}}$ be the value computed for this *JoinSplit description* in §4.3 *'JoinSplit Descriptions'* on p. 29.

4.16.1 Encryption (Sprout)

Let $\mathsf{KA}^{\mathsf{Sprout}}$ be the *key agreement scheme* instantiated in §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58.

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

Let $\mathbf{np}_{1..\mathrm{N}^{\mathsf{new}}}$ be **Sprout** *note plaintexts* defined in §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 71.

Then to encrypt:

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

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.

4.16.2 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. 27.

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

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

```
let $\mathsf{sharedSecret}_i = \mathsf{KA^{Sprout}.Agree}(\mathsf{sk_{enc}}, \mathsf{epk})$
let $\mathsf{K}^{\mathsf{enc}}_i = \mathsf{KDF^{Sprout}}(i, \mathsf{h_{Sig}}, \mathsf{sharedSecret}_i, \mathsf{epk}, \mathsf{pk_{enc}})$
return DecryptNoteSprout($\mathsf{K}^{\mathsf{enc}}_i, \mathbf{C}^{\mathsf{enc}}_i, \mathsf{cm}^{\mathsf{new}}_i, \mathsf{a_{pk}}$).
```

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

```
let $\mathbf{P}^{\mathsf{enc}}_i = \mathsf{Sym.Decrypt}_{\mathsf{K}^{\mathsf{enc}}_i}(\mathbf{C}^{\mathsf{enc}}_i)$
if $\mathbf{P}^{\mathsf{enc}}_i = \bot$, return $\bot$
extract $\mathbf{np}_i = (\mathsf{v}^{\mathsf{new}}_i \in \{0 .. 2^{\ell_{\mathsf{value}}} - 1\}, \rho^{\mathsf{new}}_i \in \mathbb{B}^{[\ell_{\mathsf{PRFSprout}}]}, \mathsf{rcm}^{\mathsf{new}}_i \in \mathsf{NoteCommit^{Sprout}.Trapdoor}, \mathsf{memo}_i \in \mathbb{B}^{\mathbb{Y}[512]})$ from $\mathbf{P}^{\mathsf{enc}}_i$
if $\mathsf{NoteCommitment^{Sprout}}((\mathsf{a_{pk}}, \mathsf{v}^{\mathsf{new}}_i, \rho^{\mathsf{new}}_i, \mathsf{rcm}^{\mathsf{new}}_i)) \neq \mathsf{cm}^{\mathsf{new}}_i$, return $\bot$, else return $\mathbf{np}_i$.
```

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

Notes:

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

See §8.7 *'In-band secret distribution'* on p. 94 for further discussion of the security and engineering rationale behind this encryption scheme.

4.17 In-band secret distribution (Sapling)

In **Sapling**, the secrets that need to be transmitted to a recipient of funds in order for them to later spend, are d, v, and rcm. A *memo field* (§3.2.1 *'Note Plaintexts and Memo Fields'* on p. 13) 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* ivk is used to reconstruct the original *note* and *memo field*.

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

For both encryption and decryption,

- let $\mathcal{A}_{\mathsf{ovk}}$ be as defined in §5.3 *'Constants'* on p. 49;
- let Sym be the scheme instantiated in §5.4.3 *'Authenticated One-Time Symmetric Encryption'* on p. 57;
- let $\mathsf{KDF^{Sapling}}$ be the *Key Derivation Function* instantiated in §5.4.4.4 *'**Sapling** Key Derivation'* on p. 59;
- let $\mathsf{KA^{Sapling}}$ be the *key agreement scheme* instantiated in §5.4.4.3 *'**Sapling** Key Agreement'* on p. 58;
- let $\mathcal{A}_{\mathbb{J}}$ and $\mathsf{repr}_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67;
- let $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ be as defined in §5.4.8.4 *'Hash Extractor for Jubjub'* on p. 68;
- let $\mathsf{PRF^{ock}}$ be as instantiated in §5.4.2 *'Pseudo Random Functions'* on p. 56.

4.17.1 Encryption (Sapling)

Let $\mathsf{pk}_{\mathsf{d}}^{\mathsf{new}} \circ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$ be the *diversified transmission key* for the intended recipient address of a new **Sapling** *note*, and let $\mathsf{g}_{\mathsf{d}}^{\mathsf{new}} \circ \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$ be the corresponding *diversified base* computed as $\mathsf{DiversifyHash}(\mathsf{d})$.

Since **Sapling** *note* encryption is used only in the context of §4.6.2 *'Sending Notes (**Sapling**)'* on p. 32, we may assume that $\mathsf{g}_{\mathsf{d}}^{\mathsf{new}}$ has already been calculated and is not $\perp$.

Let $\mathsf{ovk} \circ \mathsf{B}^{\mathsf{Y}[A_{\mathsf{ovk}}/8]} \cup \{\perp\}$ be as described in §4.6.2 *'Sending Notes (**Sapling**)'* on p. 32, 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 $\perp$.

Let **np** $= (\mathsf{d}, \mathsf{v}, \underline{\mathsf{rcm}}, \mathsf{memo})$ be the **Sapling** *note plaintext* .

np is encoded as deZned in §5.5 *'Encodings of Note Plaintexts and Memo Fields'* on p. 71.

Let $\mathsf{cv}^{\mathsf{new}}$ be the *value commitment* for the new *note*, and let $\mathsf{cm}^{\mathsf{new}}$ be the *note commitment* .

Then to encrypt:

```
    choose a uniformly random ephemeral private key $\mathsf{esk} \xleftarrow{R} \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Private} \setminus \{0\}$
    let $\mathsf{epk} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{DerivePublic}(\mathsf{esk}, \mathsf{g}_{\mathsf{d}}^{\mathsf{new}})$
    let $\mathsf{P}^{\mathsf{enc}}$ be the raw encoding of np
    let $\mathsf{sharedSecret} = \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{esk}, \mathsf{pk}_{\mathsf{d}}^{\mathsf{new}})$
    let $\mathsf{K}^{\mathsf{enc}} = \mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{sharedSecret}, \mathsf{epk})$
    let $\mathsf{C}^{\mathsf{enc}} = \mathsf{Sym.Encrypt}_{\mathsf{K}^{\mathsf{enc}}}(\mathsf{P}^{\mathsf{enc}})$
    if $\mathsf{ovk} = \perp$:
        choose random $\mathsf{ock} \xleftarrow{R} \mathsf{Sym}.\mathbf{K}$ and $\mathbf{op} \xleftarrow{R} \mathsf{B}^{\mathsf{Y}[(A_{\mathsf{J}}+256)/8]}$
    else:
        let $\mathrm{cv} = \mathsf{LEBS2OSP}_{A_{\mathsf{J}}}(\mathsf{repr}_{\mathsf{J}}(\mathsf{cv}^{\mathsf{new}}))$
        let $\mathrm{cmu} = \mathsf{LEBS2OSP}_{256}(\mathsf{Extract}_{\mathsf{J}^{(r)}}(\mathsf{cm}^{\mathsf{new}}))$
        let $\mathrm{ephemeralKey} = \mathsf{LEBS2OSP}_{A_{\mathsf{J}}}(\mathsf{repr}_{\mathsf{J}}(\mathsf{epk}))$
        let $\mathsf{ock} = \mathsf{PRF}_{\mathsf{ovk}}^{\mathsf{ock}}(\mathrm{cv}, \mathrm{cmu}, \mathrm{ephemeralKey})$
        let $\mathbf{op} = \mathsf{LEBS2OSP}_{A_{\mathsf{J}}+256}(\mathsf{repr}_{\mathsf{J}}(\mathsf{pk}_{\mathsf{d}}^{\mathsf{new}}) \,||\, \mathsf{I2LEBSP}_{256}(\mathsf{esk}))$
    let $\mathsf{C}^{\mathsf{out}} = \mathsf{Sym.Encrypt}_{\mathsf{ock}}(\mathbf{op})$
```

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

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

4.17.2 Decryption using an Incoming Viewing Key (Sapling)

Let $\mathsf{ivk} \circ \{0 \,..\, 2^{A_{\mathsf{ivk}}} - 1\}$ be the recipient's *incoming viewing key* , as speciZed in §4.2.2 *'**Sapling** Key Components'* on p. 27.

Let $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$ be the *transmitted note ciphertext* from the *Output description*. Let cmu be that Zeld of the *Output description* (encoding the *u*-coordinate of the *note commitment*).

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

```
let $\mathsf{sharedSecret} = \mathsf{KA^{Sapling}.Agree(ivk, epk)}$
let $\mathsf{K^{enc}} = \mathsf{KDF^{Sapling}}\,(\mathsf{sharedSecret, epk})$
let $\mathsf{P^{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K^{enc}}}(\mathsf{C^{enc}})$
if $\mathsf{P^{enc}} = \bot$, return $\bot$
extract $\mathbf{np} = (\mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0 .. 2^{\ell_{\mathsf{value}}} - 1\}, \underline{\mathsf{rcm}} ⦂ \mathbb{BY}^{[32]}, \mathsf{memo} ⦂ \mathbb{BY}^{[512]})$ from $\mathsf{P^{enc}}$
let $\mathsf{rcm} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{rcm}})$ and $\mathsf{g_d} = \mathsf{DiversifyHash(d)}$
if $\mathsf{rcm} \geq r_{\mathbb{J}}$ or $\mathsf{g_d} = \bot$, return $\bot$
let $\mathsf{pk_d} = \mathsf{KA^{Sapling}.DerivePublic(ivk, g_d)}$
                                         Sapling
let $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit_{rcm^{new}}}(\mathsf{repr_J(g_d), repr_J(pk_d), v}))$.
if $\mathsf{LEBS2OSP}_{256}(\mathsf{cm}_u^{r})$ Ç cmu, return $\bot$, else return $\mathbf{np}$.
```

A received **Sapling** *note* is necessarily a *positioned note*, and so its ρ value can immediately be calculated as described in §4.14 *'Note Commitments and Nullifiers'* on p. 39.

To test whether a **Sapling** *note* is unspent in a particular *block chain* also requires the *nulliber deriving key* $\mathsf{nk>}$; the coin is unspent if and only if $\mathsf{nf} = \mathsf{PRF^{nfSapling}_{nk_{\star}}}(\mathsf{repr_J}(\rho))$ is not in the *nulliber set* for that *block chain*.

Note: A *note* can change from being unspent to spent as a node's view of the best *block chain* is extended by new *transactions*. Also, *block chain* reorganizations can cause a node to switch to a different best *block chain* that does not contain the *transaction* in which a *note* was output.

4.17.3 Decryption using a Full Viewing Key (Sapling)

Let $\mathsf{ovk} ⦂ \mathbb{BY}^{[\ell_{\mathsf{ovk}}/8]}$ be the *outgoing viewing key*, as speciZed in §4.2.2 *'**Sapling** Key Components'* on p. 27, that is to be used for decryption. (If $\mathsf{ovk} = \bot$ was used for encryption, the payment is not decryptable by this method.)

Let $(\mathsf{epk}, \mathsf{C^{enc}}, \mathsf{C^{out}})$ be the *transmitted note ciphertext* , and let cv, cmu, and ephemeralKey be those Zelds of the *Output description* (encoding the *value commitment* , the *u*-coordinate of the *note commitment* , and epk).

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

```
let $\mathsf{ock} = \mathsf{PRF^{ock}_{ovk}}$ (cv, cmu, ephemeralKey)
let $\mathbf{op} = \mathsf{Sym.Decrypt_{ock}}(\mathsf{C^{out}})$
if $\mathbf{op} = \bot$, return $\bot$
extract $(\mathsf{pk>_d} ⦂ \mathbb{B}^{[\ell_{\mathbb{J}}]}, \underline{\mathsf{esk}} ⦂ \mathbb{BY}^{[32]})$ from $\mathbf{op}$
let $\mathsf{esk} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{esk}})$ and $\mathsf{pk_d} = \mathsf{abst_J}(\mathsf{pk>_d})$
if $\mathsf{esk} \geq r_{\mathbb{J}}$ or $\mathsf{pk_d}$ g $\mathsf{KA^{Sapling}.PublicPrimeOrder}$, return $\bot$
let $\mathsf{sharedSecret} = \mathsf{KA^{Sapling}.Agree(esk, pk_d)}$
let $\mathsf{K^{enc}} = \mathsf{KDF^{Sapling}}\,(\mathsf{sharedSecret, epk})$
let $\mathsf{P^{enc}} = \mathsf{Sym.Decrypt}_{\mathsf{K^{enc}}}(\mathsf{C^{enc}})$
if $\mathsf{P^{enc}} = \bot$, return $\bot$
extract $\mathbf{np} = (\mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0 .. 2^{\ell_{\mathsf{value}}} - 1\}, \underline{\mathsf{rcm}} ⦂ \mathbb{BY}^{[32]}, \mathsf{memo} ⦂ \mathbb{BY}^{[512]})$ from $\mathsf{P^{enc}}$
let $\mathsf{rcm} = \mathsf{LEOS2IP}_{256}(\underline{\mathsf{rcm}})$ and $\mathsf{g_d} = \mathsf{DiversifyHash(d)}$
if $\mathsf{rcm} \geq r_{\mathbb{J}}$ or $\mathsf{g_d} = \bot$, return $\bot$
if $\mathsf{KA^{Sapling}.DerivePublic(esk, g_d)}$ Ç $\mathsf{epk}$, return $\bot$
                                         Sapling
let $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{NoteCommit_{rcm^{new}}}(\mathsf{repr_J(g_d), repr_J(pk_d), v}))$.
if $\mathsf{LEBS2OSP}_{256}(\mathsf{cm}_u^{r})$ Ç cmu, return $\bot$, else return $\mathbf{np}$.
```

Note: For a valid *transaction* it must be the case that ephemeralKey $= \mathsf{LEBS2OSP}_{A_{\mathbb{J}}}\left(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})\right)$.

4.18 Block Chain Scanning (Sprout)

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* field, and its final status (spent or unspent).

Let $A_{\mathsf{PRFSprout}}$ be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{Note}^{\mathsf{Sprout}}$ be as defined in §3.2 *'Notes'* on p. 12.

Let $\mathsf{ivk} = (\mathsf{a_{pk}} ⦂ \mathbb{B}^{[A_{\mathsf{PRFSprout}}]}, \mathsf{sk_{enc}} ⦂ \mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private})$ 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. 27.

```
Initialize $\mathsf{ReceivedSet} ⦂ \mathbb{P}\left(\mathsf{Note}^{\mathsf{Sprout}} \times \mathbb{B}^{\mathbb{Y}[512]}\right) = \{\}$.
Initialize $\mathsf{SpentSet} ⦂ \mathbb{P}\left(\mathsf{Note}^{\mathsf{Sprout}}\right) = \{\}$.
Initialize $\mathsf{NullifierMap} ⦂ \mathbb{B}^{[A_{\mathsf{PRFSprout}}]} \to \mathsf{Note}^{\mathsf{Sprout}}$ to the empty mapping.

For each transaction $\mathsf{tx}$,
    For each JoinSplit description in $\mathsf{tx}$,
        Let $(\mathsf{epk}, \mathbf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}})$ be the transmitted notes ciphertext of the JoinSplit description.
        For $i$ in $1..\mathrm{N}^{\mathsf{new}}$,
            Attempt to decrypt the transmitted note ciphertext component $(\mathsf{epk}, \mathbf{C}^{\mathsf{enc}}_i)$ using $\mathsf{ivk}$ with the algorithm in §4.16.2 'Decryption (Sprout)' on p. 44. If this succeeds giving $\mathbf{np}$:
                Extract $\mathbf{n}$ and $\mathsf{memo} ⦂ \mathbb{B}^{\mathbb{Y}[512]}$ from $\mathbf{np}$ (taking the $\mathsf{a_{pk}}$ field of the note to be $\mathsf{a_{pk}}$ from $\mathsf{ivk}$).
                Add $(\mathbf{n}, \mathsf{memo})$ to $\mathsf{ReceivedSet}$.
                Calculate the nullifier $\mathsf{nf}$ of $\mathbf{n}$ using $\mathsf{a_{sk}}$ as described in §3.2 'Notes' on p. 12.
                Add the mapping $\mathsf{nf} \to \mathbf{n}$ to $\mathsf{NullifierMap}$.

        Let $\mathsf{nf}_{1..\mathrm{N}^{\mathsf{old}}}$ be the nullifiers of the JoinSplit description.
        For $i$ in $1..\mathrm{N}^{\mathsf{old}}$,
            If $\mathsf{nf}_i$ is present in $\mathsf{NullifierMap}$, add $\mathsf{NullifierMap}(\mathsf{nf}_i)$ to $\mathsf{SpentSet}$.

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

4.19 Block Chain Scanning (Sapling)

In **Sapling**, *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. 27.

The following algorithm can be used, given the *block chain* and $(\mathsf{nk} ⦂ \mathbb{J}^{(r)}, \mathsf{ivk} ⦂ \{0\,..\,2^{A_{\mathsf{ivk}}} - 1\})$, to obtain each *note* sent to the corresponding *shielded payment address*, its *memo field* field, and its final status (spent or unspent).

Let $A_{\mathsf{PRFnfSapling}}$ be as defined in §5.3 *'Constants'* on p. 49.

Let $\mathsf{Note}^{\mathsf{Sapling}}$ be as defined in §3.2 *'Notes'* on p. 12.

```
Initialize $\mathsf{ReceivedSet} : \mathsf{P}\big(\mathsf{Note}^{\mathsf{Sapling}} \times \mathbb{B}^{\mathbb{Y}[512]}\big) = \{\}$.
Initialize $\mathsf{SpentSet} : \mathsf{P}\big(\mathsf{Note}^{\mathsf{Sapling}}\big) = \{\}$.
Initialize $\mathsf{NullifierMap} : \mathbb{B}^{[\ell_{\mathsf{PRFnfSapling}}]} \to \mathsf{Note}^{\mathsf{Sapling}}$ to the empty mapping.

For each transaction $\mathsf{tx}$,
  For each Output description in $\mathsf{tx}$ with note position $\mathsf{pos}$,
    Attempt to decrypt the transmitted note ciphertext components $\mathsf{epk}$ and $\mathsf{C}^{\mathsf{enc}}$ using $\mathsf{ivk}$ with the algorithm
    in §4.17.2 'Decryption using an Incoming Viewing Key (Sapling)' on p. 45. If this succeeds giving $\mathbf{np}$:
      Extract $\mathbf{n}$ and $\mathsf{memo} : \mathbb{B}^{\mathbb{Y}[512]}$ from $\mathbf{np}$.
      Add $(\mathbf{n}, \mathsf{memo})$ to $\mathsf{ReceivedSet}$.
      Calculate the nulliZer $\mathsf{nf}$ of $\mathbf{n}$ using $\mathsf{nk}$ and $\mathsf{pos}$ as described in §3.2 'Notes' on p. 12.
      Add the mapping $\mathsf{nf} \to \mathbf{n}$ to $\mathsf{NullifierMap}$.

  For each Spend description in $\mathsf{tx}$,
    Let $\mathsf{nf}$ be the nulliber of the Spend description.
    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 $\mathsf{C}^{\mathsf{out}}$ *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 §4.17.3 *'Decryption using a Full Viewing Key (**Sapling**)'* on p. 46, 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 $\mathsf{C}^{\mathsf{out}}$ 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 *nullibers* are known.

5Concrete 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. 92

5.2 Integers, Bit Sequences, and Endianness

All integers in **bitzec**-speciZc encodings are unsigned, have a Zxed bit length, and are encoded in little-endian byte order *unless otherwise specified*.

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

- $\mathsf{I2LEBSP} : (\ell : \mathbb{N}) \times \{0 \,..\, 2^{\ell} - 1\} \to \mathbb{B}^{[\ell]}$, such that $\mathsf{I2LEBSP}_{\ell}(x)$ is the sequence of ℓ bits representing x in little-endian order;

- $\mathsf{I2BEBSP} : (\ell : \mathbb{N}) \times \{0 .. 2^{\ell} - 1\} \to \mathbb{B}^{[\ell]}$ such that $\mathsf{I2BEBSP}_{\ell}(x)$ is the sequence of ℓ bits representing x in big-endian order.
- $\mathsf{LEBS2IP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \{0 .. 2^{\ell} - 1\}$ such that $\mathsf{LEBS2IP}_{\ell}(S)$ is the integer represented in little-endian order by the bit sequence S of length ℓ.
- $\mathsf{LEOS2IP} : (\ell : \mathbb{N} \mid \ell \bmod 8 = 0) \times \mathbb{BY}^{[\ell/8]} \to \{0 .. 2^{\ell} - 1\}$ such that $\mathsf{LEOS2IP}_{\ell}(S)$ is the integer represented in little-endian order by the byte sequence S of length $\ell/8$.
- $\mathsf{LEBS2OSP} : (\ell : \mathbb{N}) \times \mathbb{B}^{[\ell]} \to \mathbb{BY}^{[\mathsf{ceiling}(\ell/8)]}$ 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 ℓ is given explicitly in each box, except when it is obvious (e.g. for a single bit, or for the notation $[0]^{\ell}$ representing the sequence of ℓ zero bits, or for the output of $\mathsf{LEBS2OSP}_{\ell}$).

The entire diagram represents the sequence of *bytes* formed by first concatenating these bit sequences, and then treating each subsequence of 8 bits as a byte with the bits ordered from *most significant* to *least significant*. Thus the *most significant* bit in each byte is toward the left of a diagram.(This convention is used only in descriptions of the **Sprout** design; in the **Sapling** 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:

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

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

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

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

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

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

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

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

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

$\ell_{\mathsf{PRFexpand}} : \mathbb{N} := 512$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

5.4.1.1 SHA-256 and SHA256Compress Hash Functions

SHA-256 is defined by [NIST2015].

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

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

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

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

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

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

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

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

5.4.1.2 BLAKE2 Hash Function

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

BLAKE2b-$A(p, x)$ refers to unkeyed BLAKE2b-A in sequential mode, with an output digest length of $A/8$ 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 $\mathsf{PRF}^{\mathsf{expand}}$, $\mathsf{PRF}^{\mathsf{ock}}$, $\mathsf{KDF}^{\mathsf{Sapling}}$, and in the RedJubjub *signature scheme* which instantiates SpendAuthSig and BindingSig.

$$\mathsf{BLAKE2b}\text{-}A \circ \mathsf{BY}^{[16]} \times \mathsf{BY}^{[\mathsf{N}]} \to \mathsf{BY}^{[A/8]}$$

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

BLAKE2s-$A(p, x)$ refers to unkeyed BLAKE2s-A in sequential mode, with an output digest length of $A/8$ bytes, 8-byte personalization string p, and input$x$.

BLAKE2s is used to instantiate $\mathsf{PRF}^{\mathsf{nfSapling}}$, $\mathsf{CRH}^{\mathsf{ivk}}$, and $\mathsf{GroupHash}^{\mathsf{J}^{(r)*}}$.

$$\mathsf{BLAKE2s}\text{-}A \circ \mathsf{BY}^{[8]} \times \mathsf{BY}^{[\mathsf{N}]} \to \mathsf{BY}^{[A/8]}$$

5.4.1.3 Merkle Tree Hash Function

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

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

Let SHA256Compress be as specified in §5.4.1.1 *'SHA-256 and SHA256Compress Hash Functions'* on p. 50.

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

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

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. 53.

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

$$\mathsf{MerkleCRH}^{\mathsf{Sapling}}(\mathsf{layer}, \mathsf{left}, \mathsf{right}) := \mathsf{PedersenHash}(\text{“}\mathbf{bitzec\_PH}\text{”}, l \,\|\, \mathsf{left}\ \mathsf{right})$$

where $l = \mathsf{I2LEBSP}_6\ \mathsf{MerkleDepth}_{\mathsf{Sapling}} - 1 - \mathsf{layer}$.

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 §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63.

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

hSigCRH is used to compute the value h$_{\mathsf{Sig}}$ in §4.3 *'JoinSplit Descriptions'* on p. 29.

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

where

hSigInput :=	256-bit randomSeed	256-bit $\mathsf{nf}^{\mathsf{old}}_{1}$ ...	256-bit $\mathsf{nf}^{\mathsf{old}}_{N^{\mathsf{old}}}$	256-bit joinSplitPubKey

BLAKE2b-256(p, x) is deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51.

Security requirement: BLAKE2b-256 **"bitzecComputehSig"**, x must be collision-resistant on x.

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

CRH$^{\mathsf{ivk}}$ is used to derive the *incoming viewing key* ivk for a **Sapling** *shielded payment address*. For its use when generating an address see §4.2.2 *'**Sapling** Key Components'* on p. 27, and for its use in the *Spend statement* see §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41.

