Abstract. Zcash is an implementation of the Decentralized Anonymous Payment scheme Zerocash, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin with a shielded payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK). It attempted to address the problem of mining centralization by use of the Equihash memory-hard proof-of-work algorithm. This specification defines the Zcash consensus protocol at launch, and after each of the upgrades codenamed Overwinter, Sapling, Blossom, and Heartwood. It is a work in progress. Protocol differences from Zerocash and Bitcoin are also explained.

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

3 Concepts

3.1 Payment Addresses and Keys ................................................. 12
3.2 Notes .................................................................................. 13
   3.2.1 Note Plaintexts and Memo Fields .................................. 14
3.3 The Block Chain ................................................................... 15
3.4 Transactions and Treestates .................................................. 15
3.5 JoinSplit Transfers and Descriptions .................................... 16
3.6 Spend Transfers, Output Transfers, and their Descriptions .... 16
3.7 Note Commitment Trees ....................................................... 17
3.8 Nullifier Sets ....................................................................... 18
3.9 Block Subsidy and Founders’ Reward .................................... 18
3.10 Coinbase Transactions ......................................................... 18

4 Abstract Protocol ................................................................... 18

4.1 Abstract Cryptographic Schemes .......................................... 18
   4.1.1 Hash Functions ............................................................. 18
   4.1.2 Pseudo Random Functions .......................................... 19
   4.1.3 Symmetric Encryption ................................................ 20
   4.1.4 Key Agreement ............................................................ 20
   4.1.5 Key Derivation ............................................................. 20
   4.1.6 Signature ...................................................................... 21
      4.1.6.1 Signature with Re-Randomizable Keys ...................... 22
      4.1.6.2 Signature with Private Key to Public Key Monomorphism 23
   4.1.7 Commitment ................................................................. 24
   4.1.8 Represented Group ........................................................ 25
   4.1.9 Hash Extractor .............................................................. 25
   4.1.10 Group Hash ................................................................. 26
   4.1.11 Represented Pairing ..................................................... 26
   4.1.12 Zero-Knowledge Proving System ................................. 27
4.2 Key Components ................................................................... 28
   4.2.1 Sprout Key Components ............................................... 28
   4.2.2 Sapling Key Components .............................................. 28
4.3 JoinSplit Descriptions ............................................................ 30
4.4 Spend Descriptions ............................................................... 31
4.5 Output Descriptions .............................................................. 32
4.6 Sending Notes ....................................................................... 33
   4.6.1 Sending Notes (Sprout) .................................................. 33
   4.6.2 Sending Notes (Sapling) ................................................ 33
4.7 Dummy Notes ....................................................................... 34
   4.7.1 Dummy Notes (Sprout) .................................................. 34
   4.7.2 Dummy Notes (Sapling) ................................................ 35
4.8 Merkle Path Validity .............................................................. 35
4.9 SIGMASH Transaction Hashing ........................................... 36
5.4.7 Commitment schemes ................................................. 65
  5.4.7.1 Sprout Note Commitments .................................. 65
  5.4.7.2 Windowed Pedersen commitments ......................... 65
  5.4.7.3 Homomorphic Pedersen commitments ..................... 66
5.4.8 Represented Groups and Pairings ................................. 66
  5.4.8.1 BN-254 ...................................................... 66
  5.4.8.2 BLS12-381 ................................................... 68
  5.4.8.3 Jubjub ...................................................... 69
  5.4.8.4 Hash Extractor for Jubjub .................................. 70
  5.4.8.5 Group Hash into Jubjub .................................... 71
5.4.9 Zero-Knowledge Proving Systems ................................. 72
  5.4.9.1 BCTV14 ...................................................... 72
  5.4.9.2 Groth16 ...................................................... 73
5.5 Encodings of Note Plaintexts and Memo Fields ..................... 73
5.6 Encodings of Addresses and Keys .................................. 74
  5.6.1 Transparent Addresses ....................................... 75
  5.6.2 Transparent Private Keys ..................................... 75
  5.6.3 Sprout Payment Addresses .................................... 75
  5.6.4 Sapling Payment Addresses ................................... 76
  5.6.5 Sprout Incoming Viewing Keys .................................. 76
  5.6.6 Sapling Incoming Viewing Keys ................................. 77
  5.6.7 Sapling Full Viewing Keys .................................... 77
  5.6.8 Sprout Spending Keys ......................................... 78
  5.6.9 Sapling Spending Keys ....................................... 78
5.7 BCTV14 zk-SNARK Parameters .................................... 78
5.8 Groth16 zk-SNARK Parameters .................................... 79
5.9 Randomness Beacon ................................................... 79
6 Network Upgrades ......................................................... 79
7 Consensus Changes from Bitcoin ...................................... 81
  7.1 Encoding of Transactions ........................................ 81
  7.2 Encoding of JoinSplit Descriptions ............................... 84
  7.3 Encoding of Spend Descriptions ................................ 85
  7.4 Encoding of Output Descriptions ................................ 85
  7.5 Block Header ..................................................... 86
  7.6 Proof of Work .................................................... 88
    7.6.1 Equihash ..................................................... 88
    7.6.2 Difficulty filter ............................................. 89
    7.6.3 Difficulty adjustment .......................................... 89
    7.6.4 nBits conversion ............................................. 91
    7.6.5 Definition of Work ............................................. 91
  7.7 Calculation of Block Subsidy and Founders’ Reward .......... 91
  7.8 Payment of Founders’ Reward .................................... 92
  7.9 Changes to the Script System .................................... 93
A.3.6 Homomorphic Pedersen Commitment ........................................... 141
A.3.7 BLAKE2s hashes ................................................................. 142
A.4 The Sapling Spend circuit ....................................................... 145
A.5 The Sapling Output circuit ...................................................... 147

B Batching Optimizations ............................................................. 148
B.1 RedDSA batch verification .................................................... 148
B.2 Groth16 batch verification ................................................... 149

List of Theorems and Lemmata ..................................................... 151

Index ......................................................................................... 151
1 Introduction

Zcash is an implementation of the Decentralized Anonymous Payment scheme Zerocash [BCGGMT2014], with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin [Nakamoto2008] with a shielded payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK).

Changes from the original Zerocash are explained in §8 'Differences from the Zerocash paper' on p. 94, and highlighted in magenta throughout the document.

Changes specific to the Overwinter upgrade are highlighted in blue.

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

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

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

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

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

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

This specification is structured as follows:

- Notation — definitions of notation used throughout the document;
- Concepts — the principal abstractions needed to understand the protocol;
- Abstract Protocol — a high-level description of the protocol in terms of ideal cryptographic components;
- Concrete Protocol — how the functions and encodings of the abstract protocol are instantiated;
- Network Upgrades — the strategy for upgrading the Zcash protocol.
- Consensus Changes from Bitcoin — how Zcash differs from Bitcoin at the consensus layer, including the Proof of Work;
- Differences from the Zerocash protocol — a summary of changes from the protocol in [BCGGMT2014].
- Appendix: Circuit Design — details of how the Sapling circuit is defined as a quadratic constraint program.
- Appendix: Batching Optimizations — improvements to the efficiency of verifying multiple signatures and proofs.

1.1 Caution

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

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

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

Value in Zcash is either transparent or shielded. Transfers of transparent value work essentially as in Bitcoin and have the same privacy properties. Shielded value is carried by notes\(^2\), which specify an amount and (indirectly) a shielded payment address, which is a destination to which notes can be sent. As in Bitcoin, this is associated with a private key that can be used to spend notes sent to the address; in Zcash this is called a spending key.

To each note there is cryptographically associated a note commitment. Once the transaction creating a note has been mined, the note is associated with a fixed note position in a tree of note commitments, and with a nullifier\(^2\), unique to that note. Computing the nullifier requires the associated private spending key (or the nullifier deriving key for Sapling notes). It is infeasible to correlate the note commitment or note position with the corresponding nullifier without knowledge of at least this key. An unspent valid note, at a given point on the block chain, is one for which the note commitment has been publically revealed on the block chain prior to that point, but the nullifier has not.

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

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

For each shielded input,

- [Sprout 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 nullifier and note commitment are computed correctly.

and for each shielded output,

- [Sprout 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 nullifier of the output note to collide with the nullifier of any other note.

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

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

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

\(^2\) In Zerocash [BCGGMTV2014], notes were called "coins", and nullifiers were called "serial numbers".
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.

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

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

2 Notation

\( \mathbb{B} \) means the type of bit values, i.e. \{0, 1\}. \( \mathbb{B}^\ell \) means the type of byte values, i.e. \{0..255\}.

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

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

The type of a randomized algorithm is denoted by \( S \to \mathbb{B} \to T \). The domain of a randomized algorithm may be (), indicating that it requires no arguments. Given \( f : S \to \mathbb{B} \to T \) and \( s : S \), sampling a variable \( x : T \) from the output of \( f \) applied to \( s \) is denoted by \( x \sim f(s) \).

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

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

\( T \subseteq U \) indicates that \( T \) is an inclusive subset or subtype of \( U \). \( S \cup T \) means the set union of \( S \) and \( T \).

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

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

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

\( x : T \mapsto_{\neq y} e_x : U \) means \( x : T \mapsto e_x : U \cup \{ y \} \) restricted to the domain \( \{ x : T \mid e_x \neq y \} \) and range \( U \).

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

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

---

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].
\(\mathbb{B}^\ell\) means the type of byte sequences of arbitrary length.

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

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

0x followed by a string of monospace hexadecimal digits means the corresponding integer converted from hexadecimal. \(0x00\)\(^\ell\) means the sequence of \(\ell\) zero bytes.

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

\([0]\)^\ell means the sequence of \(\ell\) zero bits. \([1]\)^\ell means the sequence of \(\ell\) one bits.

\(a..b\), used as a subscript, means the sequence of values with indices \(a\) through \(b\) inclusive. For example, \(a_{\text{new}}^{\text{new}}\) means the sequence \([a_{\text{pk,1}}^{\text{new}}, a_{\text{pk,2}}^{\text{new}}, \ldots, a_{\text{pk,N}}^{\text{new}}]\). (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.

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

concat\(_5\)\((S)\) 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.

sorted\((S)\) means the sequence formed by sorting the elements of \(S\).

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

Where there is a need to make the distinction, we denote the unique representative of a \(\mathbb{F}_n\) in the range \([0..n-1]\) (or the unique representative of a \(\mathbb{F}_n^*\) in the range \([1..n-1]\)) as \(a \mod n\). Conversely, we denote the element of \(\mathbb{F}_n\) corresponding to an integer \(k : \mathbb{Z}\) as \(k \mod n\). We also use the latter notation in the context of an equality \(k = k' \mod n\) as shorthand for \(k \mod n = k' \mod n\), and similarly \(k \neq k' \mod n\) as shorthand for \(k \mod n \neq k' \mod n\). (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. 25) 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 \(G\), \((−a) + a = 0_G\)), 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 \((a/b) \cdot b = a\).

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

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

\[\sum_{i=1}^N a_i\] means the sum of \(a_{1..N}\).

\[\prod_{i=1}^N a_i\] means the product of \(a_{1..N}\).

\[\bigoplus_{i=1}^N a_i\] means the bitwise exclusive-or of \(a_{1..N}\).

When \(N = 0\) these yield the appropriate neutral element, i.e. \(\sum_{i=1}^0 a_i = 0\), \(\prod_{i=1}^0 a_i = 1\), and \(\bigoplus_{i=1}^0 a_i = 0\) or the all-zero bit sequence of length given by the type of \(a\).
\(\sqrt{a}\), where \(a \in \mathbb{F}_q\), means the positive (i.e. in the range \(\{0 \ldots \frac{q-1}{2}\}\)) 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.

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

The \([k] P\) notation for scalar multiplication in a group is defined in § 4.1.8 ‘Represented Group’ on p. 25.

The convention of affixing * 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.

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

\(\text{bitlength}(x)\), for \(x \in \mathbb{N}\), means the smallest integer \(\ell\) such that \(2^\ell > x\).

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

The following integer constants will be instantiated in § 5.3 ‘Constants’ on p. 51:

- \(\text{MerkleDepth}^{\text{Sprout}}, \text{MerkleDepth}^{\text{Sapling}}, \text{N}^{\text{Old}}, \text{N}^{\text{New}}, \ell_{\text{value}}, \ell_{\text{MerkleSprout}}, \ell_{\text{MerkleSapling}}, \ell_{\text{hSig}}, \ell_{\text{PRFSprout}}, \ell_{\text{PRFexpand}}, \ell_{\text{PRFmSapling}}, \ell_{\text{rcm}}, \ell_{\text{Seed}}, \ell_{\text{asure}}, \ell_{\text{asg}}, \ell_{\text{sh}}, \ell_{\text{d}}, \ell_{\text{ivk}}, \ell_{\text{ovk}}, \ell_{\text{scalar}}, \text{MAX\_MONEY}, \text{BlossomActivationHeight}, \text{SlowStartInterval}, \text{PreBlossomHalvingInterval}, \text{MaxBlockSubsidy}, \text{NumFounderAddresses}, \text{PoWLimit}, \text{PoWAveragingWindow}, \text{PoWMedianBlockSpan}, \text{PoWDampingFactor}, \text{PreBlossomPoWTargetSpacing},\) and \(\text{PostBlossomPoWTargetSpacing}\).

The bit sequence constants \(\text{Uncommitted}^{\text{Sprout}} : \mathbb{B}^{\ell_{\text{MerkleSprout}}}\) and \(\text{Uncommitted}^{\text{Sapling}} : \mathbb{B}^{\ell_{\text{MerkleSapling}}}\), and rational constants \(\text{FoundersFraction}\), \(\text{PoWMaxAdjustDown}\), and \(\text{PoWMaxAdjustUp}\) will also be defined in that section.

We use the abbreviation “ctEdwards” to refer to complete twisted Edwards elliptic curves and coordinates (see § 5.4.8.3 ‘Jubjub’ on p. 69).
3 Concepts

3.1 Payment Addresses and Keys

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

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

[Sprout] The *receiving key* $sk.enc$, *incoming viewing key* $ivk = (a.pk, sk.enc)$, and *shielded payment address* $addr_{pk} = (a.pk, pk.enc)$ are derived from the *spending key* $a.sk$, as described in §4.2.1 *Sprout Key Components* on p. 28.

[Sapling onward] An *expanded spending key* is composed of a *Spend authorizing key* $ask$, a *nullifier private key* $nsk$, and an *outgoing viewing key* $ovk$. From these components we can derive an *proof authorizing key* $(ak, nsk)$, a *full viewing key* $(ak, nk, ovk)$, an *incoming viewing key* $ivk$, and a set of *diversified payment addresses* $addr_d = (d, pk_d)$, as described in §4.2.2 *Sapling Key Components* on p. 28.

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

1. Generate a *spending key* $sk$ at random and derive the *expanded spending key* $(ask, nsk, ovk)$ from it, as shown in the diagram above and described in §4.2.2 *Sapling Key Components* on p. 28.
2. Obtain an *extended spending key* as specified in [ZIP-32]; this includes a superset of the components of an *expanded spending key*. This method is used in the context of a Hierarchical Deterministic Wallet.

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

The composition of shielded payment addresses, incoming viewing keys, full viewing keys, and spending keys is a cryptographic protocol detail that should not normally be exposed to users. However, user-visible operations should be provided to obtain a shielded payment address or incoming viewing key or full viewing key from a spending key or extended spending key.
Users can accept payment from multiple parties with a single *shielded payment address* and the fact that these payments are destined to the same payee is not revealed on the *block chain*, even to the paying parties. However if two parties collude to compare a *shielded payment address* they can trivially determine they are the same. In the case that a payee wishes to prevent this they should create a distinct *shielded payment address* for each payer.

[Sapling onward] **Sapling** 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 unlinked 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 (pk$_{enc}$ or pk$_d$) need not be publicly 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. 45 and §4.17 *In-band secret distribution (Sapling)* on p. 65), since an adversary would have to know pk$_{enc}$ or some pk$_d$ in order to exploit a hypothetical weakness in that cryptosystem.

### 3.2 Notes

A *note* (denoted 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 MAX\_MONEY, $\ell_{PRFSprout}$, $\ell_{PRFnsapling}$, and $\ell_d$ be as defined in §5.3 *‘Constants’* on p. 51.

Let $NoteCommit^{Sprout}$ be as defined in §5.4.7.1 *‘Sprout Note Commitments’* on p. 65.

Let $NoteCommit^{Sapling}$ be as defined in §5.4.7.2 *‘Windowed Pedersen commitments’* on p. 65.

Let $KA^{Sapling}$ be as defined in §5.4.4.3 *‘Sapling Key Agreement’* on p. 61.

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

- $a_{pk} : \mathbb{B}^{[\ell_{PRFSprout}]}$ is the *paying key* of the recipient’s *shielded payment address*;
- $v : \{0..MAX\_MONEY\}$ is an integer representing the value of the *note* in *zatoshi* (1 ZEC = $10^8$ zatoshi);
- $\rho : \mathbb{B}^{[\ell_{PRFSprout}]}$ is used as input to PRF$_n$ to derive the *nullifier* of the *note*;
- $rcm : NoteCommit^{Sprout}.\text{Trapdoor}$ is a random commitment trapdoor as defined in §4.17 *‘Commitment’* on p. 24.

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

$$Note_{Sprout} := \mathbb{B}^{[\ell_{PRFSprout}]} \times \{0..MAX\_MONEY\} \times \mathbb{B}^{[\ell_{PRFSprout}]} \times NoteCommit^{Sprout}.\text{Trapdoor}.$$

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

- $d : \mathbb{B}^{[\ell_d]}$ is the *diversifier* of the recipient’s *shielded payment address*;
- $pk_d : KA^{Sapling}.\text{PublicPrimeOrder}$ is the *diversified transmission key* of the recipient’s *shielded payment address*;
- $v : \{0..MAX\_MONEY\}$ is an integer representing the value of the *note* in *zatoshi*;
- $rcm : NoteCommit^{Sapling}.\text{Trapdoor}$ is a random commitment trapdoor as defined in §4.17 *‘Commitment’* on p. 24.
Let $\text{Note}_{\text{Sapling}}$ be the type of a $\text{Sapling}$ note, i.e.

$$\text{Note}_{\text{Sapling}} := \mathbb{B}^{\ell_{\text{value}}} \times \mathbb{A}^{\text{PublicPrimeOrder}} \times \{0..\text{MAX\_MONEY}\} \times \text{NoteCommit}_{\text{Sapling}} \times \text{Trapdoor}.$$  

Creation of new notes is described in § 4.6 ‘Sending Notes’ on p. 33. When notes are sent, only a commitment (see § 4.1.7 ‘Commitment’ on p. 24) to the above values is disclosed publically, and added to a data structure called the note commitment tree. This allows the value and recipient to be kept private, while the commitment is used by the zk-SNARK proof when the note is spent, to check that it exists on the block chain.

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

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

where $\text{NoteCommit}^{\text{Sprout}}_{rcm}$ is instantiated in § 5.4.7.1 ‘Sprout Note Commitments’ on p. 65.

Let DiversifyHash be as defined in § 5.4.1.6 ‘DiversifyHash Hash Function’ on p. 55.

A Sapling note commitment on a note $n = (d, pk_d, v, rcm)$ is computed as

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

$$\text{NoteCommitment}^{\text{Sapling}}(n) := \begin{cases} \perp, & \text{if } g_d = \perp \\
\text{NoteCommit}^{\text{Sapling}}_{rcm}(\text{repr}_2(g_d), \text{repr}_2(pk_d), v), & \text{otherwise.} \end{cases}$$

where $\text{NoteCommit}^{\text{Sapling}}_{rcm}$ is instantiated in § 5.4.7.2 ‘Windowed Pedersen commitments’ on p. 65.

Notice that the above definition of a Sapling note does not have a $\rho$ field. There is in fact a $\rho$ value associated with each Sapling note, but this can only be computed once its position in the note commitment tree is known (see § 3.4 ‘Transactions and Treestates’ on p. 15 and § 3.7 ‘Note Commitment Trees’ on p. 17). We refer to the combination of a note and its note position pos, as a positioned note.

For a positioned note, we can compute the value $\rho$ as described in § 4.14 ‘Note Commitments and Nullifiers’ on p. 41.

A nullifier (denoted $nf$) is derived from the $\rho$ value of a note and the recipient’s spending key $a_{sk}$ or nullifier deriving key $nk$. This computation uses a Pseudo Random Function (see § 4.1.2 ‘Pseudo Random Functions’ on p. 19), as described in § 4.14 ‘Note Commitments and Nullifiers’ on p. 41.

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

### 3.2.1 Note Plaintexts and Memo Fields

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

The note plaintexts in each JoinSplit description are encrypted to the respective transmission keys $pk_{\text{enc}_i..\text{N}_{\text{enc}}}^{\text{new}}$.

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

$$(v : \{0..2^{\ell_{\text{value}}}-1\}, \rho : \mathbb{B}^{2^\ell_{\text{Sprout}}}, rcm : \text{NoteCommit}^{\text{Sprout}}_{rcm} \times \text{Trapdoor}, memo : \mathbb{B}^{512}).$$

[Sapling onward] The note plaintext in each Output description is encrypted to the diversified payment address $(d, pk_d)$.
Each Sapling note plaintext (denoted np) consists of 
\((d : \mathbb{B}^{[d]}, v : \{0..2^{\ell_{\text{val}}}-1\}, \text{rcm} : \mathbb{B}^{[32]}, \text{memo} : \mathbb{B}^{[512]})\).

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.13.

Encodings are given in §5.5 'Encodings of Note Plaintexts and Memo Fields' on p.73. 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.45 and §4.17 'In-band secret distribution (Sapling)' on p.46.

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.86).

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

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

3.4 Transactions and Treestates

Each block contains one or more transactions.

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

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

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

- a note commitment tree (§3.7 'Note Commitment Trees' on p.17);
- a nullifier set (§3.8 'Nullifier Sets' on p.18).

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

An anchor is a Merkle tree root of a note commitment tree (either the Sprout tree or the Sapling tree). It uniquely identifies a note commitment tree state given the assumed security properties of the Merkle tree’s hash function. Since the nullifier set is always updated together with the note commitment tree, this also identifies a particular state of the associated nullifier set.
In a given block chain, for each of Sprout and Sapling, treestates are chained as follows:

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

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

### 3.5 JoinSplit Transfers and Descriptions

A JoinSplit description is data included in a transaction that describes a JoinSplit transfer, i.e. a shielded value transfer. In Sprout, this kind of value transfer was the primary Zcash-specific operation performed by transactions. A JoinSplit transfer spends \( N^{\text{old}} \) notes \( n^{\text{old}}_{1..N^{\text{old}}} \) and transparent input \( v^{\text{old}}_{\text{pub}} \), and creates \( N^{\text{new}} \) notes \( n^{\text{new}}_{1..N^{\text{new}}} \) and transparent output \( v^{\text{new}}_{\text{pub}} \). It is associated with a JoinSplit statement instance (§ 4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42), for which it provides a zk-SNARK proof.

Each transaction has a sequence of JoinSplit descriptions.

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

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

For each of the \( N^{\text{old}} \) shielded inputs, a nullifier is revealed. This allows detection of double-spends as described in §3.8 ‘Nullifier Sets’ on p. 18.

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

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

Consensus rules:

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

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

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

Spend descriptions and Output descriptions are data included in a transaction that describe Spend transfers and Output transfers, respectively.
A Spend transfer spends a note \( n^{\text{old}} \). 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. 43) for which it provides a zk-SNARK proof.

An Output transfer creates a note \( n^{\text{new}} \). 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. 44) 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. 66). 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 \( v_1 + v_2 \). 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. 38) that the result commits to a value consistent with the net transparent value change. This approach allows all of the zk-SNARK statements to be independent of each other, potentially increasing opportunities for precomputation.

A Spend description includes an anchor, which refers to the output Sapling treestate of a previous block. It also reveals a nullifier, which allows detection of double-spends as described in § 3.8 ‘Nullifier Sets’ on p. 18.

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 specified in § 4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38.
- The anchor of each Spend description MUST refer to some earlier block’s final Sapling treestate.

### 3.7 Note Commitment Trees

A note commitment tree is an incremental Merkle tree of fixed depth used to store note commitments that JoinSplit transfers or Spend transfers 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. 15).

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

The index of a note’s commitment at the leafmost layer (MerkleDepth\(_{\text{Sprout,Sapling}}\)) is called its note position.
3.8 Nullifier Sets

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

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

3.9 Block Subsidy and Founders’ Reward

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

The calculations of the block subsidy, miner subsidy, and Founders’ Reward depend on the block height, as defined in § 3.3 ‘The Block Chain’ on p. 15. These calculations are described in § 7.7 ‘Calculation of Block Subsidy and Founders’ Reward’ on p. 91.

3.10 Coinbase Transactions

The first (and only the first) transaction in a block is a coinbase transaction, which collects and spends any miner subsidy and transaction fees paid by transactions included in this block. The coinbase transaction MUST also pay the Founders’ Reward as described in § 7.8 ‘Payment of Founders’ Reward’ on p. 92.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

Let MerkleDepthSprout, ℓMerkleSprout, MerkleDepthSapling, ℓMerkleSapling, ℓSeed, ℓPRFSprout, ℓHSig, and Nold be as defined in § 5.3 ‘Constants’ on p. 51.

Let ℓJ, g(ℓ)∗, rJ, and ℓJ be as defined in § 5.4.8.3 ‘Jubjub’ on p. 69.