It is deZned as follows:

$\mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}, \mathsf{nk}) := \mathsf{LEOS2IP}_{256}$ BLAKE2s-256 **"bitzecivk"**, crhInput mod $2^{\ell_{\mathsf{ivk}}}$

where

crhInput :=	$\mathsf{LEBS2OSP}_{256}(\mathsf{ak})$	$\mathsf{LEBS2OSP}_{256}(\mathsf{nk})$

BLAKE2b-256(p, x) is deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51.

Security requirement: $\mathsf{LEOS2IP}_{256}$ BLAKE2s-256 **"bitzecivk"**, x mod $2^{\ell_{\mathsf{ivk}}}$ 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-speciZc personalization string together with truncation achieve essentially the same effect as using that feature.

5.4.1.6 DiversifyHash Hash Function

DiversifyHash is used to derive a *diversibed base* from a *diversiber* in §4.2.2 *'**Sapling** Key Components'* on p. 27.

Let GroupHash$^{\mathbb{J}^{(r)*}}$ and U be as deZned in §5.4.8.5 *'Group Hash into Jubjub'* on p. 69.

DeZne

$\mathsf{DiversifyHash}(\mathsf{d}) := \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{U}$ **"bitzec_gd"**, $\mathsf{LEBS2OSP}_{\ell_{\mathsf{d}}}(\mathsf{d})$

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 *diversibers*, 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 $\mathsf{DiversifyHash}$ 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 pk, gives the same distribution of $(\mathsf{g_d}^{r}, \mathsf{pk_d}^{r})$ pairs as the distribution of ElGamal ciphertexts obtained by encrypting $\mathcal{O}_{\mathbb{J}}$ under 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 $\mathsf{DiversifyHash}$. (IK-CCA corresponds to the potentially stronger requirement that $\mathsf{DiversifyHash}$ remains Unlinkable when given Diffie-Hellman key agreement oracles for each of the candidate *diversified payment addresses*.) So if ElGamal is *key-private*, then $\mathsf{DiversifyHash}$ 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. 27), 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] 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* (§5.4.7.2 *‘Windowed Pedersen commitments’* on p. 63), and of the *hash function* for the **Sapling** *incremental Merkle tree* (§5.4.1.3 ‘$\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ *Hash Function’* on p. 51).

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 §5.4.8.3 *‘Jubjub’* on p. 67.

Let $\mathsf{Extract}_{\mathbb{J}^{(r)}} \circ \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ be as defined in §5.4.8.4 *‘Hash Extractor for Jubjub’* on p. 68.

Let $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ be as defined in §5.4.8.5 *‘Group Hash into Jubjub’* on p. 69.

Let $c := 63$.

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

$$\mathcal{I}_i^D := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}} \left(D, \boxed{\text{32-bit } i-1} \right).$$

Define $\mathsf{PedersenHashToPoint}(D \circ \mathbb{B}^{\mathbb{Y}[8]}, M \circ \mathbb{B}^{[\mathbb{N}^+]}) \to \mathbb{J}^{(r)}$ 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 $\sum_{i=1}^{n} [(M_i)]\, \mathcal{I}_i^D \circ \mathbb{J}^{(r)}$.

where $(\cdot) \circ \mathbb{B}^{[3 \cdot \{1\,..\,c\}]} \to \{-\frac{r_{\mathbb{J}}-1}{2}\,..\,\frac{r_{\mathbb{J}}-1}{2}\} \setminus \{0\}$ 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 $\mathsf{enc}(m_j) = (1 - 2 \cdot s_2^j) \cdot (1 + s_0^j + 2 \cdot s_1^j) \circ \mathbb{Z}$.

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

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

$$\mathsf{PedersenHash}(D, M) := \mathsf{Extract}_{\mathbb{J}^{(r)}}\left(\mathsf{PedersenHashToPoint}(D, M)\right).$$

See §A.3.3.9 *‘Pedersen hash’* on p. 129 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* $(\cdot)$ *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

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

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

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

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 $\mathsf{PedersenHashToPoint}$ is collision-resistant under the same assumptions and security bounds. Because $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ is injective, it follows that $\mathsf{PedersenHash}$ is equally collision-resistant.

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

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

5.4.1.8 Mixing Pedersen Hash Function

A mixing *Pedersen hash* is used to compute ρ from cm and pos in §4.14 *'Note Commitments and Nullifiers'* on p. 39. It takes as input a *Pedersen commitment* P, and hashes it with another input x.

DeZne $\mathcal{J} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\text{"bitzec\_J\_"}, \text{""})$.

We deZne $\mathsf{MixingPedersenHash} \colon \mathbb{J} \times \{0 .. r_{\mathbb{J}} - 1\} \to \mathbb{J}$ by:

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

Security requirement: The function

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

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

See §A.3.3.10 *'Mixing Pedersen hash'* on p. 132 for efZcient circuit implementation of this function.

5.4.1.9 Equihash Generator

$\mathsf{EquihashGen}_{n,k}$ is a specialized *hash function* that maps an input and an index to an output of length n bits. It is used in §7.6.1 *'Equihash'* on p. 85.

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

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

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

- $m := \mathsf{floor}\left(\frac{512}{n}\right)$;
- $h := (i - 1 \bmod m) \cdot n$;
- $T := \mathsf{BLAKE2b}\text{-}(n \cdot m)\left(\mathsf{powtag}, S \,||\, \mathsf{powcount}\left(\mathsf{floor}\left(\frac{i-1}{m}\right)\right)\right)$.

Indices of bits in T are 1-based.

$\mathsf{BLAKE2b}\text{-}A(p, x)$ is deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51.

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

Note: When $\mathsf{EquihashGen}$ is evaluated for sequential indices, as in the Equihash solving process (§7.6.1 *'Equihash'* on p. 85), the number of calls to $\mathsf{BLAKE2b}$ can be reduced by a factor of $\mathsf{floor}\left(\frac{512}{n}\right)$ in the best case (which is a factor of 2 for $n = 200$).

5.4.2 Pseudo Random Functions

$\mathsf{PRF}^{\mathsf{addr}}$, $\mathsf{PRF}^{\mathsf{nf}}$, $\mathsf{PRF}^{\mathsf{pk}}$, and PRF^{ρ}, described in §4.1.2 *'Pseudo Random Functions'* on p. 18, are all instantiated using the *SHA-256 compression function* deZned in §5.4.1.1 *'SHA-256 and SHA256Compress Hash Functions'* on p. 50:

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

Security requirements:

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

Note: The Zrst four bits –i.e. the most signiZcant four bits of the Zrst byte– are used to separate distinct uses of SHA256Compress, ensuring that the functions are independent. As well as the inputs shown here, bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function; see §5.4.7.1 *'**Sprout** Note Commitments'* on p. 62.

(The speciZc bit patterns chosen here were motivated by the possibility of future extensions that might have increased $\mathrm{N}^{\mathrm{old}}$ and/or $\mathrm{N}^{\mathrm{new}}$ to 3, or added an additional bit to 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. 27 to derive the *spend authorizing key* ask and the *proof authorizing key* nsk.

It is instantiated using the BLAKE2b *hash function* deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51:

$$\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}(t) := \mathsf{BLAKE2b\text{-}512}\,\text{“}\textbf{bitzec\_ExpandSeed}\text{”}, \mathsf{LEBS2OSP}_{256}(\mathsf{sk}) \,||\, t$$

Security requirement: BLAKE2b-512 **“bitzec_ExpandSeed”**, $\mathsf{LEBS2OSP}_{256}(\mathsf{sk}) \,||\, t$ must be a PRF for output range $\mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{PRFexpand}}/8]}$ when keyed by the bits corresponding to sk, with input in the bits corresponding to t.

$\mathsf{PRF}^{\mathsf{ock}}$ is used in §4.17.1 *'Encryption (**Sapling**)'* on p. 45 to derive the *outgoing cipher key* ock used to encrypt an *output ciphertext* .

It is instantiated using the BLAKE2b *hash function* deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51:

$$\mathsf{PRF}^{\mathsf{ock}}_{\mathsf{ovk}}(\mathrm{cv}, \mathrm{cmu}, \mathrm{ephemeralKey}) := \mathsf{BLAKE2b\text{-}256}\,\text{“}\textbf{bitzec\_Derive\_ock}\text{”}, \mathsf{ockInput}$$

where $\mathsf{ockInput} = \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{ovk})}\boxed{\text{32-byte cv}}\boxed{\text{32-byte cmu}}\boxed{\text{32-byte ephemeralKey}}$.

Security requirement: BLAKE2b-512 **“bitzec_Derive_ock”**, ockInput must be a PRF for output range $\mathsf{Sym}.\mathbf{K}$ (deZned in §5.4.3 *'Authenticated One-Time Symmetric Encryption'* on p. 57) when keyed by the bits corresponding to ovk, with input in the bits corresponding to cv, cmu, and ephemeralKey.

$\mathsf{PRF}^{\mathsf{nfSapling}}$ is used to derive the *nulliber* for a **Sapling** *note*. It is instantiated using the BLAKE2s *hash function* deZned in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51:

$$\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}y}(\rho{>}) := \mathsf{BLAKE2s\text{-}256}\,\text{“}\textbf{bitzec\_nf}\text{”}, \boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{nk}{>})}\boxed{\mathsf{LEBS2OSP}_{256}\,\rho{>}} .$$

Security requirement: BLAKE2s-256 **“bitzec_nf”**, $\boxed{\mathsf{LEBS2OSP}_{256}(\mathsf{nk}{>})}\boxed{\mathsf{LEBS2OSP}_{256}\,\rho{>}}$ must be a collision-resistant PRF for output range $\mathbb{B}^{\mathbb{Y}[32]}$ when keyed by the bits corresponding to nk>, with input in the bits corresponding to ρ>. Note that nk> $\mathbb{J}^{(r)}$ is a representation of a point in the r-order subgroup of the *Jubjub curve*,

and therefore is not uniformly distributed on $\mathbb{B}^{[\ell_{\mathbb{J}}]}$. $\mathbb{J}^{(r)}$ is deZned in §5.4.8.3 *'Jubjub'* on p. 67.

5.4.3 Authenticated One-Time Symmetric Encryption

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

Let $\mathsf{Sym}.\mathsf{Encrypt}_{\mathsf{K}}(\mathsf{P})$ 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 $\perp$ indicating failure to decrypt.

Note: The “IETF" definition of $\mathsf{AEAD\_CHACHA20\_POLY1305}$ 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 $\mathsf{ChaCha20}$.

5.4.4 Key Agreement and Derivation

5.4.4.1 Sprout Key Agreement

$\mathsf{KA}^{\mathsf{Sprout}}$ is a *key agreement scheme* as specified in §4.1.4 *‘Key Agreement’* on p. 19.

It is instantiated as $\mathsf{Curve25519}$ 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 $\mathsf{Curve25519}$ public keys (i.e. $\mathbb{B}^{\mathbb{Y}[32]}$), and let $\mathsf{KA}^{\mathsf{Sprout}}.\mathsf{Private}$ be the type of $\mathsf{Curve25519}$ secret keys.

Let $\mathsf{Curve25519}(\underline{n}, \underline{q})$ be the result of point multiplication of the $\mathsf{Curve25519}$ public key represented by the byte sequence $\underline{q}$ by the $\mathsf{Curve25519}$ 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 $\mathsf{Curve25519}$ 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 $\mathsf{Curve25519}$ private key, with bits “clamped” as described in [Bernstein2006, section 3]: “clear bits 0, 1, 2 of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte.” Here the bits of a byte are numbered such that bit b has numeric weight 2^b.

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

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

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

5.4.4.2 Sprout Key Derivation

$\mathsf{KDF}^{\mathsf{Sprout}}$ is a *Key Derivation Function* as specified in §4.1.5 *‘Key Derivation’* on p. 19.

It is instantiated using $\mathsf{BLAKE2b\text{-}256}$ as follows:

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

where:

$\mathsf{kdftag} :=$

64-bit **“bitzecKDF”**	8-bit $i-1$	$[0]^{56}$

$\mathsf{kdfinput} :=$

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

.

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 *‘BLAKE2 Hash Function’* on p. 51.

5.4.4.3 Sapling Key Agreement

$\mathsf{KA}^{\mathsf{Sapling}}$ is a *key agreement scheme* as specified in §4.1.4 *‘Key Agreement’* on p. 19.

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 §5.4.8.3 *‘Jubjub’* on p. 67.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Public} := \mathbb{J}$.

Define $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder} := \mathbb{J}^{(r)*}$.

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 $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{Agree}(\mathsf{sk}, P) := [h_{\mathbb{J}} \cdot \mathsf{sk}]\, P$.

5.4.4.4 Sapling Key Derivation

$\mathsf{KDF}^{\mathsf{Sapling}}$ is a *Key Derivation Function* as specified in §4.1.5 *'Key Derivation'* on p. 19.

It is instantiated using BLAKE2b-256 as follows:

$$\mathsf{KDF}^{\mathsf{Sapling}}(\mathsf{sharedSecret}, \mathsf{epk}) := \mathsf{BLAKE2b\text{-}256}(\textbf{“bitzec\_SaplingKDF”}, \mathsf{kdfinput}).$$

where:

$$\mathsf{kdfinput} := \begin{array}{|c|c|}\hline \mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{sharedSecret})\big) & \mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})\big) \\ \hline \end{array}.$$

$\mathsf{BLAKE2b\text{-}256}(p, x)$ is defined in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51.

5.4.5 JoinSplit Signature

JoinSplitSig is a *signature scheme* as specified in §4.1.6 *'Signature'* on p. 20.

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

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

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

Ed25519 is defined as using SHA-512 internally.

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

The encoding of a signature is:

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

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

5.4.6 RedDSA and RedJubjub

RedDSA is a Schnorr-based *signature scheme*, optionally supporting key re-randomization as described in §4.1.6.1 *'Signature with Re-Randomizable Keys'* on p. 21. It also supports a Secret Key to Public Key Homomorphism as described in §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22. 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* (§5.4.8.3 *'Jubjub'* on p. 67), using the BLAKE2b-512 hash function.

The *spend authorization signature scheme* defined in §5.4.6.1 *'Spend Authorization Signature'* on p. 62 is instantiated by RedJubjub. The *binding signature scheme* BindingSig defined in §5.4.6.2 *'Binding Signature'* on p. 62 is instantiated by RedJubjub without use of key re-randomization.

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 $A_{\mathbb{G}}$, a representation function $\mathsf{repr}_{\mathbb{G}}$, and an abstraction function $\mathsf{abst}_{\mathbb{G}}$, as specified in §4.1.8 *'Represented Group'* on p. 24;
- $\mathcal{P}_{\mathbb{G}}$, a generator of $\mathbb{G}^{(r)}$;
- a bit-length $A_{\mathsf{H}} : \mathbb{N}$ such that $2^{A_{\mathsf{H}}-128} \geq r_{\mathbb{G}}$ and $A_{\mathsf{H}} \bmod 8 = 0$;
- a cryptographic *hash function* $\mathsf{H} : \mathbb{B}^{\mathbb{Y}[\mathbb{N}]} \to \mathbb{B}^{\mathbb{Y}[A_{\mathsf{H}}/8]}$.

Its associated types are defined as follows:

$\mathsf{RedDSA.Message} := \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$

$\mathsf{RedDSA.Signature} := \mathbb{B}^{\mathbb{Y}[\mathsf{ceiling}(A_{\mathbb{G}}/8) + \mathsf{ceiling}(\mathsf{bitlength}(r_{\mathbb{G}})/8)]}$

$\mathsf{RedDSA.Public} := \mathbb{G}$

$\mathsf{RedDSA.Private} := \mathbb{F}_{r_{\mathbb{G}}}$.

$\mathsf{RedDSA.Random} := \mathbb{F}_{r_{\mathbb{G}}}$.

Define $\mathsf{H}^{\sim} : \mathbb{B}^{\mathbb{Y}[\mathbb{N}]} \to \mathbb{F}_{r_{\mathbb{G}}}$ by:

$\mathsf{H}^{\sim}(B) = \mathsf{LEOS2IP}_{A_{\mathsf{H}}}\big(\mathsf{H}(B)\big) \pmod{r_{\mathbb{G}}}$

Define $\mathsf{RedDSA.GenPrivate} : () \xrightarrow{R} \mathsf{RedDSA.Private}$ as:

Return $\mathsf{sk} \xleftarrow{R} \mathbb{F}_{r_{\mathbb{G}}}$.

Define $\mathsf{RedDSA.DerivePublic} : \mathsf{RedDSA.Private} \to \mathsf{RedDSA.Public}$ by:

$\mathsf{RedDSA.DerivePublic}(\mathsf{sk}) := [\mathsf{sk}]\, \mathcal{P}_{\mathbb{G}}$.

Define $\mathsf{RedDSA.GenRandom} : () \xrightarrow{R} \mathsf{RedDSA.Random}$ as:

Choose a byte sequence T uniformly at random on $\mathbb{B}^{\mathbb{Y}[(A_{\mathsf{H}}+128)/8]}$.

Return $\mathsf{H}^{\sim}(T)$.

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 $\mathsf{RedDSA.Sign} : (\mathsf{sk} : \mathsf{RedDSA.Private}) \times (M : \mathsf{RedDSA.Message}) \xrightarrow{R} \mathsf{RedDSA.Signature}$ as:

Choose a byte sequence T uniformly at random on $\mathbb{B}^{\mathbb{Y}[(A_{\mathsf{H}}+128)/8]}$.

Let $\underline{\mathsf{vk}} = \mathsf{LEBS2OSP}_{A_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{RedDSA.DerivePublic}(\mathsf{sk}))\big)$.

Let $r = \mathsf{H}^{\sim}(T \,||\, \underline{\mathsf{vk}} \,||\, M)$.

Let $R = [r]\, \mathcal{P}_{\mathbb{G}}$.

Let $\underline{R} = \mathsf{LEBS2OSP}_{A_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(R)\big)$.

Let $S = (r + \mathsf{H}^{\sim}(\underline{R} \,||\, \underline{\mathsf{vk}} \,||\, M) \cdot \mathsf{sk}) \bmod r_{\mathbb{G}}$.

Let $\underline{S} = \mathsf{LEBS2OSP}_{\mathsf{bitlength}(r_{\mathbb{G}})}\big(\mathsf{I2LEBSP}_{\mathsf{bitlength}(r_{\mathbb{G}})}(S)\big)$.

Return $\underline{R} \,||\, \underline{S}$.

Define $\mathsf{RedDSA.Verify} \colon (\mathsf{vk} \colon \mathsf{RedDSA.Public}) \times (M \colon \mathsf{RedDSA.Message}) \times (\sigma \colon \mathsf{RedDSA.Signature}) \to \mathbb{B}$ as:

Let $\underline{R}$ be the first $\mathsf{ceiling}\big(A_{\mathbb{G}}/8\big)$ bytes of σ, and let $\underline{S}$ be the remaining $\mathsf{ceiling}\big(\mathsf{bitlength}(r_{\mathbb{G}})/8\big)$ bytes.

Let $R = \mathsf{abst}_{\mathbb{G}}\big(\mathsf{LEOS2BSP}_{A_{\mathbb{G}}}(\underline{R})\big)$, and let $S = \mathsf{LEOS2IP}_{8 \cdot \mathsf{length}(\underline{S})}(\underline{S})$.

Let $\underline{\mathsf{vk}} = \mathsf{LEBS2OSP}_{A_{\mathbb{G}}}\big(\mathsf{repr}_{\mathbb{G}}(\mathsf{vk})\big)$.

Let $c = \mathsf{H}^{\sim}(\underline{R} \,||\, \underline{\mathsf{vk}} \,||\, M)$.

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 verification algorithm *does not* check that R is a point of order at least $r_{\mathbb{G}}$. It *does* check that $\underline{R}$ is the canonical representation (as output by $\mathsf{repr}_{\mathbb{G}}$) of a point on the curve. This is different to $\mathsf{Ed25519}$ as specified in §5.4.5 *'JoinSplit Signature'* on p. 59.
- Appendix §B.1 *'*RedDSA *batch verification'* on p. 139 describes an optimization that **MAY** be used to speed up verification of batches of RedDSA signatures.

Non-normative note: The randomization used in $\mathsf{RedDSA.RandomizePrivate}$ and $\mathsf{RedDSA.RandomizePublic}$ may interact with other uses of additive properties of keys for Schnorr-based signature schemes. In the **bitzec** 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.

The two abelian groups specified in §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22 are instantiated for RedDSA as follows:

- $\mathcal{O}_{\boxplus} := 0 \pmod{r_{\mathbb{G}}}$
- $\mathsf{sk}_1 \boxplus \mathsf{sk}_2 := \mathsf{sk}_1 + \mathsf{sk}_2 \pmod{r_{\mathbb{G}}}$
- $\mathcal{O}_{\diamondplus} := \mathcal{O}_{\mathbb{G}}$
- $\mathsf{vk}_1 \diamondplus \mathsf{vk}_2 := \mathsf{vk}_1 + \mathsf{vk}_2$.

As required, $\mathsf{RedDSA.DerivePublic}$ is a group homomorphism:

$$\begin{aligned}
\mathsf{RedDSA.DerivePublic}(\mathsf{sk}_1 \boxplus \mathsf{sk}_2) &= [\mathsf{sk}_1 + \mathsf{sk}_2 \pmod{r_{\mathbb{G}}}]\,\mathcal{P}_{\mathbb{G}} \\
&= [\mathsf{sk}_1]\,\mathcal{P}_{\mathbb{G}} + [\mathsf{sk}_2]\,\mathcal{P}_{\mathbb{G}} \text{ (since } \mathcal{P}_{\mathbb{G}} \text{ has order } r_{\mathbb{G}}) \\
&= \mathsf{RedDSA.DerivePublic}(\mathsf{sk}_1) \diamondplus \mathsf{RedDSA.DerivePublic}(\mathsf{sk}_2).
\end{aligned}$$

A RedDSA public key vk can be encoded as a bit sequence $\mathsf{repr}_{\mathbb{G}}(\mathsf{vk})$ of length $A_{\mathbb{G}}$ bits (or as a corresponding byte sequence $\underline{\mathsf{vk}}$ by then applying $\mathsf{LEBS2OSP}_{A_{\mathbb{G}}}$).

The scheme RedJubjub specializes RedDSA with:

- $\mathbb{G} := \mathbb{J}$ as defined in §5.4.8.3 *'Jubjub'* on p.67;
- $A_{\mathsf{H}} := 512$;
- $\mathsf{H}(x) := \mathsf{BLAKE2b\text{-}512}(\text{“bitzec\_RedJubjubH”}, x)$ as defined in §5.4.1.2 *'BLAKE2 Hash Function'* on p. 51.

The generator $\mathcal{P}_{\mathbb{G}} \colon \mathbb{G}^{(r)}$ is left as an unspecified parameter, which is different between $\mathsf{BindingSig}$ and $\mathsf{SpendAuthSig}$.

5.4.6.1 Spend Authorization Signature

Let RedJubjub be as deZned in §5.4.6 *'RedDSA and RedJubjub'* on p. 59.

DeZne $\mathcal{G} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}\left(\textbf{"bitzec\_G\_"}, \textbf{""}\right)$.

SpendAuthSig is instantiated as RedJubjub with key re-randomization, and with generator $\mathcal{P}_{\mathbb{G}} = \mathcal{G}$.

See §4.13 *'Spend Authorization Signature'* on p. 38 for details on the use of this *signature scheme*.

Security requirement: SpendAuthSig must be a SURK-CMA secure *signature scheme with re-randomizable keys* as deZned in §4.1.6.1 *'Signature with Re-Randomizable Keys'* on p. 21.

5.4.6.2 Binding Signature

Let RedJubjub be as deZned in §5.4.6 *'RedDSA and RedJubjub'* on p. 59.

Let $\mathcal{R}$ be the randomness base deZned in §5.4.7.3 *'Homomorphic Pedersen commitments'* on p. 63.

BindingSig is instantiated as RedJubjub, without use of key re-randomization, and with generator $\mathcal{P}_{\mathbb{G}} = \mathcal{R}$.

See §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36 for details on the use of this *signature scheme*.

Security requirement: BindingSig must be a SUF-CMA secure *signature scheme with private key to public key homomorphism* as deZned in §4.1.6.2 *'Signature with Private Key to Public Key Homomorphism'* on p. 22. A signature must prove knowledge of the discrete logarithm of the public key with respect to the base $\mathcal{R}$.

5.4.7 Commitment schemes

5.4.7.1 Sprout Note Commitments

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ speciZed in §4.1.7 *'Commitment'* on p. 23 is instantiated using SHA-256 as follows:

$$\mathsf{NoteCommit}^{\mathsf{Sprout}}_{\mathsf{rcm}}(\mathsf{a_{pk}}, \mathsf{v}, \rho) := \mathsf{SHA\text{-}256}\left(\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{256-bit } \mathsf{a_{pk}} & \text{64-bit } \mathsf{v} & \text{256-bit } \rho & \text{256-bit } \mathsf{rcm} \\ \hline \end{array}\right)$$

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

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

Security requirements:

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

5.4.7.2 Windowed Pedersen commitments

§5.4.1.7 *'Pedersen Hash Function'* on p. 53 defines a *Pedersen hash* construction. We construct "*windowed*" *Pedersen commitments* by reusing that construction, and adding a randomized point on the *Jubjub curve* (see §5.4.8.3 *'Jubjub'* on p. 67):

$$\mathsf{WindowedPedersenCommit}_r(s) := \mathsf{PedersenHashToPoint}(\textbf{"bitzec\_PH"}, s) + [r]\,\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\textbf{"bitzec\_PH"}, \textbf{"r"})$$

See §A.3.5 *'Windowed Pedersen Commitment'* on p. 132 for rationale and efficient circuit implementation of this function.

The commitment scheme $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ specified in §4.1.7 *'Commitment'* on p. 23 is instantiated as follows using $\mathsf{WindowedPedersenCommit}$:

$$\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g}\star_{\mathsf{d}}, \mathsf{pk}\star_{\mathsf{d}}, \mathsf{v}) := \mathsf{WindowedPedersenCommit}_{\mathsf{rcm}}\left([1]^6 \,\|\, \mathsf{I2LEBSP}_{64}(\mathsf{v}) \,\|\, \mathsf{g}\star_{\mathsf{d}} \,\|\, \mathsf{pk}\star_{\mathsf{d}}\right)$$

$\mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$ 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. 51). 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.

5.4.7.3 Homomorphic Pedersen commitments

The windowed Pedersen commitments defined in the preceding section are highly efficient, but they do not support the homomorphic property we need when instantiating $\mathsf{ValueCommit}$.

For more details on the use of this property, see §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36 and §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 15.

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

$$\mathsf{HomomorphicPedersenCommit}_{\mathsf{rcv}}(D, \mathsf{v}) := [\mathsf{v}]\,\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \textbf{"v"}) + [\mathsf{rcv}]\,\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, \textbf{"r"})$$

$\mathsf{ValueCommit}.\mathsf{GenTrapdoor}()$ generates the uniform distribution on $\mathbb{F}_{r_\mathbb{J}}$.

See §A.3.6 *'Homomorphic Pedersen Commitment'* on p. 133 for rationale and efficient circuit implementation of this function.

Define:

$$\mathcal{V} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\textbf{"bitzec\_cv"}, \textbf{"v"})$$

$$\mathcal{R} := \mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(\textbf{"bitzec\_cv"}, \textbf{"r"}).$$

The commitment scheme ValueCommit speciZed in §4.1.7 *'Commitment'* on p. 23 is instantiated as follows using HomomorphicPedersenCommit:

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

which is equivalent to:

$$\mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v}) := [\mathsf{v}]\, \mathcal{V} + [\mathsf{rcv}]\, \mathcal{R}.$$

Security requirements:

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

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

5.4.8 Represented Groups and Pairings

5.4.8.1 BN-254

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

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

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

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

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

Let $\mathbb{G}_1^{(r)}$ be the group (of order $r_{\mathbb{G}}$) of rational points on a Barreto–Naehrig ([BN2005]) curve $E_{\mathbb{G}_1}$ over $\mathbb{F}_{q_{\mathbb{G}}}$ with equation $y^2 = x^3 + b_{\mathbb{G}}$. This curve has embedding degree 12 with respect to $r_{\mathbb{G}}$.
Let $\mathbb{G}_2^{(r)}$ be the subgroup of order $r_{\mathbb{G}}$ in the sextic twist $E_{\mathbb{G}_2}$ of $E_{\mathbb{G}_1}$ over $\mathbb{F}_{q_{\mathbb{G}}^2}$ with equation $y^2 = x^3 + \frac{b_{\mathbb{G}}}{\xi}$, where $\xi \in \mathbb{F}_{q_{\mathbb{G}}^2}$.
We represent elements of $\mathbb{F}_{q_{\mathbb{G}}^2}$ as polynomials $a_1 \cdot t + a_0 \in \mathbb{F}_{q_{\mathbb{G}}}[t]$, modulo the irreducible polynomial $t^2 + 1$; in this representation, ξ is given by $t + 9$.
Let $\mathbb{G}_T^{(r)}$ be the subgroup of $r_{\mathbb{G}}^{\text{th}}$ roots of unity in $\mathbb{F}^*_{q_{\mathbb{G}}^{12}}$, with multiplicative identity $\mathbf{1}_{\mathbb{G}}$.

Let $\hat{e}_{\mathbb{G}}$ be the optimal ate pairing (see [Vercauter2009] and [AKLGL2010, section 2]) of type $\mathbb{G}_1^{(r)} \times \mathbb{G}_2^{(r)} \to \mathbb{G}_T^{(r)}$.
For $i \in \{1 .. 2\}$, let $\mathcal{O}_{\mathbb{G}_i}$ be the point at inZnity (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 $\mathcal{P}_{\mathbb{G}_1} \in \mathbb{G}_1^{(r)*} := (1, 2)$.
Let $\mathcal{P}_{\mathbb{G}_2} \in \mathbb{G}_2^{(r)*} := (11559732032986387107991004021392285783925812861821192530917403151452391805634 \cdot t +$

$$10857046999023057135944570762232829481370756359578518086990519993285655852781,$$

$$4082367875863433681332203403145435568316851327593401208105741076214120093531 \cdot t +$$

$$8495653923123431417604973247489272438418190587263600148770280649306958101930).$$

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

DeZne $\mathsf{I2BEBSP} :: (\ell : \mathbb{N}) \times \{0 .. 2^{\ell} - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

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

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

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

.

$\circ$ 2

- 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.

5.4.8.2 BLS12-381

The *represented pairing* $\mathsf{BLS12\text{-}381}$ is defined in this section. Parameters are taken from [Bowe2017].

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

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

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

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

(q_{S} and r_{S} are prime.)

Let $\mathbb{S}_1^{(r)}$ be the subgroup of order r_{S} of the group of rational points on a Barreto–Lynn–Scott ([BLS2002]) curve E_{S_1} over $\mathbb{F}_{q_{\mathsf{S}}}$ with equation $y^2 = x^3 + b_{\mathsf{S}}$. This curve has embedding degree 12 with respect to r_{S}.

Let $\mathbb{S}_2^{(r)}$ be the subgroup of order r_{S} in the sextic twist E_{S_2} of E_{S_1} over $\mathbb{F}_{q_{\mathsf{S}}^2}$ with equation $y^2 = x^3 + 4(i + 1)$, where $i \in \mathbb{F}_{q_{\mathsf{S}}^2}$.

We represent elements of $\mathbb{F}_{q_{\mathsf{S}}^2}$ as polynomials $a_1 \cdot t + a_0 \in \mathbb{F}_{q_{\mathsf{S}}}[t]$, 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_{S}th roots of unity in $\mathbb{F}^*_{q_{\mathsf{S}}^{12}}$, with multiplicative identity $\mathbf{1}_{\mathsf{S}}$.

Let $\hat{e}_{\mathsf{S}}$ be the optimal ate pairing of type $\mathbb{S}_1^{(r)} \times \mathbb{S}_2^{(r)} \rightarrow \mathbb{S}_T^{(r)}$.

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

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

Let $\mathcal{P}_{\mathsf{S}_2} \in \mathbb{S}_2^{(r)*} := (11559732032986387107991004021392285783925812861821192530917403151452391805634 \cdot t +$

$10857046999023057135944570762232829481370756359578518086990519993285655852781,$

$4082367875863433681332203403145435568316851327593401208105741076214120093531 \cdot t +$

$8495653923123431417604973247489272438418190587263600148770280649306958101930)$.

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

Define $\mathsf{I2BEBSP} \colon (\ell \colon \mathbb{N}) \times \{0\,..\,2^\ell - 1\} \rightarrow \mathbb{B}^{[\ell]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

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

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

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

.

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

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

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

.

Non-normative notes:

- Only the $r_{\mathbb{S}}$-order subgroups $\mathbb{S}^{(r)}_{1,2,T}$ are used in the protocol, not their containing groups $\mathbb{S}_{1,2,T}$. Points in $\mathbb{S}^{(r)*}_{1,2}$ are *always* checked to be of order $r_{\mathbb{S}}$ when decoding from external representation. (Elements of $\mathbb{S}^{(r)}_{T}$ are never represented externally.) The (r) superscripts on $\mathbb{S}^{(r)}_{1,2,T}$ 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 $\mathbb{S}^{(r)*}_{i}$, by checking that $r_{\mathbb{S}} \cdot P = \mathcal{O}_{\mathbb{S}_i}$.
- The encodings for $\mathbb{S}^{(r)*}_{1,2}$ are specific to **bitzec**.
- Algorithms for decompressing points from the encodings of $\mathbb{S}^{(r)*}_{1,2}$ are defined analogously to those for $\mathbb{G}^{(r)*}_{1,2}$ in §5.4.8.1 *'BN-254'* on p. 64, taking into account that the SORT compressed form (not the LSB compressed form) is used for $\mathbb{S}^{(r)*}_{1}$.

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

5.4.8.3 Jubjub

Sapling uses an elliptic curve designed to be efficiently implementable in *zk-SNARK circuits*, called "Jubjub" [Carroll1876]. The *represented group* Jubjub of points on this curve is defined in this section.

Let $q_{\mathbb{J}} := r_{\mathbb{S}}$, as defined in §5.4.8.2 *'BLS12-381'* on p. 66.

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

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

Define $\mathsf{I2LEBSP} :: (\ell : \mathbb{N}) \times \{0 \,..\, 2^{\ell} - 1\} \to \mathbb{B}^{[\ell]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

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

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

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 $\mathbb{J}^{(r)}_{\star} := \{\mathsf{repr}_{\mathbb{J}}(P) : \mathbb{B}^{[\ell_{\mathbb{J}}]} \mid P \in \mathbb{J}^{(r)}\}$.

Non-normative notes:

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

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

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

Note that algorithms elsewhere in this specification that use Jubjub may impose other conditions on points, for example that they have order at least $r_\mathbb{J}$.

5.4.8.4 Hash Extractor for Jubjub

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

Define $\mathsf{Extract}_{\mathbb{J}^{(r)}} : \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$ by

$$\mathsf{Extract}_{\mathbb{J}^{(r)}}(P) := \mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}(\mathcal{U}(P)).$$

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. *Let $P = (u, v) \in \mathbb{J}^{(r)}$. Then $(u, -v) \notin \mathbb{J}^{(r)}$.*

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

Theorem 5.4.3. *$\mathcal{U}$ is injective on $\mathbb{J}^{(r)}$.*

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

Since $\mathsf{I2LEBSP}_{\ell_{\mathsf{MerkleSapling}}}$ is injective, it follows that $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ is injective on $\mathbb{J}^{(r)}$.

5.4.8.5 Group Hash into Jubjub

Let GroupHash.Input := $\mathbb{B}^{\mathbb{Y}[8]} \times \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$, and let GroupHash.URSType := $\mathbb{B}^{\mathbb{Y}[64]}$.

(The input element with type $\mathbb{B}^{\mathbb{Y}[8]}$ 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 deZned in §5.9 *‘Randomness Beacon’* on p. 76.

Let BLAKE2s-256 be as deZned in §5.4.1.2 *‘BLAKE2 Hash Function’* on p. 51.

Let LEOS2IP be as deZned in §5.2 *‘Integers, Bit Sequences, and Endianness’* on p. 48.

Let $\mathbb{J}^{(r)}$, $\mathbb{J}^{(r)*}$, and $\mathsf{abst}_{\mathbb{J}}$ be as deZned in §5.4.8.3 *‘Jubjub’* on p. 67.