The functions MerkleCRH Sprout : \{0..MerkleDepth Sprout − 1\} × \B[ℓMerkleSprout] × \B[ℓPRFSprout] → \B[ℓMerkleSprout] and (for Sapling), MerkleCRH Sapling : \{0..MerkleDepth Sapling − 1\} × \B[ℓMerkleSapling] × \B[ℓPRFSapling] → \B[ℓMerkleSapling] are hash functions used in § 4.8 ‘Merkle Path Validity’ on p. 35. MerkleCRH Sapling is collision-resistant on all its arguments, and MerkleCRH Sprout is collision-resistant except on its first argument.

Both of these functions are instantiated in § 5.4.1.3 ‘Merkle Tree Hash Function’ on p. 53.

hSigCRH : \B[ℓSeed] × \B[ℓPRFSapling][Nold] × JoinSplitSig.Public → \B[ℓHSig] is a collision-resistant hash function used in § 4.3 ‘JoinSplit Descriptions’ on p. 30. It is instantiated in § 5.4.1.4 ‘hSig Hash Function’ on p. 54.

EquihashGen : (n : \N^+) × \N^+ × \B[y][N] × \N^+ → \B[n] is another hash function, used in § 7.6.1 ‘Equihash’ on p. 88 to generate input to the Equihash solver. The first two arguments, representing the Equihash parameters n and k, are written subscripted. It is instantiated in § 5.4.1.9 ‘Equihash Generator’ on p. 58.
CRH⁷\text{pk} : \mathbb{B}^{[\ell]} \times \mathbb{B}^{[\ell]} \rightarrow \{0 \ldots 2^{\ell - 1}\} is a collision-resistant hash function used in §4.2.2 ‘Sapling Key Components’ on p. 28 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. 43) to confirm use of the correct keys for the note being spent. It is instantiated in §5.4.1.5 ‘CRH⁷\text{pk} Hash Function’ on p. 54.

MixingPedersenHash : \mathbb{J} \times \{0 \ldots r_j - 1\} \rightarrow \mathbb{J} is a hash function used in §4.14 ‘Note Commitments and Nullifiers’ on p. 41 to derive the unique \(r\) value for a Sapling note. It is also used in the Spend statement to confirm use of the correct \(r\) value as an input to nullifier derivation. It is instantiated in §5.4.1.8 ‘Mixing Pedersen Hash Function’ on p. 57.

DiversifyHash : \mathbb{B}^{[s]} \rightarrow \mathbb{J}^{(r)^*} is a hash function instantiated in §5.4.1.6 ‘DiversifyHash Hash Function’ on p. 55, 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. 28.

### 4.1.2 Pseudo Random Functions

PRF₂ is a Pseudo Random Function keyed by \(x\).

Let \(\ell_{a_n}, \ell_{q_p}, \ell_{\text{vk}}, \ell_{\text{prfSapling}}, \ell_{\text{sk}}, \ell_{\text{prfExpand}}, \ell_{\text{prfSprout}}, N^{\text{old}}, \) and \(N^{\text{new}}\) be as defined in §5.3 ‘Constants’ on p. 51.

Let \(\ell_j\) and \(\mathbb{J}^{(r)^*}\) be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

Let Sym be as defined in §5.4.3 ‘Symmetric Encryption’ on p. 60.

For Sprout, four independent PRF₂ are needed:

\[
\begin{align*}
\text{PRF}_{\text{addr}}^{\text{Sprout}} & : \mathbb{B}^{[\ell_{a_n}]} \times \mathbb{B}^{[\ell_j]} \rightarrow \mathbb{B}^{[\ell_{\text{prfSprout}}]} \\
\text{PRF}^{\text{nf}} & : \mathbb{B}^{[\ell_{a_n}]} \times \mathbb{B}^{[\ell_{\text{prfSprout}}]} \rightarrow \mathbb{B}^{[\ell_{\text{prfSprout}}]} \\
\text{PRF}^{\text{pk}} & : \mathbb{B}^{[\ell_{a_n}]} \times \{1 \ldots N^{\text{old}}\} \times \mathbb{B}^{[\ell_{\text{sk}}]} \rightarrow \mathbb{B}^{[\ell_{\text{prfSprout}}]} \\
\text{PRF}^{\text{p}} & : \mathbb{B}^{[\ell_{a_n}]} \times \{1 \ldots N^{\text{new}}\} \times \mathbb{B}^{[\ell_{\text{sk}}]} \rightarrow \mathbb{B}^{[\ell_{\text{prfSprout}}]}
\end{align*}
\]

These are used in §4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42; \text{PRF}_{\text{addr}}^{\text{Sprout}} is also used to derive a shielded payment address from a spending key in §4.2.1 ‘Sprout Key Components’ on p. 28.

For Sapling, three additional PRF₂ are needed:

\[
\begin{align*}
\text{PRF}^{\text{expand}} & : \mathbb{B}^{[\ell_{a_n}]} \times \mathbb{B}^{[\ell_j]} \rightarrow \mathbb{B}^{[\ell_{\text{prfExpand}}]} \\
\text{PRF}^{\text{ock}} & : \mathbb{B}^{[\ell_{\text{vek}}]} \times \mathbb{B}^{[\ell_j]} \rightarrow \mathbb{B}^{[\ell_j]} \times \mathbb{B}^{[\ell_j]} \times \mathbb{B}^{[\ell_j]} \rightarrow \text{Sym}.K \\
\text{PRF}^{\text{nfSapling}} & : \mathbb{J}^{(r)^*} \times \mathbb{B}^{[\ell_j]} \rightarrow \mathbb{B}^{[\ell_{\text{prfSapling}}]}
\end{align*}
\]

\text{PRF}^{\text{expand}} is used in §4.2.2 ‘Sapling Key Components’ on p. 28.

\text{PRF}^{\text{ock}} is used in §4.17 ‘In-band secret distribution (Sapling)’ on p. 46.

\text{PRF}^{\text{nfSapling}} is used in §4.15.2 ‘Spend Statement (Sapling)’ on p. 43.

All of these Pseudo Random Functions are instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 58.

**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 PRF₂\(,\) PRF₂\(\text{addr}\), PRF₂\(\text{p}\), and PRF₂\(\text{nfSapling}\) be **collision-resistant** across all \(x\) – i.e. finding \((x, y) \neq (x', y')\) such that \(\text{PRF}_x(x) = \text{PRF}_x(y)\) should not be feasible, and similarly for PRF₂\(\text{addr}\) and PRF₂\(\text{p}\) and PRF₂\(\text{nfSapling}\).

**Non-normative note:** PRF₂\(\text{nf}\) was called PRF₂\(\text{mn}\) in Zerocash [BCGGMTV2014].
4.1.3 Symmetric Encryption

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

\[ \text{Sym.Encrypt} : \text{Sym.K} \times \text{Sym.P} \rightarrow \text{Sym.C} \]

\[ \text{Sym.Decrypt} : \text{Sym.K} \times \text{Sym.C} \rightarrow \text{Sym.P} \cup \{⊥\} \]

The inputs to the Sym.Decrypt algorithm may be an out of bounds ciphertext and the adversary may make many adaptive chosen ciphertext queries for a given key. However, the adversary may make many adaptive chosen ciphertext queries for a given key. One-time encryption must be IND-CPA-secure [AC07]. “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.

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* KA.Public, a type of *private keys* KA.Private, and a type of shared secrets KA.SharedSecret. Optionally, it also defines a type KA.PublicPrimeOrder \( \subseteq \text{KA.Public} \).

Optional: Let KA.FormatPrivate : \{\ell_{\text{PRFSprout}}\} \rightarrow KA.Private be a function to convert a bit string of length \( \ell_{\text{PRFSprout}} \) to a KA private key.

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

Let KA.Agree : KA.Private \times KA.Public \rightarrow KA.SharedSecret be the agreement function.

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

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

Security requirements:

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

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

\[
\text{KDF}^\text{Sprout} : \{1..N^{\text{new}}\} \times \{h_{\text{Sig}}\} \times \text{KA.Sprout}.\text{SharedSecret} \times \text{KA.Sprout}.\text{Public} \times \text{KA.Sprout}.\text{Public} \rightarrow \text{Sym.K}
\]

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

Security requirements:

- The asymmetric encryption scheme in §4.16 ‘In-band secret distribution (Sprout)’ on p. 45, constructed from \( \text{KA}^{\text{Sprout}}, \text{KDF}^{\text{Sprout}} \) 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. 46, constructed from \( \text{KA}^{\text{Sapling}}, \text{KDF}^{\text{Sapling}} \) 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 \( \text{Sig} \) defines:

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

such that for any signing key \( \text{sk} \sim \text{Sig.GenPrivate}() \) and corresponding verifying key \( \text{vk} = \text{Sig.DerivePublic}(\text{sk}) \), and any \( m : \text{Sig.Message} \) and \( s : \text{Sig.Signature} \), \( \text{Sig.Verify}_{\text{vk}}(m, s) = 1 \).

Zcash uses four signature schemes:

- one used for signatures that can be verified by script operations such as \text{OP_CHECKSIG} and \text{OP_CHECKMULTISIG} as in Bitcoin;
- one called \( \text{JoinSplitSig} \) (instantiated in §5.4.5 ‘JoinSplit Signature’ on p. 61), which is used to sign transactions that contain at least one \text{JoinSplit description};
- \[\text{Sapling onward}\] one called \( \text{SpendAuthSig} \) (instantiated in §5.4.6.1 ‘Spend Authorization Signature’ on p. 64) which is used to sign authorizations of \text{Spend transfers};
- \[\text{Sapling onward}\] one called \( \text{BindingSig} \) (instantiated in §5.4.6.2 ‘Binding Signature’ on p. 64), which is used to enforce balance of \text{Spend transfers} and \text{Output transfers}, and to prevent their replay across \text{transactions}.

The following security property is needed for \( \text{JoinSplitSig} \) and \( \text{BindingSig} \). Security requirements for \( \text{SpendAuthSig} \) are defined in the next section, §4.1.6.1 ‘Signature with Re-Randomizable Keys’ on p. 22. An additional requirement for \( \text{BindingSig} \) is defined in §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p. 23.

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

\[^{4}\text{The scheme defined in that paper was attacked in [LM2017], but this has no impact on the applicability of the definition.}\]
Non-normative notes:

- We need separate signing key generation and verifying key derivation algorithms, rather than the more conventional combined key pair generation algorithm $\text{Sig.Gen} : () \xrightarrow{\$} \text{Sig.Private} \times \text{Sig.Public}$, to support the key derivation in §4.2.2 ‘Sapling Key Components’ on p. 28. 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. 22 and §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p. 23.

- 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. 37), a one-time signature scheme would suffice for JoinSplitSig. This is also the reason why only security against non-adaptive chosen message attack is needed. In fact the instantiation of JoinSplitSig uses a scheme designed for security under adaptive attack even when multiple signatures are signed under the same key.

- [Sapling onward] The same remarks as above apply to BindingSig, except that the key is derived from the randomness of value commitments. This results in the same distribution as of freshly generated key pairs, for each transaction containing Spend descriptions or Output descriptions.

- 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 $\text{Sig}$ is a signature scheme that additionally defines:

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

such that:

- for any $\alpha \xrightarrow{\$} \text{Sig.Random}$, $\text{Sig.RandomizePrivate}_{\alpha} : \text{Sig.Private} \rightarrow \text{Sig.Private}$ is injective and easily invertible;
- $\text{Sig.RandomizePrivate}_{\alpha} : \text{Sig.Private} \rightarrow \text{Sig.Private}$ is the identity function on $\text{Sig.Private}$.
- for any $\text{sk} : \text{Sig.Private}$,
  
  $\text{Sig.RandomizePrivate}(\alpha, \text{sk}) : \alpha \xrightarrow{\$} \text{Sig.GenRandom}()$

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

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

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

For any \( sk : \text{Sig.Private} \), let

\[
O_{sk} : \text{Sig.Message} \times \text{Sig.Random} \rightarrow \text{Sig.Signature}
\]

be a signing oracle with state \( Q : \mathcal{P}(\text{Sig.Message} \times \text{Sig.Signature}) \) initialized to \( \{\} \) that records queried messages and corresponding signatures.

\[
O_{sk} := \text{var } Q \leftarrow \{\} \text{ in } (m : \text{Sig.Message}, \alpha : \text{Sig.Random}) \mapsto \\
\quad \text{let } \sigma = \text{Sig.Sign}_{\alpha} \text{Sig.RandomizePrivate}(\alpha, sk)(m) \\
\quad Q \leftarrow Q \cup \{(m, \sigma)\} \\
\quad \text{return } \sigma : \text{Sig.Signature}.
\]

For random \( sk \overset{R}{\leftarrow} \text{Sig.GenPrivate()} \) and \( vk = \text{Sig.DerivePublic}(sk) \), it must be infeasible for an adversary given \( vk \) and a new instance of \( O_{sk} \) to find \( (m', \sigma', \alpha') \) such that \( \text{Sig.Verify}_{\text{Sig.RandomizePublic}(\alpha', vk)}(m', \sigma') = 1 \) and \( (m', \sigma') \not\in O_{sk}.Q \).

Non-normative notes:

\[
\begin{align*}
\quad & \text{The randomizer and key arguments to Sig.RandomizePrivate and Sig.RandomizePublic are swapped relative to} \\
& \quad \text{[FKMSSS2016, section 3].}
\end{align*}
\]

\[
\begin{align*}
\quad & \text{The requirement for the identity randomizer } O_{\text{Sig.Random}} \text{ simplifies the definition of SURK-CMA by removing} \\
& \quad \text{the need for two oracles (because the oracle for original keys, called } O_1 \text{ in [FKMSSS2016], is a special case of} \\
& \quad \text{the oracle for randomized keys).}
\end{align*}
\]

\[
\begin{align*}
\quad & \text{Since } \text{Sig.RandomizePrivate}(\alpha, sk) : \alpha \overset{R}{\leftarrow} \text{Sig.Random} \text{ has an identical distribution to } \text{Sig.GenPrivate}(), \text{ and since} \\
& \quad \text{Sig.DerivePublic is a deterministic function, the combination of a re-randomized public key and signature(s)} \text{ under that key do not reveal the key from which it was re-randomized.}
\end{align*}
\]

\[
\begin{align*}
\quad & \text{Since } \text{Sig.RandomizePrivate} \text{ is injective and easily invertible, knowledge of } \text{Sig.RandomizePrivate}(\alpha, sk) \text{ and } \alpha \text{ implies knowledge of } sk.
\end{align*}
\]

### 4.1.6.2 Signature with Private Key to Public Key Monomorphism

A signature scheme with key monomorphism \( \text{Sig} \) is a signature scheme that additionally defines:

\[
\begin{align*}
\quad & \text{an abelian group on private keys, with operation } \boxplus : \text{Sig.Private} \times \text{Sig.Private} \rightarrow \text{Sig.Private} \text{ and identity } O_{\boxplus} \\
\quad & \text{an abelian group on public keys, with operation } \boxdot : \text{Sig.Public} \times \text{Sig.Public} \rightarrow \text{Sig.Public} \text{ and identity } O_{\boxdot}
\end{align*}
\]

such that for any \( sk_{1,2} : \text{Sig.Private}, \text{Sig.DerivePublic}(sk_{1} \boxplus sk_{2}) = \text{Sig.DerivePublic}(sk_{1}) \boxdot \text{Sig.DerivePublic}(sk_{2}) \).

In other words, \( \text{Sig.DerivePublic} \) is a monomorphism (that is, an injective homomorphism) from the private key group to the public key group.

For \( N : \mathbb{N}^+ \),

\[
\begin{align*}
\quad & \boxplus_{i=1}^{N} \text{ sk}_i \text{ means } \text{sk}_1 \boxplus \text{sk}_2 \boxplus \cdots \boxplus \text{sk}_N : \\
\quad & \boxdot_{i=1}^{N} \text{ vk}_i \text{ means } \text{vk}_1 \boxdot \text{vk}_2 \boxdot \cdots \boxdot \text{vk}_N.
\end{align*}
\]

When \( N = 0 \) these yield the appropriate group identity, i.e. \( \boxplus_{i=1}^{0} \text{ sk}_i = O_{\boxplus} \) and \( \boxdot_{i=1}^{0} \text{ vk}_i = O_{\boxdot} \).

\( \boxplus \text{ sk} \) means the private key such that \( (\boxplus \text{ sk}) \boxplus \text{ sk} = O_{\boxplus} \) and \( \text{sk}_1 \boxplus \text{sk}_2 \) means \( \text{sk}_1 \boxplus (\boxplus \text{sk}_2) \).

\( \boxdot \text{ vk} \) means the public key such that \( (\boxdot \text{ vk}) \boxdot \text{ vk} = O_{\boxdot} \) and \( \text{vk}_1 \boxdot \text{vk}_2 \) means \( \text{vk}_1 \boxdot (\boxdot \text{vk}_2) \).

With a change of notation from \( \mu \) to \( \text{Sig.DerivePublic} \), \( + \) to \( \boxplus \), and \( \cdot \) to \( \boxdot \), this is similar to the definition of a "Signature with Secret Key to Public Key Homomorphism" in [DS2016, Definition 13], except for an additional requirement for the homomorphism to be injective.
**Security requirement:** For any \( \text{sk}_1 : \text{Sig.Private} \) and an unknown \( \text{sk}_2 \overset{\$}{\Leftarrow} \text{Sig.GenPrivate()} \) chosen independently of \( \text{sk}_1 \), the distribution of \( \text{sk}_1 \oplus \text{sk}_2 \) is computationally indistinguishable from that of \( \text{Sig.GenPrivate()} \). (Since \( \oplus \) is an abelian group operation, this implies that for \( n : \mathbb{N}^+ \), \( \oplus_{i=1}^n \text{sk}_i \) is computationally indistinguishable from \( \text{Sig.GenPrivate()} \) when at least one of \( \text{sk}_{1..n} \) 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 \( \text{COMM} \) defines a type of inputs \( \text{COMM.Input} \), a type of commitments \( \text{COMM.Output} \), a type of commitment trapdoors \( \text{COMM.Trapdoor} \), and a trapdoor generator \( \text{COMM.GenTrapdoor} : () \overset{\$}{\Rightarrow} \text{COMM.Trapdoor.} \)

Let \( \text{COMM} : \text{COMM.Trapdoor} \times \text{COMM.Input} \rightarrow \text{COMM.Output} \) be a function satisfying the following security requirements.

**Security requirements:**

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

**Notes:**

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

Let \( \ell_{\text{cm}}, \ell_{\text{MerkleSprout}}, \ell_{\text{PRFSprout}}, \) and \( \ell_{\text{value}} \) be as defined in §5.3 ‘Constants’ on p. 51.

Define \( \text{NoteCommit}^{\text{Sprout}}.\text{Trapdoor} := \mathbb{B}[\ell_{\text{cm}}] \) and \( \text{NoteCommit}^{\text{Sprout}}.\text{Output} := \mathbb{B}[\ell_{\text{MerkleSprout}}]. \)

**Sprout** uses a note commitment scheme

\[
\text{NoteCommit}^{\text{Sprout}} : \text{NoteCommit}^{\text{Sprout}}.\text{Trapdoor} \times \mathbb{B}[\ell_{\text{PRFSprout}}] \times \{0..2^\ell_{\text{value}} - 1\} \times \mathbb{B}[\ell_{\text{PRFSprout}}] \rightarrow \text{NoteCommit}^{\text{Sprout}}.\text{Output},
\]

instantiated in §5.4.7.1 ‘Sprout Note Commitments’ on p. 65.

Let \( \ell_{\text{scalar}} \) be as defined in §5.3 ‘Constants’ on p. 51.

Let \( J^{(r)} \) and \( r_j \) be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

Define:

\[
\text{NoteCommit}^{\text{Sapling}}.\text{Trapdoor} := \{0..2^\ell_{\text{scalar}} - 1\} \) and \( \text{NoteCommit}^{\text{Sapling}}.\text{Output} := J;
\]

\[
\text{ValueCommit}.\text{Trapdoor} := \{0..2^\ell_{\text{scalar}} - 1\} \) and \( \text{ValueCommit}.\text{Output} := J.
\]
**Sapling** uses two additional commitment schemes:

NoteCommit\textsuperscript{Sapling} : NoteCommit\textsuperscript{Sapling}.Trapdoor × B[ℓ] × B[ℓ] × \{0..2\textsuperscript{\text{value}}−1\} → NoteCommit\textsuperscript{Sapling}.Output

ValueCommit : ValueCommit.Trapdoor × \{-\frac{r_1−1}{2}..\frac{r_2−1}{2}\} → ValueCommit.Output

NoteCommit\textsuperscript{Sapling} is instantiated in § 5.4.7.2 ‘Windowed Pedersen commitments’ on p. 65, and ValueCommit is instantiated in § 5.4.7.3 ‘Homomorphic Pedersen commitments’ on p. 66.

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

### 4.1.8 Represented Group

A represented group \( G \) consists of:

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

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

Define \( G^{(r)}_x := \{ \text{repr}_G(P) : B[\ell] | P \in G^{(r)} \} \).

For \( G : G \) we write \(-G\) for the negation of \( G \), such that \((-G) + G = O_\text{G} \). We write \( G - H \) for \( G + (-H) \).

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

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

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

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

### 4.1.9 Hash Extractor

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

**Note:** Unlike the representation function \( \text{repr}_G \), \( \text{Extract}_{G^{(r)}} \) need not have an efficiently computable left inverse.
4.1.10 Group Hash

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

- a type GroupHash.URSType of Uniform Random Strings;
- a type GroupHash.Input of inputs;
- a function GroupHash$^{G^{(r)}}$: GroupHash.URSType × GroupHash.Input → $G^{(r)}$.

In §5.4.8.5 ‘Group Hash into Jubjub’ on p. 71, we instantiate a family of group hashes into the Jubjub curve defined by §5.4.8.3 ‘Jubjub’ on p. 69.

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

Non-normative notes:

- GroupHash$^{G^{(r)}}$ is used to obtain generators of the Jubjub curve for various purposes: the bases $G$ and $H$ used in Sapling key generation, the Pedersen hash defined in §5.4.1.7 ‘Pedersen Hash Function’ on p. 56, and the commitment schemes defined in §5.4.7.2 ‘Windowed Pedersen commitments’ on p. 65 and in §5.4.7.3 ‘Homomorphic Pedersen commitments’ on p. 66.

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 GroupHash$^{G^{(r)}}_{URS}$ of the family, it is infeasible to find a sequence of distinct inputs $m_1...n : GroupHash.Input^{[n]}$ and a sequence of nonzero $x_1...n : \mathbb{F}_r^{[n]}$ such that $\sum_{i=1}^{n} [x_i] GroupHash^{G^{(r)}}_{URS}(m_i) = O_G$.

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

- GroupHash$^{G^{(r)}}$ is also used to instantiate DiversifyHash in §5.4.16 ‘DiversifyHash Hash Function’ on p. 55. We do not know how to prove the Unlinkability property defined in that section in the standard model, but in a model where GroupHash$^{G^{(r)}}$ (restricted to inputs for which it does not return ⊥) is taken as a random oracle, it is implied by the Decisional Diffie-Hellman assumption on $G^{(r)}$.

- URS is a Uniform Random String: we choose it verifiably at random (see §5.9 ‘Randomness Beacon’ on p. 79), 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 $r_p : \mathbb{N}^+$ which must be prime;
- two represented subgroups $\mathbb{P}_{1,2}^{(r)}$, both of order $r_p$;
- a group $\mathbb{P}_T^{(r)}$ of order $r_p$, written multiplicatively with operation $\cdot : \mathbb{P}_T^{(r)} \times \mathbb{P}_T^{(r)} \rightarrow \mathbb{P}_T^{(r)}$ and group identity $1_p$;
- three generators $\mathbb{P}_{1,2,T}^{(r)}$ of $\mathbb{P}_{1,2,T}^{(r)}$ respectively;
- a pairing function $\hat{e}_p : \mathbb{P}_1^{(r)} \times \mathbb{P}_2^{(r)} \rightarrow \mathbb{P}_T^{(r)}$ satisfying:
  - (Bilinearity) for all $a, b : \mathbb{F}_r^{(r)}, P : \mathbb{P}_1^{(r)}$, and $Q : \mathbb{P}_2^{(r)}$, $\hat{e}_p([a] P, [b] Q) = \hat{e}_p(P, Q)^{a b}$; and
  - (Nondegeneracy) there does not exist $P : \mathbb{P}_1^{(r)}$ such that for all $Q : \mathbb{P}_2^{(r)}$, $\hat{e}_p(P, Q) = 1_p$. 

...
4.1.12 Zero-Knowledge Proving System

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

A preprocessing zk-SNARK instance ZK defines:

- a type of zero-knowledge proving keys, ZK.ProvingKey;
- a type of zero-knowledge verifying keys, ZK.VerifyingKey;
- a type of primary inputs ZK.PrimaryInput;
- a type of auxiliary inputs ZK.AuxiliaryInput;
- a type of zk-SNARK proofs ZK.Proof;
- a type ZK.SatisfyingInputs ⊆ ZK.PrimaryInput × ZK.AuxiliaryInput of inputs satisfying the statement;
- a randomized key pair generation algorithm ZK.Gen : () ↣ ZK.ProvingKey × ZK.VerifyingKey;
- a proving algorithm ZK.Prove : ZK.ProvingKey × ZK.SatisfyingInputs → ZK.Proof;
- a verifying algorithm ZK.Verify : ZK.VerifyingKey × ZK.PrimaryInput × ZK.Proof → B;

The security requirements below are supposed to hold with overwhelming probability for (pk, vk) ↣ ZK.Gen().

Security requirements:

- **Completeness:** An honestly generated proof will convince a verifier: for any (x, w) ∈ ZK.SatisfyingInputs, if ZK.Prove_{pk}(x, w) outputs π, then ZK.Verify_{vk}(x, π) = 1.