Let $D \circ \mathbb{B}^{\mathbb{Y}[8]}$ be an 8-byte domain separator, and let $M \circ \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}$ be the hash input.
The hash $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(D, M) \circ \mathbb{J}^{(r)*}$ is calculated as follows:

```
let $\underline{H}$ = BLAKE2s-256($D$, URS || $M$)
let $P$ = $\mathsf{abst}_{\mathbb{J}}$ 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 BLAKE2s-256 chaining variable after processing URS may be precomputed.
- The use of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$ for DiversifyHash 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 sufZcient to employ a simpler random oracle instantiated by BLAKE2s-256 in the security analysis.

  $\underline{H} \circ \mathbb{B}^{\mathbb{Y}[32]} \mapsto_{\c{c}\bot} \mathsf{abst}_{\mathbb{J}}(\mathsf{LEOS2BSP}_{256}(\underline{H})) \circ \mathbb{J}$ is injective, and both it and its inverse are efZciently computable.

  $[h_{\mathbb{J}}]\, P \circ \mathbb{J}^{(r)*}$ is exactly $h_{\mathbb{J}}$-to-1, and both it and its inverse relation are efZciently computable.

  $P \circ \mathbb{J} \mapsto_{\c{c}\mathcal{O}_{\mathbb{J}}} [h_{\mathbb{J}}]\, P \circ \mathbb{J}^{(r)*}$ is exactly $h_{\mathbb{J}}$-to-1, and both it and its inverse relation are efZciently computable.

  It follows that when $(D \circ \mathbb{B}^{\mathbb{Y}[8]}, M \circ \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}) \mapsto \mathsf{BLAKE2s\text{-}256}(D, \mathsf{URS} \,\|\, M) \circ \mathbb{B}^{\mathbb{Y}[32]}$ is modelled as a random oracle, $(D \circ \mathbb{B}^{\mathbb{Y}[8]}, M \circ \mathbb{B}^{\mathbb{Y}[\mathbb{N}]}) \mapsto_{\c{c}\bot} \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(D, M) \circ \mathbb{J}^{(r)*}$ also acts as a random oracle.

DeZne $\mathsf{first} \circ (\mathbb{B}^{\mathbb{Y}} \to T \cup \{\bot\}) \to T \cup \{\bot\}$ so that $\mathsf{first}(f) = f(i)$ where i is the least integer in $\mathbb{B}^{\mathbb{Y}}$ such that $f(i)$ Ç $\bot$, or $\bot$ if no such i exists.

DeZne $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}(D, M)$: $= \mathsf{first}(i \circ \mathbb{B}^{\mathbb{Y}} \mapsto \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(D, M \,\|\, [i]) \circ \mathbb{J}^{(r)*} \cup \{\bot\})$.

Note: For random input, $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ returns $\bot$ with probability approximately 2^{-256}. In the **bitzec** pro- tocol, most uses of $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$ are for constants and do not return ; the only use that could potentially return is in the computation of a *default diversibed payment address* in §4.2.2 ***‘Sapling** Key Components’* on p. 27.

5.4.9 Zero-Knowledge Proving Systems

5.4.9.1 PHGR13

Before **Sapling** activation, **bitzec** uses *zk-SNARKs* generated by a fork of *libsnark* [bitzec-libsnark] with the PHGR13 *proving system* described in [BCTV2015], which is a reZnement of the systems in [PHGR2013] and [BCGTV2013].
A PHGR13 proof consists of $(\pi_A \circ \mathbb{G}^{(r)*}_1, \pi'_A \circ \mathbb{G}^{(r)*}_1, \pi_B \circ \mathbb{G}^{(r)*}_2, \pi'_B \circ \mathbb{G}^{(r)*}_1, \pi_C \circ \mathbb{G}^{(r)*}_1, \pi'_C \circ \mathbb{G}^{(r)*}_1, \pi_K \circ \mathbb{G}^{(r)*}_1, \pi_H \circ \mathbb{G}^{(r)*}_1)$
It is computed as described in [BCTV2015, Appendix B], using the pairing parameters speciZed in §5.4.8.1 *‘BN-254’* on p. 64.

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* [BCTV2015, section 2.3] [WCBTV2015], are not specified in this document. In practice it will be necessary to use the specific proving and verification keys given in §5.7 *'**Sprout** zk-SNARK Parameters'* on p. 76 that were generated for the **bitzec** production *block chain*, together with a *proving system* implementation that is interoperable with the **bitzec** fork of *libsnark*, to ensure compatibility.

Encoding of PHGR13 Proofs A PHGR13 proof is encoded by concatenating the encodings of its elements; for the BN-254 pairing this is:

264-bit π_A	264-bit π^{r}_A	520-bit π_B	264-bit π^{r}_B	264-bit π_C	264-bit π^{r}_C	264-bit π_K	264-bit π_H

The resulting proof size is 296 bytes.

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

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

5.4.9.2 Groth16

After **Sapling** activation, **bitzec** uses *zk-SNARKs* with the *proving system* described in [Groth2016]. These are used in *transaction version* 4 and later (§7.1 *'Encoding of Transactions'* on p. 78) for proofs 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 consists of $(\pi_A \circ \mathbb{S}^{(r)*}_1, \pi_B \circ \mathbb{S}^{(r)*}_2, \pi_C \circ \mathbb{S}^{(r)*}_1)$. It is computed as described in [Groth2016, section 3.2], using the pairing parameters specified in §5.4.8.2 *'BLS12-381'* on p. 66. 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. 119. 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 verification keys generated for the **bitzec** production *block chain* (see §5.8 *'**Sapling** zk-SNARK Parameters'* on p. 76), and a *proving system* implementation that is interoperable with the *bellman* library used by **bitzec**, 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 π_A	768-bit π_B	384-bit π_C

The resulting proof size is 192 bytes.

In addition to the steps to verify a proof given in [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 π_B) two integers in that range;
- the encoding represents a point in $\mathbb{S}_1^{(r)*}$ or (in the case of π_B) $\mathbb{S}_2^{(r)*}$, including checking that it is of order $r_{\mathbb{S}}$ in each case.

5.5 Encodings of Note Plaintexts and Memo Fields

As explained in §3.2.1 *'Note Plaintexts and Memo Fields'* on p. 13, transmitted *notes* are stored on the *block chain* in encrypted form.

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

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

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

$$(\mathsf{d} ⦂ \mathbb{B}^{[\ell_{\mathsf{d}}]}, \mathsf{v} ⦂ \{0 .. 2^{\ell_{\mathsf{value}}}-1\}, \mathsf{rcm} ⦂ \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Output}, \mathsf{memo} ⦂ \mathbb{B}^{\mathbb{Y}[512]})$$

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

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

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

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

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

Other fields are as defined in §3.2 *'Notes'* on p. 12.

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

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

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

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

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

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

5.6 Encodings of Addresses and Keys

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

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

For **Sapling**-speciZc key and address formats, Bech32 [BIP-173] is used instead of Base58Check. All conformance requirements of BIP 173 apply except for the limit of 90 characters on an encoded Bech32 string (which does not hold for **Sapling** viewing keys, for example), and requirements speciZc to Bitcoin’s Segwit addresses.

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

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

5.6.1 Transparent Addresses

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

The raw encoding of a P2SH address consists of:

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

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

The raw encoding of a P2PKH address consists of:

8-bit 0x1C	8-bit 0xB8	160-bit public key hash

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

Notes:

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

5.6.2 Transparent Private Keys

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

5.6.3 Sprout Shielded Payment Addresses

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{pk_{enc}}$ is a $\mathsf{KA^{Sprout}.Public}$ key (see §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58), for use with the encryption scheme defined in §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43. These components are derived from a *spending key* as described in §4.2.1 *'**Sprout** Key Components'* on p. 27.

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

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

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

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

5.6.4 Sapling Shielded Payment Addresses

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

$\mathsf{pk_d}$ is an encoding of a $\mathsf{KA^{Sapling}}$ public key of type $\mathsf{KA^{Sapling}.PublicPrimeOrder}$ (see §5.4.4.3 *'**Sapling** Key Agreement'* on p. 58), for use with the encryption scheme defined in §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44. d is a sequence of 11 bytes. These components are derived as described in §4.2.2 *'**Sapling** Key Components'* on p. 27.

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

$\mathsf{LEBS2OSP_{88}(d)}$	$\mathsf{LEBS2OSP_{256}}\big(\underline{\mathsf{repr}}_{\mathbb{J}}(\mathsf{pk_d})\big)$

- 11 bytes specifying d.
- 32 bytes specifying the compressed Edwards encoding of $\mathsf{pk_d}$ (see §5.4.8.3 *'Jubjub'* on p. 67).

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

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

5.6.5 Sprout Incoming Viewing Keys

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

$\mathsf{a_{pk}}$ is a *SHA-256 compression* output. $\mathsf{sk_{enc}}$ is a $\mathsf{KA^{Sprout}.Private}$ key (see §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58), for use with the encryption scheme defined in §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43. These components are derived from a *spending key* as described in §4.2.1 *'**Sprout** Key Components'* on p. 27.

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

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

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

$\mathsf{sk_{enc}}$ **MUST** be "clamped" using $\mathsf{KA^{Sprout}.FormatPrivate}$ as specified in §4.2.1 *'**Sprout** Key Components'* on p. 27. That is, a decoded *incoming viewing key* **MUST** be considered invalid if $\mathsf{sk_{enc}} \neq \mathsf{KA^{Sprout}.FormatPrivate}(\mathsf{sk_{enc}})$.

$\mathsf{KA^{Sprout}.FormatPrivate}$ is defined in §5.4.4.1 *'**Sprout** Key Agreement'* on p. 58.

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

5.6.6 Sapling Incoming Viewing Keys

Let ℓ_{ivk} be as defined in §5.3 *'Constants'* on p. 49.

A **Sapling** *incoming viewing key* consists of $\mathsf{ivk} ⦂ \{0 .. 2^{\ell_{\mathsf{ivk}}} - 1\}$.

ivk is a $\mathsf{KA^{Sapling}.Private}$ key (restricted to ℓ_{ivk} bits), derived as described in §4.2.2 *'**Sapling** Key Components'* on p. 27. It is used with the encryption scheme defined in §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44.

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

256-bit ivk

- 32 bytes (little-endian) specifying ivk, padded with zeros in the most significant bits.

ivk **MUST** be in the range $\{0 .. 2^{\ell_{\mathsf{ivk}}} - 1\}$ as specified in §4.2.2 *'**Sapling** Key Components'* on p. 27. That is, a decoded *incoming viewing key* **MUST** be considered invalid if ivk is not in this range.

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

5.6.7 Sapling Full Viewing Keys

A **Sapling** *full viewing key* consists of $\mathsf{ak} ⦂ \mathbb{J}^{(r)*}$, $\mathsf{nk} ⦂ \mathbb{J}^{(r)}$, and $\mathsf{ovk} ⦂ \mathbb{B}^{\mathbb{Y}[\ell_{\mathsf{ovk}}/8]}$.

ak and nk are points on the *Jubjub curve* (see §5.4.8.3 *'Jubjub'* on p. 67). They are derived as described in §4.2.2 *'**Sapling** Key Components'* on p. 27.

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

$\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})\big)$	$\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})\big)$	32-byte ovk

- 32 bytes specifying the compressed Edwards encoding of ak (see §5.4.8.3 *'Jubjub'* on p. 67).
- 32 bytes specifying the compressed Edwards encoding of nk.
- 32 bytes specifying the *outgoing viewing key* ovk.

When decoding this representation, the key is not valid if $\mathsf{abst}_{\mathbb{J}}$ returns ⊥ for either ak or nk, or if ak g $\mathbb{J}^{(r)*}$, or if nk g $\mathbb{J}^{(r)}$.

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

5.6.8 Sprout Spending Keys

A **Sprout** *spending key* consists of $\mathsf{a_{sk}}$, which is a sequence of 252 bits (see §4.2.1 *'**Sprout** Key Components'* on p. 27).

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 **bitzec** *spending key* on the production network. (Addresses on the test network use [0xAC, 0x08] instead.)
- 32 bytes: 4 zero padding bits and 252 bitsspecifying $\mathsf{a_{sk}}$.

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

Notes:

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

5.6.9 Sapling Spending Keys

A **Sapling** *spending key* consists of sk ˙ $\mathbb{B}^{[A_{\mathsf{sk}}]}$ (see §4.2.2 *'**Sapling** Key Components'* on p. 27).

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

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

- 32 bytes specifying sk.

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

5.7 Sprout zk-SNARK Parameters

For the **bitzec** production *block chain* and testnet, 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 [BGG2016]. They are used only before **Sapling** activation.

5.8 Sapling 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 [BGM2018].

5.9 Randomness Beacon

Let URS := **“096b36a5804bfacef1691e173c366a47ff5ba84a44f26ddd7e8d9f79d5b42df0”**.

This value is used in the definition of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$ in §5.4.8.5 *‘Group Hash into Jubjub’* on p. 69, and in the multi-party computation to obtain the **Sapling** parameters given in §5.8 *‘**Sapling** zk-SNARK Parameters’* on p. 76.

It is derived as described in [Bowe2018]:

- Take the hash of the **Bitcoin** *block* at height 514200 in RPC byte order [Bitcoin-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.

6Network Upgrades

bitzec launched with a protocol revision that we call **Sprout**. A Zrst network upgrade, called **Overwinter**, activated on the production **bitzec** network on 26 June 2018 at block height 347500 [Swihart2018]. At the time of writing, a further upgrade called **Sapling** is planned to activate on the production network in late October 2018. This section summarizes the strategy for upgrading from **Sprout** to **Overwinter**, and then to **Sapling** and future upgrades.

The upgrade mechanism is described in [ZIP-200].The speciZcations of the **Overwinter** upgrade are described in this document, [ZIP-201], [ZIP-202], [ZIP-203], and [ZIP-143].The speciZcations of the **Sapling** upgrade are described in this document and [ZIP-243].

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

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

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

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

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

7Consensus Changes from Bitcoin

7.1 Encoding of Transactions

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

Version	Bytes	Name	Data Type	Description
≥ 1	4	header	uint32	Contains: · fOverwintered flag (bit 31) · version (bits 30 .. 0) – *transaction version*.
≥ 3	4	nVersionGroupId	uint32	Version group ID (nonzero).
≥ 1	*Varies*	tx_in_count	compactSize uint	Number of *transparent* inputs in this *transaction*.
≥ 1	*Varies*	tx_in	tx_in	*Transparent* inputs, encoded as in **Bitcoin**.
≥ 1	*Varies*	tx_out_count	compactSize uint	Number of *transparent* outputs in this *transaction*.
≥ 1	*Varies*	tx_out	tx_out	*Transparent* outputs, encoded as in **Bitcoin**.
≥ 1	4	lock_time	uint32	A Unix epoch time (UTC) or *block height* , encoded as in **Bitcoin**.
≥ 3	4	nExpiryHeight	uint32	A *block height* in the range $\{1 .. 499999999\}$ after which the *transaction* will expire, or 0 to disable expiry ([ZIP-203]).
≥ 4	8	valueBalance	int64	The net value of **Sapling** *Spend transfers* minus *Output transfers*.
≥ 4	*Varies*	nShieldedSpend	compactSize uint	The number of *Spend descriptions* in vShieldedSpend.
≥ 4	384· nShieldedSpend	vShieldedSpend	SpendDescription [nShieldedSpend]	A sequence of *Spend descriptions*, each encoded as in §7.3 *'Encoding of Spend Descriptions'* on p. 82.
≥ 4	*Varies*	nShieldedOutput	compactSize uint	The number of *Output descriptions* in vShieldedOutput.
≥ 4	948· nShieldedOutput	vShieldedOutput	OutputDescription [nShieldedOutput]	A sequence of *Output descriptions*, each encoded as in §7.4 *'Encoding of Output Descriptions'* on p. 82.
≥ 2	*Varies*	nJoinSplit	compactSize uint	The number of *JoinSplit descriptions* in vJoinSplit.
2 .. 3	1802· nJoinSplit	vJoinSplit	JSDescriptionPHGR13 [nJoinSplit]	A sequence of *JoinSplit descriptions* using PHGR13 proofs, each encoded as in §7.2 *'Encoding of JoinSplit Descriptions'* on p. 81.
≥ 4	1698· nJoinSplit	vJoinSplit	JSDescriptionGroth16 [nJoinSplit]	A sequence of *JoinSplit descriptions* using Groth16 proofs, each encoded as in §7.2 *'Encoding of JoinSplit Descriptions'* on p. 81.
≥ 2 †	32	joinSplitPubKey	char[32]	An encoding of a JoinSplitSig public verification key.
≥ 2 †	64	joinSplitSig	char[64]	A signature on a prefix of the *transaction* encoding, to be verified using joinSplitPubKey.
≥ 4 ‡	64	bindingSig	char[64]	A signature on the *SIGHASH transaction hash*, to be verified as specified in §5.4.6.2 *'Binding Signature'* on p. 62.

† The joinSplitPubKey and joinSplitSig fields are present if and only if version ≥ 2 and nJoinSplit > 0. The encoding of joinSplitPubKey and the data to be signed are specified in §4.10 *'Non-malleability (**Sprout**)'* on p. 35.

‡ The bindingSig field is present if and only if version ≥ 4 and nShieldedSpend $+$ nShieldedOutput > 0.

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.
- [Pre-**Sapling**] The encoded size of the *transaction* **MUST** be less than or equal to 100000 bytes.
- [Pre-**Sapling**] If version $= 1$ or nJoinSplit $= 0$, then tx_in_count **MUST NOT** be 0.
- [**Sapling** onward] At least one of tx_in_count, nShieldedSpend, and nJoinSplit **MUST** be nonzero.
- A *transaction* with one or more inputs from *coinbase transactions* **MUST** have no *transparent* outputs (i.e. tx_out_count **MUST** be 0). Note that inputs from *coinbase transactions* include *Founders' Reward* outputs.
- If version ≥ 2 and nJoinSplit > 0, then:
  - joinSplitPubKey **MUST** represent a valid $\mathsf{Ed25519}$ public key encoding (§5.4.5 *'JoinSplit Signature'* on p. 59).
  - joinSplitSig **MUST** represent a valid signature under joinSplitPubKey of $\mathsf{dataToBeSigned}$, as defined in §4.10 *'Non-malleability (**Sprout**)'* on p. 35.
- [**Sapling** onward] If version ≥ 4 and nShieldedSpend $+$ nShieldedOutput > 0, then:
  - let bvk and $\mathsf{SigHash}$ be as defined in §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36;
  - bindingSig **MUST** represent a valid signature under the *transaction binding verification key* bvk of $\mathsf{SigHash}$ — i.e. $\mathsf{BindingSig.Verify}_{\mathsf{bvk}}(\mathsf{SigHash}, \text{bindingSig}) = 1$.
- [**Sapling** onward] If version ≥ 4 and nShieldedSpend $+$ nShieldedOutput $= 0$, then valueBalance **MUST** be 0.
- A *coinbase transaction* **MUST NOT** have any *JoinSplit descriptions*, *Spend descriptions*, or *Output descriptions*.
- A *transaction* **MUST NOT** spend an output of a *coinbase transaction* (necessarily a *transparent* output) from a *block* less than 100 *blocks* prior to the spend. Note that outputs of *coinbase transactions* include *Founders' Reward* outputs.
- [**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 $\{-\mathsf{MAX\_MONEY} .. \mathsf{MAX\_MONEY}\}$.
- TODO: Other rules inherited from **Bitcoin**.

In addition, consensus rules associated with each *JoinSplit description* (§7.2 *'Encoding of JoinSplit Descriptions'* on p. 81), each *Spend description* (§7.3 *'Encoding of Spend Descriptions'* on p. 82), and each *Output description* (§7.4 *'Encoding of Output Descriptions'* on p. 82) **MUST** be followed.

Notes:

- Previous versions of this specification defined what is now the header field as a signed int32 field which was required to be positive. The consensus rule that the fOverwintered flag **MUST NOT** be set before **Overwinter** has activated, has the same effect.
- The semantics of *transactions* with *transaction version number* not equal to 1, 2, 3, or 4 is not currently defined. Miners **MUST NOT** create *blocks* before the **Overwinter** *activation block height* containing *transactions* with version other than 1 or 2.

- The exclusion of *transactions* with *transaction version number* *greater than* 2 is not a consensus rule before **Overwinter** activation. Such *transactions* may exist in the *block chain* and **MUST** be treated identically to version 2 *transactions*.
- [**Overwinter** onward] Once **Overwinter** has activated, limits on the maximum *transaction version number* are consensus rules.
- Note that a future upgrade might use *any* *transaction version number* 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 deZned in [ZIP-200], and in that case the fOverwintered and version Zelds alone may be insufZcient 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 speciZed in [BIP-68]. **bitzec** was forked from **Bitcoin** v0.11.2 and does not currently support BIP68.

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

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

[Pre-**Overwinter**] Software that creates *transactions* **SHOULD** use version 1 for *transactions* with no *JoinSplit descriptions*.

7.2 Encoding of JoinSplit Descriptions

An abstract *JoinSplit description*, as described in §3.5 *'JoinSplit Transfers and Descriptions'* on p. 15, is encoded in a *transaction* as an instance of a JoinSplitDescription type as follows:

Bytes	Name	Data Type	Description
8	vpub_old	uint64	A value $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$ that the *JoinSplit transfer* removes from the *transparent value pool* .
8	vpub_new	uint64	A value $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$ that the *JoinSplit transfer* inserts into the *transparent value pool* .
32	anchor	char[32]	A *root* rt of the **Sprout** *note commitment tree* at some *block height* in the past, or the *root* produced by a previous *JoinSplit transfer* in this *transaction*.
64	nullifiers	char[32][$\mathrm{N}^{\mathsf{old}}$]	A sequence of *nullibers* of the input *notes* $\mathsf{nf}^{\mathsf{old}}_{1..\mathrm{N}^{\mathsf{old}}}$.
64	commitments	char[32][$\mathrm{N}^{\mathsf{new}}$]	A sequence of *note commitments* for the output *notes* $\mathsf{cm}^{\mathsf{new}}_{1..\mathrm{N}^{\mathsf{new}}}$.
32	ephemeralKey	char[32]	A Curve25519 public key epk.
32	randomSeed	char[32]	A 256-bit seed that must be chosen independently at random for each *JoinSplit description*.
64	vmacs	char[32][$\mathrm{N}^{\mathsf{old}}$]	A sequence of message authentication tags $\mathsf{h}_{1..\mathrm{N}^{\mathsf{old}}}$ binding $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the *JoinSplit description*, computed as described in §4.10 *'Non-malleability (**Sprout**)'* on p. 35.
296 †	zkproof	char[296]	An encoding of the *zero-knowledge proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see §5.4.9.1 *'PHGR13'* on p. 69).
192 ‡	zkproof	char[192]	An encoding of the *zero-knowledge proof* $\pi_{\mathsf{ZKJoinSplit}}$ (see §5.4.9.2 *'Groth16'* on p. 70).
1202	encCiphertexts	char[601][$\mathrm{N}^{\mathsf{new}}$]	A sequence of ciphertext components for the encrypted output *notes*, $\mathsf{C}^{\mathsf{enc}}_{1..\mathrm{N}^{\mathsf{new}}}$.

† PHGR13 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 ≥ 4, i.e. after **Sapling** activation.

The ephemeralKey and encCiphertexts Zelds together form the *transmitted notes ciphertext* , which is computed as described in §4.16 *'In-band secret distribution (**Sprout**)'* on p. 43.

Consensus rules applying to a *JoinSplit description* are given in §4.3 *'JoinSplit Descriptions'* on p. 29.

7.3 Encoding of Spend Descriptions

Let $\mathsf{LEBS2OSP}$ be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

Let $\mathsf{repr}_{\mathbb{J}}$ and $q_{\mathbb{J}}$ be as defined in §5.4.8.3 *'Jubjub'* on p. 67.

An abstract *Spend description*, as described in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 15, is encoded in a *transaction* as an instance of a SpendDescription type as follows:

Bytes	Name	Data Type	Description
32	cv	char[32]	A *value commitment* to the value of the input *note*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{cv})\big)$.
32	anchor	char[32]	A *root* of the **Sapling** *note commitment tree* at some *block height* in the past, $\mathsf{LEBS2OSP}_{256}(\mathsf{rt})$.
32	nullifier	char[32]	The *nullifier* of the input *note*, $\mathsf{LEBS2OSP}_{256}(\mathsf{nf})$.
32	rk	char[32]	The randomized public key for spendAuthSig, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{rk})\big)$.
192	zkproof	char[192]	An encoding of the *zero-knowledge proof* π_{ZKSpend} (see §5.4.9.2 *'Groth16'* on p. 70).
64	spendAuthSig	char[64]	A signature authorizing this spend.

Consensus rule: $\mathsf{LEOS2IP}_{256}(\text{anchor})$ **MUST** be less than $q_{\mathbb{J}}$.

Other consensus rules applying to a *Spend description* are given in §4.4 *'Spend Descriptions'* on p. 30.

7.4 Encoding of Output Descriptions

Let $\mathsf{LEBS2OSP}$ be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

Let $\mathsf{repr}_{\mathbb{J}}$ and $q_{\mathbb{J}}$ be as in §5.4.8.3 *'Jubjub'* on p. 67, and $\mathsf{Extract}_{\mathbb{J}^{(r)}}$ as in §5.4.8.5 *'Group Hash into Jubjub'* on p. 69.

An abstract *Output description*, described in §3.6 *'Spend Transfers, Output Transfers, and their Descriptions'* on p. 15, is encoded in a *transaction* as an instance of an OutputDescription type as follows:

Bytes	Name	Data Type	Description
32	cv	char[32]	A *value commitment* to the value of the output *note*, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{cv})\big)$.
32	cmu	char[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	char[32]	An encoding of an ephemeral Jubjub public key, $\mathsf{LEBS2OSP}_{256}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{epk})\big)$.
580	encCiphertext	char[580]	A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{enc}}$.
80	outCiphertext	char[80]	A ciphertext component for the encrypted output *note*, $\mathsf{C}^{\mathsf{out}}$.
192	zkproof	char[192]	An encoding of the *zero-knowledge proof* π_{ZKOutput} (see §5.4.9.2 *'Groth16'* on p. 70).

The ephemeralKey, encCiphertext, and outCiphertext fields together form the *transmitted note ciphertext*, which is computed as described in §4.17 *'In-band secret distribution (**Sapling**)'* on p. 44.

Consensus rule: $\mathsf{LEOS2IP}_{256}(\mathsf{cmu})$ **MUST** be less than $q_{\mathbb{J}}$.

Other consensus rules applying to an *Output description* are given in §4.5 *'Output Descriptions'* on p. 31.

7.5 Block Header

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

Bytes	Name	Data Type	Description
4	nVersion	int32	The *block version number* indicates which set of *block* validation rules to follow. The current and only deZned *block version number* for **bitzec** is 4.
32	hashPrevBlock	char[32]	A *SHA-256d* hash in internal byte order of the previous *block* 's *header* . This ensures no previous *block* can be changed without also changing this *block*'s *header*.
32	hashMerkleRoot	char[32]	A *SHA-256d* hash in internal byte order. The merkle root is derived from the hashes of all *transactions* included in this *block* , ensuring that none of those *transactions* can be modiZed without modifying the *header* .
32	hashReserved / hashFinalSaplingRoot	char[32]	[Pre-**Sapling**] A reserved Zeld which should be ignored.[**Sapling** onward] The *root* $\mathsf{LEBS2OSP}_{256}(\mathsf{rt})$ of the **Sapling** *note commitment tree* corresponding to the Znal **Sapling** *treestate* of this *block* .
4	nTime	uint32	The *block time* is a Unix epoch time (UTC) when the miner started hashing the *header* (according to the miner).
4	nBits	uint32	An encoded version of the *target threshold* this *block* 's *header* hash must be less than or equal to, in the same nBits format used by **Bitcoin**. [Bitcoin-nBits]
32	nNonce	char[32]	An arbitrary Zeld that miners can change to modify the *header* hash in order to produce a hash less than or equal to the *target threshold* .
3	solutionSize	compactSize uint	The size of an Equihash solution in bytes (always 1344).
1344	solution	char[1344]	The Equihash solution.

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

Let $\mathsf{ThresholdBits}$ be as deZned in §7.6.3 *'Difficulty adjustment'* on p. 86, and let $\mathsf{PoWMedianBlockSpan}$ be the constant deZned in §5.3 *'Constants'* on p.49.

Consensus rules:

- The *block version number* **MUST** be greater than or equal to 4.
- For a *block* at *block height* height, nBits **MUST** be equal to ThresholdBits(height).
- The *block* **MUST** pass the difficulty filter defined in §7.6.2 *'Difficulty filter'* on p. 86.
- solution **MUST** represent a valid Equihash solution as defined in §7.6.1 *'Equihash'* on p. 85.
- nTime **MUST** be strictly greater than the median time of the previous PoWMedianBlockSpan *blocks*.
- The size of a *block* **MUST** be less than or equal to 2000000 bytes.
- [**Sapling** onward] hashFinalSaplingRoot **MUST** be $\mathsf{LEBS2OSP}_{256}(\mathsf{rt})$ where rt is the *root* of the **Sapling** *note commitment tree* for the final **Sapling** *treestate* of this *block*.
- TODO: Other rules inherited from **Bitcoin**.

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

Notes:

- The semantics of blocks with *block version number* not equal to 4 is not currently defined. Miners **MUST NOT** create such *blocks*, and **SHOULD NOT** mine other blocks that chain to them.
- The exclusion of *blocks* with *block version number* *greater than* 4 is not a consensus rule; such *blocks* may exist in the *block chain* and **MUST** be treated identically to version 4 *blocks* by *full validators*. Note that a future upgrade might use *block version number* either greater than or less than 4. It is likely that such an upgrade will change the *block* header and/or *transaction* format, and software that parses *blocks* **SHOULD** take this into account.
- The nVersion field is a signed integer. (It was specified as unsigned in a previous version of this specification.) A future upgrade might use negative values for this field, or otherwise change its interpretation.
- There is no relation between the values of the version field of a *transaction*, and the nVersion field of a *block header*.
- Like other serialized fields of type compactSize uint, the solutionSize field **MUST** be encoded with the minimum number of bytes (3 in this case), and other encodings **MUST** be rejected. This is necessary to avoid a potential attack in which a miner could test several distinct encodings of each Equihash solution against the difficulty filter, rather than only the single intended encoding.
- As in **Bitcoin**, the nTime field **MUST** represent a time *strictly greater than* the median of the timestamps of the past PoWMedianBlockSpan *blocks*. The Bitcoin Developer Reference [Bitcoin-Block] was previously in error on this point, but has now been corrected.
- There are no changes to the *block version number* or format for **Overwinter**.
- Although the *block version number* does not change for **Sapling**, the previously reserved (and ignored) field hashReserved has been repurposed for hashFinalSaplingRoot. There are no other format changes.

The changes relative to **Bitcoin** version 4 blocks as described in [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 char[32].
- The maximum *block* size has been doubled to 2000000 bytes.

7.6 Proof of Work

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

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

- The solution field encodes a *valid Equihash solution* according to §7.6.1 *'Equihash'* on p. 85.
- The *block header* satisfies the difficulty check according to §7.6.2 *'Difficulty filter'* on p. 86.

7.6.1 Equihash

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

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

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

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

Let powheader :=

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

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

EquihashGen is instantiated in §5.4.1.9 *'Equihash Generator'* on p. 56.

Define $\mathsf{I2BEBSP} \colon (A \colon \mathbb{N}) \times \{0\,..\,2^A - 1\} \to \mathbb{B}^{[A]}$ as in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

A *valid Equihash solution* is then a sequence $i \colon \{1\,..\,N\}^{2^k}$ that satisfies the following conditions:

Generalized Birthday condition $\bigoplus_{j=1}^{2^k} X_{i_j} = 0$.

Algorithm Binding conditions

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

Notes:

- This does not include a difficulty condition, because here we are defining validity of an Equihash solution independent of difficulty.
- Previous versions of this specification incorrectly specified the range of r to be $\{1\,..\,k-1\}$ for both parts of the algorithm binding condition. The implementation in bitzecd 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. 48 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">· · ·</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).

7.6.2 Difficulty filter

Let ToTarget be as defined in §7.6.4 *'nBits conversion'* on p. 87.

Difficulty is defined in terms of a *target threshold* , which is adjusted for each *block* according to the algorithm defined in §7.6.3 *'Difficulty adjustment'* on p. 86.

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

7.6.3 Difficulty adjustment

bitzec uses a difficulty adjustment algorithm based on DigiShield v3/v4 [DigiByte-PoW], with simplifications and altered parameters, to adjust difficulty to target the desired 2.5-minute block time. Unlike **Bitcoin**, the difficulty adjustment occurs after every block.

The constants PoWLimit, PoWAveragingWindow, PoWMaxAdjustDown, PoWMaxAdjustUp, PoWDampingFactor, and PoWTargetSpacing are instantiated in §5.3 *'Constants'* on p. 49.

Let ToCompact and ToTarget be as defined in §7.6.4 *'nBits conversion'* on p. 87.

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

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

Block header fields are specified in §7.5 *'Block Header'* on p. 83.

Define:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.6.4 nBits conversion

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

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

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

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

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

7.6.5Debnition of Work

As explained in §3.3 *‘The Block Chain’* on p. 14, a node chooses the “best” *block chain* visible to it by Znding the chain of valid *blocks* with the greatest total work.

Let ToTarget be as deZned in §7.6.4 *‘nBits conversion’* on p. 87.

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

7.7 Calculation of Block Subsidy and Founders’ Reward

§3.9 *'Block Subsidy and Founders' Reward'* on p. 17 defines the *block subsidy* , *miner subsidy* , and *Founders' Reward* . Their amounts in *zatoshi* are calculated from the *block height* using the formulae below. The constants SlowStartInterval, HalvingInterval, MaxBlockSubsidy, and FoundersFraction are instantiated in §5.3 *'Constants'* on p. 49.

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

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

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

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

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

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

7.8 Payment of Founders' Reward

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

For the production network, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ is:

**["t3Vz22vK5z2LcKEdg16Yv4FFneEL1zg9ojd", "t3cL9AucCajm3HXDhb5jBnJK2vapVoXsop3",
"t3fqvkzrrNaMcamkQMwAyHRjfDdM2xQvDTR", "t3TgZ9ZT2CTSK44AnUPi6qeNaHa2eC7pUyF",
"t3SpkcPQPfuRYHsP5vz3Pv86PgKo5m9KVmx", "t3Xt4oQMRPagwbpQqkgAViQgtST4VoSWR6S",
"t3ayBkZ4w6kKXynwoHZFUSSgXRKtogTXNgb", "t3adJBQuaa21u7NxbR8YMzp3km3TbSZ4MGB",
"t3K4aLYagSSBySdrfAGGeUd5H9z5Qvz88t2", "t3RYnsc5nhEvKiva3ZPhfRSk7eyh1CrA6Rk",
"t3Ut4KUq2ZSMTPNE67pBU5LqYCi2q36KpXQ", "t3ZnCNAvgu6CSyHm1vWtrx3aiN98dSAGpnD",
"t3fB9cB3eSYim64BS9xfwAHQUKLgQQroBDG", "t3cwZfKNNj2vXMAHBQeewm6pXhKFdhk18kD",
"t3YcoujXfspWy7rbNUsGKxFEWZqNstGpeG4", "t3bLvCLigc6rbNrUTS5NwkgyVrZcZumTRa4",
"t3VvHWa7r3oy67YtU4LZKGCWa2J6eGHvShi", "t3eF9X6X2dSo7MCvTjfZEzwWrVzquxRLNeY",
"t3esCNwwmcyc8i9qQfyTbYhTqmYXZ9AwK3X", "t3M4jN7hYE2e27yLsuQPPjuVek81WV3VbBj",
"t3gGWxdC67CYNoBbPjNvrrWLAWxPqZLxrVY", "t3LTWeoxeWPbmdkUD3NWBquk4WkazhFBmvU",
"t3P5KKX97gXYFSaSjJPiruQEX84yF5z3Tjq", "t3f3T3nCWsEpzmD35VK62JgQfFig74dV8C9",
"t3Rqonuzz7afkF7156ZA4vi4iimRSEn41hj", "t3fJZ5jYsyxDtvNrWBeoMbvJaQCj4JJgbgX",
"t3Pnbg7XjP7FGPBUuz75H65aczphHgkpoJW", "t3WeKQDxCijL5X7rwFem1MTL9ZwVJkUFhpF",
"t3Y9FNi26J7UtAUC4moaETLbMo8KS1Be6ME", "t3aNRLLsL2y8xcjPheZZwFy3Pcv7CsTwBec",
"t3gQDEavk5VzAAHK8TrQu2BWDLxEiF1unBm", "t3RbykhxlTUFrgXrmBYrAJe2STxRKFL7G9r",
"t3aaW4aTdP7a8d1VTE1Bod2yhbeggHgMajR", "t3YEiAa6uEjXwFL2v5ztU1fn3yKgzMQqNyo",
"t3g1yUUwt2PbmDvMDevTCPWUcbDatL2iQGP", "t3dPWnep6YqGPuY1CecgbeZrY9iUwH8Yd4z",
"t3QRZXHDPh2hwU46iQs2776kRuuWfwFp4dV", "t3enhACRxi1ZD7e8ePomVGKn7wp7N9fFJ3r",
"t3PkLgT71TnF112nSwBToXsD77yNbx2gJJY", "t3LQtHUDoe7ZhhvddRv4vnaoNAhCr2f4oFN",
"t3fNcdBUbycvbCtsD2n9q3LuxG7jVPvFB8L", "t3dKojUU2EMjs28nHV84TvkVEUDu1M1FaEx",
"t3aKH6NiWN1ofGd8c19rZiqgYpkJ3n679ME", "t3MEXDF9Wsi63KwpPuQdD6by32Mw2bNTbEa",
"t3WDhPfik343yNmPTqtkZAoQZeqA83K7Y3f", "t3PSn5TbMMAEw7Eu36DYctFezRzpX1hzf3M",
"t3R3Y5vnBLrEn8L6wFjPjBLnxSUQsKnmFpv", "t3Pcm737EsVkGTbhsu2NekKtJeG92mvYyoN"]**

For the test network, $\mathsf{FounderAddressList}_{1..\mathsf{NumFounderAddresses}}$ is:

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

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

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

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

Define:

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

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

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

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

Notes:

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

7.9 Changes to the Script System

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

7.10 Bitcoin Improvement Proposals

In general, Bitcoin Improvement Proposals (BIPs) do not apply to **bitzec** unless otherwise speciZed 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 **bitzec**: [BIP-11], [BIP-14], [BIP-31], [BIP-35], [BIP-37], [BIP-61].

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

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

[BIP-13] applies with the changes to address version bytes described in §5.6.1 *'Transparent Addresses'* on p. 72.

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

- references to protocol version 70002 are to be replaced by 170003;
- references to protocol version 70011 are to be replaced by 170004;
- the reference to protocol version 70000 is to be ignored (**bitzec** nodes have supported Bloom-Zltered 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 **bitzec**, there is only the original **Bitcoin** transaction type, which is extended to contain a sequence of zero or more **bitzec**-speciZc operations.

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

Computation of *SIGHASH transaction hashes*, as described in §4.9 *'SIGHASH Transaction Hashing'* on p. 35, was changed to clean up handling of an error case for SIGHASH_SINGLE, to remove the special treatment of OP_CODESEPARATOR, and to include **bitzec**-speciZc Zelds in the hash [ZIP-76].

8.2 Memo Fields

bitzec adds a *memo beld* 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. 71.

8.3 Unibcation 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 $\mathsf{N}^{\mathsf{old}}$ *shielded notes* as input, and produces up to $\mathsf{N}^{\mathsf{new}}$ *shielded notes* and a *transparent* UTXO as output.

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

[Pre-**Sapling**] In **bitzec**, the sequence of operations added to a *transaction* (see §8.1 *'Transaction Structure'* on p. 90) 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 **bitzec** *transaction* that takes input from an UTXO can produce up to $\mathsf{N}^{\mathsf{new}}$ 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 uniZcation also simpliZes the Zx 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 §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36 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 signiZcantly 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 bx

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

An adversary can use this to mislead a *note* recipient, by sending two *notes* both of which are veriZed as valid by $\mathsf{Receive}$ (as deZned 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 deZnitions given in [BCGGMTV2014]. The issue could be framed as a problem either with the deZnition of Completeness, or the deZnition of Balance:

- The Completeness property asserts that a validly received *note* can be spent provided that its *nulliber* 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 *nulliber*. That is, the security deZnition depends on a protocol detail –*nullibers*– 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 **bitzec** addresses the immediate attack.

It would be possible to address the attack by requiring that a recipient remember all of the ρ values for all *notes* they have ever received, and reject duplicates (as proposed in [GGM2016]). However, this requirement would interfere with the intended **bitzec** 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, **bitzec** enforces that an adversary must choose distinct values for each ρ, by making use of the fact that all of the *nullifiers* in *JoinSplit descriptions* that appear in a *valid block chain* must be distinct. This is true regardless of whether the *nullifiers* corresponded to real or dummy notes (see §4.7.1 *'Dummy Notes (**Sprout**)'* on p. 33). 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. 29. ($\mathsf{h_{Sig}}$ was already used in **Zerocash** in a way that requires it to be unique in order to maintain indistinguishability of *JoinSplit descriptions*; adding the *nullifiers* to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the *transaction* creator is an adversary.)

[**Sprout**] The ρ value for each output *note* is then derived from a random private seed φ and $\mathsf{h_{Sig}}$ using $\mathsf{PRF}^{\rho}_{\varphi}$. The correct construction of ρ for each output *note* is enforced by §4.15.1 *'Uniqueness of ρ^{new}_i'* on p. 40 in the *JoinSplit statement* .

[**Sprout**] Now even if the creator of a *JoinSplit description* does not choose φ randomly, uniqueness of *nullifiers* and collision resistance of both $\mathsf{hSigCRH}$ and PRF^{ρ} will ensure that the derived ρ values are unique, at least for any two *JoinSplit descriptions* that get into a *valid block chain*. This is sufficient to prevent the Faerie Gold attack.

A variation on the attack attempts to cause the *nullifier* of a sent *note* to be repeated, without repeating ρ. However, since the *nullifier* is computed as $\mathsf{PRF}^{\mathsf{nf}}_{a_{\mathsf{sk}}}(\rho)$ (or $\mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}}(\rho>)$ for **Sapling**), this is only possible if the adversary finds a collision across both inputs on $\mathsf{PRF}^{\mathsf{nf}}$ (or $\mathsf{PRF}^{\mathsf{nfSapling}}$), which is assumed to be infeasible — see §4.1.2 *'Pseudo Random Functions'* on p. 18.

[**Sprout**] Crucially, "*nullifier* integrity" is enforced whether or not the $\mathsf{enforceMerklePath}_i$ flag is set for an input *note* (§4.15.1 *'Nullifier integrity'* on p. 40). If this were not the case then an adversary could perform the attack by creating a zero-valued *note* with a repeated *nullifier* , since the *nullifier* would not depend on the value.

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

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

8.5 Internal hash collision attack and fix

The **Zerocash** security proof requires that the composition of $\mathsf{COMM_{rcm}}$ and $\mathsf{COMM_s}$ is a computationally binding commitment to its inputs $\mathsf{a_{pk}}$, v, and ρ. However, the instantiation of $\mathsf{COMM_{rcm}}$ and $\mathsf{COMM_s}$ in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of $\mathsf{a_{pk}}$ and ρ is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of 2^{64}, to Znd distinct pairs $(\mathsf{a}_{\mathsf{pk}}, \rho)$ and $(\mathsf{a}^{r}_{\mathsf{pk}}, \rho^{r})$ 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].

bitzec 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 veriZed without requiring a *zero-knowledge proof* ([BCGGMTV2014, section 1.3, under step 3]). Since **bitzec** combines "Mint" and "Pour" transactions into generalized *JoinSplit transfers* (for **Sprout**),or *Spend transfers* and *Output transfers* (for **Sapling**), and each transfer always uses a *zero-knowledge proof* , **bitzec** does not require the nesting. A side beneZt 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, **bitzec** 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 deZne 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 beneZts.

[**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 bitzecd). Note that *diversibed 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. 56 has changed relative to **Zerocash**. There is also a requirement for another PRF, PRF^{ρ}, which must be domain-separated from the others.

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

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

In more detail:

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

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

As further evidence that it is essential for the proofs to explicitly take any such truncations into account, consider a slightly modiZed protocol in which ρ is truncated in the input to $\mathsf{COMM}_{\mathsf{rcm}}$ but not in the input to $\mathsf{PRF}^{\mathsf{sn}}$. In that case, it would be possible to violate balance by creating two *notes* for which ρ differs only in the truncated bits. These *notes* would have the same *note commitment* but different *nullibers*, 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. 91, **bitzec** depends on collision resistance of hSigCRH and PRF^{ρ} (instantiated using BLAKE2b-256 and SHA256Compress respectively). Collision resistance of a truncated hash does not follow from collision resistance of the original hash, even if the truncation is only by one bit. This motivated avoiding truncation along any path from the inputs to the computation of $\mathsf{h}_{\mathsf{Sig}}$ to the uses of ρ.

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

Sapling uses *Pedersen hashes* and 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 $\mathsf{CRH}^{\mathsf{ivk}}$ is truncated, requiring a security assumption on BLAKE2s truncated to 251 bits (see §5.4.1.5 *'$\mathsf{CRH}^{\mathsf{ivk}}$ Hash Function'* on p. 52).

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 $\mathsf{h}_{\mathsf{Sig}}$ 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 notes ciphertext* unless $\mathsf{pk}_{\mathsf{enc}}$ 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 $\mathsf{BLAKE2b\text{-}256}$ 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 $\mathsf{AEAD\_CHACHA20\_POLY1305}$ 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 **bitzec** 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 ρ), the adversary is able to double-spend the note, once with each $\mathsf{a_{sk}}$. This is not detected because each spend reveals a different *nullifier* . The *JoinSplit statements* are still valid because they can only check that the $\mathsf{a_{sk}}$ in the witness is *some* preimage of the $\mathsf{a_{pk}}$ used in the *note commitment*.

The error is in the proof of Balance in [BCGGMTV2014, Appendix D.3]. For the "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 "A violates Condition II" case.

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

The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of $\mathsf{PRF^{addr}}$ (on both its arguments) to argue that distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{sk},2}$ together with constraint 1(b) of the *JoinSplit statement* (see §4.15.1 *'Spend authority'* on p. 40), implies distinctness of $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},1}$ and $\mathsf{a}^{\mathsf{old}}_{\mathsf{pk},2}$, therefore distinct openings of the *note commitment* when Condition I or II is violated.

8.9 Miscellaneous

- The paper defines a *note* as $((\mathsf{a_{pk}}, \mathsf{pk_{enc}}), \mathsf{v}, \rho, \mathsf{rcm}, \mathsf{s}, \mathsf{cm})$, whereas this specification defines a **Sprout** *note* as $(\mathsf{a_{pk}}, \mathsf{v}, \rho, \mathsf{rcm})$. The instantiation of $\mathsf{COMM_s}$ in section 5.1 of the paper did not actually use s, and neither does the new instantiation of $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ in **Sprout**. $\mathsf{pk_{enc}}$ is also not needed as part of a *note*: it is not an input to $\mathsf{NoteCommit}^{\mathsf{Sprout}}$ nor is it constrained by the **Zerocash** POUR *statement* or the **bitzec** *JoinSplit statement* . cm can be computed from the other fields.(The definition of *notes* for **Sapling** is different again.)
- The length of proof encodings given in the paper is 288 bytes. [**Sprout**] This differs from the 296 bytes specified in §5.4.9.1 *'PHGR13'* on p. 69, 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 **bitzec** uses this standard encoding rather than the less efficient (uncompressed) one used by upstream *libsnark* .In **Sapling**, a customized encoding is used for $\mathsf{BLS12\text{-}381}$ points in $\mathsf{Groth16}$ proofs to minimize length.
- The range of monetary values differs. In **bitzec** 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 **bitzec** 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, and George Tankersley. 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 **bitzec** protocol designs; in addition to the preceding, this includes Mike Perry, Isis Lovecruft, Leif Ryge, Andrew Miller, Samantha Hulsey, jl777, Ben Blaxill, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, Alison Stevenson, John Tromp, Paige Peterson, Maureen Walsh, Jack Gavigan, Filippo Valsorda, Zaki Manian, Tracy Hu, Brian Warner, Mary Maller, Michael Dixon, Andrew Poelstra, and no doubt others. We would also like to thank the designers and developers of **Bitcoin**.

bitzec has benefited from security audits performed by NCC Group, Coinspect, Least Authority, Mary Maller, Kudelski Security, and QED-it.

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. 93 were also found by Daira Hopwood.

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

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.

Many of the ideas used in **bitzec** —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 **bitzec** is a refinement of their "Auditable, Anonymous Electronic Cash" proposal in [ST1999].

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

10Change History

2018.0-beta-33

- No changes to **Sprout**.
- Complete §A.4 *'The Sapling Spend circuit'* on p. 136.
- Add §A.5 *'The Sapling Output circuit'* on p. 138.
- Change the description of window lookup in §A.3.3.7 *'Fixed-base affine-Edwards scalar multiplication'* on p. 128 to match sapling-crypto.
- Describe 2-bit window lookup with conditional negation in §A.3.3.9 *'Pedersen hash'* on p. 129.
- 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

- No changes to **Sprout**.
- Correct the input to $\mathsf{H}^{\sim}$ 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. 132.

2018.0-beta-31 2018-09-30

- No changes to **Sprout**.
- Correct some uses of $r_{\mathbb{J}}$ that should have been $r_{\mathbb{S}}$ or q.
- Correct uses of $\mathsf{LEOS2IP}_A$ in $\mathsf{RedDSA.Verify}$ and $\mathsf{RedDSA.BatchVerify}$ to ensure that A 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 $\mathsf{E}_{\mathbb{A}}$.
- Correct a use of $\mathbb{J}$ that should have been $\mathbb{M}$ in the proof of Theorem A.3.4 on p. 126, and make a minor tweak to the theorem statement (k_2 Ç $\pm k_1$ instead of k_1 Ç $\pm k_2$) to make the contradiction derived by the proof clearer.
- Clarify notation in the proof of Theorem A.3.3 on p. 125.
- Address some of the findings of the QED-it report:
  - Improved cross-referencing in §5.4.1.7 *'Pedersen Hash Function'* on p. 53.
  - Clarify the notes concerning domain separation of prefixes in §5.4.1.3 *'$\mathsf{MerkleCRH}^{\mathsf{Sapling}}$ Hash Function'* on p. 51 and §5.4.7.2 *'Windowed Pedersen commitments'* on p. 63.
  - Correct the statement and proof of Theorem A.3.2 on p. 125.
- Add the QED-it report to the acknowledgements.

2018.0-beta-30 2018-09-02

- No changes to **Sprout**.
- 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 [BGM2018] for the **Sapling** *zk-SNARK* parameters.
- Write §A.4 *'The Sapling Spend circuit'* on p. 136 (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 Edwards coordinates are on the curve'* on p. 124 and §A.3.3.6 *'Affine-Edwards nonsmall-order check'* on p. 127 accurately reflect the sapling-crypto implementation.
- Minor correction to the non-normative note in §A.3.2.2 *'Range check'* on p. 122.
- Clarify the non-normative note in §4.1.7 *'Commitment'* on p. 23 about the definitions of ValueCommit.Output and NoteCommit$^{\mathsf{Sapling}}$.Output.
- Clarify that the signer of a *spend authorization signature* is supposed to choose the *spend authorization randomizer*, α, itself. Only step 4 in the procedure in §4.13 *'Spend Authorization Signature'* on p. 38 may securely be delegated.
- Add a non-normative note to §5.4.6 *'RedDSA and RedJubjub'* on p. 59 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

- No changes to **Sprout**.
- Finish §A.3.2.2 *'Range check'* on p. 122.
- Change §A.3.7 *'BLAKE2s hashes'* on p. 133 to correct the constraint count and to describe batched equality checks performed by the sapling-crypto implementation.

2018.0-beta-28 2018-08-14

- No changes to **Sprout**.
- Finish §A.3.7 *'BLAKE2s hashes'* on p. 133.
- Minor corrections to §A.3.3.8 *'Variable-base affine-Edwards scalar multiplication'* on p. 129.

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 > an affix instead of a superscript.
- Clarify that when validating a Groth16 proof, it is necessary to perform a subgroup check for π_A and π_C as well as for π_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

- No changes to **Sprout**.
- Add §B.2 *'Groth16 batch verification'* on p. 140.

2018.0-beta-25 2018-08-05

- No changes to **Sprout**.
- Add the hashes of parameter files for **Sapling**.
- Add cross references for parameters and functions used in RedDSA batch verification.
- Makefile changes: name the PDF file for the **Sprout** version of the specification as sprout.pdf, and make protocol.pdf link to the **Sapling** version.

2018.0-beta-24 2018-07-31

- No changes to **Sprout**.
- Add a missing consensus rule for version 4 *transactions*: if there are no **Sapling** spends or outputs, then valueBalance **MUST** be 0.

2018.0-beta-23 2018-07-27

- No changes to **Sprout**.
- Update RedDSA verification to use cofactor multiplication. This is necessary in order for the output of batch verification to match that of unbatched verification in all cases.
- Add §B.1 *'RedDSA batch verification'* on p. 139.

2018.0-beta-22 2018-07-18

- No changes to **Sprout**.
- Update §6 *'Network Upgrades'* on p. 77 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. 130.
- Add a note about redundancy in the nonsmall-order checking of 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-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.
- Brieay 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 *‘Transparent Addresses’* on p. 72 — they use a hash of a compressed, not an uncompressed ECDSA key representation.
- Clarify the wording of the caveat[3] about the claimed security of shielded *transactions*.
- Correct the deZnition of set difference ($S \setminus T$).
- Add a note concerning malleability of *zero-knowledge proofs*.
- Clarify attribution of the **bitzec** protocol design.
- Acknowledge Alex Biryukov and Dmitry Khovratovich as the designers of Equihash.
- Acknowledge ShaZ 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 Zeld of an *Output description* to cmu, reaecting the fact that it is a *Jubjub curve* u-coordinate.
- Add explicit consensus rules that the anchor Zeld of a *Spend description* and the cmu Zeld of an *Output description* must be canonical encodings.
- Enforce that esk in outCiphertext is a canonical encoding.
- Add consensus rules that cv in a *Spend description*, and cv and epk in an *Output description*, are not of small order. Exclude 0 from the range of esk when encrypting **Sapling** notes.
- Add a consensus rule that valueBalance is in the range $\{-\mathsf{MAX\_MONEY} \,..\, \mathsf{MAX\_MONEY}\}$.
- Enforce stronger constraints on the types of key components $\mathsf{pk_d}$, ak, and 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 v^* is in range in §4.12 *‘Balance and Binding Signature (**Sapling**)’* on p. 36.
- Correct an error in the algorithm for RedDSA.Verify: the public key vk is given directly to this algorithm and should not be computed from the unknown private key sk.
- Correct or improve the types of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$, $\mathsf{FindGroupHash}^{\mathbb{J}^{(r)*}}$, $\mathsf{Extract}_{\mathbb{J}^{(r)}}$, $\mathsf{PRF}^{\mathsf{expand}}$, $\mathsf{PRF}^{\mathsf{ock}}$, and $\mathsf{CRH}^{\mathsf{ivk}}$.
- Instantiate $\mathsf{PRF}^{\mathsf{ock}}$ using BLAKE2b-256.
- Change the syntax of a *commitment scheme* to add 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* (ovk was missing).
- Ensure that **Sprout** functions and values are given **Sprout**-speciZc types where appropriate.
- Improve cross-referencing.

- Clarify the use of PHGR13 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 *'Edwards ↔ Montgomery conversion'* on p. 125.)
- Model the group hash as a random oracle. This appears to be unavoidable in order to allow proving unlinkability of DiversifyHash. 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. 119 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 *balancing value* in §4.12 *'Balance and Binding Signature (**Sapling**)'* on p. 36.
- Correct a type error in §5.4.8.5 *'Group Hash into Jubjub'* on p. 69.
- Correct a type error in RedDSA.Sign in §5.4.6 *'RedDSA and RedJubjub'* on p. 59.
- Ensure $\mathcal{G}$ is defined in §5.4.6.1 *'Spend Authorization Signature'* on p. 62.
- Make the public 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 H^{y} to $\mathsf{H}^{\sim}$ in §5.4.6 *'RedDSA and RedJubjub'* on p. 59, to avoid confusion with the ^{y} convention for representations of group elements.
- 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

- No changes to **Sprout**.
- Minor clarifications.

2018.0-beta-18 2018-04-23

- No changes to **Sprout**.
- Clarify the security argument for balance in **Sapling**.
- Correct a subtle problem with the type of the value input to ValueCommit: although it is only directly used to commit to values in $\{0 .. 2^{A\mathsf{value}} - 1\}$, the security argument depends on a sum of commitments being binding on ’$-\frac{r_{\mathbb{J}}-1}{2} .. \frac{r_{\mathbb{J}}-1}{2}$’.
- Fix the loss of tightness in the use of $\mathsf{PRF}^{\mathsf{nfSapling}}$ by specifying the keyspace more precisely.
- Correct type ambiguities for $\mathsf{\rho}$.
- Specify the representation of i in group $\mathbb{G}_2$ of BLS12-381.

2018.0-beta-17 2018-04-21

- No changes to **Sprout**.
- 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 ℓ.
- Specify support for [BIP-111] (the NODE_BLOOM service bit) in network protocol version 170004.
- Give references [Vercauter2009] and [AKLGL2010] for the optimal ate pairing.
- Give references for 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. 7 and link to [Peterson2017] and [Quesnelle2017].
- Do not require a generator as part of the specification of a *represented group*; instead, define it in the *represented pairing* or scheme using the group.
- Refactor the abstract definition of a *signature scheme* to allow derivation of verifying keys independent of key pair generation.
- Correct the explanation in §1.2 *'High-level Overview'* on p. 7 to apply to **Sapling**.
- Add the definition of a private key to public 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 *binding signatures* and *spend authorization signatures*.
- Specify the randomness beacon.

- Add *output ciphertexts* and ock.
- Define $\mathsf{DefaultDiversifier}$.
- 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. 126 (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 *nulliber* is computed and separating it from the *randomized spend verifying key* (rk).
- Clarify conversions between bit and byte sequences for sk, $\mathsf{repr}_{\mathbb{J}}(\mathsf{ak})$, and $\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$.
- Change the Makefile to avoid multiple reloads in PDF readers while rebuilding the PDF.
- Spacing and pagination improvements.

2018.0-beta-14 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* 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

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

2018.0-beta-11 2018-02-26

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

2018.0-beta-10 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. 48.
- 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. 126) 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

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

2018.0-beta-7 2018-02-07

- Specify the 100000-byte limit on *transaction* size. (The implementation in bitzecd 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

- No changes to **Sprout**.
- **Sapling** work in progress, mainly on Appendix A *‘Circuit Design’* on p. 119.

2018.0-beta-5 2018-01-30

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

2018.0-beta-4 2018-01-25

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

2018.0-beta-3 2018-01-22

- Explain how the chosen Zx 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 $\mathsf{sk_{enc}}$ 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 π_B.
- Initial version of draft **Sapling** protocol speciZcation.

2017.0-beta-2.7 2017-07-10

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

2017.0-beta-2.6 2017-05-09

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

2017.0-beta-2.5 2017-03-07

- Clarify the consensus rule preventing double-spends.
- Clarify what a *note commitment* opens to in §8.8 *‘Omission in **Zerocash** security proof’* on p. 95.
- Correct the order of arguments to COMM in §5.4.7.1 *‘**Sprout** Note Commitments’* on p. 62.
- Correct a statement about indistinguishability of *JoinSplit descriptions*.
- Change the *Founders’ Reward* addresses, for the test network only, to reaect the hard-fork upgrade described in [bitzec-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 $(\mathsf{a}_{\mathsf{pk}}, \rho)$ rather than just on ρ.
- Rename $\mathsf{enforce}_i$ to $\mathsf{enforceMerklePath}_i$.

2017.0-beta-2.3 2017-02-12

- Specify the security requirements on the *SHA-256 compression* function in order for the scheme in §5.4.7.1 *'**Sprout** Note Commitments'* on p. 62 to be a secure commitment.
- Specify 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 [BGG2016].

2017.0-beta-2.1 2017-02-06

- A_{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 *nulliber* integrity.
- Make the description of *block chains* more consistent with upstream **Bitcoin** documentation (referring to "best" chains rather than using the concept of a *block chain view*).
- Define how nodes select a best chain.

2016.0-beta-1.13 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. 27.

2016.0-beta-1.12 2017-01-09

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

2016.0-beta-1.11 2016-12-19

- Specify a check on the order of π_B in a *zero-knowledge proof* .
- Note that due to an oversight, the **bitzec** *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 the production network.
- 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 **bitzec**.
- 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 $\mathsf{BLAKE2b}\text{-}A$ is different from truncated $\mathsf{BLAKE2b\text{-}512}$.

- Clarify a note about SU-CMA security for signatures.
- Add a note about $\mathsf{PRF}^{\mathsf{nf}}$ corresponding to $\mathsf{PRF}^{\mathsf{sn}}$ in **Zerocash**.
- Add a paragraph about key length in §8.7 *'In-band secret distribution'* on p. 94.
- Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

2016.0-beta-1.5 2016-09-22

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

2016.0-beta-1.4 2016-09-19

- Specify the *block subsidy*, *miner subsidy*, and the *Founders' Reward*.
- Specify *coinbase transaction* outputs to *Founders' Reward* addresses.
- Improve notation (for example "·" for multiplication and "$T^{[A]}$" 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 uint encodings must be canonical.
- Add a note about conformance language in the introduction.
- Add acknowledgements for Solar Designer, Ling Ren and Alison Stevenson, and for the NCC Group and Coinspect security audits.

2016.0-beta-1.2 2016-09-11

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

2016.0-beta-1.1 2016-09-05

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

2016.0-beta-1 2016-09-04

- Major reorganization to separate the abstract cryptographic protocol from the algorithm instantiations.
- Add type declarations.
- Add a "High-level Overview" section.
- Add a section specifying the *zero-knowledge proving system* and the encoding of proofs. Change the encoding of points in proofs to follow IEEE Std 1363[a].
- Add a section on consensus changes from **Bitcoin**, and the specification of Equihash.

- Complete the “Differences from the **Zerocash** paper” section.
- Correct the Merkle tree depth to 29.
- Change the length of *memo belds* to 512 bytes.
- Switch the *JoinSplit signature* scheme to Ed25519, with consequent changes to the computation of $\mathsf{h}_{\mathsf{Sig}}$.
- Fix the lead bytes in *shielded payment address* and *spending key* encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $\mathsf{v}^{\mathsf{old}}_{\mathsf{pub}}$and $\mathsf{v}^{\mathsf{new}}_{\mathsf{pub}}$
- Clarify cryptographic security requirements and added deZnitions relating to the in-band secret distribution.
- Add various citations: the “Fixing Vulnerabilities in the bitzec Protocol” and “Why Equihash?” blog posts, several crypto papers for security deZnitions, 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 toQuattrocento.

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 **bitzec** 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. 95).

- Add speciZcation of path computation for the *incremental Merkle tree*.
- Add a note in §4.15.1 *'Merkle path validity'* on p. 40 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 Difbe-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) (↑ p19, 94, 95).

[AGRRT2017] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. *MiMC: Efbcient 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) (↑ p135).

[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) (↑ p64, 102).

[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) (↑ p51, 133).

[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. (↑ p20, 53, 94, 98).

[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) (↑ p125, 126).

[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, 18, 36, 40, 44, 91, 93, 95).

[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. (↑ p69).

[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. (↑ p53).

[BCTV2014]Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. “Scalable Zero Knowledge via Cycles of Elliptic Curves (extended version)”. In: *Advances in Cryptology - CRYPTO 2014*. Vol. 8617. Lecture Notes in Computer Science. Springer, 2014, pages 276–294. URL: https://www.cs.tau.ac.il/~tromer/papers/scalablezk-20140803.pdf (visited on 2016-09-01) (↑ p26).

[BCTV2015]Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. *Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture*. Cryptology ePrint Archive: Report 2013/879. Last revised May 19, 2015. URL: https://eprint.iacr.org/2013/879 (visited on 2016-08-21) (↑ p69,70, 119).

[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) (↑ p20).

[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. (↑ p18).

[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. (↑ p59, 140).

[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. (↑ p140, 141).

[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. (↑ p95).

[Bernstein2006]Daniel Bernstein. “Curve25519: new DifZe-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. (↑ p19, 58, 73, 74, 94).

[BGG-mpc]Sean Bowe, Ariel Gabizon, and Matthew Green. *GitHub repository ‘bitzec/mpc’: zk-SNARK parameter multi-party computation protocol.* URL: https://github.com/bitzec/mpc (visited on 2017-01-06) (↑ p76).

[BGG1995]Mihir Bellare, Oded Goldreich, and ShaZ 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) (↑ p53, 55, 129).

[BGG2016]Sean Bowe, Ariel Gabizon, and Matthew Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. November 24, 2016. URL: https://github.com/bitzec/mpc/blob/master/whitepaper.pdf (visited on 2017-02-11) (↑ p76, 107).

[BGM2018]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) (↑ p76, 98).

[BIP-11]Gavin Andresen. *M-of-N Standard Transactions*. Bitcoin Improvement Proposal 11. Created October 18, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-13]Gavin Andresen. *Address Format for pay-to-script-hash*. Bitcoin Improvement Proposal 13. Created October 18, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki (visited on 2016-09-24) (↑ p72, 90).

[BIP-14]Amir Taaki and Patrick Strateman. *Protocol Version and User Agent*. Bitcoin Improvement Proposal 14. Created November 10, 2011. URL: https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-16]Gavin Andresen. *Pay to Script Hash*. Bitcoin Improvement Proposal 16. Created January 3, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-30]Pieter Wuille. *Duplicate transactions*. Bitcoin Improvement Proposal 30. Created February 22, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-31]Mike Hearn. *Pong message*. Bitcoin Improvement Proposal 31. Created April 11, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-32]Pieter Wuille. *Hierarchical Deterministic Wallets*. Bitcoin Improvement Proposal 32. Created February 11, 2012. Last updated January 15, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki (visited on 2016-09-24) (↑ p73).

[BIP-34]Gavin Andresen. *Block v2, Height in Coinbase*. Bitcoin Improvement Proposal 34. Created July 6, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki (visited on 2016-10-02) (↑ p90, 108).

[BIP-35]Jeff Garzik. *mempool message*. Bitcoin Improvement Proposal 35. Created August 16, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-37]Mike Hearn and Matt Corallo. *Connection Bloom bltering*. Bitcoin Improvement Proposal 37. Created October 24, 2012. URL: https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-61]Gavin Andresen. *Reject P2P message*. Bitcoin Improvement Proposal 61. Created June 18, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-62]Pieter Wuille. *Dealing with malleability*. Bitcoin Improvement Proposal 62. Withdrawn November 17, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki (visited on 2016-09-05) (↑ p21).

[BIP-65]Peter Todd. *OP_CHECKLOCKTIMEVERIFY*. Bitcoin Improvement Proposal 65. Created October 10, 2014. URL: https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-66]Pieter Wuille. *Strict DER signatures*. Bitcoin Improvement Proposal 66. Created January 10, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki (visited on 2016-10-02) (↑ p90).

[BIP-68]Mark Friedenbach, BtcDrak, Nicolas Dorier, and kinoshitajona. *Relative lock-time using consensus-enforced sequence numbers*. Bitcoin Improvement Proposal 68. Last revised November 21, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki (visited on 2016-09-02) (↑ p80).

[BIP-111]Matt Corallo and Peter Todd. *NODE_BLOOM service bit*. Bitcoin Improvement Proposal 111. Created August 20, 2015. URL: https://github.com/bitcoin/bips/blob/master/bip-0111.mediawiki (visited on 2018-04-02) (↑ p90, 102).

[BIP-173]Pieter Wuille and Greg Maxwell. *Base32 address format for native v0-16 witness outputs*. Bitcoin Improvement Proposal 173. Last revised September 24, 2017. URL: https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki (visited on 2018-01-22) (↑ p72, 101).

[Bitcoin-Base58] *Base58Check encoding — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Base58Check_encoding (visited on 2016-01-26) (↑ p72, 73).

[Bitcoin-Block] *Block Headers — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#block-headers (visited on 2017-04-25) (↑ p84).

[Bitcoin-CoinJoin] *CoinJoin — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/CoinJoin (visited on 2016-08-17) (↑ p9).

[Bitcoin-Format] *Raw Transaction Format — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#raw-transaction-format (visited on 2016-03-15) (↑ p80).

[Bitcoin-Multisig] *P2SH multisig (debnition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2sh-multisig (visited on 2016-08-19) (↑ p89).

[Bitcoin-nBits] *Target nBits — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#target-nbits (visited on 2016-08-13) (↑ p83, 87).

[Bitcoin-Order] *Hash Byte Order — Bitcoin Developer Reference*. URL: https://bitcoin.org/en/developer-reference#hash-byte-order (visited on 2018-02-09) (↑ p76).

[Bitcoin-P2PKH] *P2PKH (debnition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2pkh (visited on 2016-08-24) (↑ p72).

[Bitcoin-P2SH] *P2SH (debnition) — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#term-p2sh (visited on 2016-08-24) (↑ p72).

[Bitcoin-Protocol] *Protocol documentation — Bitcoin Wiki*. URL: https://en.bitcoin.it/wiki/Protocol_documentation (visited on 2016-10-02) (↑ p8).

[Bitcoin-SigHash] *Signature Types — Bitcoin Developer Guide*. URL: https://bitcoin.org/en/developer-guide#signature-hash-types (visited on 2018-06-10) (↑ p35).

[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) (↑ p59, 68).

[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, 85, 108).

[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) (↑ p94, 116).

[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) (↑ p119, 125, 126, 127).

[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) (↑ p66, 102).

[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) (↑ p64, 102).

[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) (↑ p19).

[Bowe-bellman]Sean Bowe. *bellman: zk-SNARK library*. URL: https://github.com/ebfull/bellman (visited on 2018-04-03) (↑ p70, 76).

[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) (↑ p66).

[Bowe2018]Sean Bowe. *Random Beacon*. March 22, 2018. URL: https://github.com/bitzecFoundation/powersoftau-attestations/tree/master/0088 (visited on 2018-04-08) (↑ p76).

[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) (↑ p67).

[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) (↑ p97, 101).

[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) (↑ p53).

[CvHP1991]David Chaum, Eugène van Heijst, and Birgit PZtzmann. *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. (↑ p53, 129).

[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) (↑ p38, 98).

[deRooij1995]Peter de Rooij. “EfZcient 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) (↑ p140, 141).

[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) (↑ p95).

[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) (↑ p86).

[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) (↑ p22).

[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) (↑ p26, 35).

[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) (↑ p53).

[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).

[FKMSSS2016]Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. *Efbcient 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. (↑ p21, 22, 59).

[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) (↑ p92).

[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) (↑ p70, 71, 140).

[Hopwood2018]Daira Hopwood. *GitHub repository 'daira/jubjub': Supporting evidence for security of the Jubjub curve to be used in bitzec*. 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. (↑ p94).

[HW2016]Taylor Hornby and Zooko Wilcox. *Fixing Vulnerabilities in the bitzec Protocol*. bitzec blog. April 26, 2016. URL: https://blog.z.cash/fixing-bitzec-vulns/ (visited on 2018-04-15). Updated December 26, 2017. (↑ p93).

[IEEE2000]IEEE Computer Society. *IEEE Std 1363-2000: Standard Specibcations 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) (↑ p65).

[IEEE2004]IEEE Computer Society. *IEEE Std 1363a-2004: Standard Specibcations 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) (↑ p65, 94, 96).

[Jedusor2016]Tom Elvis Jedusor. *Mimblewimble*. July 19, 2016. URL: http://diyhpl.us/~bryan/papers2/bitcoin/mimblewimble.txt (visited on 2018-04-03) (↑ p38).

[KvE2013]Kaa1el and Hagen von Eitzen. *If a group tt 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. (↑ p68).

[KYMM2018]George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. *An Empirical Analysis of Anonymity in bitzec*. 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 (↑ p91).

[libsodium-Seal] *Sealed boxes — libsodium*. URL: https://download.libsodium.org/doc/public-key_cryptography/sealed_boxes.html (visited on 2016-02-01) (↑ p94).

[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) (↑ p20).

[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) (↑ p94).

[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 2018-02-14) (↑ p50, 72).

[Peterson2017]Paige Peterson. *Transaction Linkability*. bitzec blog. January 25, 2017. URL: https://blog.z.cash/transaction-linkability/ (visited on 2018-04-15) (↑ p9, 102).

[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) (↑ p69).

[Quesnelle2017]Jeffrey Quesnelle. *On the linkability of bitzec transactions*. arXiv:1712.01210 [cs.CR]. December 4, 2017. URL: https://arxiv.org/abs/1712.01210 (visited on 2018-04-15) (↑ p9, 102).

[RFC-2119]Scott Bradner. *Request for Comments 7693: Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF). March 1997. URL: https://tools.ietf.org/html/rfc2119 (visited on 2016-09-14) (↑ 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://tools.ietf.org/html/rfc7539 (visited on 2016-09-02). As modiZed by veriZed errata at https://www.rfc-editor.org/errata_search.php?rfc=7539 (visited on 2016-09-02). (↑ p57, 58).

[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) (↑ p72).

[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) (↑ p96, 100).

[Swihart2018]Josh Swihart. *Overwinter Activated Successfully*. bitzec blog. June 26, 2018. URL: https://blog.z.cash/overwinter-activated-successfully/ (visited on 2018-07-18) (↑ p77).

[Unicode]The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, 2016. URL: http://www.unicode.org/versions/latest/ (visited on 2016-08-31) (↑ p71).

[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. (↑ p64, 102).

[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 2018-05-22) (↑ p70, 119).

[WG2016]Zooko Wilcox and Jack Grigg. *Why Equihash?* bitzec blog. April 15, 2016. URL: https://blog.z.cash/why-equihash/ (visited on 2018-04-15). Updated December 14, 2017. (↑ p85).

[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) (↑ p95).

[bitzec-Issue2113]Simon Liu. *GitHub repository 'bitzec/bitzec': Issue 2113*. URL: https://github.com/bitzec/bitzec/issues/2113 (visited on 2017-02-20) (↑ p89, 106).

[bitzec-libsnark] *libsnark: C++ library for zkSNARK proofs (bitzec fork)*. URL: https://github.com/bitzec / bitzec/tree/master/src/snark (visited on 2018-02-04) (↑ p69).

[ZIP-32]Jack Grigg and Daira Hopwood. *Shielded Hierarchical Deterministic Wallets*. bitzec Improvement Proposal 32 (in progress). (↑ p29, 32, 45, 53, 61, 101).

[ZIP-76]Jack Grigg and Daira Hopwood. *Transaction Signature Veribcation before Overwinter*. bitzec Improvement Proposal 76 (in progress). (↑ p35, 90).

[ZIP-143]Jack Grigg and Daira Hopwood. *Transaction Signature Veribcation for Overwinter*. bitzec Improvement Proposal 143. Created December 27, 2017. URL: https://github.com/bitzec/zips/blob/master/zip-0143.rst (visited on 2018-03-01) (↑ p35, 51, 77).

[ZIP-200]Jack Grigg. *Network Upgrade Mechanism*. bitzec Improvement Proposal 200. Created January 8, 2018. URL: https://github.com/bitzec/zips/blob/master/zip-0200.rst (visited on 2018-03-01) (↑ p77, 80).

[ZIP-201]Simon Liu. *Network Peer Management for Overwinter*. bitzec Improvement Proposal 201. Created January 15, 2018. URL: https://github.com/bitzec/zips/blob/master/zip-0201.rst (visited on 2018-03-01) (↑ p77).

[ZIP-202]Simon Liu. *Version 3 Transaction Format for Overwinter*. bitzec Improvement Proposal 202. Created January 10, 2018. URL: https://github.com/bitzec/zips/blob/master/zip-0202.rst (visited on 2018-03-01) (↑ p77).

[ZIP-203]Jay Graber. *Transaction Expiry*. bitzec Improvement Proposal 203. Created January 9, 2018. URL: https://github.com/bitzec/zips/blob/master/zip-0203.rst (visited on 2018-03-01) (↑ p77, 78).

[ZIP-243] Jack Grigg and Daira Hopwood. *Transaction Signature Verification for Sapling*. bitzec Improvement Proposal 243. Created April 10, 2018. URL: https://github.com/bitzec/zips/blob/master/zip-0243.rst (visited on 2018-04-15) (↑ p35, 37, 39, 51, 77).

Appendices

A Circuit Design

A.1 Quadratic Constraint Programs

Sapling defines two circuits, Spend and Output, each implementing an abstract *statement* described in §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41 and §4.15.3 *'Output Statement (**Sapling**)'* on p. 42 respectively. It also adds a Groth16 circuit for the *JoinSplit statement* described in §4.15.1 *'JoinSplit Statement (**Sprout**)'* on p. 40.

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 PHGR13 or Groth16 proving systems, this program is translated to a *Quadratic Arithmetic Program* [BCTV2015, section 2.3] [WCBTV2015]. The circuit descriptions given here are necessary to compute witness elements for each circuit, as well as the proving and verification keys.

Let $\mathsf{F}_{r_\mathbb{S}}$ be the finite field over which Jubjub is defined, as given in §5.4.8.3 *'Jubjub'* on p. 67.

A *quadratic constraint program* consists of a set of constraints over variables in $\mathsf{F}_{r_\mathbb{S}}$, each of the form:

$$(A) \times (B) = (C)$$

where A, B, and C are *linear combinations* of variables and constants in $\mathsf{F}_{r_\mathbb{S}}$.

Here × and · both represent multiplication in the field $\mathsf{F}_{r_\mathbb{S}}$, but we use × for multiplications corresponding to gates of the circuit, and · for multiplications by constants in the terms of a *linear combination*. × should not be confused with × which is defined as cartesian product in §2 *'Notation'* on p. 9.

A.2 Elliptic curve background

The **Sapling** circuits make use of a twisted Edwards curve, Jubjub, and also a Montgomery curve M that is birationally equivalent to Jubjub. From here on we omit "twisted" when referring to the Edwards Jubjub curve or coordinates. Following the notation in [BL2017] we use (u, v) for affine coordinates on the Edwards curve, and (x, y) for affine coordinates on the Montgomery curve.

A point P is normally represented by two $\mathsf{F}_{r_\mathbb{S}}$ variables, which we name as (P^u, P^v) for an affine Edwards point, for instance.

The implementations of scalar multiplication require the scalar to be represented as a bit sequence. We therefore allow the notation $[k>]\, P$ meaning $[\mathsf{LEBS2IP}_{\mathsf{length}(k>)}(k>)]\, P$. There will be no ambiguity because variables representing bit sequences are named with a > suffix.

The Montgomery curve M has parameters $A_{\mathsf{M}} = 40962$ and $B_{\mathsf{M}} = 1$. We use an affine representation of this curve with the formula:

$$B_{\mathsf{M}} \cdot y^2 = x^3 + A_{\mathsf{M}} \cdot x^2 + x$$

Usually, elliptic curve arithmetic over prime fields is implemented using some form of projective coordinates, in order to reduce the number of expensive inversions required. In the circuit, it turns out that a division can be implemented at the same cost as a multiplication, i.e. one constraint. Therefore it is beneficial to use affine coordinates for both curves.

We define the following types representing affine Edwards and Montgomery coordinates respectively:

$$\mathsf{AffineEdwardsJubjub} := (u ⦂ \mathbb{F}_{r_{\mathbb{S}}}) \times (v ⦂ \mathbb{F}_{r_{\mathbb{S}}}) : a_{\mathsf{J}} \cdot u^2 + v^2 = 1 + d_{\mathsf{J}} \cdot u^2 \cdot v^2$$
$$\mathsf{AffineMontJubjub} := (x ⦂ \mathbb{F}_{r_{\mathbb{S}}}) \times (y ⦂ \mathbb{F}_{r_{\mathbb{S}}}) : B_{\mathsf{M}} \cdot y^2 = x^3 + A_{\mathsf{M}} \cdot x^2 + x$$

We also define a type representing compressed, *not necessarily valid*, Edwards coordinates:

$$\mathsf{CompressedEdwardsJubjub} := (\tilde{u} ⦂ \mathbb{B}) \times (v ⦂ \mathbb{F}_{r_{\mathbb{S}}})$$

See §5.4.8.3 *'Jubjub'* on p. 67 for how this type is represented as a byte sequence in external encodings.

We use affine Montgomery arithmetic in parts of the circuit because it is more efficient, in terms of the number of constraints, than affine Edwards arithmetic.

An important consideration when using Montgomery arithmetic is that the addition formula is not complete, that is, there are cases where it produces the wrong answer. We must ensure that these cases do not arise.

We will need the theorem below about y-coordinates of points on Montgomery curves.

Fact: $A_{\mathsf{M}}{}^2 - 4$ is a nonsquare in $\mathbb{F}_{r_{\mathbb{S}}}$.

Theorem A.2.1. *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:

$$(a) \times (b - c) = (0)$$

A.3.1.3 Selection constraints

A selection constraint $(b \; ? \; x : y) = z$, where b $\mathbb{B}$ has been boolean-constrained, can be implemented as:

$$(b) \times (y - x) = (y - z)$$

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, $a_{\mathsf{inv}} = a^{-1} \pmod{r_\mathbb{S}}$:

$$(a_{\mathsf{inv}}) \times (a) = (1)$$

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 $*$ $\prod_{i=0}^{n-1}$ $*$ 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 $\mathbb{B}$ are already boolean-constrained, can be implemented in one constraint as:

$$(2 \cdot a) \times (b) = (a + b - c)$$

This automatically boolean-constrains c. Its correctness can be seen by checking the truth table of (a, b).

A.3.2 Operations on multiple bits

A.3.2.1 [Un]packing modulo $r_{\mathbb{S}}$

Let $n \in \mathbb{N}^+$ be a constant. The operation of converting a field element, $a \in \mathbb{F}_{r_{\mathbb{S}}}$, to a sequence of boolean variables $b_{0\,..\,n-1} \in \mathbb{B}^{[n]}$ 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:

$$\left(\sum_{i=0}^{n-1} b_i \cdot (2^i \bmod r_{\mathbb{S}})\right) \times \left(1\right) = \left(a\right)$$

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 $a \in \mathbb{F}_{r_{\mathbb{S}}}$ 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 $n \in \mathbb{N}^+$ be a constant, and let $a = \sum_{i=0}^{n-1} a_i \cdot 2^i \in \mathbb{N}$. Suppose we want to constrain $a \leq c$ for some *constant* $c = \sum_{i=0}^{n-1} c_i \cdot 2^i \in \mathbb{N}$.

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 $\Pi_m = \bigwedge_{i=m}^{n-1} (c_i = 0 \vee a_i = 1)$ for $m \in \{0\,..\,n-1\}$. Notice that for any $m < n-1$ such that $c_m = 0$, we have $\Pi_m = \Pi_{m+1}$, and so it is only necessary to allocate separate variables for the Π_m such that $m < n-1$ and $c_m = 1$. Furthermore if $c_{n-2\,..\,0}$ has $t > 0$ trailing 1 bits, then we do not need to allocate variables for $\Pi_{0\,..\,t-1}$ because those variables will not be used below.

More explicitly:

Let $\Pi_{n-1} = a_{n-1}$.

For i from $n - 2$ down to t,

- if $c_i = 0$, then let $\Pi_i = \Pi_{i+1}$;
- if $c_i = 1$, then constrain $(\Pi_{i+1}) \times (a_i) = (\Pi_i)$.

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. 121.

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. *Assume $c_{0..n-1} \in \mathbb{B}^{[n]}$ and $c_{n-1} = 1$. Debne $A_m := \sum_{i=m}^{n-1} a_i \cdot 2^{i-m}$ and $C_m := \sum_{i=m}^{n-1} c_i \cdot 2^{i-m}$. 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 satisbed. 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 $(1 - \Pi_{i+1} - a_i) \times (a_i) = (0)$ 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 preZxes 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 fulZls the theorem since $A_{n-1} \leq C_{n-1}$ is always satisZed.

Inductive case $m < n - 1$:

- If $A_{m+1} > C_{m+1}$, then by the inductive hypothesis the constraint system must fail, which fulZls 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 $\Pi_{m+1} = \bigwedge_{i=m+1}^{n-1} (c_i = 0 \vee a_i = 1) = \bigwedge_{i=m+1}^{n-1} (a_i \geq c_i)$.
  - 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 fulZls the theorem.
    When $c_m = 0$, a_m is constrained to be 0 which fulZls 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 fulZls 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 *'Edwards [de]compression and validation'* on p. 124 uses range checks with $c = r_{\mathbb{S}} - 1$ to validate compressed Edwards points. 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 $\Pi_{\ell\,..\,n-2}$ further. Notice that Π_m is only used when m is the index of the last bit of a run of 1 bits in c. So for each such run of 1 bits $c_{m\,..\,m+N-2}$ of length $N-1$, it is sufficient to compute an N-ary AND of $a_{m\,..\,m+N-2}$ and Π_{m+N-1}: $R = \prod_{i=0}^{N-1} X_i$. This can be computed in 3 constraints for any N; boolean-constrain the output R, and then add constraints

$$\left(N - \sum_{i=0}^{N-1} X_i\right) \times \left(\mathsf{inv}\right) = \left(1 - R\right) \text{ to enforce that } \sum_{i=0}^{N-1} X_i \neq N \text{ when } R = 0;$$

$$\left(N - \sum_{i=0}^{N-1} X_i\right) \times \left(R\right) = \left(0\right) \text{ to enforce that } \sum_{i=0}^{N-1} X_i = N \text{ when } R = 1.$$

where inv is witnessed as $\left(N - \sum_{i=0}^{N-1} X_i\right)^{-1}$ if $R = 0$ or is unconstrained otherwise. (Since $N < r_{\mathbb{S}}$, the sums cannot overflow.)

In fact the last constraint is not needed in this context because it is sufficient to compute an upper bound on each Π_m (i.e. it does not benefit a malicious prover to witness $R = 1$ when the result of the AND should be 0). So the cost of computing Π variables for an arbitrarily long run of 1 bits can be reduced to 2 constraints. For example, for $c = r_{\mathbb{S}} - 1$ the overall cost would be reduced to $255 + 68 = 323$ constraints.

These optimizations are not used in **Sapling**.

A.3.3 Elliptic curve operations

A.3.3.1 Checking that affine Edwards coordinates are on the curve

To check that (u, v) is a point on the Edwards curve, the **Sapling** circuit uses 4 constraints:

$$\begin{aligned}
(u) \times (u) &= (uu) \\
(v) \times (v) &= (vv) \\
(uu) \times (vv) &= (uuvv) \\
(a_{\mathbb{J}} \cdot uu + vv) \times (1) &= (1 + d_{\mathbb{J}} \cdot uuvv)
\end{aligned}$$

Non-normative note: The last two constraints can be combined into $(d_{\mathbb{J}} \cdot uu) \times (vv) = (a_{\mathbb{J}} \cdot uu + vv - 1)$. The **Sapling** circuit does not use this optimization.

A.3.3.2 Edwards [de]compression and validation

Define $\mathsf{DecompressValidate} ⦂ \mathsf{CompressedEdwardsJubjub} \to \mathsf{AffineEdwardsJubjub}$ as follows:

```
$\mathsf{DecompressValidate}(\tilde{u}, v)$ :
    // Prover supplies the $u$-coordinate.
    Let $u ⦂ \mathbb{F}_{r_{\mathbb{S}}}$.
```

// §A.3.3.1 *'Checking that affine Edwards coordinates are on the curve'* on p. 124.

Check that (u, v) is a point on the Edwards curve.

// §A.3.2.1 *'[Un]packing modulo $r_{\mathbb{S}}$'* on p. 122.

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. 122.

Check that $\sum_{i=0}^{254} u_i \cdot 2^i \leq r_{\mathbb{S}} - 1$.

Return (u, v).

This costs 4 constraints for the curve equation check, 1 constraint for the unpacking, and 387 constraints for the range check (as computed in §A.3.2.2 *'Range check'* on p. 122) 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.

Note: The point-on-curve check could be omitted if (u, v) were already known to be on the curve. However, the **Sapling** circuit never omits it; this provides a consistency check on the elliptic curve arithmetic.

A.3.3.3 Edwards ↔ Montgomery conversion

Define EdwardsToMont : AffineEdwardsJubjub → AffineMontJubjub as follows:

$$\mathsf{EdwardsToMont}(u, v) = \left(\frac{1 + v}{1 - v}, \sqrt{-40964} \cdot \frac{1 + v}{(1 - v) \cdot u}\right) \quad [1 - v \neq 0 \text{ and } u \neq 0]$$

Define MontToEdwards : AffineMontJubjub → AffineEdwardsJubjub as follows:

$$\mathsf{MontToEdwards}(x, y) = \left(\sqrt{-40964} \cdot \frac{x}{y}, \frac{x - 1}{x + 1}\right) \quad [x + 1 \neq 0 \text{ and } y \neq 0]$$

Either of these conversions can be implemented by the same *quadratic constraint program*:

$$\left(y\right) \times \left(u\right) = \left(\sqrt{-40964} \cdot x\right)$$
$$\left(x + 1\right) \times \left(v\right) = \left(x - 1\right)$$

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. *Let (u, v) be an affine point on a complete twisted Edwards curve $E_{\mathsf{Edwards}(a,d)}$. 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. *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 complete twisted Edwards curve. Then $x + 1 \neq 0$, and the only point (x, y) with $y = 0$ is $(0, 0)$ of order 2.*

Proof. That the only point with $y = 0$ is $(0, 0)$ is proven by Theorem A.2.1 on p. 120.

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 complete twisted Edwards curve birationally equivalent to the Montgomery curve (see [BL2017, section 4.3.5]). For all complete twisted Edwards curves, d is nonsquare, so this equation has no solutions for y, hence $x + 1 \neq 0$. □

(When the theorem is applied with $E_{\mathsf{Mont}(A,B)} = \mathbb{M}$ defined in §A.2 *'Elliptic curve background'* on p. 119, the complete twisted Edwards 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:

$$x_3 = B_{\mathbb{M}} \cdot \lambda^2 - A_{\mathbb{M}} - x_1 - x_2$$
$$y_3 = (x_1 - x_3) \cdot \lambda - y_1$$
$$\text{where } \lambda = \begin{cases} \dfrac{3 \cdot x_1^2 + 2 \cdot A_{\mathbb{M}} \cdot x_1 + 1}{2 \cdot B_{\mathbb{M}} \cdot y_1}, & \text{if } x_1 = x_2 \\ \dfrac{y_2 - y_1}{x_2 - x_1}, & \text{otherwise.} \end{cases}$$

The following theorem helps to determine when these incomplete addition formulae can be safely used:

Theorem A.3.4. *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 $\left[-\frac{s-1}{2}\,..\,\frac{s-1}{2}\right] \setminus \{0\}$. 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*

$$\left(x_2 - x_1\right) \times \left(\lambda\right) = \left(y_2 - y_1\right)$$
$$\left(B_{\mathbb{M}} \cdot \lambda\right) \times \left(\lambda\right) = \left(A_{\mathbb{M}} + x_1 + x_2 + x_3\right)$$
$$\left(x_1 - x_3\right) \times \left(\lambda\right) = \left(y_3 + y_1\right)$$

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 $k \in \left[-\frac{s-1}{2}\,..\,\frac{s-1}{2}\right] \mapsto [k]\,Q \in \mathbb{M}$ 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:

$$\left(x\right) \times \left(x\right) = \left(xx\right)$$
$$\left(2 \cdot B_{\mathbb{M}} \cdot y\right) \times \left(\lambda\right) = \left(3 \cdot xx + 2 \cdot A_{\mathbb{M}} \cdot x + 1\right)$$
$$\left(B_{\mathbb{M}} \cdot \lambda\right) \times \left(\lambda\right) = \left(A_{\mathbb{M}} + 2 \cdot x + x_3\right)$$
$$\left(x - x_3\right) \times \left(\lambda\right) = \left(y_3 + y\right)$$

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. 120.

A.3.3.5 Affine-Edwards arithmetic

Formulae for affine-Edwards addition are given in [BBJLP2008, section 6]. With a change of variable names to match our convention, the formulae for $(u_1, v_1) + (u_2, v_2) = (u_3, v_3)$ are:

$$u_3 = \frac{u_1 \cdot v_2 + v_1 \cdot u_2}{1 + d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2}$$
$$v_3 = \frac{v_1 \cdot v_2 - a_{\mathbb{J}} \cdot u_1 \cdot u_2}{1 - d_{\mathbb{J}} \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2}$$

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]:

$$\left(u_1 + v_1\right) \times \left(v_2 - a_J \cdot u_2\right) = \left(T\right)$$
$$\left(u_1\right) \times \left(v_2\right) = \left(A\right)$$
$$\left(v_1\right) \times \left(u_2\right) = \left(B\right)$$
$$\left(d_J \cdot A\right) \times \left(B\right) = \left(C\right)$$
$$\left(1 + C\right) \times \left(u_3\right) = \left(A + B\right)$$
$$\left(1 - C\right) \times \left(v_3\right) = \left(T - A + a_J \cdot B\right)$$

The correctness of this implementation can be seen by expanding $T - A + a_J \cdot B$:

$$\begin{aligned} T - A + a_J \cdot B &= (u_1 + v_1) \cdot (v_2 - a_J \cdot u_2) - u_1 \cdot v_2 + a_J \cdot v_1 \cdot u_2 \\ &= v_1 \cdot v_2 - a_J \cdot u_1 \cdot u_2 + u_1 \cdot v_2 - a_J \cdot v_1 \cdot u_2 - u_1 \cdot v_2 + a_J \cdot v_1 \cdot u_2 \\ &= v_1 \cdot v_2 - a_J \cdot u_1 \cdot u_2 \end{aligned}$$

The above addition formulae are “uniZed”, that is, they can also be used for doubling. AfZne-Edwards doubling $[2]\,(u, v) = (u_3, v_3)$ can also be implemented slightly more efZciently as:

$$\left(u + v\right) \times \left(v - a_J \cdot u\right) = \left(T\right)$$
$$\left(u\right) \times \left(v\right) = \left(A\right)$$
$$\left(d_J \cdot A\right) \times \left(A\right) = \left(C\right)$$
$$\left(1 + C\right) \times \left(u_3\right) = \left(2 \cdot A\right)$$
$$\left(1 - C\right) \times \left(v_3\right) = \left(T + (a_J - 1) \cdot A\right)$$

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 Afbne-Edwards nonsmall-order check

In order to avoid small-subgroup attacks, we check that certain points used in the circuit are not of small order. In practice the **Sapling** circuit uses this in combination with a check that the coordinates are on the curve (§A.3.3.1 *‘Checking that affine Edwards coordinates are on the curve’* on p. 124), 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-Edwards arithmetic’* on p. 126) and checks that the resulting u-coordinate is not 0 (as in §A.3.1.4 *‘Nonzero constraints’* on p. 121).

On a twisted Edwards curve, only the zero poin$\mathcal{O}_J$, 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 $\mathcal{O}_J$.

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 mentioned in §A.3.3.1 *'Checking that affine Edwards coordinates are on the curve'* on p. 124; 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-Edwards scalar multiplication

If the base point B is fixed for a given scalar multiplication $[k]\,B$, we can fully precompute window tables for each window position.

It is most efficient to use 3-bit fixed windows. Since the length of $r_{\mathbb{J}}$ is 252 bits, we need 84 windows.

Express k in base 8, i.e. $k = \sum_{i=0}^{83} k_i \cdot 8^i$.

Then $[k]\,B = \sum_{i=0}^{83} w_{(B,\, i,\, k_i)}$, where $w_{(B,\, i,\, k_i)} = [k_i \cdot 8^i]\,B$.

We precompute all of $w_{(B,\, i,\, s)}$ for $i \in \{0 \,..\, 83\}$, $s \in \{0 \,..\, 7\}$.

To look up a given window entry $w_{(B,\, i,\, s)} = (u_s, v_s)$, where $s = 4 \cdot s_2 + 2 \cdot s_1 + s_0$, we use:

$$(s_1) \times (s_2) = (s_{\&})$$

$$(s_0) \times (-u_0 \cdot s_{\&} + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_{\&} - u_2 \cdot s_1 + u_4 \cdot s_{\&} - u_4 \cdot s_2 - u_6 \cdot s_{\&} + u_1 \cdot s_{\&} - u_1 \cdot s_2 - u_1 \cdot s_1 + u_1 - u_3 \cdot s_{\&} + u_3 \cdot s_1 - u_5 \cdot s_{\&} + u_5 \cdot s_2 + u_7 \cdot s_{\&}) =$$
$$(u_s - u_0 \cdot s_{\&} + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_{\&} - u_2 \cdot s_1 + u_4 \cdot s_{\&} - u_4 \cdot s_2 - u_6 \cdot s_{\&})$$

$$(s_0) \times (-v_0 \cdot s_{\&} + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_{\&} - v_2 \cdot s_1 + v_4 \cdot s_{\&} - v_4 \cdot s_2 - v_6 \cdot s_{\&} + v_1 \cdot s_{\&} - v_1 \cdot s_2 - v_1 \cdot s_1 + v_1 - v_3 \cdot s_{\&} + v_3 \cdot s_1 - v_5 \cdot s_{\&} + v_5 \cdot s_2 + v_7 \cdot s_{\&}) =$$
$$(v_s - v_0 \cdot s_{\&} + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_{\&} - v_2 \cdot s_1 + v_4 \cdot s_{\&} - v_4 \cdot s_2 - v_6 \cdot s_{\&})$$

For a full-length (252-bit) scalar this costs 3 constraints for each of 84 window lookups, plus 6 constraints for each of 83 Edwards additions (as in §A.3.3.5 *'Affine-Edwards arithmetic'* on p. 126), 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. 133 uses a 64-bit scalar for the v input to $\mathsf{ValueCommit}$, 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. 132 uses a 32-bit scalar for the pos input to $\mathsf{MixingPedersenHash}$, 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. 129. However since there are only three instances of fixed-base scalar multiplication in the *Spend circuit* and two in the *Output circuit* [5], the additional complexity was not considered justified for **Sapling**.

[5] A Pedersen commitment uses fixed-base scalar multiplication as a subcomponent.

- 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_{T} in the lookup for the most signiZcant 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 afbne-Edwards scalar multiplication

When the base point B is not Zxed, the method in the preceding section cannot be used. Instead we use a naïve double-and-add method.

$$= \sum^{250}_{i}$$