- **Knowledge Soundness:** For any adversary \( \mathcal{A} \) able to find an \( x \in ZK.PrimaryInput \) and proof \( \pi \in ZK.Proof \) such that ZK.Verify_{vk}(x, π) = 1, there is an efficient extractor \( \mathcal{E}_A \) such that if \( \mathcal{E}_A(vk, pk) \) returns \( w \), then the probability that \( (x, w) \not\in ZK.SatisfyingInputs \) 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:

  \[
  \Pr \left[ (x, w) \in ZK.SatisfyingInputs \mid \mathcal{D}(\pi) = 1 \right] = \Pr \left[ (pk, vk) \overset{\$}{\leftrightarrow} ZK.Gen() \mid (x, w) \overset{\$}{\leftrightarrow} D(pk, vk) \mid \pi \overset{\$}{\leftrightarrow} ZK.Prove_{pk}(x, w) \right] \\
  \text{and} \quad \Pr \left[ (x, w) \in ZK.SatisfyingInputs \mid \mathcal{D}(\pi) = 1 \right] = \Pr \left[ (pk, vk) \overset{\$}{\leftrightarrow} S() \mid (x, w) \overset{\$}{\leftrightarrow} D(pk, vk) \mid \pi \overset{\$}{\leftrightarrow} S(x) \right]
  \]

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

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

**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.

27
Zcash uses two proving systems:

- BCTV14 (§ 5.4.9.1 ‘BCTV14’ on p. 72) is used with the BN-254 pairing (§ 5.4.8.1 ‘BN-254’ on p. 66), to prove and verify the Sprout JoinSplit statement (§ 4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42) before Sapling activation.
- Groth16 (§ 5.4.9.2 ‘Groth16’ on p. 73) is used with the BLS12-381 pairing (§ 5.4.8.2 ‘BLS12-381’ on p. 68), to prove and verify the Sapling Spend statement (§ 4.15.2 ‘Spend Statement (Sapling)’ on p. 43) and Output statement (§ 4.15.3 ‘Output Statement (Sapling)’ on p. 44). It is also used to prove and verify the JoinSplit statement after Sapling activation.

These specializations are: ZKJoinSplit for the Sprout JoinSplit statement (with BCTV14 and BN-254, or Groth16 and BLS12-381); ZKSpend for the Sapling Spend statement; and ZKOutput for the Sapling Output statement.

We omit key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be either the BCTV14 proving key and verifying key defined in § 5.7 ‘BCTV14 zk-SNARK Parameters’ on p. 78, or the sprout-groth16.params Groth16 proving key and verifying key defined in § 5.8 ‘Groth16 zk-SNARK Parameters’ on p. 79, 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 defined in § 5.8 ‘Groth16 zk-SNARK Parameters’ on p. 79.

4.2 Key Components

4.2.1 Sprout Key Components

Let \( \ell_{ak} \) be as defined in § 5.3 ‘Constants’ on p. 51.

Let PRF\(_{\text{addr}}\) be a Pseudo Random Function, instantiated in § 5.4.2 ‘Pseudo Random Functions’ on p. 58.

Let KA\(_{\text{Sprout}}\) be a key agreement scheme, instantiated in § 5.4.1 ‘Sprout Key Agreement’ on p. 60.

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

\[ a_{pk}, sk_{\text{enc}}, \text{ and } pk_{\text{enc}} \text{ are derived from } a_{sk} \text{ as follows:} \]

\[ a_{pk} := \text{PRF}_{a_{sk}}(i) \]
\[ sk_{\text{enc}} := \text{KA}_{\text{Sprout}}.\text{FormatPrivate}(\text{PRF}_{a_{sk}}(1)) \]
\[ pk_{\text{enc}} := \text{KA}_{\text{Sprout}}.\text{DerivePublic}(sk_{\text{enc}}, \text{KA}_{\text{Sprout}}.\text{Base}). \]

4.2.2 Sapling Key Components

Let \( \ell_{\text{PRFexpand}}, \ell_{sk}, \ell_{ovk}, \text{ and } \ell_{d} \) be as defined in § 5.3 ‘Constants’ on p. 51.

Let PRF\(_{\text{expand}}\) and PRF\(_{\text{gsk}}\) be Pseudo Random Functions instantiated in § 5.4.2 ‘Pseudo Random Functions’ on p. 58.

Let KA\(_{\text{Sapling}}\) be a key agreement scheme, instantiated in § 5.4.4.3 ‘Sapling Key Agreement’ on p. 61.

Let CRH\(_{ivk}\) be a hash function, instantiated in § 5.4.1.5 ‘CRH\(_{ivk}\) Hash Function’ on p. 54.

Let DiversifyHash be a hash function, instantiated in § 5.4.16 ‘DiversifyHash Hash Function’ on p. 55.

Let SpendAuthSig, instantiated in § 5.4.6.1 ‘Spend Authorization Signature’ on p. 64, be a signature scheme with re-randomizable keys.

Let repr\(_{r}^{(r)}, j_{r}^{(r)}\), and \( j_{r}^{(r)}\) be as defined in § 5.4.8.3 ‘Jubjub’ on p. 69, and let FindGroupHash\(_{j}^{(r)}\) be as defined in § 5.4.8.5 ‘Group Hash into Jubjub’ on p. 71.
As explained in §3.1 ‘Integers, Bit Sequences, and Endianness’ on p.12, Sapling allows the efficient creation of multiple diversified payment addresses with the same spending authority. A group of such addresses shares the same full viewing key and incoming viewing key.

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

\[
p_d := \text{KA}^{\text{Sapling}}.\text{DerivePublic}(ivk, g_d).
\]

The resulting diversified payment address is \((d : \mathbb{B}[\ell], p_d : \text{KA}^{\text{Sapling}}.\text{PublicPrimeOrder})\).

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

Let first : (\(\mathbb{B} \rightarrow T \cup \{\bot\}\)) \(\rightarrow T \cup \{\bot\}\) be as defined in §5.4.8.5 ‘Group Hash into Jubjub’ on p.71. Define:

\[
\text{CheckDiversifier}(d : \mathbb{B}[\ell]) := \begin{cases} 
\bot, & \text{if } \text{DiversifyHash}(d) = \bot \\
\text{d}, & \text{otherwise}
\end{cases}
\]

\[
\text{DefaultDiversifier}(sk : \mathbb{B}[\ell]) := \text{first}(i : \mathbb{B} \mapsto \text{CheckDiversifier}(\text{truncate}(\ell/8)(\text{PRF}^{\text{expand}}([3, i]))) : \mathbb{J}^*) \cup \{\bot\})
\]

For a random spending key, DefaultDiversifier returns \(\bot\) with probability approximately \(2^{-256}\); if this happens, discard the key and repeat with a different sk.
4.3 JoinSplit Descriptions

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

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

Let $\ell_{\text{MerkleSprout}}, \ell_{\text{PRFSprout}}, \ell_{\text{Seed}}, N^{\text{old}}, N^{\text{new}},$ and MAX\_MONEY be as defined in §5.3 ‘Constants’ on p.51.

Let $h_{\text{SigCRH}}$ be as defined in §4.1.1 ‘Hash Functions’ on p.18.

Let $\text{NoteCommit}^{\text{Sprout}}$ be as defined in §4.1.7 ‘Commitment’ on p.24.

Let $\text{KASprout}$ be as defined in §4.1.4 ‘Key Agreement’ on p.20.

Let $\text{Sym}$ be as defined in §4.1.3 ‘Symmetric Encryption’ on p.20.

Let $\text{ZKJoinSplit}$ be as defined in §4.1.12 ‘Zero-Knowledge Proving System’ on p.27.

A JoinSplit description consists of $(v_{\text{pub}}^{\text{old}}, v_{\text{pub}}^{\text{new}}, rt_{1..N^{\text{old}}}, cm_{1..N^{\text{new}}}, epk, \text{randomSeed}, h_{1..N^{\text{old}}}, \pi_{\text{ZKJoinSplit}}, C_{1..N^{\text{new}}})$ where

- $v_{\text{pub}}^{\text{old}} : \{0..\text{MAX\_MONEY}\}$ is the value that the JoinSplit transfer removes from the transparent transaction value pool;
- $v_{\text{pub}}^{\text{new}} : \{0..\text{MAX\_MONEY}\}$ is the value that the JoinSplit transfer inserts into the transparent transaction value pool;
- $rt : \mathbb{F}^{\ell_{\text{MerkleSprout}}}$ is an anchor, as defined in §3.3 ‘The Block Chain’ on p.15, for the output treestate of either a previous block, or a previous JoinSplit transfer in this transaction.
\[ n_{\text{old}} \cdot 1_1 \cdot N_{\text{old}} : B_{\text{Sapling}}^{\ell_{\text{MerkleSapling}}}[N_{\text{old}}] \] is the sequence of nullifiers for the input notes;

\[ cn_{\text{new}} \cdot 1_1 \cdot N_{\text{new}} : \text{NoteCommit}^{\text{Sprout}}. \text{Output}[N_{\text{new}}] \] is the sequence of note commitments for the output notes;

\[ epk : \text{KA}^{\text{Sprout}}. \text{Public} \text{ is a key agreement public key, used to derive the key for encryption of the transmitted notes ciphertext (§4.16 ‘In-band secret distribution (Sprout)’ on p.49);} \]

\[ \text{randomSeed} : B_{\text{seed}} \] is a seed that must be chosen independently at random for each JoinSplit description;

\[ h_{\text{old}} \cdot 1_1 \cdot N_{\text{old}} : B_{\text{seed}}^{\ell_{\text{seed}}}[N_{\text{old}}] \] is a sequence of tags that bind hSig to each a_k of the input notes;

\[ \pi_{\text{ZKJoinSplit}}^{\text{ZKJoinSplit}}. \text{Proof} \text{ is a zk proof with primary input } (rt, n_{\text{old}} \cdot 1_1 \cdot N_{\text{old}}, cm_{\text{new}} \cdot 1_1 \cdot N_{\text{new}}, v_{\text{pub}}, v_{\text{pub}}, h_{\text{Sig}}, h_{1_1 \cdot N_{\text{old}}}) \text{ for the JoinSplit statement defined in §4.15.1 ‘JoinSplit Statement (Sprout)’ on p.42 (this is a BCTV14 proof before Sapling activation, and a Groth16 proof after Sapling activation);} \]

\[ C_{\text{new}}^{\text{enc}} : \text{Sym}. C_{\text{new}}^{\text{enc}} \] is a sequence of ciphertext components for the encrypted output notes.

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

The value hSig is also computed from randomSeed, n_{\text{old}} \cdot 1_1 \cdot N_{\text{old}}, and the JoinSplitPubKey of the containing transaction:

\[ h_{\text{Sig}} := h_{\text{Sig}} \text{CRH}(\text{randomSeed}, n_{\text{old}} \cdot 1_1 \cdot N_{\text{old}}, \text{JoinSplitPubKey}). \]

Consensus rules:

\[ \text{Elements of a JoinSplit description MUST have the types given above (for example: } 0 \leq v_{\text{new}} \leq \text{MAX\_MONEY} \text{ and } 0 \leq v_{\text{pub}} \leq \text{MAX\_MONEY}). \]

\[ \text{Either } v_{\text{pub}} \text{ Or } v_{\text{pub}} \text{ MUST be zero.} \]

\[ \text{The proof } \pi_{\text{ZKJoinSplit}}^{\text{ZKJoinSplit}} \text{ MUST be valid given a primary input formed from the relevant other fields and } h_{\text{Sig}} \text{ — i.e. ZKJoinSplit. Verify( (rt, n_{\text{old}} \cdot 1_1 \cdot N_{\text{old}}, cm_{\text{new}} \cdot 1_1 \cdot N_{\text{new}}, v_{\text{pub}}, v_{\text{pub}}, h_{\text{Sig}}, h_{1_1 \cdot N_{\text{old}}}, \pi_{\text{ZKJoinSplit}}) = 1.} \]

### 4.4 Spend Descriptions

A Spend transfer, as specified in §3.6 ‘Spend Transfers, Output Transfers, and their Descriptions’ on p.16, is encoded in transactions as a Spend description.

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

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

Let \( \ell_{\text{MerkleSapling}} \) and \( \ell_{\text{PRFSapling}} \) be as defined in §5.3 ‘Constants’ on p.51.


Let SpendAuthSig be as defined in §4.13 ‘Spend Authorization Signature’ on p.40.

Let ZKSpend be as defined in §4.1.12 ‘Zero-Knowledge Proving System’ on p.27.

A Spend description consists of (cv, rt, nf, rk, \( \pi_{\text{ZKSpend}}^{\text{SpendAuthSig}} \)) where

\[ \text{cv} : \text{ValueCommit}. \text{Output is the value commitment to the value of the input note;} \]

\[ \text{rt} : B_{\text{MerkleSapling}}^{\ell_{\text{MerkleSapling}}} \text{ is an anchor, as defined in §3.3 ‘The Block Chain’ on p.15, for the output treestate of a previous block;} \]

\[ \text{nf} : B_{\text{PRFSapling}}^{\ell_{\text{PRFSapling}}} \text{ is the nullifier for the input note;} \]

\[ \text{rk} : \text{SpendAuthSig}. \text{Public is a randomized public key that should be used to verify spendAuthSig.} \]

\[ \pi_{\text{ZKSpend}}^{\text{ZKSpend}} \text{Proof is a zk-SNARK proof with primary input } (\text{cv}, \text{rt}, \text{nf}, \text{rk}) \text{ for the Spend statement defined in §4.15.2 ‘Spend Statement (Sapling)’ on p.43;} \]

\[ \text{spendAuthSig} : \text{SpendAuthSig}. \text{Signature is as specified in §4.13 ‘Spend Authorization Signature’ on p.40.} \]
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. \([h_{j}]\) cv MUST NOT be \(O_{j}\) and \([h_{j}]\) rk MUST NOT be \(O_{j}\).
- The proof \(\pi_{\text{ZKspend}}\) MUST be valid given a primary input formed from the other fields except spendAuthSig — i.e. \(\text{ZKspend}.\text{Verify}((\text{cv}.\text{rt}, \text{nf}.\text{rk}), \pi_{\text{ZKspend}}) = 1\).
- Let SigHash be the SIGHASH transaction hash of this transaction, not associated with an input, as defined in §4.9 ‘SIGHASH Transaction Hashing’ on p. 36 using SIGHASH_ALL.
  The spend authorization signature MUST be a valid SpendAuthSig signature over SigHash using rk as the public key — i.e. \(\text{SpendAuthSig}.\text{Verify}_{\text{rk}}(\text{SigHash}, \text{spendAuthSig}) = 1\).

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. 16, 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 ValueCommit.Output be as defined in §4.1.7 ‘Commitment’ on p. 24.
Let \(f_{\text{MerkleSapling}}\) be as defined in §5.3 ‘Constants’ on p. 51.
Let \(KA^{\text{Sapling}}\) be as defined in §4.14 ‘Key Agreement’ on p. 20.
Let Sym be as defined in §4.1.3 ‘Symmetric Encryption’ on p. 20.
Let \(ZKOutput\) be as defined in §4.1.12 ‘Zero-Knowledge Proving System’ on p. 27.

An Output description consists of \((\text{cv.cmv}, \text{epk}, C^{\text{enc}}, C^{\text{out}}, \pi_{\text{ZKOutput}})\) where

- \(\text{cv} : \text{ValueCommit}.\text{Output}\) is the value commitment to the value of the output note;
- \(\text{cmv} : E^{\{f_{\text{MerkleSapling}}\}}\) is the result of applying \(\text{Extract}_{\text{J}^{(r)}}\) (defined in §5.4.8.3 ‘Hash Extractor for Jubjub’ on p. 70) to the note commitment for the output note;
- \(\text{epk} : KA^{\text{Sapling}}\text{.Public}\) 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. 46);
- \(C^{\text{enc}} : \text{Sym}.\text{C}\) is a ciphertext component for the encrypted output note;
- \(C^{\text{out}} : \text{Sym}.\text{C}\) is a ciphertext component that allows the holder of a full viewing key to recover the recipient diversified transmission key \(pk_{d}\) and the ephemeral private key esk (and therefore the entire note plaintext);
- \(\pi_{\text{ZKOutput}} : ZKOutput.\text{Proof}\) is a zk-SNARK proof with primary input \((\text{cv, cmv, epk})\) for the Output statement defined in §4.15.3 ‘Output Statement (Sapling)’ on p. 44.

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. \([h_{j}]\) cv MUST NOT be \(O_{j}\) and \([h_{j}]\) epk MUST NOT be \(O_{j}\).
- The proof \(\pi_{\text{ZKOutput}}\) MUST be valid given a primary input formed from the other fields except \(C^{\text{enc}}\) and \(C^{\text{out}}\) — i.e. \(\text{ZKspend}.\text{Verify}((\text{cv, cmv, epk}), \pi_{\text{ZKOutput}}) = 1\).
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 JoinSplitSig key pair:

\[
\begin{align*}
&\text{joinSplitPrivKey} \leftarrow \text{JoinSplitSig.GenPrivate()} \\
&\text{joinSplitPubKey} := \text{JoinSplitSig.DerivePublic(joinSplitPrivKey)}.
\end{align*}
\]

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

\[
\begin{align*}
&\text{Choose uniformly random } rcm_{i_{\text{new}}} \leftarrow \text{NoteCommit^{Sprout}.GenTrapdoor()} \\
&\text{Compute } \rho_{i_{\text{new}}} = \text{PRF}_{\varphi}(i, h_{\text{Sig}}). \\
&\text{Compute } cm_{i_{\text{new}}} = \text{NoteCommit^{Sprout}_{rcm_{i_{\text{new}}}}(a_{pk,i}, v_{i_{\text{new}}}, \rho_{i_{\text{new}}})}. \\
&\text{Let } np_i = (v_{i_{\text{new}}}, \rho_{i_{\text{new}}}, rcm_{i_{\text{new}}}, \text{memo}_i).
\end{align*}
\]

$np_{1..N_{\text{new}}}$ are then encrypted to the recipient’s transmission keys $pk_{\text{new}}^{\text{enc}}$, $1..N_{\text{new}}$, giving the transmitted notes ciphertext $(epk, C_{\text{enc}}^{1..N_{\text{new}}})$, as described in § 4.16 ‘In-band secret distribution (Sprout)’ on p. 45.

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 $\text{dataToBeSigned} : \mathbb{G}^{[Y]}$ as described in § 4.10 ‘Non-malleability (Sprout)’ on p. 37, and signs it with the private JoinSplit signing key:

\[
\begin{align*}
&\text{joinSplitSig} \leftarrow \text{JoinSplitSig.Sign(joinSplitPrivKey)(dataToBeSigned)}.
\end{align*}
\]

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 $\text{ValueCommit}$ and $\text{NoteCommit^{Sapling}}$ be as specified in § 4.1.7 ‘Commitment’ on p. 24.

Let $\text{KA^{Sapling}}$ be as defined in § 4.1.4 ‘Key Agreement’ on p. 20.

Let $\text{DiversifyHash}$ be as defined in § 4.1.1 ‘Hash Functions’ on p. 18.

Let $\text{repr}_J, r_J$, and $h_J$ be as defined in § 5.4.8.3 ‘Jubjub’ on p. 69.

Let $ovk$ be an outgoing viewing key that is intended to be able to decrypt this payment. This may be one of:

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

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

33
For each Output description, the sender selects a value \( v^{\text{new}} \) \( \in \{0..\text{MAX\_MONEY}\} \) and a destination Sapling shielded payment address \((d, pk_d)\), and then performs the following steps:

- Check that \( pk_d \) is of type \( \text{KA}^{\text{Sapling}} \) \( . \text{PublicPrimeOrder} \), i.e. it is a valid ctEdwards curve point on the Jubjub curve (as defined in §5.4.8.3 ’Jubjub’ on p. 69) not equal to \( O_3 \), and \( [r_j] pk_d = O_j \).
- Calculate \( g_d = \text{DiversifyHash}(d) \) and check that \( g_d \neq \bot \).
- Choose independent uniformly random commitment trapdoors:
  \[
  \begin{align*}
  rcv^{\text{new}} & \overset{\$}{\leftarrow} \text{ValueCommit} . \text{GenTrapdoor}(), \\
  rcm^{\text{new}} & \overset{\$}{\leftarrow} \text{NoteCommit}^{\text{Sapling}} . \text{GenTrapdoor}().
  \end{align*}
  \]
- Calculate
  \[
  \begin{align*}
  cv^{\text{new}} & \overset{\$}{=} \text{ValueCommit}_{rcv^{\text{new}}} (v^{\text{new}}) \\
  cm^{\text{new}} & \overset{\$}{=} \text{NoteCommit}_{rcm^{\text{new}}} (\text{repr}_j(g_d), \text{repr}_j(pk_d), v^{\text{new}})
  \end{align*}
  \]
- Let \( \text{np} = (d, v^{\text{new}}, rcm, \text{memo}) \), where \( rcm = \text{LEBS2OSP}_{256}(12 \text{LEBS2OSP}_{256}(rcm^{\text{new}})) \).
- Encrypt \( \text{np} \) to the recipient diversified transmission key \( pk_d \) with diversified base \( g_d \), and to the outgoing viewing key ovk, giving the transmitted note ciphertext \((\text{epk}, C^{\text{enc}}, C^{\text{out}})\). This procedure is described in §4.171 ’Encryption (Sapling)’ on p. 47; it also uses \( cv^{\text{new}} \) and \( cm^{\text{new}} \) to derive the outgoing cipher key.
- Generate a proof \( \pi_{\text{ZKOutput}} \) for the Output statement in §4.15.3 ’Output Statement (Sapling)’ on p. 44.
- Return \((cv^{\text{new}}, cm^{\text{new}}, \text{epk}, C^{\text{enc}}, C^{\text{out}}, \pi_{\text{ZKOutput}})\).

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 \( N^{\text{old}} \) input notes, and \( N^{\text{new}} \) 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 \( \ell_{\text{ak}} \) and \( \ell_{\text{PRF}^{\text{Sprout}}} \) be as defined in §5.3 ’Constants’ on p. 51.

Let \( \text{PRF}^{\text{nf}} \) be as defined in §4.12 ’Pseudo Random Functions’ on p. 19.

Let NoteCommit\(^{\text{Sprout}} \) be as defined in §4.17 ’Commitment’ on p. 24.

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

- Generate a new uniformly random spending key \( a^{\text{old}}_{pk,i} \overset{\$}{\leftarrow} \mathbb{B}^{[\ell_{\text{ak}}]} \) and derive its paying key \( a^{\text{old}}_{pk,i} \).
- Set \( \nu^{\text{old}}_i = 0 \).
- Choose uniformly random \( \rho^{\text{old}}_i \overset{\$}{\leftarrow} \mathbb{B}^{[\ell_{\text{PRF}^{\text{Sprout}}}]} \) and \( rcm^{\text{old}}_i \overset{\$}{\leftarrow} \text{NoteCommit}^{\text{Sprout}} . \text{GenTrapdoor}() \).
- Compute \( \text{nf}^{\text{old}}_i = \text{PRF}^{\text{nf}}_{a^{\text{old}}_{ak,i}}(\rho^{\text{old}}_i) \).
- 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 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 \( \ell_{sk} \) be as defined in § 5.3 ‘Constants’ on p. 51.
Let \( r_j \) and \( \text{repr}_j \) be as defined in § 5.4.8.3 ‘Jubjub’ on p. 69.
Let \( \mathcal{H} \) be as defined in § 4.2.2 Sapling Key Components’ on p. 28.
Let \( \text{PRF}^{nf\text{Sapling}} \) be as defined in § 4.1.2 ‘Pseudo Random Functions’ on p. 19.
Let \( \text{NoteCommit}^{\text{Sapling}} \) be as defined in § 4.1.7 ‘Commitment’ on p. 24.

A dummy Sapling input note is constructed as follows:

- Choose uniformly random \( sk \leftarrow \mathbb{R}[\ell_{sk}] \).
- Generate a new diversified payment address \( (d, pk_d) \) for \( sk \) as described in § 4.2.2 Sapling Key Components’ on p. 28.
- Set \( v^\text{old} = 0 \), and set \( pos = 0 \).
- Choose uniformly random \( rcm \leftarrow \mathbb{R} \text{NoteCommit}^{\text{Sapling}}.\text{GenTrapdoor}() \), and \( nsk \leftarrow \mathbb{R} \text{F}_r \).
- Compute \( nk = [nsk] \mathcal{H} \) and \( nk^* = \text{repr}_j(nk) \).
- Compute \( \rho = cm^\text{old} = \text{NoteCommit}^{\text{Sapling}}(\text{repr}_j(g_d), \text{repr}_j(pk_d), v^\text{old}) \).
- Compute \( nf^\text{old} = \text{PRF}^{nf\text{Sapling}}(\text{repr}_j(\rho)) \).
- Construct a dummy Merkle path for use in the auxiliary input to the Spend statement (this will not be checked, because \( v^\text{old} = 0 \)).

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 \( \text{MerkleDepth} \) be \( \text{MerkleDepth}^{\text{Sprout}} \) for the Sprout note commitment tree, or \( \text{MerkleDepth}^{\text{Sapling}} \) for the Sapling note commitment tree. These constants are defined in § 5.3 ‘Constants’ on p. 51.

Similarly, let \( \text{MerkleCRH} \) be \( \text{MerkleCRH}^{\text{Sprout}} \) for Sprout, or \( \text{MerkleCRH}^{\text{Sapling}} \) 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 \( M_i^h \) be the hash value associated with the node at index \( i \) in layer \( h \).

The nodes at layer \( \text{MerkleDepth} \) are called leaf nodes. When a note commitment is added to the tree, it occupies the leaf node hash value \( M_i^{\text{MerkleDepth}} \) for the next available \( i \).

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

35
The nodes at layers 0 to MerkleDepth – 1 inclusive are called internal nodes, and are associated with MerkleCRH outputs. Internal nodes are computed from their children in the next layer as follows: for 0 ≤ h < MerkleDepth and 0 ≤ i < 2^h.

\[ M^h_i := \text{MerkleCRH}(M^{h+1}_{2i}, M^{h+1}_{2i+1}). \]

A Merkle path from leaf node \( M^{\text{MerkleDepth}}_i \) in the incremental Merkle tree is the sequence

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

where

\[ \text{sibling}(h, i) := \text{floor}\left( \frac{i}{2^{\text{MerkleDepth} - h}} \right) + 1 \]

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

### 4.9 SIGHASH Transaction Hashing

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

In the case of Zcash, the BCTV14 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. Zcash has been designed to mitigate malleability attacks, as described in §4.10 ‘Non-malleability (Sprout)’ on p. 37, §4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38, and §4.13 ‘Spend Authorization Signature’ on p. 40.

To provide additional flexibility when combining Spend authorizations from different sources, Bitcoin defines several SIGHASH types that cover various parts of a transaction [Bitcoin-SigHash]. One of these types is SIGHASH_ALL, which is used for Zcash-specific signatures, i.e. JoinSplit signatures, spend authorization signatures, 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-Zcash-specific parts of the transaction.

In Zcash, all SIGHASH types are extended to cover the Zcash-specific fields nJoinSplit, vJoinSplit, and if present joinSplitPubKey. These fields are described in §7.1 ‘Encoding of Transactions’ on p. 81. 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 Zcash 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 be written).

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

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

[Blossom onward] The SIGHASH algorithm used after Blossom activation is the same as for Sapling, but using the Blossom consensus branch ID 0x2BB40E60 as defined in [ZIP-206].
Heartwood onward] The SIGHASH algorithm used after Heartwood activation is the same as for Sapling, but using the Heartwood consensus branch ID 0xF5B9230B as defined in [ZIP-250].

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 v\textsuperscript{new}\textsubscript{pub} and v\textsuperscript{old}\textsubscript{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. 61.

If nJoinSplit is zero, the joinSplitPubKey and joinSplitSig fields are omitted. Otherwise, a transaction has a correct JoinSplit signature if and only if JoinSplitSig.Verify(joinSplitPubKey, dataToBeSigned, joinSplitSig) = 1.

Let h\textsubscript{Sig} be computed as specified in § 4.3 ‘JoinSplit Descriptions’ on p. 30.

Let PRF\textsuperscript{pk} be as defined in § 4.1.2 ‘Pseudo Random Functions’ on p. 19.

For each \( i \in \{1..N^\text{old}\} \), the creator of a JoinSplit description calculates \( h_i = PRF_{a^\text{old}_{sk,i}}(i, h_{Sig}) \).

The correctness of \( h_{1..N^\text{old}} \) is enforced by the JoinSplit statement given in § 4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42. This ensures that a holder of all of the \( a^\text{old}_{sk,1..N^\text{old}} \) 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 inputs minus transparent outputs is transferred to the miner of the block containing the transaction; it is added to the miner subsidy in the coinbase transaction of the block.

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

Unlike original Zerocash [BCGGMTV2014], Zcash does not have a distinction between Mint and Pour operations. The addition of v\textsuperscript{old}\textsubscript{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. 30, either v\textsuperscript{old}\textsubscript{pub} or v\textsuperscript{new}\textsubscript{pub} MUST be zero. No generality is lost because, if a transaction in which both v\textsuperscript{old}\textsubscript{pub} and v\textsuperscript{new}\textsubscript{pub} were nonzero were allowed, it could be replaced by an equivalent one in which \( \min(v\textsuperscript{old}_{pub}, v\textsuperscript{new}_{pub}) \) is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between transactions according to client implementation.
4.12 Balance and Binding Signature (Sapling)

Sapling adds Spend transfers and Output transfers to the transparent and JoinSplit transfers present in Sprout. The net value of Spend transfers minus Output transfers in a transaction is called the balancing value, measured in zatoshi as a signed integer $v_{\text{balance}}$. $v_{\text{balance}}$ is encoded explicitly in a transaction as the field valueBalance; see §7.1 ‘Encoding of Transactions’ on p.81.

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

Consistency of $v_{\text{balance}}$ with the value commitments in Spend descriptions and Output descriptions is enforced by the binding signature. This signature has a dual role in the Sapling protocol:

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

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

Let $J(r)$, $J^*(r)$, and $r_{\text{new}}$ be as defined in §5.4.8.3 ‘Jubjub’ on p.69.

Let ValueCommit, $V$, and $R$ be as defined in §5.4.7.3 ‘Homomorphic Pedersen commitments’ on p.66:

\[
\text{ValueCommit} : \text{ValueCommit}.\text{Trapdoor} \times \left\{ -\frac{r_{\text{old}}-1}{2}, ..., \frac{r_{\text{new}}-1}{2} \right\} \to \text{ValueCommit}.\text{Output};
\]

$V : J(r)^*$ is the value base in ValueCommit;

$R : J^*(r)^*$ is the randomness base in ValueCommit.

BindingSig, $\Phi$, and $\bowtie$ are instantiated in §5.4.6.2 ‘Binding Signature’ on p.64. These and the derived notation $\Box$, $\bigodot_{i=1}^N \bowtie$, and $\bigodot_{i=1}^N \bowtie$ are specified in §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p.23.

Suppose that the transaction has:

- $n$ Spend descriptions with value commitments $cv_{1..n}^{\text{old}}$, committing to values $v_{1..n}^{\text{old}}$ with randomness $rcv_{1..n}^{\text{old}}$;
- $m$ Output descriptions with value commitments $cv_{1..m}^{\text{new}}$, committing to values $v_{1..m}^{\text{new}}$ with randomness $rcv_{1..m}^{\text{new}}$;
- balancing value $v_{\text{balance}}$.

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

Instead, validators calculate the transaction binding verification key as:

\[
bvk := \bigodot_{i=1}^N cv_{i}^{\text{old}} \bowtie \left( \bigodot_{j=1}^m cv_{j}^{\text{new}} \right) \bowtie \text{ValueCommit}_0(v_{\text{balance}}).
\]

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

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

\[
bsk := \bigodot_{i=1}^N rcv_{i}^{\text{old}} \bowtie \left( \bigodot_{j=1}^m rcv_{j}^{\text{new}} \right).
\]
In order to check for implementation faults, the signer SHOULD also check that
\[ \text{bvk} = \text{BindingSig}.\text{DerivePublic}(\text{bsk}). \]

Let \( \text{SigHash} \) be the SIGHASH transaction hash as defined in [ZIP-243], not associated with an input, using the SIGHASH type SIGHASH_ALL.

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

We now explain why this works.

A binding signature proves knowledge of the discrete logarithm \( \text{bsk} \) of \( \text{bvk} \) with respect to \( R \). That is, \( \text{bvk} = [\text{bsk}] R \).

So the value 0 and randomness \( \text{bsk} \) is an opening of the Pedersen commitment \( \text{bvk} = \text{ValueCommit}_{\text{bsk}}(0) \). 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 an opening to more than one value for any of those commitments, i.e. we may assume that \( v_{1\ldots n}^{\text{old}} \) are determined by \( \text{cv}_{1\ldots n}^{\text{old}} \), and that \( v_{1\ldots m}^{\text{new}} \) are determined by \( \text{cv}_{1\ldots m}^{\text{new}} \). We may also assume, from Knowledge Soundness of Groth16, that the Spend proofs could not have been generated without knowing \( r_{cv_{1\ldots n}}^{\text{old}} (\mod r_j) \), and the Output proofs could not have been generated without knowing \( r_{cv_{1\ldots m}}^{\text{new}} (\mod r_j) \).

Using the fact that \( \text{ValueCommit}_{cv}(v) = [v] \mathcal{V} \Phi [r_{cv}] R \), the expression for \( \text{bvk} \) above is equivalent to:

\[
\text{bvk} = \left[ \left( \bigoplus_{i=1}^{n} v_i^{\text{old}} \right) \bigoplus \left( \bigoplus_{j=1}^{m} v_j^{\text{new}} \right) \right] \mathcal{V} \Phi \left[ \left( \bigoplus_{i=1}^{n} r_{cv_i}^{\text{old}} \right) \bigoplus \left( \bigoplus_{j=1}^{m} r_{cv_j}^{\text{new}} \right) \right] R.
\]

Let \( v^* = \sum_{i=1}^{n} v_i^{\text{old}} - \sum_{j=1}^{m} v_j^{\text{new}} - v^{\text{balance}} \).

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

The above argument shows only that \( v^* = 0 (\mod r_j) \); in order to show that \( v^* = 0 \), we will also demonstrate that it does not overflow \( \{ -r_j - 1, \ldots, -1 \} \).

The Spend statements prove that all of \( v_{1\ldots n}^{\text{old}} \) are in \( \{ 0 \ldots 2^\ell_{\text{value}} - 1 \} \). Similarly the Output statements prove that all of \( v_{1\ldots m}^{\text{old}} \) are in \( \{ 0 \ldots 2^\ell_{\text{value}} - 1 \} \). \( v^{\text{balance}} \) is encoded in the transaction as a signed two’s complement 64-bit integer in the range \( \{-2^{63} \ldots 2^{63} - 1 \} \). \( \ell_{\text{value}} \) is defined as 64, so \( v^* \) is in the range \( \{-m \cdot (2^{64} - 1) - 2^{63} + 1 \ldots n \cdot (2^{64} - 1) + 2^{63} \} \). The maximum transaction size of 2 MB limits \( n \) to at most floor \( \frac{20000000}{384} \) = 5208 and \( m \) to at most floor \( \frac{20000000}{948} \) = 2109, ensuring \( v^* \in \{ -38913406623490299131842 \ldots 96079866507916199586728 \} \) which is a subrange of \( \{ -r_{j\cdot2^L} - 1, \ldots, -1 \} \).

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 \( \bigoplus \) sum of the value commitment randomnesses, authorized a transaction with the given SIGHASH transaction hash by signing 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) \( rcv_{1..n} \) and \( rcv_{1..m} \); this technique is also used in Bulletproofs [Dalek-notes].

4.13 Spend Authorization Signature

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. 64.

Knowledge of the spending key could have been proven directly in the Spend statement, similar to the check in § 4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42 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 BCTV14 or 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 SigHash be the SIGHASH transaction hash as defined in [ZIP-243], not associated with an input, using the SIGHASH type SIGHASH_ALL.

Let ask be the spend authorization private key as defined in § 4.2.2 'Sapling Key Components' on p. 28.

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

1. Choose \( \alpha \leftarrow \text{SpendAuthSig.GenRandom()} \).
2. Let rsk = SpendAuthSig.RandomizePrivate(\( \alpha \), ask).
3. Let rk = SpendAuthSig.DerivePublic(rsk).
4. Generate a proof \( \pi_{ZKSpend} \) of the Spend statement (§ 4.15.2 ‘Spend Statement (Sapling)’ on p. 43), with \( \alpha \) in the auxiliary input and rk in the primary input.
5. Let spendAuthSig = SpendAuthSig.Sign(rsk, SigHash).

The resulting spendAuthSig and \( \pi_{ZKSpend} \) 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-SNARK proof. In this case privacy will be lost to that party since it needs ak and the proof authorizing key rsk; 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. 18).

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:

\[ \rho := \text{MixingPedersenHash}(cm, pos). \]

MixingPedersenHash is defined in §5.4.1.8 ‘Mixing Pedersen Hash Function’ on p. 57.

Let PRF_{nf} and PRF_{nfSapling} be as instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 58.

For a Sprout note, the nullifier is derived as \( PRF_{a_{sk}}(\rho) \), where \( a_{sk} \) is the spending key associated with the note.

For a Sapling note, the nullifier is derived as \( PRF_{nk^*\mathcal{S}}(\rho^*) \), where \( nk^* \) is a representation of the nullifier deriving key associated with the note and \( \rho^* = \text{repr}_J(\rho) \).
4.15 Zk-SNARK Statements

4.15.1 JoinSplit Statement (Sprout)

Let \( \ell_{\text{MerkleSprout}}, \ell_{\text{PRFSprout}}, \text{MerkleDepth}^{\text{Sprout}}, \ell_{\text{value}}, \ell_{\text{a}}, \ell_{\text{q}}, \ell_{\text{hSig}}, N_{\text{old}}, N_{\text{new}} \) be as defined in §5.3 ‘Constants’ on p. 51.

Let \( \text{PRF}^{\text{addr}}, \text{PRF}^{n}, \text{PRF}^{pk}, \) and \( \text{PRF}^{0} \) be as defined in §4.1.2 ‘Pseudo Random Functions’ on p. 19.

Let \( \text{NoteCommitment}^{\text{Sprout}} \) be as defined in §4.1.7 ‘Commitment’ on p. 24. and let \( \text{NoteSprout} \) and \( \text{NoteCommitment}^{\text{Sprout}} \) be as defined in §3.2 ‘Notes’ on p. 13.

A valid instance of a JoinSplit statement, \( \pi_{\text{ZKJoinSplit}} \), ensures that given a primary input:

\[
\begin{align*}
(\text{rt} & : \mathbb{B}^{[\text{MerkleSprout}]],} \\
\text{n}_{\text{old}} & : \mathbb{B}^{[\ell_{\text{PRFSprout}}, \text{MerkleDepth}^{\text{Sprout}}, N_{\text{old}}]}, \\
\text{cm}_{\text{new}} & : \text{NoteCommitment}^{\text{Sprout}}.\text{Output}^{[N_{\text{new}}]}, \\
\text{v}_{\text{old}} & : \{0..2^{\ell_{\text{value}}}-1\}, \\
\text{v}_{\text{new}} & : \{0..2^{\ell_{\text{value}}}-1\}, \\
\text{h}_{\text{Sig}} & : \mathbb{B}^{[\text{hSig}]} \\
\text{h}_{1..N_{\text{old}}} & : \mathbb{B}^{[\ell_{\text{PRFSprout}}, \text{MerkleDepth}^{\text{Sprout}}, N_{\text{old}}]}, \\
\phi & : \mathbb{B}^{[\ell_{\phi}]} \\
\end{align*}
\]

\( \text{enforceMerklePath}_{1..N_{\text{old}}} : \mathbb{B}^{[N_{\text{old}}]} \)

where:

- for each \( i \in \{1..N_{\text{old}}\} \): \( \text{n}_{\text{old}} = (a_{\text{old},i}^{pk}, v_{\text{old},i}^{pk}, \rho_{\text{old},i}^{pk}, rcm_{\text{old},i}) \)
- for each \( i \in \{1..N_{\text{new}}\} \): \( \text{n}_{\text{new}} = (a_{\text{new},i}^{pk}, v_{\text{new},i}^{pk}, \rho_{\text{new},i}^{pk}, rcm_{\text{new},i}) \)

such that the following conditions hold:

**Merkle path validity** for each \( i \in \{1..N_{\text{old}}\} \) \( \text{enforceMerklePath}_{i} = 1 \) is a valid Merkle path (see §4.8 ‘Merkle Path Validity’ on p. 35) of depth \( \text{MerkleDepth}^{\text{Sprout}} \) from \( \text{NoteCommitment}^{\text{Sprout}}(\text{n}_{\text{old}}) \) to the anchor \( \text{rt} \).

**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 \( i \in \{1..N_{\text{old}}\} \) if \( v_{\text{old},i} \neq 0 \) then \( \text{enforceMerklePath}_{i} = 1 \).

**Balance** \( v_{\text{old}} + \sum_{i=1}^{N_{\text{old}}} v_{\text{old},i}^{pk} = v_{\text{new}} + \sum_{i=1}^{N_{\text{new}}} v_{\text{new},i}^{pk} \in \{0..2^{\ell_{\text{value}}}-1\} \).

**Nullifier integrity** for each \( i \in \{1..N_{\text{old}}\} \): \( nf_{\text{old},i} = \text{PRF}_{\text{addr}}^{nf} \left( a_{\text{old},i}^{pk} \right) \).

**Spend authority** for each \( i \in \{1..N_{\text{old}}\} \): \( a_{\text{old},i}^{pk} = \text{PRF}_{\text{addr}}^{\text{addr}}(0) \).

**Non-malleability** for each \( i \in \{1..N_{\text{old}}\} \): \( h_{\text{old}} = \text{PRF}_{\text{addr}}^{\text{addr}} (i, h_{\text{Sig}}) \).

**Uniqueness of \( \rho_{\text{new}} \)** for each \( i \in \{1..N_{\text{new}}\} \): \( \rho_{\text{new},i}^{pk} = \text{PRF}_{\text{addr}}^{\text{addr}}(i, h_{\text{Sig}}) \).

**Note commitment integrity** for each \( i \in \{1..N_{\text{new}}\} \): \( \text{cm}_{\text{new}}^{\text{new}} = \text{NoteCommitment}^{\text{Sprout}}(\text{n}_{\text{new}}) \).

For details of the form and encoding of proofs, see §5.4.9.1 ‘BCTV14’ on p. 72.
4.15.2 Spend Statement (Sapling)

Let \( \ell_{\text{MerkleSapling}} \), \( \ell_{\text{PRFnfSapling}} \), and \( \ell_{\text{scalar}} \) be as defined in §5.3 ‘Constants’ on p. 51.

Let ValueCommit and NoteCommit\(^{\text{Sapling}}\) be as specified in §4.1.7 ‘Commitment’ on p. 24.

Let SpendAuthSig be as defined in §5.4.6.1 ‘Spend Authorization Signature’ on p. 64.

Let \( \mathcal{J}, \mathcal{J}^{(r)} \), \( \mathcal{B} \), \( \mathcal{R}_{\text{nfSapling}} \), and \( h_1 \) be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

Let \( \text{Extract}_{\mathcal{J}^{(r)}} : \mathcal{J}^{(r)} \rightarrow \mathbb{B}[\ell_{\text{MerkleSapling}}] \) be as defined in §5.4.8.4 ‘Hash Extractor for Jubjub’ on p. 70.

Let \( \mathcal{H} \) be as defined in §4.2.2 ‘Sapling Key Components’ on p. 28.

A valid instance of a Spend statement, \( \pi_{\text{ZKSpend}} \), assures that given a primary input:

\[
(\mathbf{rt} : \mathbb{B}[\ell_{\text{MerkleSapling}}], \mathbf{cv}^{\text{old}} : \text{ValueCommit}.\text{Output}, \mathbf{nf}^{\text{old}} : \mathbb{B}[\ell_{\text{PRFnfSapling}}], \mathbf{rk} : \text{SpendAuthSig}.\text{Public})
\]

the prover knows an auxiliary input:

\[
(\mathbf{path} : \mathbb{B}[\ell_{\text{MerkleDepth}}^{\text{Sapling}}], \mathbf{pos} : \{0 \ldots 2^{\ell_{\text{MerkleDepth}}^{\text{Sapling}}} - 1\}, \mathbf{g}_d : \mathcal{J}, \mathbf{pk}_d : \mathcal{J}, \mathbf{v}^{\text{old}} : \{0 \ldots 2^{\ell_{\text{value}} - 1}\}, \mathbf{rcv}^{\text{old}} : \{0 \ldots 2^{\ell_{\text{scalar}} - 1}\}, \mathbf{cm}^{\text{old}} : \mathcal{J}, \mathbf{rcm}^{\text{old}} : \{0 \ldots 2^{\ell_{\text{scalar}} - 1}\}, \mathbf{\alpha} : \{0 \ldots 2^{\ell_{\text{scalar}} - 1}\}, \mathbf{ak} : \text{SpendAuthSig}.\text{Public}, \mathbf{nsk} : \{0 \ldots 2^{\ell_{\text{scalar}} - 1}\})
\]

such that the following conditions hold:

Note commitment integrity \( \mathbf{cm}^{\text{old}} = \text{NoteCommit}_{\text{Sapling}}^{\text{nfSapling}} (\mathbf{repr}_d(g_d), \mathbf{repr}_d(pk_d), \mathbf{v}^{\text{old}}) \).

Merkle path validity Either \( \mathbf{v}^{\text{old}} = 0 \); or (\( \mathbf{path}, \mathbf{pos} \)) is a valid Merkle path of depth \( \ell_{\text{MerkleDepth}}^{\text{Sapling}} \), as defined in §4.8 ‘Merkle Path Validity’ on p. 35, from \( \mathbf{cm}_u = \text{Extract}_{\mathcal{J}^{(r)}}(\mathbf{cm}^{\text{old}}) \) to the anchor \( \mathbf{rt} \).

Value commitment integrity \( \mathbf{cv}^{\text{old}} = \text{ValueCommit}_{\text{nfSapling}}^{\text{nfSapling}}(\mathbf{v}^{\text{old}}) \).

Small order checks \( \mathbf{g}_d \) and \( \mathbf{ak} \) are not of small order, i.e. \( [h_j] \mathbf{g}_d \neq \mathcal{O}_j \) and \( [h_j] \mathbf{ak} \neq \mathcal{O}_j \).

Nullifier integrity \( \mathbf{nf}^{\text{old}} = \text{PRF}_{\text{nfSapling}}(\mathbf{\rho}^*) \) where \( \mathbf{nk}^* = \text{repr}_d([\mathbf{nsk}] \mathcal{H}) \)
\( \mathbf{\rho}^* = \text{repr}_d(\text{MixingPedersenHash}(\mathbf{cm}^{\text{old}}, \mathbf{pos})) \).

Spend authority \( \mathbf{rk} = \text{SpendAuthSig}.\text{RandomizePublic}(\mathbf{\alpha}, \mathbf{ak}) \).

Diversified address integrity \( \mathbf{pk}_d = [\mathbf{ivk}] \mathbf{g}_d \) where \( \mathbf{ivk} = \text{CRH}_{\mathcal{ivk}}(\mathbf{ak}^*, \mathbf{nk}^*) \)
\( \mathbf{ak}^* = \text{repr}_d(\mathbf{ak}) \).

For details of the form and encoding of Spend statement proofs, see §5.4.9.2 ‘Groth16’ on p. 73.
Notes:

- Public and auxiliary inputs MUST be constrained to have the types specified. In particular, see §A.3.3.2 'ctEdwards (de)compression and validation' on p.133, for required validity checks on compressed representations of Jubjub curve points.

  The ValueCommit.Output and 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 \( \{0..r - 1\} \)) of the integer from the previous layer.

- It is not checked in the Spend statement that \( r_k \) is not of small order. However, this is checked outside the Spend statement, as specified in §4.4 'Spend Descriptions' on p.31.

- It is not checked that \( rcv^{old} < r_j \) or that \( rcm^{old} < r_j \).

- SpendAuthSig.RandomizePublic(\( \alpha, ak \)) = \( ak + [\alpha] G \). (\( G \) is as defined in §5.4.6.1 'Spend Authorization Signature' on p.64.)

4.15.3 Output Statement (Sapling)

Let \( \ell_{\text{MerkleSapling}} \), \( \ell_{\text{PRFnSapling}} \), and \( \ell_{\text{scalar}} \) be as defined in §5.3 'Constants' on p.51.

Let ValueCommit and NoteCommit \( \text{Sapling} \) be as specified in §4.1.7 'Commitment' on p.24.

Let \( \mathbb{J}, \text{repr}_J, \text{and } h_J \) be as defined in §5.4.8.3 'Jubjub' on p.69.

A valid instance of an Output statement, \( \pi_{\text{ZKOutput}} \), assures that given a primary input:

\[
\begin{align*}
(cv^{new} : \text{ValueCommit}.Output, \\
cm_u : \mathbb{B}[\ell_{\text{MerkleSapling}}], \\
epk : \mathbb{J})
\end{align*}
\]

the prover knows an auxiliary input:

\[
\begin{align*}
(g_d : \mathbb{J}, \\
pk*_{d} : \mathbb{B}[\ell_{J}], \\
v^{new} : \{0..2^{\ell_{value}} - 1\}, \\
r cv^{new} : \{0..2^{\ell_{scalar}} - 1\}, \\
r cm^{new} : \{0..2^{\ell_{scalar}} - 1\}, \\
esk : \{0..2^{\ell_{scalar}} - 1\})
\end{align*}
\]

such that the following conditions hold:

Note commitment integrity \( cm_u = \text{Extract}_{J^{0,1}}(\text{NoteCommit}^{\text{Sapling}}(g_{*d}, pk*_{d}, v^{new})) \), where \( g_{*d} = \text{repr}_J(g_d) \).

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

Small order check \( g_d \) is not of small order, i.e. \( [h_J] g_d \neq O_J \).

Ephemeral public key integrity \( epk = [esk] g_d \).

For details of the form and encoding of Output statement proofs, see §5.4.9.2 'Groth16' on p.73.
Notes:

- Public and auxiliary inputs MUST be constrained to have the types specified. In particular, see §A.3.3.2 ‘ctEdwards [de]compression and validation’ on p.133, for required validity checks on compressed representations of Jubjub curve points.

  The ValueCommit.Output type also represents points, i.e. J.

- The validity of pk⋆d is not checked in this circuit.

- It is not checked that rcv^old < r_j or that rcm^old < r_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, ρ, and rcm. A memo field (§3.2.1 ‘Note Plaintexts and Memo Fields’ on p.14) is also transmitted.

To transmit these secrets securely to a recipient without requiring an out-of-band communication channel, the transmission key 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 N^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 ‘Symmetric Encryption’ on p.60;
- let KDF^Sprout be the Key Derivation Function instantiated in §5.4.4.2 ‘Sprout Key Derivation’ on p.60;
- let KA^Sprout be the key agreement scheme instantiated in §5.4.4.1 ‘Sprout Key Agreement’ on p.60;
- let h_Sig be the value computed for this JoinSplit description in §4.3 ‘JoinSplit Descriptions’ on p.30.