```
Given
    // $\mathsf{Base}_i = [2^i]\,B$
    let $\mathsf{Base}_0 = B$
    let $\mathsf{Acc}^u_0 = k_0\ ?\ \mathsf{Base}^u_0 : 0$
    let $\mathsf{Acc}^v_0 = k_0\ ?\ \mathsf{Base}^v_0 : 1$
    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 $\mathsf{Addend}^u_i = k_i\ ?\ \mathsf{Base}^u_i : 0$
        let $\mathsf{Addend}^v_i = k_i\ ?\ \mathsf{Base}^v_i : 1$
        let $\mathsf{Acc}_i = \mathsf{Acc}_{i-1} + \mathsf{Addend}_i$
    let $R = \mathsf{Acc}_{250}$.
```

This costs 5 constraints for each of 250 Edwards doublings, 6 constraints for each of 250 Edwards additions, and 2 constraints for each of 251 point selections, for a total of 3252 constraints.

Non-normative note: It would be more efZcient to use 2-bit Zxed windows, and/or to use arithmetic on the Montgomery curve in a similar way to §A.3.3.9 *'Pedersen hash'* on p. 129. 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 justiZed for **Sapling**.

A.3.3.9 Pedersen hash

The speciZcation of the *Pedersen hashes* used in **Sapling** is given in §5.4.1.7 *'Pedersen Hash Function'* on p. 53. 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 Digits {=− 4 ..}4 {0 }. This allows a more efZcient lookup of the window entry for each chunk than if the set{1 .. 8 had been used, because a point can be conditionally negated using only a single constraint.

Next, we optimize the cost of point addition by allowing as many additions as possible to be performed on the Montgomery curve. An incomplete Montgomery addition costs 3 constraints, in comparison with an Edwards addition which costs 6constraints.

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. 126 is met. This requires splitting the input into segments (each using an independent generator), calculating an intermediate result for each segment, and then converting to the Edwards curve and summing the intermediate results using Edwards addition.

Abstracting away the changes of curve, this calculation can be written as:

$$\mathsf{PedersenHashToPoint}(D, M) = \sum_{j=1}^{N} [\langle M_j \rangle]\, \mathcal{I}^D_j$$

where $\langle\cdot\rangle$ and $\mathcal{I}^D_j$ are defined as in §5.4.1.7 *'Pedersen Hash Function'* on p. 53.

We have to prove that:

- the Montgomery-to-Edwards 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. 54 showed that all indices of addition inputs are in the range $\left\{ -\frac{r_{\mathbb{J}}-1}{2} \,..\, \frac{r_{\mathbb{J}}-1}{2} \right\} \setminus \{0\}$.

Because the $\mathcal{I}^D_j$ (which are outputs of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$) are all of prime order, and $\langle M_j \rangle \neq 0 \pmod{r_{\mathbb{J}}}$, it is guaranteed that all of the terms $[\langle M_j \rangle]\, \mathcal{I}^D_j$ to be converted to Edwards form are of prime order. From Theorem A.3.3 on p. 125, 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. *For all disjoint nonempty subsets S and S' of $\{1 \,..\, c\}$, all $m \in \mathbb{B}^{[3][c]}$, and all $\Theta \in \{-1, 1\}$:*

$$\sum_{j \in S} \mathsf{enc}(m_j) \cdot 2^{4\cdot(j-1)} \neq \Theta \cdot \sum_{j' \in S'} \mathsf{enc}(m_{j'}) \cdot 2^{4\cdot(j'-1)}.$$

Proof. Suppose for a contradiction that S, S', m, Θ is a counterexample. Taking the multiplication by Θ on the right hand side inside the summation, we have:

$$\sum_{j \in S} \mathsf{enc}(m_j) \cdot 2^{4\cdot(j-1)} = \sum_{j' \in S'} \Theta \cdot \mathsf{enc}(m_{j'}) \cdot 2^{4\cdot(j'-1)}.$$

Define $\mathsf{enc}' : \{-1, 1\} \times \mathbb{B}^{[3]} \to \{0 \,..\, 8\} \setminus \{4\}$ as $\mathsf{enc}'_\theta(m_i) := 4 + \theta \cdot \mathsf{enc}(m_i)$.

Let $\Delta = \sum_{i=1}^{c} 4 \cdot 2^{4\cdot(i-1)}$ as in the proof of Theorem 5.4.1 on p. 54. By adding Δ to both sides, we get

$$\sum_{j \in S} \mathsf{enc}'_1(m_j) \cdot 2^{4\cdot(j-1)} + \sum_{j \in \{1..c\} \setminus S} 4 \cdot 2^{4\cdot(j-1)} = \sum_{j' \in S'} \mathsf{enc}'_\Theta(m_{j'}) \cdot 2^{4\cdot(j'-1)} + \sum_{j' \in \{1..c\} \setminus S'} 4 \cdot 2^{4\cdot(j'-1)}$$

where all of the $\mathsf{enc}'_1(m_j)$ and $\mathsf{enc}'_\Theta(m_{j'})$ are in $\{0 \,..\, 8\} \setminus \{4\}$.

Each term on the left and on the right affects the single hex digit indexed by j and j' respectively. Since S and S' are disjoint subsets of $\{1 \,..\, c\}$ and S is nonempty, $S \cap (\{1 \,..\, c\} \setminus S')$ is nonempty. Therefore the left hand side has at least one hex digit not equal to 4 such that the corresponding right hand side digit is 4; contradiction. $\square$

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. 126. (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 $s_\& = s_0 \mathbin{\&} s_1$:

$$\left(s_0\right) \times \left(s_1\right) = \left(s_\&\right)$$

Let $(x_k, y_k) = [k]\, P$ for $k \in \{1\,..\,4\}$. Define each coordinate of $(x_S, y_R) = [1 + s_0 + 2 \cdot s_1]\, P$ as a linear combination of s_0, s_1, and $s_\&$:

$$\begin{aligned} &\text{let } x_S = x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_\& \\ &\text{let } y_R = y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\& \end{aligned}$$

We implement the conditional negation as $\left(2 \cdot y_R\right) \times \left(s_2\right) = \left(y_R - y_S\right)$. After substitution of y_R this becomes:

$$\begin{aligned} &\left(2 \cdot (y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\&)\right) \times \left(s_2\right) = \\ &\qquad \left(y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_\& - y_S\right) \end{aligned}$$

Then we substitute x_S into the Montgomery addition constraints from §A.3.3.4 *'Affine-Montgomery arithmetic'* on p. 126, as follows:

$$\begin{aligned} &\left(x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_\& - x_A\right) \times \left(\lambda\right) = \left(y_S - y_A\right) \\ &\left(B_{\mathsf{M}} \cdot \lambda\right) \times \left(\lambda\right) = \left(A_{\mathsf{M}} + x_A + x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_\& + x_C\right) \\ &\left(x_A - x_C\right) \times \left(\lambda\right) = \left(y_C + y_A\right) \end{aligned}$$

(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 A bits (where A includes the fixed bits) is as follows. The number of chunks is $c = \mathsf{ceiling}\left(\frac{A}{3}\right)$ and the number of segments is $n = \mathsf{ceiling}\left(\frac{A}{3 \cdot 63}\right)$.

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-Edwards conversions;
- $6 \cdot (n - 1)$ constraints for Edwards 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 $A = 516$, 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 $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ (§5.4.7.2 *'Windowed Pedersen commitments'* on p. 63), $A = 6 + \ell_{\mathsf{value}} + 2 \cdot \ell_{\mathsf{J}} = 582$, $c = 194$, 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 cm and pos in §4.14 *‘Note Commitments and Nullifiers’* on p. 39. It takes as input a *Pedersen commitment* P , and hashes it with another input x.

Let $\mathcal{J}$ be as deZned in §5.4.1.8 *‘Mixing Pedersen Hash Function’* on p. 55.

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

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

This costs 92 constraints for the scalar multiplication (§A.3.3.7 *‘Fixed-base affine-Edwards scalar multiplication’* on p. 128), and 6 constraints for the Edwards addition (§A.3.3.5 *‘Affine-Edwards arithmetic’* on p. 126), for a total of 98 constraints.

A.3.4 Merkle path check

Checking each layer of a Merkle authentication path, as described in §4.8 *‘Merkle path validity’* on p. 34, 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. 122; that is, it is *not* necessary to ensure that the left or right inputs to the hash represent integers in the range $\{0\,..\,r_{\mathbb{S}} - 1\}$. Since the root of the Merkle tree is calculated outside the circuit using the canonical representations, and since the *Pedersen hashes* are collision-resistant on arbitrary bit-sequence inputs, an attempt by an adversarial prover to use a non-canonical input would result in the wrong root being calculated, and the overall path check would fail.

For each layer, the cost is $1 + 2\ 255$ 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. 129), 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. 129, and adding a randomized point:

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

This can be implemented in:

- $5 \cdot c + 5 \cdot n - 6$ constraints for the Pedersen hash applied to $A = 6 + \mathsf{length}(s)$ bits, where $c = \mathsf{ceiling}\left(\frac{A}{3}\right)$ and $n = \mathsf{ceiling}\left(\frac{A}{3 \cdot 63}\right)$;
- 750 constraints for the Zxed-base scalar multiplication;
- 6 constraints for the Znal Edwards addition.

When 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. 129, 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 ValueCommit.

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

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

In the case that we need for ValueCommit, v has 64 bits[6]. 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 Edwards 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 "*tt* function", defined as follows:

$tt : \{0 \,..\, 9\} \times \{0 \,..\, 2^{32} - 1\}^{[4]} \to \{0 \,..\, 2^{32} - 1\}^{[4]}$

$tt(a, b, c, d, x, y) = (a^{rr}, b^{rr}, c^{rr}, d^{rr})$ where

$a^{r} = (a + b + x) \bmod 2^{32}$

$d^{r} = (d \oplus a^{r}) \gg 16$

$c^{r} = (c + d^{r}) \bmod 2^{32}$

$b^{r} = (b \oplus c^{r}) \gg 12$

$a^{rr} = (a^{r} + b^{r} + y) \bmod 2^{32}$

$d^{rr} = (d^{r} \oplus a^{rr}) \gg 8$

$c^{rr} = (c^{r} + d^{rr}) \bmod 2^{32}$

$b^{rr} = (b^{r} \oplus c^{rr}) \gg 7$

[6] It would be sufficient to use 51 bits, which accomodates the range {0 .. MAX_MONEY}, but the **Sapling** circuit uses 64.

The following table is used to determine which message words the x and y arguments to tt are selected from:

$$\begin{array}{lllllllllllllllll}
\sigma_0 = [\,0, & 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10, & 11, & 12, & 13, & 14, & 15\,] \\
\sigma_1 = [\,14, & 10, & 4, & 8, & 9, & 15, & 13, & 6, & 1, & 12, & 0, & 2, & 11, & 7, & 5, & 3\,] \\
\sigma_2 = [\,11, & 8, & 12, & 0, & 5, & 2, & 15, & 13, & 10, & 14, & 3, & 6, & 7, & 1, & 9, & 4\,] \\
\sigma_3 = [\,7, & 9, & 3, & 1, & 13, & 12, & 11, & 14, & 2, & 6, & 5, & 10, & 4, & 0, & 15, & 8\,] \\
\sigma_4 = [\,9, & 0, & 5, & 7, & 2, & 4, & 10, & 15, & 14, & 1, & 11, & 12, & 6, & 8, & 3, & 13\,] \\
\sigma_5 = [\,2, & 12, & 6, & 10, & 0, & 11, & 8, & 3, & 4, & 13, & 7, & 5, & 15, & 14, & 1, & 9\,] \\
\sigma_6 = [\,12, & 5, & 1, & 15, & 14, & 13, & 4, & 10, & 0, & 7, & 6, & 3, & 9, & 2, & 8, & 11\,] \\
\sigma_7 = [\,13, & 11, & 7, & 14, & 12, & 1, & 3, & 9, & 5, & 0, & 15, & 4, & 8, & 6, & 2, & 10\,] \\
\sigma_8 = [\,6, & 15, & 14, & 9, & 11, & 3, & 0, & 8, & 12, & 2, & 13, & 7, & 1, & 4, & 10, & 5\,] \\
\sigma_9 = [\,10, & 2, & 8, & 4, & 7, & 6, & 1, & 5, & 15, & 11, & 9, & 14, & 3, & 12, & 13, & 0\,]
\end{array}$$

The Initialization Vector is defined as:

$$\begin{aligned}\mathsf{IV} ⦂ \{0\,..\,2^{32}-1\}^{[8]} := [\,&\text{0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A}\\ &\text{0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19}\,]\end{aligned}$$

The full hash function applied to an 8-byte personalization string and a single 64-byte block, in sequential mode with 32-byte output, can be expressed as follows.

Define $\mathsf{BLAKE2s\text{-}256} ⦂ (p ⦂ \mathbb{B}^{\mathbb{Y}^{[8]}}) \times (x ⦂ \mathbb{B}^{\mathbb{Y}^{[64]}}) \to \mathbb{B}^{\mathbb{Y}^{[32]}}$ as:

```
let $\mathsf{PB} ⦂ \mathbb{B}^{\mathbb{Y}^{[32]}} = [32, 0, 1, 1] \,||\, [\text{0x00}]^{20} \,||\, p$
let $[\,t_0, t_1, f_0, f_1\,] ⦂ \{0\,..\,2^{32}-1\}^{[4]} = [\,0, 0, 0, \text{0xFFFFFFFF}, 0\,]$
let $h ⦂ \{0\,..\,2^{32}-1\}^{[8]} = [\,\mathsf{LEOS2IP}_{32}(\mathsf{PB}_{4 \cdot i\,..\,4 \cdot i+3}) \oplus \mathsf{IV}_i$ for $i$ from 0 up to 7 $]$
let $v ⦂ \{0\,..\,2^{32}-1\}^{[16]} = h \,||\, [\,\mathsf{IV}_0, \mathsf{IV}_1, \mathsf{IV}_2, \mathsf{IV}_3, t_0 \oplus \mathsf{IV}_4, t_1 \oplus \mathsf{IV}_5, f_0 \oplus \mathsf{IV}_6, f_1 \oplus \mathsf{IV}_7\,]$
let $m ⦂ \{0\,..\,2^{32}-1\}^{[16]} = [\,\mathsf{LEOS2IP}_{32}(x_{4 \cdot i\,..\,4 \cdot i+3})$ for $i$ from 0 up to 15 $]$
for $r$ from 0 up to 9:
    set $(v_0, v_4, v_8, v_{12}) := tt(v_0, v_4, v_8, v_{12}, m_{\sigma_{r,0}}, m_{\sigma_{r,1}})$
    set $(v_1, v_5, v_9, v_{13}) := tt(v_1, v_5, v_9, v_{13}, m_{\sigma_{r,2}}, m_{\sigma_{r,3}})$
    set $(v_2, v_6, v_{10}, v_{14}) := tt(v_2, v_6, v_{10}, v_{14}, m_{\sigma_{r,4}}, m_{\sigma_{r,5}})$
    set $(v_3, v_7, v_{11}, v_{15}) := tt(v_3, v_7, v_{11}, v_{15}, m_{\sigma_{r,6}}, m_{\sigma_{r,7}})$
    set $(v_0, v_5, v_{10}, v_{15}) := tt(v_0, v_5, v_{10}, v_{15}, m_{\sigma_{r,8}}, m_{\sigma_{r,9}})$
    set $(v_1, v_6, v_{11}, v_{12}) := tt(v_1, v_6, v_{11}, v_{12}, m_{\sigma_{r,10}}, m_{\sigma_{r,11}})$
    set $(v_2, v_7, v_8, v_{13}) := tt(v_2, v_7, v_8, v_{13}, m_{\sigma_{r,12}}, m_{\sigma_{r,13}})$
    set $(v_3, v_4, v_9, v_{14}) := tt(v_3, v_4, v_9, v_{14}, m_{\sigma_{r,14}}, m_{\sigma_{r,15}})$
return $\mathsf{LEBS2OSP}_{256}\,\mathsf{concat}_{\mathbb{B}}\,[\,\mathsf{I2LEBSP}_{32}(h_i \oplus v_i \oplus v_{i+8})$ for $i$ from 0 up to 7 $]$ □
```

In practice the message and output will be expressed as bit sequences. In the **Sapling** circuit, the personalization string will be constant for each use.

Each 32-bit exclusive-or is implemented in 32 constraints, one for each bit position $a \oplus b = c$ as in §A.3.1.5 *'Exclusive-or constraints'* on p. 121.

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 $\sum_{i=0}^{31} (a_i + b_i) \cdot 2^i = \sum_{i=0}^{32} c_i \cdot 2^i$.

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 $\sum_{i=0}^{31} (a_i + b_i + m_i) \cdot 2^i = \sum_{i=0}^{33} c_i \cdot 2^i$.

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 $\mathbb{F}_{r_S}$ field element are equated together using one constraint. This allows 7 such checks per constraint.

Each *tt* 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 21262 constraints:

- $10 \cdot 8 \cdot 262 = 20960$ constraints for 80 *tt* evaluations, excluding equality checks;
- $\mathsf{ceiling}\left(\frac{10 \cdot 8 \cdot 4}{7}\right) = 46$ 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 $\mathbb{F}_{r_S}$ 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.4The Sapling Spend circuit

The **Sapling** Spend *statement* is deZned in §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41.

The primary input is

$$\dot{}\ \mathsf{rt} : \mathsf{B}^{[A_{\mathsf{MerkleSapling}}]},$$
$$\mathsf{cv}^{\mathsf{old}} : \mathsf{ValueCommit.Output},$$
$$\mathsf{nf}^{\mathsf{old}} : \mathsf{B}^{[A_{\mathsf{PRFnfSapling}}]},$$
$$\mathsf{rk} : \mathsf{SpendAuthSig.Public}^{\square},$$

which is encoded as 8 $\mathsf{F}_{r_\mathbb{S}}$ elements (starting with the Zxed element 1 required by Groth16):

$$[1, \mathcal{U}(\mathsf{rk}), \mathcal{V}(\mathsf{rk}), \mathcal{U}(\mathsf{cv}^{\mathsf{old}}), \mathcal{V}(\mathsf{cv}^{\mathsf{old}}), \mathsf{LEBS2IP}_{A_{\mathsf{MerkleSapling}}}(\mathsf{rt}), \mathsf{LEBS2IP}_{251}\ \dot{}\ \mathsf{nf}^{\mathsf{old}}_{0\,..\,250}{}^{\square}, \mathsf{LEBS2IP}_{5}\ \dot{}\ \mathsf{nf}^{\mathsf{old}}_{251\,..\,255}{}^{\square}\ ]$$

The auxiliary input is

$$\dot{}\ \mathsf{path} : \mathsf{B}^{[A_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sapling}}]},$$
$$\mathsf{pos} : \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}} - 1\},$$
$$\mathsf{g_d} : \mathsf{J},$$
$$\mathsf{pk_d} : \mathsf{J},$$
$$\mathsf{v}^{\mathsf{old}} : \{0\,..\,2^{A_{\mathsf{value}}} - 1\},$$
$$\mathsf{rcv}^{\mathsf{old}} : \{0\,..\,2^{A_{\mathsf{scalar}}} - 1\},$$
$$\mathsf{cm}^{\mathsf{old}} : \mathsf{J},$$
$$\mathsf{rcm}^{\mathsf{old}} : \{0\,..\,2^{A_{\mathsf{scalar}}} - 1\},$$
$$\alpha : \{0\,..\,2^{A_{\mathsf{scalar}}} - 1\},$$
$$\mathsf{ak} : \mathsf{SpendAuthSig.Public},$$
$$\mathsf{nsk} : \{0\,..\,2^{\ \mathsf{scalar}} - 1\}^{\square}\ .$$

ValueCommit.Output and SpendAuthSig.Public are 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 ValueCommit;
- $\mathsf{cm}^{\mathsf{old}}$ will be constrained to an output of $\mathsf{NoteCommit}^{\mathsf{Sapling}}$;
- rk will be constrained to $[\alpha]\,\mathrm{G} + \mathsf{ak}$;
- $\mathsf{pk_d}$ 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, nk> and ρ> used in **Nulliber integrity** are compressed representations of *Jubjub curve* points. TODO: explain why these are implemented as §A.3.3.2 *'Edwards [de]compression and validation'* on p. 124 even though the statement spec doesn't explicitly say to do validation.

Therefore we have $\mathsf{g_d}$, ak, nk, and ρ that need to be constrained to valid *Jubjub curve* points as described in §A.3.3.2 *'Edwards [de]compression and validation'* on p. 124.

In order to aid in comparing the implementation with the speciZcation, we present the checks needed in the order in which they are implemented in the sapling-crypto code:

Check	Implements	Cost	Reference
ak is on the curve TODO: FIXME also decompressed below	ak ◦ SpendAuthSig.Public	4	§A.3.3.1 on p. 124
ak is not small order	**Small order checks**	16	§A.3.3.6 on p. 127
$\alpha>$ ◦ $\mathsf{B}^{[A_{\mathsf{scalar}}]}$	α ◦ $\{0\,..\,2^{A_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\alpha^{\mathsf{r}} = [\alpha>]\,\mathcal{G}$	**Spend authority**	750	§A.3.3.7 on p. 128
$\mathsf{rk} = \alpha^{\mathsf{r}} + \mathsf{ak}$		6	§A.3.3.5 on p. 126
inputize rk TODO: not ccteddecompressvalidate => wrong count	rk ◦ SpendAuthSig.Public	392?	§A.3.3.2 on p. 124
nsk> ◦ $\mathsf{B}^{[A_{\mathsf{scalar}}]}$	nsk ◦ $\{0\,..\,2^{A_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{nk} = [\mathsf{nsk}>]\,\mathcal{H}$	**Nulliber integrity**	750	§A.3.3.7 on p. 128
$\mathsf{ak}> = \mathsf{repr}_{\mathbb{J}}(\mathsf{ak}$ ◦ $\mathbb{J})$	**Diversibed address integrity**	392	§A.3.3.2 on p. 124
$\mathsf{nk}> = \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$ TODO: spec doesn't say to validate nk since it's calculated	**Nulliber integrity**	392	§A.3.3.2 on p. 124
$\mathsf{ivk}> = \mathsf{I2LEBSP}_{251}\ \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}, \mathsf{nk})$ †	**Diversibed address integrity**	21262	§A.3.7 on p. 133
$\mathsf{g_d}$ is on the curve	$\mathsf{g_d}$ ◦ $\mathbb{J}$	4	§A.3.3.1 on p. 124
$\mathsf{g_d}$ is not small order	**Small order checks**	16	§A.3.3.6 on p. 127
$\mathsf{pk_d} = [\mathsf{ivk}>]\,\mathsf{g_d}$	**Diversibed address integrity**	3252	§A.3.3.8 on p. 129
$\mathsf{v}^{\mathsf{old}}_{>}$ ◦ $\mathsf{B}^{[64]}$	$\mathsf{v}^{\mathsf{old}}$ ◦ $\{0\,..\,2^{64}-1\}$	64	§A.3.1.1 on p. 121
rcv> ◦ $\mathsf{B}^{[A_{\mathsf{scalar}}]}$	rcv ◦ $\{0\,..\,2^{A_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{cv} = \mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}})$	**Value commitment integrity**	947	§A.3.6 on p. 133
inputize cv		?	
rcm> ◦ $\mathsf{B}^{[A_{\mathsf{scalar}}]}$	rcm ◦ $\{0\,..\,2^{A_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g_d}, \mathsf{pk_d}, \mathsf{v}^{\mathsf{old}})$	**Note commitment integrity**	1740	§A.3.5 on p. 132
$\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm})$	**Merkle path validity**	0	
rt^{r} is the root of a Merkle tree with leaf cm_u, and authentication path (path, pos>)		32 ·1380	§A.3.4 on p. 132
$\mathsf{pos}> = \mathsf{I2LEBSP}_{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}(\mathsf{pos})$		1	§A.3.2.1 on p. 122
if $\mathsf{v}^{\mathsf{old}}$ Ç 0 then $\mathsf{rt}^{\mathsf{r}} = \mathsf{rt}$		1	§A.3.1.2 on p. 121
inputize rt		?	
$\rho = \mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})$	**Nulliber integrity**	98	§A.3.3.10 on p. 132
$\rho> = \mathsf{repr}_{\mathbb{J}}\ \rho$ TODO: spec doesn't say to validate ρ since it's calculated		392	§A.3.3.2 on p. 124
$\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}y}(\rho>)$		21262	§A.3.7 on p. 133
pack $\mathsf{nf}^{\mathsf{old}}_{0\,..\,250}$ and $\mathsf{nf}^{\mathsf{old}}_{251\,..\,255}$ into two $\mathbb{F}_{r_{\mathbb{S}}}$ inputs	input encoding	2	§A.3.2.1 on p. 122

† This is implemented by taking the output of BLAKE2s-256 as a bit sequence and dropping the most signiZcant 5 bits, not by converting to an integer and back to a bit sequence as literally speciZed.

Note: The implementation represents α>, nsk>, ivk>, rcm>, rcv>, and $\mathsf{v}^{\mathsf{old}}$ as bit sequences rather than integers.

A.5The Sapling Output circuit

The **Sapling** Output *statement* is deZned in §4.15.3 *'Output Statement (**Sapling**)'* on p. 42.

The primary input is

$$\begin{aligned}&(\mathsf{cv}^{\mathsf{new}} ⦂ \mathsf{ValueCommit.Output},\\ &\ \mathsf{cm}_u ⦂ \mathbb{B} \quad ,\\ &\ \mathsf{epk} ⦂ \mathbb{J} \quad ,\end{aligned}$$

which is encoded as 6 $\mathbb{F}_{r_{\mathbb{S}}}$ elements (starting with the Zxed element 1 required by Groth16):

$$[1, \mathcal{U}(\mathsf{cv}^{\mathsf{new}}), \mathcal{V}(\mathsf{cv}^{\mathsf{new}}), \mathcal{U}(\mathsf{epk}), \mathcal{V}(\mathsf{epk}), \mathsf{LEBS2IP}_{\ell_{\mathsf{MerkleSapling}}}(\mathsf{cm}_u)]$$

The auxiliary input is

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

ValueCommit.Output is $\mathbb{J}$, so we have $\mathsf{cv}^{\mathsf{new}}$, epk, and $\mathsf{g_d}$ that represent *Jubjub curve* points. However,

- $\mathsf{cv}^{\mathsf{new}}$ will be constrained to an output of ValueCommit;
- epk will be constrained to $[\mathsf{esk}]\,\mathsf{g_d}$

so $\mathsf{cv}^{\mathsf{new}}$ and epk do not need to be explicitly checked to be on the curve.

Therefore we have only $\mathsf{g_d}$ that needs to be constrained to a valid *Jubjub curve* point as described in §A.3.3.2 *'Edwards [de]compression and validation'* on p. 124.

Note: $\mathsf{pk}>_{\mathsf{d}}$ 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
$\mathsf{v}^{\mathsf{old}}_{>} : \mathbb{B}^{[64]}$	$\mathsf{v}^{\mathsf{old}} : \{0 .. 2^{64}-1\}$	64	§A.3.1.1 on p. 121
$\mathsf{rcv}_{>} : \mathbb{B}^{[\ell_{\mathsf{scalar}}]}$	$\mathsf{rcv} : \{0 .. 2^{\ell_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{cv} = \mathsf{ValueCommit}_{\mathsf{rcv}}(\mathsf{v}^{\mathsf{old}})$	**Value commitment integrity**	947	§A.3.6 on p. 133
inputize cv		?	
$\mathsf{g}_{>\mathsf{d}} = \mathsf{repr}_{\mathbb{J}}(\mathsf{g}_{\mathsf{d}} : \mathbb{J})$	**Note commitment integrity**	392	§A.3.3.2 on p. 124
g_{d} is not small order	**Small order checks**	16	§A.3.3.6 on p. 127
$\mathsf{esk}_{>} : \mathbb{B}^{[\ell_{\mathsf{scalar}}]}$	$\mathsf{esk} : \{0 .. 2^{\ell_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{epk} = [\mathsf{esk}_{>}]\,\mathsf{g}_{\mathsf{d}}$	**Ephemeral public key integrity**	3252	§A.3.3.8 on p. 129
inputize epk		?	
$\mathsf{pk}_{>\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$	$\mathsf{pk}_{>\mathsf{d}} : \mathbb{B}^{[\ell_{\mathbb{J}}]}$	256	§A.3.1.1 on p. 121
$\mathsf{rcm}_{>} : \mathbb{B}^{[\ell_{\mathsf{scalar}}]}$	$\mathsf{rcm} : \{0 .. 2^{\ell_{\mathsf{scalar}}}-1\}$	252	§A.3.1.1 on p. 121
$\mathsf{cm} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{g}_{\mathsf{d}}, \mathsf{pk}_{\mathsf{d}}, \mathsf{v}^{\mathsf{old}})$	**Note commitment integrity**	1740	§A.3.5 on p. 132
pack inputs		?	

Note: The implementation represents $\mathsf{esk}_{>}$, $\mathsf{pk}_{>\mathsf{d}}$, $\mathsf{rcm}_{>}$, $\mathsf{rcv}_{>}$, and $\mathsf{v}^{\mathsf{old}}_{>}$ as bit sequences rather than integers.

BBatching Optimizations

B.1 RedDSA batch verification

The reference verification algorithm for RedDSA signatures is defined in §5.4.6 *'RedDSA and RedJubjub'* on p. 59.

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 verification of a batch of signatures, i.e. to determine whether all signatures in a batch are valid. Its input is a sequence of N "*batch entries*", each of which is a (public key, message, signature) triple.

Let LEOS2BSP, LEOS2IP, and LEBS2OSP be as defined in §5.2 *'Integers, Bit Sequences, and Endianness'* on p. 48.

Define $\mathsf{RedDSA.BatchEntry} := \mathsf{RedDSA.Public} \times \mathsf{RedDSA.Message} \times \mathsf{RedDSA.Signature}$.

Define $\mathsf{RedDSA.BatchVerify} : (\mathsf{entry}_{0..N-1} : \mathsf{RedDSA.BatchEntry}^{[N]}) \to \mathbb{B}$ 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 $\mathsf{ceiling}(\ell_{\mathbb{G}}/8)$ bytes of $\sigma_j$, and let $\underline{S_j}$ be the remaining $\mathsf{ceiling}(\mathsf{bitlength}(r_{\mathbb{G}})/8)$ bytes.
    Let $R_j = \mathsf{abst}_{\mathbb{G}}(\mathsf{LEOS2BSP}_{\ell_{\mathbb{G}}}(\underline{R_j}))$, and let $S_j = \mathsf{LEOS2IP}_{8 \cdot \mathsf{length}(\underline{S_j})}(\underline{S_j})$.
    Let $\underline{\mathsf{vk}_j} = \mathsf{LEBS2OSP}_{\ell_{\mathbb{G}}}(\mathsf{repr}_{\mathbb{G}}(\mathsf{vk}_j))$.
```