4.16.1 Encryption (Sprout)

Let KA^Sprout be the key agreement scheme instantiated in §5.4.4.1 ‘Sprout Key Agreement’ on p.60.

Let pk^new_enc,1..N^new be the transmission keys for the intended recipient addresses of each new note.

Let np,1..N^new be Sprout note plaintexts defined in §5.5 ‘Encodings of Note Plaintexts and Memo Fields’ on p.73.

Then to encrypt:

- Generate a new KA^Sprout (public, private) key pair (epk, esk).
- For i ∈ {1..N^new},
  - Let P_{i}^enc be the raw encoding of np_{i}.
  - Let sharedSecret_{i} := KA^Sprout.Agree(esk, pk^new_enc,i).
  - Let K_{i}^enc := KDF^Sprout(i, h_Sig, sharedSecret_{i}, epk, pk^new_enc,i).
  - Let C_{i}^enc := Sym.Encrypt_{K_{i}^enc}(P_{i}^enc).

The resulting transmitted notes ciphertext is (epk, C_{1..N^new}^enc).

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

Let ivk = (a_{pk}, sk_{enc}) be the recipient’s incoming viewing key, and let pk_{enc} be the corresponding transmission key derived from sk_{enc} as specified in §4.2.1 ‘Sprout Key Components’ on p. 28.

Let cm_i^{\text{new}} be the note commitments of each output coin.

Then for each i ∈ {1..N^{\text{new}}}, the recipient will attempt to decrypt that ciphertext component (epk, C_i^{\text{enc}}) as follows:

let sharedSecret_i = KA_{\text{Sprout}} . Agree(sk_{enc}, epk)
let K_i^{\text{enc}} = KDF_{\text{Sprout}} (i, h_{Sig}, sharedSecret_i, epk, pk_{enc})
return DecryptNoteSprout(K_i^{\text{enc}}, C_i^{\text{enc}}, cm_i^{\text{new}}, a_{pk}).

DecryptNoteSprout(K_i^{\text{enc}}, C_i^{\text{enc}}, cm_i^{\text{new}}, a_{pk}) is defined as follows:

let P_i^{\text{enc}} = Sym. Decrypt_{K_i^{\text{enc}}}(C_i^{\text{enc}})
if P_i^{\text{enc}} = \bot, return \bot
extract np_i = (v_i^{\text{new}} : \{0..2^{\text{value}} - 1\}, p_i^{\text{new}} : \mathbb{B}[{\text{PRFSprout}}], rcm_i^{\text{new}} : \text{NoteCommit}_{\text{Sprout}} . \text{Trapdoor}, \text{memo}_i : \mathbb{B}[{\text{Sprout}}]) from P_i^{\text{enc}}
if \text{NoteCommit}_{\text{Sprout}} ((a_{pk}, v_i^{\text{new}}, p_i^{\text{new}}, rcm_i^{\text{new}})) \neq cm_i^{\text{new}}, return \bot, else return np_i.

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

Notes:

- The decryption algorithm corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in [BCGGMTV2014, Figure 2].

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

See §8.7 ‘In-band secret distribution’ on p. 98 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. 14) is also transmitted.

To transmit these secrets securely to a recipient without requiring an out-of-band communication channel, the diversified transmission key pk_{iv} 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 by a fresh ephemeral public key.

For both encryption and decryption,

- let \ell_{\text{epk}} be as defined in §5.3 ‘Constants’ on p. 51;
- let Sym be the scheme instantiated in §5.4.3 ‘Symmetric Encryption’ on p. 60;
- let KDF_{\text{Sapling}} be the Key Derivation Function instantiated in §5.4.4.4 ‘Sapling Key Derivation’ on p. 61;
- let KA_{\text{Sapling}} be the key agreement scheme instantiated in §5.4.3.3 ‘Sapling Key Agreement’ on p. 61;
- let \ell_J and repr_J be as defined in §5.4.8.3 ‘Jubjub’ on p. 69;
- let Extract_{\ell_J} be as defined in §5.4.8.4 ‘Hash Extractor for Jubjub’ on p. 70;
- let PRF_{\ell_J} be as instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 58.
4.17.1 Encryption (Sapling)

Let \( pk_d^{\text{new}} : KA_{\text{Sapling}} \). PublicPrimeOrder be the _diversified transmission key_ for the intended recipient address of a new Sapling note, and let \( g_d^{\text{new}} : KA_{\text{Sapling}} \). PublicPrimeOrder be the corresponding _diversified base_ computed as DiversifyHash(d).

Since Sapling note encryption is used only in the context of §4.6.2 ‘Sending Notes (Sapling)’ on p. 33, we may assume that \( g_d^{\text{new}} \) has already been calculated and is not \perp.

Let \( ovk : \mathbb{G}^{\ell \cdot (\ell + 256)/8} \cup \{ \perp \} \) be as described in §4.6.2 ‘Sending Notes (Sapling)’ on p. 33, 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 \( np = (d, v, rcm, memo) \) be the Sapling note plaintext.

\( np \) is encoded as defined in §5.5 ‘Encodings of Note Plaintexts and Memo Fields’ on p. 73.

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

Then to encrypt:

choose a uniformly random ephemeral private key \( esk \overset{\$}{\in} \mathbb{K} A_{\text{Sapling}} \). Private \( \setminus \{ 0 \} \)

let \( epk = KA_{\text{Sapling}} \). DerivePublic(\( esk, g_d^{\text{new}} \))

let \( P^{\text{enc}} \) be the raw encoding of \( np \)

let \( sharedSecret = KA_{\text{Sapling}} \). Agree(\( esk, pk_d^{\text{new}} \))

let \( K^{\text{enc}} = KDF_{\text{Sapling}}(sharedSecret, epk) \)

let \( C^{\text{enc}} = \text{Sym.Encrypt}_{k^{\text{new}}}(P^{\text{enc}}) \)

if \( ovk = \perp \):

choose random \( ock \overset{\$}{\in} \text{Sym.K} \) and \( op \overset{\$}{\in} \mathbb{G}^{\ell + 256}/8 \)

else:

let \( cv = \text{LEBS2OSP}_{\ell_j}(\text{repr}_j(cm^{\text{new}})) \)

let \( cmu = \text{LEBS2OSP}_{256}(\text{Extract}_{\ell_j}(cm^{\text{new}})) \)

let \( ephemeralKey = \text{LEBS2OSP}_{\ell_j}(\text{repr}_j(epk)) \)

let \( ock = \text{PRF}_{ovk}(cv, cmu, ephemeralKey) \)

let \( op = \text{LEBS2OSP}_{\ell_j + 256}(\text{repr}_j(pk_d^{\text{new}}) \parallel \text{I2LEBS}_{256}(esk)) \)

let \( C^{\text{out}} = \text{Sym.Encrypt}_{ock}(op) \)

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

**Note:** It is technically possible to replace \( C^{\text{enc}} \) 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 \( ivk : \{ 0 .. 2^{ivk} - 1 \} \) be the recipient’s incoming viewing key, as specified in §4.2.2 ‘Sapling Key Components’ on p. 28.

Let \( (epk, C^{\text{enc}}, C^{\text{out}}) \) be the transmitted note ciphertext from the Output description. Let \( cmu \) be that field of the Output description (encoding the \( u \)-coordinate of the note commitment).
The recipient will attempt to decrypt the epk and $C^{\text{enc}}$ components of the transmitted note ciphertext as follows:

let $\text{sharedSecret} = K^A_{\text{Sapling}}.\text{Agree}(ivk, epk)$
let $K^{\text{enc}} = \text{KDF}^\text{Sapling}(\text{sharedSecret, epk})$
let $P^{\text{enc}} = \text{Sym}.\text{Decrypt}_{K^{\text{enc}}}(C^{\text{enc}})$
if $P^{\text{enc}} = \perp$, return $\perp$
extract $np = (d : \mathbb{B}[|t|], v : \{0 .. 2^{t_{\text{value}}}-1 \}, rcm : \mathbb{B}^{[32]}, \text{memo} : \mathbb{B}^{[512]})$ from $P^{\text{enc}}$
let $rcm = \text{LEOS2IP}_{256}(rcm)$ and $g_d = \text{DiversifyHash}(d)$
if $rcm \geq r_j$ or $g_d = \perp$, return $\perp$
let $pk_d = K^A_{\text{Sapling}}.\text{DerivePublic}(ivk, g_d)$
let $cm'_u = \text{Extract}_{32}(\text{NoteCommit}^\text{Sapling}_{rcm}(\text{repr}_j(g_d), \text{repr}_j(pk_d), v))$.
if $\text{LEBS2OSP}_{256}(cm'_u) \neq cm_u$, return $\perp$, else return $np$.

A received Sapling note is necessarily a positioned note, and so its $p$ value can immediately be calculated as described in §4.14 ‘Note Commitments and Nullifiers’ on p. 41.

To test whether a Sapling note is unspent in a particular block chain also requires the nullifier deriving key $nk^*$; the coin is unspent if and only if $nf = \text{PRF}^\text{Sapling}(\text{repr}_j(p))$ is not in the nullifier set for that block chain.

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

4.17.3 Decryption using a Full Viewing Key (Sapling)

Let $ovk : \mathbb{B}^{[\text{ovk}/8]}$ be the outgoing viewing key, as specified in §4.2.2 ‘Sapling Key Components’ on p. 28, that is to be used for decryption. (If $ovk = \perp$ was used for encryption, the payment is not decryptable by this method.)

Let $(epk, C^{\text{enc}}, C^{\text{out}})$ be the transmitted note ciphertext, and let $cv, cmu$, and ephemeralKey be those fields 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 $ock = \text{PRF}^\text{Sapling}_{ovk}(cv, cmu, ephemeralKey)$
let $op = \text{Sym}.\text{Decrypt}_{ock}(C^{\text{out}})$
if $op = \perp$, return $\perp$
extract $(pk_*d : B[|t|], esk : \mathbb{B}^{[32]})$ from $op$
let $esk = \text{LEOS2IP}_{256}(esk)$ and $pk_d = \text{abst}_3(pk_*d)$
if $esk \geq r_j$ or $pk_d \notin K^A_{\text{Sapling}}.\text{PublicPrimeOrder}$, return $\perp$
let $\text{sharedSecret} = K^A_{\text{Sapling}}.\text{Agree}(esk, pk_d)$
let $K^{\text{enc}} = \text{KDF}^\text{Sapling}(\text{sharedSecret, epk})$
let $P^{\text{enc}} = \text{Sym}.\text{Decrypt}_{K^{\text{enc}}}(C^{\text{enc}})$
if $P^{\text{enc}} = \perp$, return $\perp$
extract $np = (d : \mathbb{B}[|t|], v : \{0 .. 2^{t_{\text{value}}}-1 \}, rcm : \mathbb{B}^{[32]}, \text{memo} : \mathbb{B}^{[512]})$ from $P^{\text{enc}}$
let $rcm = \text{LEOS2IP}_{256}(rcm)$ and $g_d = \text{DiversifyHash}(d)$
if $rcm \geq r_j$ or $g_d = \perp$, return $\perp$
if $K^A_{\text{Sapling}}.\text{DerivePublic}(esk, g_d) \neq epk$, return $\perp$
let $cm'_u = \text{Extract}_{32}(\text{NoteCommit}^\text{Sapling}_{rcm}(\text{repr}_j(g_d), \text{repr}_j(pk_d), v))$.
if $\text{LEBS2OSP}_{256}(cm'_u) \neq cm_u$, return $\perp$, else return $np$.

Note: For a valid transaction it must be the case that ephemeralKey = $\text{LEBS2OSP}_{64}(\text{repr}_j(epk))$. 
4.18 Block Chain Scanning (Sprout)

Let $\ell_{\text{PRFSprout}}$ be as defined in § 5.3 ‘Constants’ on p. 51.
Let $\text{Note}^{\text{Sprout}}$ be as defined in § 3.2 ‘Notes’ on p. 13.
Let $\text{KA}^{\text{Sprout}}$ be as defined in § 5.4.1 ‘Sprout Key Agreement’ on p. 60.

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

Let $ivk = (a_{pk} \cdot B^{[\ell_{\text{PRFSprout}}]}, sk_{\text{enc}} \cdot \text{KA}^{\text{Sprout}}.\text{Private})$ be the incoming viewing key corresponding to $a_{sk}$, and let $pk_{\text{enc}}$ be the associated transmission key, as specified in § 4.2.1 ‘Sprout Key Components’ on p. 28.

Initialize $\text{ReceivedSet} : \mathcal{P}(\text{Note}^{\text{Sprout}} \times B^{[512]}) = \{}$.
Initialize $\text{SpentSet} : \mathcal{P}(\text{Note}^{\text{Sprout}}) = \{}$.
Initialize $\text{NullifierMap} : B^{[\ell_{\text{PRFSprout}}]} \rightarrow \text{Note}^{\text{Sprout}}$ to the empty mapping.

For each transaction $tx$,
For each JoinSplit description in $tx$,
Let $(epk, C^{\text{enc}}_{\text{new}})$ be the transmitted notes ciphertext of the JoinSplit description.
For $i$ in $1..N_{\text{new}}$,
 Attempt to decrypt the transmitted notes ciphertext component $(epk, C^{\text{enc}}_{i})$ using $ivk$ with the algorithm in § 4.16.2 ‘Decryption (Sprout)’ on p. 46. If this succeeds giving $np$:
 Extract $n$ and memo : $B^{[512]}$ from $np$ (taking the $a_{pk}$ field of the note to be $a_{pk}$ from $ivk$).
 Add $(n, \text{memo})$ to $\text{ReceivedSet}$.
 Calculate the nullifier $nf$ of $n$ using $a_{sk}$ as described in § 3.2 ‘Notes’ on p. 13.
 Add the mapping $nf \rightarrow n$ to $\text{NullifierMap}$.

Let $nf_{1..N_{\text{old}}}$ be the nullifiers of the JoinSplit description.
For $i$ in $1..N_{\text{old}}$,
 If $nf_i$ is present in $\text{NullifierMap}$, add $\text{NullifierMap}(nf_i)$ to $\text{SpentSet}$.

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

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. 28.

Let $\ell_{\text{PRFSnapling}}$ be as defined in § 5.3 ‘Constants’ on p. 51.
Let $\text{Note}^{\text{Sapling}}$ be as defined in § 3.2 ‘Notes’ on p. 13.
Let $\text{KA}^{\text{Sapling}}$ be as defined in § 5.4.3 ‘Sapling Key Agreement’ on p. 61.
The following algorithm can be used, given the block chain and \((nk : \mathbb{J}^{r}, ivk : \{0..2^{64} - 1\})\), to obtain each note sent to the corresponding shielded payment address, its memo field field, and its final status (spent or unspent).

Initialize ReceivedSet : \(\mathcal{P}(\text{Note}^{\text{Sapling}} \times \mathbb{E}^{[512]}) = \{}\).

Initialize SpentSet : \(\mathcal{P}(\text{Note}^{\text{Sapling}}) = \{}\).

Initialize NullifierMap : \(\mathbb{F}^{[\text{NullifierSapling}]} \rightarrow \text{Note}^{\text{Sapling}}\) to the empty mapping.

For each transaction \(tx\).

For each Output description in \(tx\) with note position \(pos\),

Attempt to decrypt the transmitted note ciphertext components \(epk\) and \(C_{\text{enc}}\) using \(ivk\) with the algorithm in § 4.17.2 'Decryption using an Incoming Viewing Key (Sapling)' on p. 47. If this succeeds giving \(np\):

Extract \(n\) and \(memo : \mathbb{E}^{[512]}\) from \(np\).

Add \((n, memo)\) to ReceivedSet.

Calculate the nullifier \(nf\) of \(n\) using \(nk\) and \(pos\) as described in § 3.2 'Notes' on p. 13.

Add the mapping \(nf \rightarrow n\) to NullifierMap.

For each Spend description in \(tx\),

Let \(nf\) be the nullifier of the Spend description.

If \(nf\) is present in NullifierMap, add NullifierMap(\(nf\)) to SpentSet.

Return \((\text{ReceivedSet}, \text{SpentSet})\).

Non-normative notes:

- The above algorithm does not use the ovk key component, or the \(C_{\text{out}}\) 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. 48, 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 \(C_{\text{out}}\) components for each transaction, in order to obtain information about notes that were spent in the scanned period but received outside it.

- The above algorithm does not detect notes that were sent "out-of-band" or with incorrect transmitted note ciphertexts. It is possible to detect whether such notes were spent only if their nullifiers are known.

5 Concrete Protocol

5.1 Caution

TODO: Explain the kind of things that can go wrong with linkage between abstract and concrete protocol. E.g. § 8.5 'Internal hash collision attack and fix' on p. 96

5.2 Integers, Bit Sequences, and Endianness

All integers in Zcash-specific encodings are unsigned, have a fixed bit length, and are encoded in little-endian byte order unless otherwise specified.
The following functions convert between sequences of bits, sequences of bytes, and integers:

- **I2LEBSP** : \((\ell : N) \times \{0 .. 2^\ell - 1\} \rightarrow B[\ell]\), such that \(\text{I2LEBSP}_\ell(x)\) is the sequence of \(\ell\) bits representing \(x\) in little-endian order;
- **I2BEBSP** : \((\ell : N) \times \{0 .. 2^\ell - 1\} \rightarrow B[\ell]\) such that \(\text{I2BEBSP}_\ell(x)\) is the sequence of \(\ell\) bits representing \(x\) in big-endian order.
- **LEBS2IP** : \((\ell : N) \times B[\ell] \rightarrow \{0 .. 2^\ell - 1\}\) such that \(\text{LEBS2IP}_\ell(S)\) is the integer represented in little-endian order by the bit sequence \(S\) of length \(\ell\).
- **LEOS2IP** : \((\ell : N) \times B[\ell] \rightarrow \{0 .. 2^\ell - 1\}\) such that \(\text{LEOS2IP}_\ell(S)\) is the integer represented in little-endian order by the byte sequence \(S\) of length \(\ell/8\).
- **LEBS2OSP** : \((\ell : N) \times B[\ell] \rightarrow B[\text{ceiling}(\ell/8)]\) defined as follows: pad the input on the right with \(8 \cdot \text{ceiling}(\ell/8) - \ell\) 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.
- **LEOS2BSP** : \((\ell : N) \times B[\text{ceiling}(\ell/8)] \rightarrow B[\ell]\) defined as follows: convert each byte to a group of 8 bits with the least significant bit first, and concatenate the resulting groups in the same order as the bytes.

In bit layout diagrams, each box of the diagram represents a sequence of bits. Diagrams are read from left-to-right, with lines read from top-to-bottom; the breaking of boxes across lines has no significance. The bit length \(\ell\) is given explicitly in each box, except when it is obvious (e.g. for a single bit, or for the notation \([0]^{\ell}\) representing the sequence of \(\ell\) zero bits, or for the output of \(\text{LEBS2OSP}_\ell\)).

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:

```
MerkleDepth^{Sprout} : N := 29
MerkleDepth^{Sapling} : N := 32
N^{old} : N := 2
N^{new} : N := 2
\ell^{value} : N := 64
\ell^{MerkleSprout} : N := 256
\ell^{MerkleSapling} : N := 255
\ell^{hSig} : N := 256
\ell^{PRFSprout} : N := 256
\ell^{PRFexpand} : N := 512
\ell^{PRFmSapling} : N := 256
\ell^{rcm} : N := 256
\ell^{Seed} : N := 256
\ell^{sk} : N := 252
\ell^{ip} : N := 252
\ell^{sk} : N := 256
```
\[ \ell_d : N := 88 \]
\[ \ell_{ivk} : N := 251 \]
\[ \ell_{ovk} : N := 256 \]
\[ \ell_{scalar} : N := 252 \]

Uncommitted^Sprout_ : \mathbb{B}^{[\ell_{MerkleSprout}]} := \{0\}^{\ell_{MerkleSprout}}

Uncommitted^Sapling_ : \mathbb{B}^{[\ell_{MerkleSapling}]} := \text{I2LEBSP}_{\ell_{MerkleSapling}}(1)

\[ \text{MAX\_MONEY} : N := 2.1 \cdot 10^{15} \text{ (zatoshi)} \]

\[ \text{BlossomActivationHeight} : N := \begin{cases} 653600, & \text{for the production network} \\ 584000, & \text{for the test network} \end{cases} \]

\[ \text{SlowStartInterval} : N := 20000 \]
\[ \text{PreBlossomHalvingInterval} : N := 840000 \]
\[ \text{MaxBlockSubsidy} : N := 1.25 \cdot 10^9 \text{ (zatoshi)} \]
\[ \text{NumFounderAddresses} : N := 48 \]
\[ \text{FoundersFraction} : Q := \frac{1}{5} \]
\[ \text{PoWLimit} : N := \begin{cases} 2^{243} - 1, & \text{for the production network} \\ 2^{251} - 1, & \text{for the test network} \end{cases} \]
\[ \text{PoWAverageWindow} : N := 17 \]
\[ \text{PoWMedianBlockSpan} : N := 11 \]
\[ \text{PoWMaxAdjustDown} : Q := \frac{32}{100} \]
\[ \text{PoWMaxAdjustUp} : Q := \frac{16}{100} \]
\[ \text{PoWDampingFactor} : N := 4 \]
\[ \text{PreBlossomPoWTargetSpacing} : N := 150 \text{ (seconds)}. \]
\[ \text{PostBlossomPoWTargetSpacing} : N := 75 \text{ (seconds)}. \]

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

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

SHA-256 is defined by [NIST2015].

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

\[ \text{SHA-256} : \mathbb{B}^{[N]} \rightarrow \mathbb{B}^{[32]} \]

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

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

\[ \text{SHA-256d} : \mathbb{B}^{[N]} \rightarrow \mathbb{B}^{[32]} \]
Zcash also uses the SHA-256 compression function, SHA256Compress. This operates on a single 512-bit block and 
*excludes* the padding step specified in [NIST2015, section 5.1].

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

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

\[
\text{SHA256Compress} : \mathbb{B}^{[512]} \rightarrow \mathbb{B}^{[256]}
\]

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

### 5.4.1.2 BLAKE2 Hash Functions

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

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

BLAKE2b is used to instantiate \(h\text{SigCRH}, \text{EquihashGen}, \text{and KDF}\text{Sprout}\). 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 \(\text{PRF}\text{expand}, \text{PRFlock}, \text{KDF}\text{Sapling}\), and in the RedJubjub *signature scheme* which instantiates 
\(\text{SpendAuthSig}\) and \(\text{BindingSig}\).

\[
\text{BLAKE2b-}\ell : \mathbb{B}^{[16]} \times \mathbb{B}^{[N]} \rightarrow \mathbb{B}^{[\ell/8]}
\]

**Note:** BLAKE2b-\(\ell\) is not the same as BLAKE2b-512 truncated to \(\ell\) bits, because the digest length is encoded in the
parameter block.

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

BLAKE2s is used to instantiate \(\text{PRF}_{\text{nfSapling}}, \text{CRH}^{ivk}, \text{and GroupHash}^{j(r)}\).

\[
\text{BLAKE2s-}\ell : \mathbb{B}^{[8]} \times \mathbb{B}^{[N]} \rightarrow \mathbb{B}^{[\ell/8]}
\]

### 5.4.1.3 Merkle Tree Hash Function

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

**MerkleCRH\textsuperscript{Sprout} Hash Function**

Let SHA256Compress be as specified in §5.4.1.1 ‘SHA-256, SHA-256d, and SHA256Compress Hash Functions’ on p. 52.

MerkleCRH\textsuperscript{Sprout} : \(\{0 \ldots \text{MerkleDepth}^{\text{Sprout}} - 1\} \times \mathbb{B}^{[\text{MerkleDepth}^{\text{Sprout}}]} \times \mathbb{B}^{[\text{MerkleDepth}^{\text{Sprout}}]} \rightarrow \mathbb{B}^{[\text{MerkleDepth}^{\text{Sprout}}]}\) is defined as follows:

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

**Security requirement:** SHA256Compress must be *collision-resistant*, and it must be infeasible to find a preimage
\(x\) such that 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.
MerkleCRH\textsuperscript{Sapling} Hash Function

Let PedersenHash be as specified in §5.4.1.7 ‘Pedersen Hash Function’ on p. 56.

\[
\text{MerkleCRH}\textsuperscript{Sapling} : \{0 \ldots \text{MerkleDepth}\textsuperscript{Sapling} - 1\} \times \mathbb{B}^{\lceil \text{MerkleDepth}\textsuperscript{Sapling} \rceil} \times \mathbb{B}^{\lceil \text{MerkleDepth}\textsuperscript{Sapling} \rceil} \to \mathbb{B}^{\lceil \text{MerkleDepth}\textsuperscript{Sapling} \rceil}
\]

is defined as follows:

\[
\text{MerkleCRH}\textsuperscript{Sapling}(\text{layer}, \text{left}, \text{right}) := \text{PedersenHash}(\text{“Zcash\_PH”}, l || \text{left} || \text{right})
\]

where \(l = \text{I2LEBS}\_6(\text{MerkleDepth}\textsuperscript{Sapling} - 1 - \text{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. NoteCommit\textsuperscript{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. 65.

5.4.1.4 \(h_{\text{Sig}}\) Hash Function

\(h_{\text{SigCRH}}\) is used to compute the value \(h_{\text{Sig}}\) in §4.3 ‘JoinSplit Descriptions’ on p. 30.

\[
\text{hSigCRH}(\text{randomSeed}, n_{\text{old}}^{1..N_{\text{old}}}, \text{joinSplitPubKey}) := \text{BLAKE2}-256(\text{“ZcashComputehSig”}, \text{hSigInput})
\]

where

\[
\text{hSigInput} := \begin{array}{|c|c|c|}
\text{256-bit randomSeed} & \text{256-bit } n_{\text{old}}^{1} & \ldots & \text{256-bit } n_{\text{old}}^{N_{\text{old}}} & \text{256-bit joinSplitPubKey}
\end{array}
\]

BLAKE2b-256(\(p, x\)) is defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53.

Security requirement: BLAKE2b-256(“ZcashComputehSig”, \(x\)) must be collision-resistant on \(x\).

5.4.1.5 \(CRH^{ivk}\) Hash Function

\(CRH^{ivk}\) is used to derive the incoming viewing key \(ivk\) for a Sapling shielded payment address. For its use when generating an address see §4.2.2 ‘Sapling Key Components’ on p. 28, and for its use in the Spend statement see §4.15.2 ‘Spend Statement (Sapling)’ on p. 43.

It is defined as follows:

\[
\text{CRH}^{ivk}(ak^*, nk^*) := \text{LEOS}\_2\_256(\text{BLAKE2s}-256(\text{“Zcashivk”}, \text{crhInput})) \mod 2^{f_{ivk}}
\]

where

\[
\text{crhInput} := \begin{array}{|c|c|}
\text{LEBS}\_2\_256(ak^*) & \text{LEBS}\_2\_256(nk^*)
\end{array}
\]

BLAKE2b-256(\(p, x\)) is defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53.

Security requirement: \(\text{LEOS}\_2\_256(\text{BLAKE2s}-256(\text{“Zcashivk”}, x)) \mod 2^{f_{ivk}}\) must be collision-resistant on a 64-byte input \(x\). Note that this does not follow from collision resistance of BLAKE2s-256 (and the best possible concrete security is that of a 251-bit hash rather than a 256-bit hash), but it is a reasonable assumption given the design, structure, and cryptanalysis to date of BLAKE2s.

Non-normative note: BLAKE2s has a variable output digest length feature, but it does not support arbitrary bit lengths, otherwise it would have been used rather than external truncation. However, the protocol-specific personalization string together with truncation achieve essentially the same effect as using that feature.
5.4.1.6 DiversifyHash Hash Function

DiversifyHash is used to derive a diversified base from a diversifier in §4.2.2 ‘Sapling Key Components’ on p. 28.

Let GroupHash\(_{(\cdot)^*}\) and \(U\) be as defined in §5.4.8.5 ‘Group Hash into Jubjub’ on p. 71.

Define

\[
\text{DiversifyHash}(d) := \text{GroupHash}\(_{U}(\text{"Zcash\_gd", LEB2OSP}_{t_a}(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 diversifiers, it is not possible to tell which authority the third address was derived from.

Non-normative notes:

- Suppose that GroupHash\(_{(\cdot)^*}\) (restricted to inputs for which it does not return \(\perp\)) is modelled as a random oracle from diversifiers to points of order \(r_2\) on the Jubjub curve. In this model, Unlinkability of 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 \(O_2\). (ElGamal was originally defined for \(\mathbb{F}_p^\times\) 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 GroupHash\(_{(\cdot)^*}\), then generating a new diversified payment address from a given address \(pk\), gives the same distribution of \((g_d', pk_d')\) pairs as the distribution of ElGamal ciphertexts obtained by encrypting \(O_2\) 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 DiversifyHash. (IK-CCA corresponds to the potentially stronger requirement that 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 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 \((g_d, pk_d)\) 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. 28), 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

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 group. 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. 65], and of the Pedersen hash for the Sapling *incremental Merkle tree* ([5.4.1.3 ‘MerkleCRH\(^{\text{Sapling}}\) Hash Function’ on p. 54]).

Let \(J, J^{(r)}, D, M, q, r, a, b, c\) be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

Let \(\text{Extract}_{J^{(r)}} : J^{(r)} \rightarrow \mathbb{B}^{\ell_{\text{MarkleSapling}}}\) be as defined in §5.4.8.4 ‘Hash Extractor for Jubjub’ on p. 70.

Let \(\text{FindGroupHash}_{J^{(r)}}\) be as defined in §5.4.8.5 ‘Group Hash into Jubjub’ on p. 71.

Let \(\text{Uncommitted}_{\text{Sapling}}\) be as defined in §5.3 ‘Constants’ on p. 51.

Let \(c\) be the largest integer such that \(4 \cdot \frac{q + c}{15} \leq \frac{r_2 - 1}{2}\), i.e. \(c := 63\).

Define \(I : \mathbb{B}^{[8]} \times \mathbb{N} \rightarrow J^{(r)}\) by:

\[
I^D_i := \text{FindGroupHash}^{J^{(r)}} (D, \underbrace{32\text{-bit } i - 1}_{0}) .
\]

Define \(\text{PedersenHashToPoint}(D : \mathbb{B}^{[8]}, M : \mathbb{B}^{[N^\top]}) \rightarrow J^{(r)}\) as follows:

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

Let \(n = \text{ceiling}(\frac{\text{length}(M')}{3 \cdot c})\).

Split \(M'\) into \(n\) "segments" \(M_1 \ldots n\) so that \(M' = \text{concat}_8(M_1 \ldots n)\), and each of \(M_1 \ldots n-1\) is of length \(3 \cdot c\) bits. (\(M_n\) may be shorter.)

Return \(\sum_{i=1}^{n} \langle M_i \rangle I^D_i : J^{(r)}\).

where \((\cdot) : \mathbb{B}^{[3 \cdot (1 \ldots c)]} \rightarrow \{-\frac{r_2 - 1}{2} \ldots \frac{r_2 - 1}{2}\} \setminus \{0\}\) is defined as:

- Let \(k_i = \text{length}(M_i)/3\).
- Split \(M_i\) into 3-bit "chunks" \(m_{1 \ldots k_i}\) so that \(M_i = \text{concat}_3(m_{1 \ldots k_i})\).
- Write each \(m_j\) as \([s_0, s_1, s_2]\), and let \(\text{enc}(m_j) = (1 - 2 \cdot s_2) \cdot (1 + s_0 + 2 \cdot s_1) \in \mathbb{Z}\).
- Let \(\langle M_i \rangle = \sum_{j=1}^{k_i} \text{enc}(m_j) \cdot 2^{4(j-1)}\).

Finally, define \(\text{PedersenHash} : \mathbb{B}^{[8]} \times \mathbb{B}^{[N^\top]} \rightarrow \mathbb{B}^{\ell_{\text{MarkleSapling}}}\) by:

\[
\text{PedersenHash}(D, M) := \text{Extract}_{J^{(r)}} (\text{PedersenHashToPoint}(D, M)) .
\]

See §A.3.3.9 ‘Pedersen hash’ on p.138 for rationale and efficient circuit implementation of these functions.
Security requirement: PedersenHash and PedersenHashToPoint are required to be collision-resistant between inputs of fixed length, for a given personalization input $D$. No other security properties commonly associated with hash functions are needed.

Non-normative note: These hash functions are not collision-resistant for variable-length inputs.

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

Proof. We first check that the range of $\sum_{j=1}^{k_i} \text{enc}(m_j) \cdot 2^{4(j-1)}$ is a subset of the allowable range $\{-\frac{r_2-1}{2} \ldots \frac{r_2-1}{2}\} \setminus \{0\}$.

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

When $c = 63$, we have

$$4 \cdot \frac{2^{4c} - 1}{15} = 0x73EDA753299D7D48339D80809A1D8053341049E6640841684B7F6B7B965B$$

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

$\text{enc}$ is injective. In order to prove that $\langle \cdot \rangle$ is injective, consider $\langle \cdot \rangle^\Delta : \mathbb{B}^{[3 \ldots c]} \rightarrow \{0 \ldots 2 \cdot \Delta\}$ such that $\langle M_i \rangle^\Delta = \langle M_i \rangle + \Delta$.

With $k_i$ and $m_j$ defined as above, we have $\langle M_i \rangle^\Delta = \sum_{j=1}^{k_i} \text{enc}'(m_j) \cdot 2^{4(j-1)}$ where $\text{enc}'(m_j) = \text{enc}(m_j) + 4$ is in $\{0 \ldots 8\}$ and $\text{enc}'$ is injective. Express this sum in hexadecimal; then each $m_j$ affects only one hex digit, and it is easy to see that $\langle \cdot \rangle^\Delta$ 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 PedersenHashToPoint is collision-resistant under the same assumptions and security bounds. Because Extract_{j(r)} is injective, it follows that PedersenHash is equally collision-resistant.

Theorem 5.4.2. UncommittedSapling is not in the range of PedersenHash.

Proof. UncommittedSapling is defined as I2LEBSP_{MarkSapling}(1). By injectivity of I2LEBSP_{MarkSapling} and definitions of PedersenHash and Extract_{j(r)}, I2LEBSP_{MarkSapling}(1) can be in the range of PedersenHash only if there exist $D : \mathbb{B}^{[8]}$ and $M : \mathbb{B}^{[N]}$ such that $\mathcal{U}$(PedersenHashToPoint($D$, $M$)) = 1. The latter can only be the affine-ctEdwards $u$-coordinate of a point in $\mathbb{J}$. We show that there are no points in $\mathbb{J}$ with affine-ctEdwards $u$-coordinate 1. Suppose for a contradiction that $(u, v) \in \mathbb{J}$ for $u = 1$ and some $v : \mathbb{F}_p$. By writing the curve equation as $v^2 = (1 - a_j \cdot u^2)/(1 - d_j \cdot u^2)$, and noting that $1 - d_j \cdot u^2 \neq 0$ because $d_j$ is nonsquare, we have $v^2 = (1 - a_j)/(1 - d_j)$. The right-hand-side is a nonsquare in $\mathbb{F}_p$ (for the Jubjub curve parameters), so there are no solutions for $v$ (contradiction).

5.4.1.8 Mixing Pedersen Hash Function

A mixing Pedersen hash is used to compute $\rho$ from $cm$ and $pos$ in §4.14 ‘Note Commitments and Nullifiers’ on p. 41. It takes as input a Pedersen commitment $P$, and hashes it with another input $x$.

Define $\mathcal{J} := \text{FindGroupHash}^{\text{ctSapling}}("Zcash_J_-", \ldots)$.

We define MixingPedersenHash : $\mathbb{J} \times \{0 \ldots r_2 - 1\} \rightarrow \mathbb{J}$ by:

$$\text{MixingPedersenHash}(P, x) := P + [x] \mathcal{J}.$$
Security requirement: The function
\[(r, M, x) : \{0..r - 1\} \times \mathbb{B}^{[N]} \times \{0..r - 1\} \mapsto \text{MixingPedersenHash}(\text{WindowedPedersenCommit}_r(M), x) : \mathbb{J}\]
must be collision-resistant on \((r, M, x)\).

See §A.3.3.10 ‘Mixing Pedersen hash’ on p.140 for efficient circuit implementation of this function.

5.4.1.9 Equihash Generator

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.88.

Let \(\text{powtag} := \) 64-bit “ZcashPoW” 32-bit \(n\) 32-bit \(k\).

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

Let \(\text{EquihashGen}_{n,k}(S, i) := T_{h+1..h+n}\), where
\[
\begin{align*}
  m &:= \text{floor} \left( \frac{512}{n} \right); \\
  h &:= (i - 1 \mod m) \cdot n; \\
  T &:= \text{BLAKE2b} - (n \cdot m)(\text{powtag}, S || \text{powcount} (\text{floor} (\frac{i-1}{m}))).
\end{align*}
\]

Indices of bits in \(T\) are 1-based.

\(\text{BLAKE2b-}\ell(p, x)\) is defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p.53.

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

Note: When \(\text{EquihashGen}\) is evaluated for sequential indices, as in the Equihash solving process (§7.6.1 ‘Equihash’ on p.88), the number of calls to \(\text{BLAKE2b}\) can be reduced by a factor of \(\text{floor} \left( \frac{512}{n} \right)\) in the best case (which is a factor of 2 for \(n = 200\)).

5.4.2 Pseudo Random Functions

Let \(\text{SHA256Compress}\) be as defined in §5.4.1.1 ‘SHA-256, SHA-256d, and SHA256Compress Hash Functions’ on p.52.

The Pseudo Random Functions \(\text{PRF}^{\text{addr}}, \text{PRF}^{\text{nf}}, \text{PRF}^{\text{pk}},\) and \(\text{PRF}^{\rho}\), described in §4.1.2 ‘Pseudo Random Functions’ on p.19, are all instantiated using \(\text{SHA256Compress}\):

\[
\begin{align*}
\text{PRF}^{\text{addr}}(t) &:= \text{SHA256Compress}(1 1 0 0 252\text{-bit } x 8\text{-bit } t [0]^{248}) \\
\text{PRF}^{\text{nf}}(\rho) &:= \text{SHA256Compress}(1 1 1 0 252\text{-bit } a_{sk} 256\text{-bit } \rho) \\
\text{PRF}^{\text{pk}}(i, h_{Sig}) &:= \text{SHA256Compress}(0 1 0 0 252\text{-bit } a_{sk} 256\text{-bit } h_{Sig}) \\
\text{PRF}^{\rho}(i, h_{Sig}) &:= \text{SHA256Compress}(0 1 1 0 252\text{-bit } \varphi 256\text{-bit } h_{Sig})
\end{align*}
\]
Security requirements:

- SHA256Compress must be collision-resistant.
- SHA256Compress must be a PRF when keyed by the bits corresponding to \( x, a_k \) or \( \varphi \) in the above diagrams, with input in the remaining bits.

Note: The first four bits – i.e. the most significant four bits of the first byte – are used to separate distinct uses of SHA256Compress, ensuring that the functions are independent. As well as the inputs shown here, bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function; see §5.4.7.1 ‘Sprout Note Commitments’ on p. 65.

(The specific bit patterns chosen here were motivated by the possibility of future extensions that might have increased \( N^{old} \) and/or \( N^{new} \) to 3, or added an additional bit to \( a_k \) 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.)

PRF\textsuperscript{expand} is used in §4.2.2 ‘Sapling Key Components’ on p. 28 to derive the Spend authorizing key \( a_k \) and the proof authorizing key \( nsk \).

It is instantiated using the BLAKE2b hash function defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53:

\[
\text{PRF}_{sk}^\text{expand}(t) := \text{BLAKE2b}-512(“Zcash\_ExpandSeed”, \text{LEBS2OSP}\_256(sk) || t)
\]

Security requirement: \( \text{BLAKE2b}-512(“Zcash\_ExpandSeed”, \text{LEBS2OSP}\_256(sk) || t) \) must be a PRF for output range \( \text{Sym} \cdot \text{K} \) when keyed by the bits corresponding to \( sk \), with input in the bits corresponding to \( t \).

PRF\textsuperscript{ock} is used in §4.17.1 ‘Encryption (Sapling)’ on p. 47 to derive the outgoing cipher key \( ock \) used to encrypt an Output ciphertext.

It is instantiated using the BLAKE2b hash function defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53:

\[
\text{PRF}_{ock}^\text{ock}(cv, cmu, ephemeralKey) := \text{BLAKE2b}-256(“Zcash\_Derive\_ock”, \text{ockInput})
\]

where \( \text{ockInput} = \text{LEBS2OSP}\_256(ovk) |\rangle \text{32-byte cv} \text{ LEBS2OSP}\_256(cm\mu) |\rangle \text{32-byte ephemeralKey} \).

Security requirement: \( \text{BLAKE2b}-512(“Zcash\_Derive\_ock”, \text{ockInput}) \) must be a PRF for output range \( \text{Sym} \cdot \text{K} \) when keyed by the bits corresponding to \( ovk \), with input in the bits corresponding to \( cv, cmu, \) and \( ephemeralKey \).

PRF\textsuperscript{nfSapling} is used to derive the nullifier for a Sapling note. It is instantiated using the BLAKE2s hash function defined in §5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53:

\[
\text{PRF}_{nk\ast}^\text{nfSapling}(\rho\ast) := \text{BLAKE2s}-256(“Zcash\_nf”, \text{LEBS2OSP}\_256(nk\ast) |\rangle \text{LEBS2OSP}\_256(\rho\ast) )
\]

Security requirement: \( \text{BLAKE2s}-256(“Zcash\_nf”, \text{LEBS2OSP}\_256(nk\ast) |\rangle \text{LEBS2OSP}\_256(\rho\ast) ) \) must be a collision-resistant PRF for output range \( \mathbb{B}^{[32]} \) when keyed by the bits corresponding to \( nk\ast \), with input in the bits corresponding to \( \rho\ast \). Note that \( nk\ast : \mathbb{B}^{[32]} \) is a representation of a point in the \( r_J \)-order subgroup of the Jubjub curve, and therefore is not uniformly distributed on \( \mathbb{B}^{[32]} \). \( \mathbb{B}^{[32]} \) is defined in §5.4.8.3 ‘Jubjub’ on p. 69.
5.4.3 Symmetric Encryption

Let \( \text{Sym}.K := \mathbb{B}^{256} \), \( \text{Sym}.P := \mathbb{B}^{256} \), and \( \text{Sym}.C := \mathbb{B}^{256} \).

Let the authenticated one-time symmetric encryption scheme \( \text{Sym}.\text{Encrypt}_K(P) \) be authenticated encryption using AEAD_CHACHA20_POLY1305 [RFC-7539] encryption of plaintext \( P \in \text{Sym}.P \), with empty "associated data", all-zero nonce \( [0]^{96} \), and 256-bit key \( K \in \text{Sym}.K \).

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

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

5.4.4 Key Agreement And Derivation

5.4.4.1 Sprout Key Agreement

\( \text{KA}^{\text{Sprout}} \) is a key agreement scheme as specified in § 4.1.4 ‘Key Agreement’ on p. 20.

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

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

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

Let \( \text{KA}^{\text{Sprout}}.\text{Base} := 9 \) be the public byte sequence representing the Curve25519 base point.

Let clamp_{Curve25519}(\( x \)) take a 32-byte sequence \( x \) as input and return a byte sequence representing a Curve25519 private key, with bits “clamped” as described in [Bernstein2006, section 3]: “clear bits 0, 1, 2 of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte.” Here the bits of a byte are numbered such that bit \( b \) has numeric weight \( 2^b \).

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

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

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

5.4.4.2 Sprout Key Derivation

\( \text{KDF}^{\text{Sprout}} \) is a Key Derivation Function as specified in § 4.1.5 ‘Key Derivation’ on p. 20.

It is instantiated using BLAKE2b-256 as follows:

\[
\text{KDF}^{\text{Sprout}}(i, h, \text{sharedSecret}, d, \text{epk}, \text{pk}^{\text{new}}) := \text{BLAKE2b-256}(\text{kdftag}, \text{kdfinput})
\]

where:

\[
\text{kdftag} := \begin{bmatrix} 64 \text{-bit “ZcashKDF”} & 8 \text{-bit } i - 1 & [0]^{36} \end{bmatrix}
\]

\[
\text{kdfinput} := \begin{bmatrix} 256 \text{-bit } h & 256 \text{-bit sharedSecret} & 256 \text{-bit epk} & 256 \text{-bit } \text{pk}^{\text{new}} \end{bmatrix}
\]

BLAKE2b-256(\( p, x \)) is defined in § 5.4.1.2 ‘BLAKE2 Hash Functions’ on p. 53.
5.4.4.3 Sapling Key Agreement

KA_{Sapling} is a key agreement scheme as specified in §4.1.4 ‘Key Agreement’ on p. 20. It is instantiated as Diffie–Hellman with cofactor multiplication on Jubjub as follows:

Let $j, j^r, j^{(r)}$, and the cofactor $h_j$ be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

Define $KA_{Sapling}.Public := \mathcal{J}$.

Define $KA_{Sapling}.PublicPrimeOrder := j^{(r)}$.

Define $KA_{Sapling}.SharedSecret := j^r$.

Define $KA_{Sapling}.Private := \mathbb{F}_j$.

Define $KA_{Sapling}.DerivePublic(sk, B) := [sk] B$.

Define $KA_{Sapling}.Agree(sk, P) := [h_j \cdot sk] P$.

5.4.4.4 Sapling Key Derivation

KDF_{Sapling} is a Key Derivation Function as specified in §4.1.5 ‘Key Derivation’ on p. 20. It is instantiated using BLAKE2b-256 as follows:

$$KDF_{Sapling}(\text{sharedSecret}, \text{epk}) := \text{BLAKE2b-256("Zcash_SaplingKDF"}, \text{kdfinput})$$

where:

$$\text{kdfinput} := \text{LEBS2OSP}_{256}(\text{repr}_3(\text{sharedSecret})) | \text{LEBS2OSP}_{256}(\text{repr}_3(\text{epk}))$$

5.4.5 JoinSplit Signature

JoinSplitSig is a signature scheme as specified in §4.1.6 ‘Signature’ on p. 21.

Let ExcludedPointEncodings : $\mathcal{P}(\mathbb{G}_2^{[32]}) = \{
\}

Let $\ell = 2^{252} + 2774231777737235355851937790883648493$ (the order of the Ed25519 curve's prime order subgroup). JoinSplitSig is instantiated as the Ed25519 signature scheme [BLSY2012], with the additional requirements that for a signature $(R, S)$:

- $R$ MUST NOT be in ExcludedPointEncodings;
- $S$ MUST represent an integer less than $\ell$. 

61
If these requirements are not met then the signature is considered invalid. Note that it is not required that the integer encoding the \( y \)-coordinate\(^5\) of the point represented by \( R \) is less than \( 2^{255} - 19 \).

\( \text{Ed25519} \) is defined as using SHA-512 internally.

A valid \( \text{JoinSplitSig} \) public key is defined as a sequence of 32 bytes, not in ExcludedPointEncodings, that encodes a \( \text{Ed25519} \) public key as specified in [BDLSY2012]. Again, it is not required that the integer encoding the \( y \)-coordinate\(^5\) of the point represented by the public key is less than \( 2^{255} - 19 \).

The encoding of a signature is:

\[
\begin{array}{c|c}
256 \text{-bit } R & 256 \text{-bit } S \\
\end{array}
\]

where \( R \) and \( S \) are as defined in [BDLSY2012].

Non-normative note: The exclusion of ExcludedPointEncodings from \( R \) and public key encodings is due to a quirk of version 1.0.15 of the libsodium library [libsodium] which was initially used to implement \( \text{JoinSplitSig} \) signature validation in \zcash. (The \text{ED25519\_COMPAT} compile-time option was not set.) The intent was to exclude points of order less than \( \ell \); however, not all such points were covered. It is possible, with due attention to detail, to reproduce this quirk without using libsodium 1.0.15.

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. 22. It also supports a Secret Key to Public Key Monomorphism as described in §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p. 23. It is based on a scheme from [FKMSS2016, section 3], with some ideas from EdDSA [BILSY2015].

RedJubjub is a specialization of RedDSA to the Jubjub curve (§5.4.8.3 ‘Jubjub’ on p. 69), using the BLAKE2b-512 hash function.

The spend authorization signature scheme defined in §5.4.6.1 ‘Spend Authorization Signature’ on p. 64 is instantiated by RedJubjub. The binding signature scheme \( \text{BindingSig} \) defined in §5.4.6.2 ‘Binding Signature’ on p. 64 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 \( G \), which also defines a subgroup \( G^{(r)} \) of order \( r_G \), a cofactor \( h_G \), a group operation \( + \), an additive identity \( O_G \), a bit-length \( \ell_G \), a representation function \( \text{repr}_G \), and an abstraction function \( \text{abst}_G \), as specified in §4.1.8 ‘Represented Group’ on p. 25;
- \( \mathcal{P}_G \), a generator of \( G^{(r)} \);
- a bit-length \( \ell_H : \mathbb{N} \) such that \( 2^{64} - 128 \geq r_G \) and \( \ell_H \mod 8 = 0 \);
- a cryptographic hash function \( H : \mathbb{B}^{[N]} \rightarrow \mathbb{B}^{[(\ell_H/8)]} \).

Its associated types are defined as follows:

- \( \text{RedDSA.Message} := \mathbb{B}^{[N]} \)
- \( \text{RedDSA.Signature} := \mathbb{B}^{[\text{ceiling}(\ell_G/8) + \text{ceiling}(\text{bitlength}(r_G)/8)]} \)
- \( \text{RedDSA.Public} := G \)
- \( \text{RedDSA.Private} := F_{r_G} \)
- \( \text{RedDSA.Random} := F_{r_G} \)

\(^5\) Here we use the \((x, y)\) naming of coordinates in [BDLSY2012], which is different from the \((u, v)\) naming used for coordinates of \text{ctEdwards} curves in §4.2 ‘Elliptic curve background’ on p. 128.
Define $H^R : \mathbb{B}^{[N]} \to \mathbb{F}_{r_G}$ by:

$$H^R(B) = \text{LEOS2IP}_{\ell_G}(H(B)) \pmod{r_G}$$

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

Return $sk \xleftarrow{} \mathbb{F}_{r_G}$.

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

$\text{RedDSA.DerivePublic}(sk) := [sk] \mathcal{P}_G$.

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

Choose a byte sequence $T$ uniformly at random on $\mathbb{B}^{(\ell_G + 128)/8}$.

Return $H^R(T)$.

Define $O_{\text{RedDSA.Random}} := 0 \pmod{r_G}$.

Define $\text{RedDSA.RandomizePrivate} : \text{RedDSA.Random} \times \text{RedDSA.Private} \to \text{RedDSA.Private}$ by:

$\text{RedDSA.RandomizePrivate}(\alpha, sk) := sk + \alpha \pmod{r_G}$.

Define $\text{RedDSA.RandomizePublic} : \text{RedDSA.Random} \times \text{RedDSA.Public} \to \text{RedDSA.Public}$ as:

$\text{RedDSA.RandomizePublic}(\alpha, vk) := vk + [\alpha] \mathcal{P}_G$.

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

Choose a byte sequence $T$ uniformly at random on $\mathbb{B}^{(\ell_G + 128)/8}$.