Let $c_j = \mathsf{H}^{\sim}(\underline{R}_j \,\|\, \underline{\mathsf{vk}}_j \,\|\, M_j)$

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
- $[h_{\mathbb{G}}]\left(-\left(\sum_{j=0}^{N-1} (z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\right) \mathcal{P}_{\mathbb{G}} + \sum_{j=0}^{N-1}\left([z_j] R_j + [z_j \cdot c_j \pmod{r_{\mathbb{G}}}] \mathsf{vk}_j\right)\right) = \mathcal{O}_{\mathbb{G}}$,

otherwise 0.

The z_j values **MUST** be chosen independently of the batch entries.

The performance benefit of this approach arises partly from replacing the per-signature scalar multiplication of the base $\mathcal{P}$ 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 (§5.4.6.1 *'Spend Authorization Signature'* on p. 62) and binding signatures (§5.4.6.2 *'Binding Signature'* on p. 62) use different bases $\mathcal{P}_{\mathbb{G}}$. It is straightforward to adapt the above procedure to handle multiple bases; there will be one $\left(-\sum_j (z_j \cdot S_j) \pmod{r_{\mathbb{G}}}\right) \mathcal{P}$ term for each base $\mathcal{P}$. The benefit of this relative to using separate batches is that the multiscalar multiplication can be extended across a larger batch.

B.2 Groth16 batch verification

The reference verification algorithm for Groth16 proofs is defined in §5.4.9.2 *'Groth16'* on p. 70.

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 §5.4.8.2 *'BLS12-381'* on p. 66.

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:

$$\hat{e}_{\mathbb{S}}(P, Q) = \mathsf{FinalExp}_{\mathbb{S}}\left(\mathsf{MillerLoop}_{\mathbb{S}}(P, Q)\right)$$

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 consists of 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 $a_{0\,..\,\ell} \,⦂\, \mathbb{F}^{[\ell+1]}_{r_{\mathbb{S}}}$ requires checking the equation

$$\hat{e}_{\mathbb{S}}(\pi_A, \pi_B) \stackrel{?}{=} \hat{e}_{\mathbb{S}}(\pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\left(\sum_{i=0}^{\ell} [a_i] \Psi_i, \Gamma\right) \cdot Y$$

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:

$$\hat{e}_{\mathbb{S}}(\pi_A, -\pi_B) \cdot \hat{e}_{\mathbb{S}}(\pi_C, \Delta) \cdot \hat{e}_{\mathbb{S}}\left(\sum_{i=0}^{\ell} [a_i] \Psi_i, \Gamma\right) \cdot Y \stackrel{?}{=} \mathbf{1}_{\mathbb{S}}.$$

Raising to the power of random $z \neq 0$ gives:

$$\hat{e}_{\mathbb{S}}\big([z]\,\pi_A, -\pi_B\big) \cdot \hat{e}_{\mathbb{S}}\big([z]\,\pi_C, \Delta\big) \cdot \hat{e}_{\mathbb{S}}\Big(\sum_{i=0}^{A}[z \cdot a_i]\,\Psi_i, \Gamma\Big) \cdot Y^{z} = \mathbf{1}_{\mathbb{S}}.$$

This justifies the following optimized procedure for performing faster verification of a batch of Groth16$_{\mathbb{S}}$ proofs. Implementations **MAY** use this procedure to determine whether all proofs in a batch are valid.

Define Groth16$_{\mathbb{S}}$.BatchEntry := Groth16$_{\mathbb{S}}$.Proof × Groth16$_{\mathbb{S}}$.PrimaryInput.

Define Groth16$_{\mathbb{S}}$.BatchVerify $:$ (entry$_{0\,..\,N-1}$ $\circ$ Groth16$_{\mathbb{S}}$.BatchEntry$^{[N]}$) → $\mathbb{B}$ as:

```
For each $j \in \{0 .. N-1\}$:
    Let $((\pi_{j,A}, \pi_{j,B}, \pi_{j,C}), a_{j,\,0\,..\,A}) = \mathsf{entry}_j$.
    Choose random $z_j \circ \mathbb{F}^*_{r_{\mathbb{G}}} \xleftarrow{R} \{1 .. 2^{128} - 1\}$.

Let $\mathsf{Accum}_{AB} = \prod_{j=0}^{N-1} \mathsf{MillerLoop}_{\mathbb{S}}\big([z_j]\,\pi_{j,A}, -\pi_{j,B}\big)$.
Let $\mathsf{Accum}_{\Delta} = \sum_{j=0}^{N-1} [z_j]\,\pi_{j,C}$.
Let $\mathsf{Accum}_{\Gamma,i} = \sum_{j=0}^{N-1} (z_j \cdot a_{j,i}) \pmod{r_{\mathbb{S}}}$ for $i \in \{0 .. A\}$.
Let $\mathsf{Accum}_{Y} = \sum_{j=0}^{N-1} z_j \pmod{r_{\mathbb{S}}}$.

Return 1 if
    $\mathsf{FinalExp}_{\mathbb{S}}\Big(\mathsf{Accum}_{AB} \cdot \mathsf{MillerLoop}_{\mathbb{S}}\big(\mathsf{Accum}_{\Delta}, \Delta\big) \cdot \mathsf{MillerLoop}_{\mathbb{S}}\Big(\sum_{i=0}^{A}[\mathsf{Accum}_{\Gamma,i}]\,\Psi_i, \Gamma\Big)\Big) \cdot Y^{\mathsf{Accum}_Y} = \mathbf{1}_{\mathbb{S}}$,
otherwise 0.
```

The z_j values **MUST** be chosen independently of the 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 §4.15.2 *'Spend Statement (**Sapling**)'* on p. 41) and output proofs (of the *statement* in §4.15.3 *'Output Statement (**Sapling**)'* on p. 42) use different verification keys, with different parameters Δ, Γ, Y, and $\Psi_{0\,..\,A}$. It is straightforward to adapt the above procedure to handle multiple verification keys; the accumulator variables Accum_{Δ}, $\mathsf{Accum}_{\Gamma,i}$, and Accum_{Y} are duplicated, with one term in the verification equation for each variable, while Accum_{AB} is shared.

Neglecting multiplications in $\mathbb{S}_T^{(r)}$ and $\mathbb{F}_{r_{\mathbb{S}}}$, and other trivial operations, the cost of batched verification is therefore

- for each proof: the cost of decoding the proof representation to the form Groth16$_{\mathbb{S}}$.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;
- for each verification key: two Miller loops; an exponentiation in $\mathbb{S}_T^{(r)}$; a multiscalar multiplication with N 128-bit terms to compute Accum_{Δ}; and a multiscalar multiplication with $A + 1$ 255-bit terms to compute $\sum_{i=0}^{A}[\mathsf{Accum}_{\Gamma,i}]\,\Psi_i$;
- one final exponentiation.