Let $vk = \text{LEBS2OSP}_{\ell_G}(\text{repr}_G(\text{RedDSA.DerivePublic}(sk)))$.

Let $r = H^R(T \mid vk \mid M)$.

Let $R = [r] \mathcal{P}_G$.

Let $R = \text{LEBS2OSP}_{\ell_G}(\text{repr}_G(R))$.

Let $S = (r + H^R(R \mid vk \mid M) \cdot sk) \pmod{r_G}$.

Let $S = \text{LEBS2OSP}_{\text{bitlength}(r_G)}(\text{I2LEBSP}_{\text{bitlength}(r_G)}(S))$.

Return $R \mid S$.

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

Let $R$ be the first ceiling ($\ell_G/8$) bytes of $\sigma$, and let $S$ be the remaining ceiling (bitlength($r_G$)/8) bytes.

Let $R = \text{abst}_{\ell_G}(\text{LEOS2BSP}_{\ell_G}(R))$, and let $S = \text{LEOS2IP}_{\text{bitlength}(r_G)}(S)$.

Let $vk = \text{LEBS2OSP}_{\ell_G}(\text{repr}_G(vk))$.

Let $c = H^R(R \mid vk \mid M)$.

Return 1 if $R \neq \bot$ and $S < r_G$ and $[h_G] (−[S] \mathcal{P}_G + R + [c] vk) = O_G$, otherwise 0.

Notes:

- The verification algorithm does not check that $R$ is a point of order at least $r_G$. It does check that $R$ is the canonical representation (as output by repr$_G$) of a point on the curve. This is different to Ed25519 as specified in §5.4.5 ‘JoinSplit Signature’ on p. 61.
- Appendix §B.1 ‘RedDSA batch verification’ on p. 148 describes an optimization that MAY be used to speed up verification of batches of RedDSA signatures.
The randomization used in RedDSA.RandomizePrivate and RedDSA.RandomizePublic may interact with other uses of additive properties of keys for Schnorr-based signature schemes. In the Zcash protocol, such properties are used for binding signatures but not at the same time as key randomization. They are also used in [ZIP-32] when deriving child extended keys, but this does not result in any practical security weakness as long as the security recommendations of ZIP-32 are followed. If RedDSA is reused in other protocols making use of these additive properties, careful analysis of potential interactions is required.

The two abelian groups specified in §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p. 23 are instantiated for RedDSA as follows:

- \( O := 0 \pmod{r_G} \)
- \( sk_1 \oplus sk_2 := sk_1 + sk_2 \pmod{r_G} \)
- \( O_P := O_G \)
- \( vk_1 \oplus vk_2 := vk_1 + vk_2 \).

As required, RedDSA.DerivePublic is a group monomorphism, since it is injective and:

\[
\text{RedDSA.DerivePublic}(sk_1 \oplus sk_2) = [sk_1 + sk_2 \pmod{r_G}] P_G = [sk_1] P_G + [sk_2] P_G \quad \text{(since P_G has order r_G)}
\]

A RedDSA public key \( vk \) can be encoded as a bit sequence \( \text{repr}_G(vk) \) of length \( \ell_G \) bits (or as a corresponding byte sequence \( vk \) by then applying \( \text{LEBS2OSP}_{\ell_G} \)).

The scheme RedJubjub specializes RedDSA with:

- \( G := J \) as defined in §5.4.8.3 ‘Jubjub’ on p. 69;
- \( \ell_H := 512 \);
- \( H(x) := \text{BLAKE2b-512}("Zcash_RedJubjub",x) \) as defined in §5.4.12 ‘BLAKE2 Hash Functions’ on p. 53.

The generator \( P_G : G^{(r)} \) is left as an unspecified parameter, which is different between BindingSig and SpendAuthSig.

### 5.4.6.1 Spend Authorization Signature

Let RedJubjub be as defined in §5.4.6 ‘RedDSA and RedJubjub’ on p. 62.

Define \( G := \text{FindGroupHash}^{(r)}("Zcash_G",")" \).

The spend authorization signature scheme, SpendAuthSig, is instantiated as RedJubjub with key re-randomization, and with generator \( P_G = G \).

See §4.13 ‘Spend Authorization Signature’ on p. 40 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 defined in §4.1.6.1 ‘Signature with Re-Randomizable Keys’ on p. 22.

### 5.4.6.2 Binding Signature

Let RedJubjub be as defined in §5.4.6 ‘RedDSA and RedJubjub’ on p. 62.

Let \( R \) be the randomness base defined in §5.4.7.3 ‘Homomorphic Pedersen commitments’ on p. 66.

The binding signature scheme, BindingSig, is instantiated as RedJubjub without use of key re-randomization, and with generator \( P_G = R \).

See §4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38 for details on the use of this signature scheme.
Security requirement: BindingSig must be a SUF-CMA secure signature scheme with key monomorphism as defined in §4.1.6.2 ‘Signature with Private Key to Public Key Monomorphism’ on p. 23. A signature must prove knowledge of the discrete logarithm of the public key with respect to the base \( R \).

5.4.7 Commitment schemes

5.4.7.1 Sprout Note Commitments

The commitment scheme \( \text{NoteCommit}^{\text{Sprout}} \) specified in §4.17 ‘Commitment’ on p. 24 is instantiated using SHA-256 as follows:

\[
\text{NoteCommit}^{\text{Sprout}}_{\text{rcm}}(a_{\text{pk}}, v, \rho) := \text{SHA}-256\left( 101000 | 256\text{-bit } a_{\text{pk}} | 64\text{-bit } v | 256\text{-bit } \rho | 256\text{-bit } \text{rcm} \right)
\]

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

Security requirements:
- SHA256Compress must be collision-resistant.
- SHA256Compress must be a PRF when keyed by the bits corresponding to the position of \( \text{rcm} \) 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. 56 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. 69):

\[
\text{WindowedPedersenCommit}_r(s) := \text{PedersenHashToPoint}(\text{"Zcash\_PH"}, s) + [r] \text{FindGroupHash}^{(\text{\"Zcash\_PH\", "r"})}
\]

See §A.3.5 ‘Windowed Pedersen Commitment’ on p. 141 for rationale and efficient circuit implementation of this function.

The commitment scheme \( \text{NoteCommit}^{\text{Sapling}} \) specified in §4.17 ‘Commitment’ on p. 24 is instantiated as follows using WindowedPedersenCommit:

\[
\text{NoteCommit}^{\text{Sapling}}_{\text{rcm}}(g^*d, pk^*d, v) := \text{WindowedPedersenCommit}_r(\text{rcm}) \left( [1]^6 \| \text{I2LEBS}_{64}(v) \| g^*d \| pk^*d \right)
\]

Security requirements:
- WindowedPedersenCommit, and hence \( \text{NoteCommit}^{\text{Sapling}} \), must be computationally binding and at least computationally hiding commitment schemes.

(They are in fact unconditionally hiding commitment schemes.)

Notes:
- MerkleCRH\(^{\text{Sapling}}\) is also defined in terms of PedersenHashToPoint (see §5.4.1.3 ‘Merkle Tree Hash Function’ on p. 53). The prefix \([1]^6\) distinguishes the use of WindowedPedersenCommit in \( \text{NoteCommit}^{\text{Sapling}} \) from the layer prefix used in MerkleCRH\(^{\text{Sapling}}\). That layer prefix is a 6-bit little-endian encoding of an integer in the range \( \{0..\text{MerkleDepth}^{\text{Sapling}} - 1\} \); because \( \text{MerkleDepth}^{\text{Sapling}} \) < 64, it cannot collide with \([1]^6\).
- The arguments to \( \text{NoteCommit}^{\text{Sapling}} \) are in a different order to their encodings in 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 ValueCommit.

For more details on the use of this property, see § 4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38 and § 3.6 ‘Spend Transfers, Output Transfers, and their Descriptions’ on p. 16.

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

\[
\text{HomomorphicPedersenCommit}_{rcv}(D, v) := [v \text{FindGroupHash}^{(r)}(D, "v") + [rcv] \text{FindGroupHash}^{(r)}(D, "r")}
\]

\[
\text{ValueCommit.GenTrapdoor}() \text{ generates the uniform distribution on } \mathbb{F}_{q_r}.
\]

See § A.3.6 ‘Homomorphic Pedersen Commitment’ on p. 141 for rationale and efficient circuit implementation of this function.

Define:

\[
\mathcal{V} := \text{FindGroupHash}^{(r)}("\text{Zcash_cv}", "v")
\]

\[
\mathcal{R} := \text{FindGroupHash}^{(r)}("\text{Zcash_cv}", "r")
\]

The commitment scheme ValueCommit specified in § 4.1.7 ‘Commitment’ on p. 24 is instantiated as follows using HomomorphicPedersenCommit:

\[
\text{ValueCommit}_{rcv}(v) := \text{HomomorphicPedersenCommit}_{rcv}("\text{Zcash_cv}", v).
\]

which is equivalent to:

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

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 defined in this section.

Let \( q_G := 2188824287183927522224640574525727508869631115729782366268903789465226208583 \).

Let \( r_G := 21888242871839275222246405745257275088548364400416034343698204186575808495617 \).

Let \( b_G := 3 \).

(\( q_G \) and \( r_G \) are prime.)

Let \( G_1^{(r)} \) be the group (of order \( r_G \)) of rational points on a Barreto–Naehrig (BN2005) curve \( E_{G_1} \) over \( \mathbb{F}_{q_G} \) with equation \( y^2 = x^3 + b_G \). This curve has embedding degree 12 with respect to \( r_G \).

Let \( G_2^{(r)} \) be the subgroup of order \( r_G \) in the sextic twist \( E_{G_2} \) of \( E_{G_1} \) over \( \mathbb{F}_{q_G} \) with equation \( y^2 = x^3 + b_G / \xi \). where \( \xi \in \mathbb{F}_{q_G}^* \).
We represent elements of $F_{q_r}$ as polynomials $a_1 \cdot t + a_0 : F_{q_r}[t]$, modulo the irreducible polynomial $t^2 + 1$; in this representation, $\xi$ is given by $t + 9$.

Let $G_T^{(r)}$ be the subgroup of $r_G$th roots of unity in $F_{q_r^{12}}^*$, with multiplicative identity $1_G$.

Let $\hat{c}_G$ be the optimal ate pairing (see [Vercauter2009] and [AKLGL2010, section 2]) of type $G_1^{(r)} \times G_2^{(r)} \rightarrow G_T^{(r)}$.

For $i \in \{1, 2\}$, let $O_{G_i}$ be the point at infinity (which is the additive identity) in $G_i^{(r)}$, and let $G_i^{(r)*} := G_i^{(r)} \setminus \{O_{G_i}\}$.

Let $P_{G_1} : G_1^{(r)*} := (1, 2)$.

Let $P_{G_2} : G_2^{(r)*} := (1155973203298638710799100402132285783925812861821192530917403151452391805634 \cdot t + 1085704699902305713594457076223282941837075635957851808699051999328565852781, 4082367875866343368133220340314535568316851327593401280105741076214120093531 \cdot t + 849565392312341317604973247489272438418190587263600148770280649306958101930).$

$P_{G_1}$ and $P_{G_2}$ are generators of $G_1^{(r)}$ and $G_2^{(r)}$ respectively.

Define I2BEBSP : $(\ell : N) \times \{0, 2^\ell - 1\} \rightarrow \mathbb{B}[\ell]$ as in § 5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 50.

For a point $P : G_1^{(r)*} = (x_P, y_P)$:

- The field elements $x_P$ and $y_P : F_q$ are represented as integers $x$ and $y : \{0..q-1\}$.
- Let $\bar{y} = y \mod 2$.
- $P$ is encoded as $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \ -1 \ \bar{y} \ 256 \text{-bit I2BEBSP}_{256}(x) \end{bmatrix}$.

For a point $P : G_2^{(r)*} = (x_P, y_P)$:

- Define $FE2IP : F_{q_2}[t]/(t^2 + 1) \rightarrow \{0..q_2^2 - 1\}$ 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 $\bar{y} = \begin{cases} 1, & \text{if } y > y' \\ 0, & \text{otherwise} \end{cases}$.
- $P$ is encoded as $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \ -1 \ \bar{y} & 512 \text{-bit I2BEBSP}_{512}(x) \end{bmatrix}$.

Non-normative notes:

- Only the $r_G$-order subgroups $G_2^{(r,2,T)}$ are used in the protocol, not their containing groups $G_2$. Points in $G_2^{(r)*}$ are always checked to be of order $r_G$ when decoding from external representation. (The group of rational points $G_1$ on $E_{G_1}/F_{q_1}$ is of order $r_G$ so no subgroup checks are needed in that case, and elements of $G_2^{(r)}$ are never represented externally.) The $(r)$ superscripts on $G_1^{(r)}$ are used for consistency with notation elsewhere in this specification.
- The points at infinity $O_{G_1,2}$ never occur in proofs and have no defined encodings in this protocol.
- A rational point $P \neq O_{G_2}$ on the curve $E_{G_2}$ can be verified to be of order $r_G$, and therefore in $G_2^{(r)*}$, by checking that $r_G \cdot P = O_{G_2}$.
- The use of big-endian order by I2BEBSP is different from the encoding of most other integers in this protocol. The encodings for $G_2^{(r)*}$ 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 $G_1^{(r)*}$, and the SORT compressed form (i.e. EC2OSP-XS) for points in $G_2^{(r)*}$.
The represented pairing BLS12-381 is defined in this section. Parameters are taken from [Bowe2017].

Let \( q_S := 400240955522166739441778982575904156568828199390678533205813612403165049083784642687629129015664037894272559787 \)

Let \( r_S := 52435875175126190479447740508185965837690552500527637822603658699938581184513 \).

Let \( u_S := -15132376222941642752 \).

Let \( b_S := 4 \).

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

Let \( S^{(r)} \) be the subgroup of order \( r_S \) in the sextic twist \( E_{S^{(r)}} \) of \( E_{S^{(r)}} \) over \( \mathbb{F}_{q_S} \) with equation \( y^2 = x^3 + 4(i + 1) \), where \( i \) is given by \( t \).

We represent elements of \( \mathbb{F}_{q_S} \) as polynomials \( a_1 \cdot t + a_0 : \mathbb{F}_{q_S} \Rightarrow \mathbb{F}_q \), modulo the irreducible polynomial \( t^2 + 1 \); in this representation, \( i \) is given by \( t \).

Let \( S^{(r)}_{T} \) be the subgroup of \( r_S \) th roots of unity in \( \mathbb{F}_{q_S}^* \), with multiplicative identity \( 1_{S^{(r)}} \).

Let \( \hat{c}_S \) be the optimal ate pairing of type \( S^{(r)} \times S^{(r)} \rightarrow S^{(r)} \).

For \( i \in \{ 1, \ldots, 2 \} \), let \( O_{S_{i}}^{(r)} \) be the point at infinity in \( S_{i}^{(r)} \), and let \( S_{i}^{(r)} \) be the point at infinity in \( S^{(r)} \), and let \( S_{i}^{(r)} := S_{i}^{(r)} \setminus \{ O_{S_{i}}^{(r)} \} \).

Let \( P_{S_{1}}^{(r)} := \langle S^{(r)}_{T} \cup S^{(r)}_{1} \rangle \) where

\[
\begin{align*}
(3685416753713338701678108831518307775976162079578234640989457837868806792378736318836054974676345821548104186454507, \\
13395065449444764730204713739419412212584393759384692042654373641651142395633350647272465535336534992391756441569).
\end{align*}
\]

Let \( P_{S_{2}}^{(r)} := \langle S^{(r)}_{T} \cup S^{(r)}_{2} \rangle \) where

\[
\begin{align*}
(305914434424213709671259814753871636898407325476647558593732062916353247689584324335096310434701783788576365758 \cdot t + \\
352701069587466618178139116101060144980929527977254201990864423979378573571502687373476004386515795261926303160, \\
92755366549232454720196576003780757740193453929790250279787939768770920675564980094928972795756557543344219582 \cdot t + \\
198515060228729193556805452117717163830086897821555730835978666663442763738237184238691042633398464149434037905).
\end{align*}
\]

\( P_{S_{1}} \) and \( P_{S_{2}} \) are generators of \( G_{S_{1}}^{(r)} \) and \( G_{S_{2}}^{(r)} \) respectively.

Define \( \text{i2BEBSP} : (\ell : \mathbb{N}) \times \{ 0 \ldots 2^\ell - 1 \} \rightarrow \mathbb{E}[\ell] \) as in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 50.
For a point \( P : S_1^{(r)*} = (x_P, y_P) \):

- The field elements \( x_P \) and \( y_P : \mathbb{F}_{q_3} \) are represented as integers \( x \) and \( y \): \( \{0..q_3-1\} \).

- Let \( \tilde{y} = \begin{cases} 1, & \text{if } y > q_3 - y \\ 0, & \text{otherwise.} \end{cases} \)

- \( P \) is encoded as \( \begin{bmatrix} 1 \mid 0 \mid 1\text{-bit } \tilde{y} \end{bmatrix} \) 381-bit I2BEBS381(x).

For a point \( P : S_2^{(r)*} = (x_P, y_P) \):

- Define \( \text{FE2IPP} : \mathbb{F}_{q_3}[t]/(t^2 + 1) \rightarrow \{0..q_3-1\}^{[2]} \) such that \( \text{FE2IPP}(a_{w,1} \cdot t + a_{w,0}) = [a_{w,1}, a_{w,0}] \).

- Let \( x = \text{FE2IPP}(x_P) \), \( y = \text{FE2IPP}(y_P) \), and \( y' = \text{FE2IPP}(-y_P) \).

- Let \( \tilde{y} = \begin{cases} 1, & \text{if } y > y' \text{ lexicographically} \\ 0, & \text{otherwise.} \end{cases} \)

- \( P \) is encoded as \( \begin{bmatrix} 1 \mid 0 \mid 1\text{-bit } \tilde{y} \end{bmatrix} \) 381-bit I2BEBS381(x) 384-bit I2BEBS384(x).

Non-normative notes:

- Only the \( r_3 \)-order subgroups \( S_{1,2,i} \) are used in the protocol, not their containing groups \( S_{1,2,i} \). Points in \( S_{1,2,i}^{(r)*} \) are always checked to be of order \( r_3 \) when decoding from external representation. (Elements of \( S_{1,2,i}^{(r)} \) never represented externally.) The \( (r) \) superscripts on \( S_{1,2,i} \) are used for consistency with notation elsewhere in this specification.

- The points at infinity \( \mathcal{O}_{S_{1,2,i}} \) never occur in proofs and have no defined encodings in this protocol.

- In contrast to the corresponding BN-254 curve, \( E_{S_1} \) over \( \mathbb{F}_{q_3} \) is not of prime order.

- A rational point \( P \neq \mathcal{O}_{S_i} \) on the curve \( E_{S_i} \) for \( i \in \{1, 2\} \) can be verified to be of order \( r_3 \), and therefore in \( S_{i}^{(r)*} \), by checking that \( r_3 \cdot P = \mathcal{O}_{S_i} \).

- The encodings for \( S_{1,2,i}^{(r)*} \) are specific to Zcash.

- Algorithms for decompressing points from the encodings of \( S_{1,2,i}^{(r)*} \) are defined analogously to those for \( G_1^{(r)*} \) in \$5.4.8.1 ‘BN-254’ on p. 66, taking into account that the SORT compressed form (not the LSB compressed form) is used for \( S_{1,2,i}^{(r)*} \).

When computing square roots in \( \mathbb{F}_{q_3} \) or \( \mathbb{F}_{q_2} \) in order to decompress a point encoding, the implementation MUST NOT assume that the square root exists, or that the encoding represents a point on the curve.

5.4.8.3 Jubjub

"You boil it in sawdust; you salt it in glue:
You condense it with locusts and tape:
Still keeping one principal object in view—
To preserve its symmetrical shape."

— Lewis Carroll, "The Hunting of the Snark" [Carroll1876]

Sapling uses an elliptic curve, Jubjub, designed to be efficiently implementable in zk-SNARK circuits. The represented group \( \mathbb{G} \) of points on this curve is defined in this section.

A complete twisted Edwards elliptic curve, as defined in [BL2017, section 4.3.4], is an elliptic curve \( E \) over a non-binary field \( \mathbb{F}_q \), parameterized by distinct \( a, d : \mathbb{F}_q \setminus \{0\} \) such that \( a \) is square and \( d \) is nonsquare, with equation \( E : a \cdot u^2 + v^2 = 1 + d \cdot u^2 \cdot v^2 \). We use the abbreviation “ctEdwards” to refer to complete twisted Edwards elliptic curves and coordinates.
Let $q_j := r_j$, as defined in §5.4.8.2 ‘BLS12-381’ on p. 68.
Let $r_j := 655448396890773809930967563523245729705921265872317281365359162392183254199$.
($q_j$ and $r_j$ are prime.)
Let $d_j := 8$.
Let $a_j := -1$.
Let $d_j := -10240/10241 \pmod{q_j}$.
Let $J$ be the group of points $(u, v)$ on a *ctEdwards curve* $E_J$ over $\mathbb{F}_{q_j}$ with equation $a_j \cdot u^2 + v^2 = 1 + d_j \cdot u^2 \cdot v^2$. The zero point with coordinates $(0, 1)$ is denoted $O_J$. $J$ has order $h_j \cdot r_j$.
Let $\ell_j := 256$.
Define $\text{I2LEBSP} : (\ell : \mathbb{N}) \times \{0 \ldots 2^\ell - 1\} \rightarrow \mathbb{B}^{[\ell]}$ as in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 50.
Define $\text{repr}_J : J \rightarrow \mathbb{B}^{[\ell]}$ such that $\text{repr}_J(u, v) = \text{I2LEBSP}_{256}(v + 2^{255} \cdot \bar{u})$, where $\bar{u} = u \mod 2$.
Let $\text{abst}_J : \mathbb{B}^{[\ell]} \rightarrow J \cup \{\bot\}$ be the left inverse of $\text{repr}_J$ such that if $S$ is not in the range of $\text{repr}_J$, then $\text{abst}_J(S) = \bot$.
Define $J^r$ as the order-$r_j$ subgroup of $J$. Note that this includes $O_J$. For the set of points of order $r_j$ (which excludes $O_J$), we write $J^{(r)}$.
Define $J^\bot := \{\text{repr}_J(P) : \mathbb{B}^{[\ell]} | P \in J^{(r)}\}$.

Non-normative notes:
- The *ctEdwards compressed encoding* used here is consistent with that used in EdDSA [BILSY2015] for public keys and the $R$ element of a signature.
- [BILSY2015, "Encoding and parsing curve points"] gives algorithms for decompressing points from the encoding of $J$.

When computing square roots in $\mathbb{F}_{q_j}$ 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 [BILSY2015, "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 $r_j$.

### 5.4.8.4 Hash Extractor for Jubjub

Let $\mathcal{U}((u, v)) = u$ and let $\mathcal{V}((u, v)) = v$.
Define $\text{Extract}_{\mathcal{J}^{(r)}} : J^{(r)} \rightarrow \mathbb{B}^{[\text{MerkleSapling}]}$ by

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

**Facts:** The point $(0, 1) = O_J$, and the point $(0, -1)$ has order 2 in $J$. $J^{(r)}$ is of odd-prime order.

**Lemma 5.4.3.** Let $P = (u, v) \in J^{(r)}$. Then $(u, -v) \notin J^{(r)}$.

**Proof.** If $P = O_J$ then $(u, -v) = (0, -1) \notin J^{(r)}$. 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 $[4] P = (0, 1) = O_J$: contradiction since then $P$ would not be of odd-prime order.) Therefore, $−v \neq v$. Now suppose $(u, −v) = Q$ is a point in $J^{(r)}$. 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 $J^{(r)}$ (contradiction since $J^{(r)}$ is of odd order [KvE2013]).
**Theorem 5.4.4.**  \( \mathcal{U} \) is injective on \( \mathcal{J}^{(r)} \).

**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 cEdwards 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 \( \mathcal{J}^{(r)} \). Therefore, \( \mathcal{U} \) is injective on points in \( \mathcal{J}^{(r)} \).  

Since I2LEBSP\(_{\text{MerkleSapling}} \) is injective, it follows that Extract\(_{\mathcal{J}^{(r)}} \) is injective on \( \mathcal{J}^{(r)} \).

### 5.4.8.5 Group Hash into Jubjub

Let GroupHash.\text{Input} := \mathbb{B}^{[8]} \times \mathbb{B}^{[N]}, and let GroupHash.\text{URS}Type := \mathbb{B}^{[64]}.

(The input element with type \( \mathbb{B}^{[8]} \) is intended to act as a “personalization” parameter to distinguish uses of the group hash for different purposes.)

Let URS be the MPC randomness beacon defined in §5.9 ‘Randomness Beacon’ on p. 79.

Let BLAKE2s-256 be as defined in §5.4.12 ‘BLAKE2 Hash Functions’ on p. 53.

Let LEOS2IP be as defined in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 50.

Let \( \mathcal{J}^{(r)}, \mathcal{J}^{(s)} \), and abst\(_{j} \) be as defined in §5.4.8.3 ‘Jubjub’ on p. 69.

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

The hash \( \text{GroupHash}_{\text{URS}}^{(r)}(D, M) : \mathcal{J}^{(r)} \) is calculated as follows:

- let \( H = \text{BLAKE2s-256}(D, \text{URS} || \, M) \)
- let \( P = \text{abst}_{j}(\text{LEOS2BSP}_{256}(H)) \)
  - if \( P = \bot \) then return \( \bot \)
  - let \( Q = [h_{j}] \, P \)
  - if \( Q = O_{j} \) then return \( \bot \), else return \( Q \).

**Notes:**

- The BLAKE2s-256 chaining variable after processing URS may be precomputed.
- The use of GroupHash\(_{\text{URS}}^{(r)} \) for DiversifyHash and to generate independent bases needs a random oracle (for inputs on which GroupHash\(_{\text{URS}}^{(r)} \) does not return \( \bot \); here we show that it is sufficient to employ a simpler random oracle instantiated by BLAKE2s-256 in the security analysis.

\( H : \mathbb{B}^{[64]} \mapsto \mathcal{J}^{(r)} \) : \( \mathcal{J} \) is injective, and both it and its inverse are efficiently computable.

\( P : \mathcal{J} \mapsto \mathcal{J}^{(s)} \) : \( h_{j} \)-to-1, and both it and its inverse relation are efficiently computable.

It follows that when \( (D : \mathbb{B}^{[8]}, \, M : \mathbb{B}^{[N]}) \mapsto \text{BLAKE2s-256}(D, \, \text{URS} || \, M) : \mathbb{B}^{[64]} \) is modelled as a random oracle, \( (D : \mathbb{B}^{[8]}, \, M : \mathbb{B}^{[N]}) \mapsto \mathcal{J}^{(r)} \) also acts as a random oracle.

Define first \( : (\mathbb{B}^{[8]} \mapsto T \cup \{ \bot \}) \mapsto T \cup \{ \bot \} \) so that first\((f) = f(i) \) where \( i \) is the least integer in \( \mathbb{B}^{[8]} \) such that \( f(i) \neq \bot \), or \( \bot \) if no such \( i \) exists.

Define \( \text{FindGroupHash}^{(r)}(D, M) := \text{first}(i : \mathbb{B}^{[8]} \mapsto \text{GroupHash}_{\text{URS}}^{(r)}(D, M || [i]) : \mathcal{J}^{(r)} \cup \{ \bot \}) \).

**Note:** For random input, \( \text{FindGroupHash}^{(r)} \) returns \( \bot \) with probability approximately \( 2^{-256} \). In the Zcash protocol, most uses of \( \text{FindGroupHash}^{(r)} \) are for constants and do not return \( \bot \); the only use that could potentially return \( \bot \) is in the computation of a default diversified payment address in §4.2.2 ‘Sapling Key Components’ on p. 28.
5.4.9 Zero-Knowledge Proving Systems

5.4.9.1 BCTV14

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

A BCTV14 proof consists of $(\pi_A \cdot G_1^{(r)*}, \pi_A^{-1} \cdot G_1^{(r)*}, \pi_B \cdot G_2^{(r)*}, \pi_B^{-1} \cdot G_1^{(r)*}, \pi_C \cdot G_1^{(r)*}, \pi_C^{-1} \cdot G_2^{(r)*}, \pi_K \cdot G_1^{(r)*}, \pi_H \cdot G_1^{(r)*})$.

It is computed as described in [BCTV2014a, Appendix B], using the pairing parameters specified in §5.4.8.1 ‘BN-254’ on p. 66.

Note: Many details of the proving system are beyond the scope of this protocol document. For example, the quadratic constraint program verifying the JoinSplit statement, or its translation to a Quadratic Arithmetic Program [BCTV2014a, section 2.3], are not specified in this document. In 2015, Bryan Parno found a bug in this translation, which is corrected by the libsnark implementation [WCBTV2015] [Parno2015] [BCTV2014a, Remark 2.5]. In practice it will be necessary to use the specific proving and verification keys that were generated for the Zcash production block chain, given in §5.7 ‘BCTV14 zk-SNARK Parameters’ on p. 78, together with a proving system implementation that is interoperable with the Zcash fork of libsnark, to ensure compatibility.

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

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

The vulnerability does not affect the Zero Knowledge property of the scheme (as described in any version of [BCTV2014a] or as implemented in any version of libsnark that has been used in Zcash), even under subversion of the parameter generation [BGG2017, Theorem 4.10].

Encoding of BCTV14 Proofs

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

| 264-bit $\pi_A$ | 264-bit $\pi_A^{-1}$ | 520-bit $\pi_B$ | 264-bit $\pi_B^{-1}$ | 264-bit $\pi_C$ | 264-bit $\pi_C^{-1}$ | 264-bit $\pi_K$ | 264-bit $\pi_H$ |

The resulting proof size is 296 bytes.

In addition to the steps to verify a proof given in [BCTV2014a, Appendix B], the verifier MUST check, for the encoding of each element, that:

- the lead byte is of the required form;
- the remaining bytes encode a big-endian representation of an integer in $\{0..q_5-1\}$ or (in the case of $\pi_B$) $\{0..q_5^2-1\}$;
- the encoding represents a point in $G_1^{(r)*}$ or (in the case of $\pi_B$) $G_2^{(r)*}$, including checking that it is of order $r_5$ in the latter case.

---

$^6$ Confusingly, the bug found by Bryan Parno was fixed in libsnark in 2015, but that fix was incompletely described in the May 2015 update [BCTV2014a-old, Theorem 2.4]. It is described completely in [BCTV2014a, Theorem 2.4] and in [Gabizon2019].
After Sapling activation, Zcash uses zk-SNARK with the Groth16 proving system described in [BGM2017], which is a modification of the system in [Groth2016]. An independent security proof of this system and its setup is given in [Maller2018]. These zk-SNARK are used in transaction version 4 and later (§ 7.1 ‘Encoding of Transactions’ on p. 81) for proofs both in Sprout JoinSplit descriptions, and in Sapling Spend descriptions and Output descriptions. They are generated by the bellman library.

A Groth16 proof consists of \((π_A : S_1^{(r)^*}, \pi_B : S_2^{(r)^*}, π_C : S_1^{(r)^*})\). It is computed as described in [Groth2016, section 3.2], using the pairing parameters specified in § 5.4.8.2 ‘BLS12-381’ on p. 68. 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. 128. 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 Zcash production block chain (see § 5.8 ‘Groth16 zk-SNARK Parameters’ on p. 79), and a proving system implementation that is interoperable with the bellman library used by Zcash, to ensure compatibility.

### Encoding of Groth16 Proofs

A Groth16 proof is encoded by concatenating the encodings of its elements; for the BLS12-381 pairing this is:

- 384-bit \(π_A\)
- 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 \(\{0..q_S-1\}\) or (in the case of \(π_B\)) two integers in that range;
- the encoding represents a point in \(S_1^{(r)^*}\) or (in the case of \(π_B\)) \(S_2^{(r)^*}\), including checking that it is of order \(r_S\) 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. 14, 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 \(pk_{enc,1..N}^{new}\). Each Sprout note plaintext (denoted \(np\)) consists of:

\[
(v : \{0..2^{e_{val}}-1\}, \rho : \mathbb{B}^{[\mathbb{F}_{Sprout}]}, rcm : \text{NoteCommit}^{Sprout}.\text{Output}, \text{memo} : \mathbb{B}^{[512]})
\]

[Sapling onward] The note plaintext in each Output description is encrypted to the diversified transmission key \(pk_d\). Each Sapling note plaintext (denoted \(np\)) consists of:

\[
(d : \mathbb{B}^{[d]}, v : \{0..2^{e_{val}}-1\}, rcm : \text{NoteCommit}^{Sapling}.\text{Output}, \text{memo} : \mathbb{B}^{[512]})
\]

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 as one of:

- a UTF-8 human-readable string [Unicode], padded by appending zero bytes; or
- the byte 0xF6 followed by 511 0x00 bytes, indicating "no memo"; or
- any other sequence of 512 bytes starting with a byte value 0xF5 or greater (which is therefore not a valid UTF-8 string), as specified in [ZIP-302].

When the first byte value is less than 0xF5, 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 other cases, the contents of the memo field SHOULD NOT be displayed unless otherwise specified by [ZIP-302].

Other fields are as defined in § 3.2 ‘Notes’ on p. 13.

The encoding of a Sprout note plaintext consists of:

| 8-bit 0x00 | 64-bit \(v\) | 256-bit \(\rho\) | 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 \(\rho\).
- 32 bytes specifying \(rcm\).
- 512 bytes specifying memo.

The encoding of a Sapling note plaintext consists of:

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

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

### 5.6 Encodings of Addresses and Keys

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

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

For Sapling-specific key and address formats, Bech32 [ZIP-173] is used instead of Base58Check.
Non-normative note: ZIP 173 is similar to Bitcoin's BIP 173, except for dropping the limit of 90 characters on an encoded Bech32 string (which does not hold for Sapling viewing keys, for example), and requirements specific to Bitcoin's Segwit addresses.

SHA256Compress outputs are always represented as sequences of 32 bytes.
The language consisting of the following encoding possibilities is prefix-free.

5.6.1 Transparent Addresses

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

The raw encoding of a P2SH address consists of:

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

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

The raw encoding of a P2PKH address consists of:

| 8-bit 0x1C | 8-bit 0xB8 | 160-bit public key hash |

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

Notes:

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

5.6.2 Transparent Private Keys

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

5.6.3 Sprout Payment Addresses

A Sprout shielded payment address consists of $a_{pk} \leftarrow B^{[\mathsf{Sprout}]^{-1}}$ and $p_{k_{enc}} : KA^{\mathsf{Sprout}.\text{Public}}$.

$a_{pk}$ is a SHA256Compress output. $p_{k_{enc}}$ is a $KA^{\mathsf{Sprout}.\text{Public}}$ key (see §5.4.4.1 ‘Sprout Key Agreement’ on p. 60), for use with the encryption scheme defined in §4.16 ‘In-band secret distribution (Sprout)’ on p. 45. These components are derived from a spending key as described in §4.2.1 ‘Sprout Key Components’ on p. 28.
The raw encoding of a Sprout shielded payment address consists of:

<table>
<thead>
<tr>
<th>8-bit 0x16</th>
<th>8-bit 0x9A</th>
<th>256-bit $a_{pk}$</th>
<th>256-bit $p_{k_{enc}}$</th>
</tr>
</thead>
</table>

- 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 $a_{pk}$.
- 32 bytes specifying $p_{k_{enc}}$, using the normal encoding of a Curve25519 public key [Bernstein2006].

Note: For addresses on the production network, the lead bytes and encoded length cause the first two characters of the Base58Check encoding to be fixed as “zc”. For the test network, the first two characters are fixed as “zt”.

5.6.4 Sapling Payment Addresses

A Sapling shielded payment address consists of $d : \mathbb{B}^{f_{sl}}$ and $p_{kd} : \text{KA}_{\text{Sapling}.\text{PublicPrimeOrder}}$.

$p_{kd}$ is an encoding of a KA$_{\text{Sapling}}$ public key of type KA$_{\text{Sapling}.\text{PublicPrimeOrder}}$ (see § 5.4.4.3 ‘Sapling Key Agreement’ on p. 61), for use with the encryption scheme defined in § 4.17 ‘In-band secret distribution (Sapling)’ on p. 46. $d$ is a sequence of 11 bytes. These components are derived as described in § 4.2.2 ‘Sapling Key Components’ on p. 28.

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

<table>
<thead>
<tr>
<th>LEBS2OSP$_{88}(d)$</th>
<th>LEBS2OSP$<em>{256}(\text{repr}</em>{J}(p_{kd}))$</th>
</tr>
</thead>
</table>

- 11 bytes specifying $d$.
- 32 bytes specifying the ctEdwards compressed encoding of $p_{kd}$ (see § 5.4.8.3 ‘Jubjub’ on p. 69).

When decoding the representation of $p_{kd}$, the address is not valid if abst$\_j$ returns $\perp$ or if the resulting $p_{kd}$ is not of prime order.

For addresses on the production network, the Human-Readable Part (as defined in [ZIP-173]) is “zs”. For addresses on the test network, the Human-Readable Part is “zttestsapling”.

5.6.5 Sprout Incoming Viewing Keys

A Sprout incoming viewing key consists of $a_{pk} : \mathbb{B}^{f_{sp}}$ and $s_{k_{enc}} : \text{KA}_{\text{Sprout}.\text{Private}}$.

$a_{pk}$ is a SHA256Compress output. $s_{k_{enc}}$ is a KA$_{\text{Sprout}.\text{Private}}$ key (see § 5.4.4.1 ‘Sprout Key Agreement’ on p. 60), for use with the encryption scheme defined in § 4.16 ‘In-band secret distribution (Sprout)’ on p. 45. These components are derived from a spending key as described in § 4.2.1 ‘Sprout Key Components’ on p. 28.

The raw encoding of a Sprout incoming viewing key consists of, in order:

<table>
<thead>
<tr>
<th>8-bit 0xA8</th>
<th>8-bit 0xAB</th>
<th>8-bit 0xD3</th>
<th>256-bit $a_{pk}$</th>
<th>256-bit $s_{k_{enc}}$</th>
</tr>
</thead>
</table>

- Three bytes [0xA8, 0xAB, 0xD3], indicating this version of the raw encoding of a Zcash incoming viewing key on the production network. (Addresses on the test network use [0xA8, 0xAC, 0x0C] instead.)
- 32 bytes specifying $a_{pk}$.
- 32 bytes specifying $s_{k_{enc}}$, using the normal encoding of a Curve25519 private key [Bernstein2006].
sk_enc MUST be “clamped” using KA_Sprout.FormatPrivate as specified in §4.2.1 ‘Sprout Key Components’ on p. 28. That is, a decoded incoming viewing key MUST be considered invalid if $sk_enc \neq KA_{Sprout}.FormatPrivate(sk_enc)$.

KA_{Sprout}.FormatPrivate is defined in §5.4.4.1 ‘Sprout Key Agreement’ on p. 60.

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 $\ell_{ivk}$ be as defined in §5.3 ‘Constants’ on p. 51.

A Sapling incoming viewing key consists of $ivk : \{0 .. 2^\ell_{ivk} - 1\}$.

$ivk$ is a KA_Sapling.Private key (restricted to $\ell_{ivk}$ bits), derived as described in §4.2.2 ‘Sapling Key Components’ on p. 28. It is used with the encryption scheme defined in §4.17 ‘In-band secret distribution (Sapling)’ on p. 46.

The raw encoding of a Sapling 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 $\{0 .. 2^\ell_{ivk} - 1\}$ as specified in §4.2.2 ‘Sapling Key Components’ on p. 28. 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 $ak : \mathbb{J}^{(r)*}$, $nk : \mathbb{J}^{(r)}$, and $ovk : \mathbb{B}^{\lfloor \ell_{ovk}/8 \rfloor}$.

$ak$ and $nk$ are points on the Jubjub curve (see §5.4.8.3 ‘Jubjub’ on p. 69). They are derived as described in §4.2.2 ‘Sapling Key Components’ on p. 28.

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

- 32 bytes specifying the ctEdwards compressed encoding of ak (see §5.4.8.3 ‘Jubjub’ on p. 69).
- 32 bytes specifying the ctEdwards compressed encoding of nk.
- 32 bytes specifying the outgoing viewing key ovk.

When decoding this representation, the key is not valid if $abst_j$ returns \perp for either ak or nk, or if ak $\notin \mathbb{J}^{(r)*}$, or if nk $\notin \mathbb{J}^{(r)}$.

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”.

77
5.6.8 Sprout Spending Keys

A Sprout spending key consists of \( a_{sk} \), which is a sequence of 252 bits (see § 4.2.1 ‘Sprout Key Components’ on p. 28).

The raw encoding of a Sprout spending key consists of:

| 8-bit 0xAB | 8-bit 0x36 | \([0]^4\) | 252-bit \( a_{sk} \) |

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

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

Notes:
- If an implementation represents \( a_{sk} \) internally as a sequence of 32 bytes with the 4 bits of zero padding intact, it will be in the correct form for use as an input to PRF\(^{addr}\), PRF\(^{nf}\), and 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 \( sk^{[\ell_{sk}]} \) (see § 4.2.2 ‘Sapling Key Components’ on p. 28).

The raw encoding of a Sapling spending key consists of:

\[
\text{LEBS2OSP}_{256}(sk)
\]

- 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 BCTV14 zk-SNARK Parameters

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

- sprout-proving.key: `8bc20a7f013b2b58970cddd2e7ea028975c88ae7ceb9259a5344a16bc2c0eeef7`
- sprout-verifying.key: `4bd498dae0aacfd8e98dc306338d017d9c08dd0918ead18172bd0aec2fc5df82`

These parameters were obtained by a multi-party computation described in [BGG-mpc] and [BGG2017]. They are used only before Sapling activation. Due to the security vulnerability described in § 5.4.9.1 ‘BCTV14’ on p. 72, it is not recommended to use these parameters in new protocols, and it is recommended to stop using them in protocols other than Zcash where they are currently used.
5.8 Groth16 zk-SNARK Parameters

*bellman [Bowe-bellman]* encodes the **proving key** and **verifying key** for a **zk-SNARK circuit** in a single parameters file. The BLAKE2b-512 hashes of this file for the **Sapling** Spend circuit and Output circuit, and for the implementation of the Sprout JoinSplit circuit used after Sapling activation, are respectively:

```
8270785a1a0d0bc77196f000e6d221c9c8984f55307bd9357c3f0105d31ca63
919ab91324160d8f53e2bd3c2633a6eb8d5205d822e7f3f73edac5b1b70c  sapling-spend.params
657e3d38db5cb5e7dd2970e8b03d69b47877dd907285b5a7f0790cc8072f60b
f593b32cc2d1c0306e00ff5a64bf84c5c3eb84ddc041d48264b4171744d028  sapling-output.params
e92b38411bd6c0ec4791e9d04245ec350c9c5744f5610dfce43656ca49dfef
d505e371842b3f88fa1b9d7e8e075249b3ebabd167fa8b03f161292d36c180a  sprout-groth16.params
```

These parameters were obtained by a multi-party computation described in [BGM2017].

5.9 Randomness Beacon

Let \( URS = \text{"096b36a5804bfacef1691e173c366a47ff5b8a84a44f26dd7e8d9f79d5b42d0f"} \). This value is used in the definition of **GroupHash**\(^{(B)}\) in § 5.4.8.5 ‘**Group Hash into Jubjub**’ on p. 71, and in the multi-party computation to obtain the Sapling parameters given in § 5.8 ‘Groth16 zk-SNARK Parameters’ on p. 79.

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 0x000000000000000000000000000034b33e842ac1c50456abe65fa92b60f6b3dfc5d247f7b58.
- Apply SHA-256 \(2^{42}\) times.
- Convert to a US-ASCII lowercase hexadecimal string.

**Note:** \( URS \) is a 64-byte US-ASCII string, i.e. the first byte is 0x30, not 0x09.

6 Network Upgrades

**Zcash** launched with a protocol revision that we call **Sprout**. A first network upgrade, called **Overwinter**, activated on the production Zcash network on 26 June 2018 at block height 347500 [Swihart2018] [ZIP-201]. A second upgrade, called **Sapling**, activated on the production network on 28 October 2018 at block height 419200 [Hamdon2018] [ZIP-205]. A third upgrade, called **Blossom**, activated on the production network on 11 December 2019 at block height 653600 [Zcash-Blossom] [ZIP-206]. A fourth upgrade, called **Heartwood**, is planned to activate on the production network in mid-July 2020, at block height 903000 [ZIP-250]. A fifth upgrade, called **Canopy**, is planned to activate on the production network in November 2020, at block height 1046400 (coinciding with the first block subsidy halving) [ZIP-251].

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

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

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

The specifications of the **Sapling** upgrade are described in this document, [ZIP-205], and [ZIP-243].

The specifications of the **Blossom** upgrade are described in this document, [ZIP-206], and [ZIP-208].

The specifications of the **Heartwood** upgrade are described in this document, [ZIP-250], [ZIP-213], and [ZIP-221].
Each network upgrade is introduced as a "bilateral consensus rule change". In this kind of upgrade,

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

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

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

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

7.1 Encoding of Transactions

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| ≥ 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. |
| ≥ 1     | Varies| tx_in                 | tx_in     | Transparent inputs, encoded as in Bitcoin. |
| ≥ 1     | Varies| tx_out_count          | compactSize uint | Number of transparent outputs. |
| ≥ 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. 85. |
| ≥ 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. 85. |
| ≥ 2     | Varies| nJoinSplit            | compactSize uint | The number of JoinSplit descriptions in vJoinSplit. |
| 2..3    | 1802·nJoinSplit | vJoinSplit            | JSDescriptionBCTV14 \([nJoinSplit]\) | A sequence of JoinSplit descriptions using BCTV14 proofs, each encoded as in §7.2 'Encoding of JoinSplit Descriptions' on p. 84. |
| ≥ 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. 84. |
| ≥ 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. 64. |

† 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. 37.

‡ 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]** 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 transparent inputs from coinbase transactions **MUST** have no transparent outputs (i.e. tx_out_count **MUST** be 0). Note that inputs from coinbase transactions include Founders' Reward outputs.
- If version ≥ 2 and nJoinSplit > 0, then:
  - joinSplitPubKey **MUST** represent a valid Ed25519 public key encoding (§ 5.4.5 ‘JoinSplit Signature’ on p. 6).
  - joinSplitSig **MUST** represent a valid signature under joinSplitPubKey of dataToBeSigned, as defined in §4.10 ‘Non-malleability (Sprout)’ on p. 37.
- **[Sapling onward]** If version ≥ 4 and nShieldedSpend + nShieldedOutput > 0, then:
  - let bvk and SigHash be as defined in §4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38;
  - bindingSig **MUST** represent a valid signature under the transaction binding verification key bvk of SigHash – i.e. BindingSig.Verify_bvk(SigHash, 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 or Spend descriptions.
- **[Pre-Heartwood]** A coinbase transaction also **MUST NOT** have any Output descriptions.
- A transaction **MUST NOT** spend a transparent output of a coinbase transaction from a block less than 100 blocks prior to the spend. Note that transparent outputs of coinbase transactions include Founders' Reward outputs.
- **[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 {−MAX_MONEY..MAX_MONEY}.
- **[Heartwood onward]** All Sapling outputs in coinbase transactions **MUST** have valid note commitments when recovered using a sequence of 32 zero bytes as the outgoing viewing key.
- TODO: Other rules inherited from Bitcoin.

In addition, consensus rules associated with each JoinSplit description (§7.2 ‘Encoding of JoinSplit Descriptions’ on p. 84), each Spend description (§7.3 ‘Encoding of Spend Descriptions’ on p. 85), and each Output description (§7.4 ‘Encoding of Output Descriptions’ on p. 85) **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 Overwintered 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 defined in [ZIP-200], and in that case the Overwintered and version fields alone may be insufficient to determine the format to be used for parsing.

- A transaction version number of 2 does not have the same meaning as in Bitcoin, where it is associated with support for OP_CHECKSEQUENCEVERIFY as specified in [BIP-68]. Zcash was forked from Bitcoin v0.11.2 and does not currently support BIP 68.

- Prior to the Heartwood network upgrade, it was not possible for coinbase transactions to have shielded outputs, and therefore the "coinbase maturity" rule and the requirement to spend coinbase outputs only in transactions with no transparent outputs, applied to all coinbase outputs.

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

- Transaction version 0 is not supported.

- A version 1 transaction is equivalent to a version 2 transaction with nJoinSplit = 0.

- The nJoinSplit, vJoinSplit, joinSplitPubKey, and joinSplitSig fields have been added.

- [Overwinter onward] The nVersionGroupId field has been added.

- [Sapling onward] The nShieldedSpend, vShieldedSpend, nShieldedOutput, vShieldedOutput, and bindingSig fields have been added.

- In Zcash it is permitted for a transaction to have no transparent inputs, provided at least one of nJoinSplit, nShieldedSpend, and nShieldedOutput are nonzero.

- 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. 16, is encoded in a *transaction* as an instance of a *JoinSplitDescription* type as follows:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>vpub_old</td>
<td>uint64</td>
<td>A value $v_{pub}^{old}$ that the <em>JoinSplit</em> transfer removes from the transparent transaction value pool.</td>
</tr>
<tr>
<td>8</td>
<td>vpub_new</td>
<td>uint64</td>
<td>A value $v_{pub}^{new}$ that the <em>JoinSplit</em> transfer inserts into the transparent transaction value pool.</td>
</tr>
<tr>
<td>32</td>
<td>anchor</td>
<td>char[32]</td>
<td>A root $rt$ of the Sprout note commitment tree at some block height in the past, or the root produced by a previous <em>JoinSplit</em> transfer in this transaction.</td>
</tr>
<tr>
<td>64</td>
<td>nullifiers</td>
<td>char[32][N^old]</td>
<td>A sequence of <em>nullifiers</em> of the input notes $n_{i}^{old}$..</td>
</tr>
<tr>
<td>64</td>
<td>commitments</td>
<td>char[32][N^new]</td>
<td>A sequence of <em>note commitments</em> for the output notes $cm_{i}^{new}$.</td>
</tr>
<tr>
<td>32</td>
<td>randomSeed</td>
<td>char[32]</td>
<td>A 256-bit seed that must be chosen independently at random for each <em>JoinSplit</em> description.</td>
</tr>
<tr>
<td>64</td>
<td>vmacs</td>
<td>char[32][N^old]</td>
<td>A sequence of message authentication tags $h_{i}^{1..N^{old}}$ binding $h_{Sig}$ to each $a_{sk}$ of the <em>JoinSplit description</em>, computed as described in §4.10 ‘<em>Non-malleability (Sprout)</em>’ on p. 37.</td>
</tr>
<tr>
<td>296†</td>
<td>zkproof</td>
<td>char[296]</td>
<td>An encoding of the zk-SNARK proof $\pi_{ZKJoinSplit}$ (see §5.4.9.1 ‘<em>BCTV14</em>’ on p. 72).</td>
</tr>
<tr>
<td>192‡</td>
<td>zkproof</td>
<td>char[192]</td>
<td>An encoding of the zk-SNARK proof $\pi_{ZKJoinSplit}$ (see §5.4.9.2 ‘<em>Groth16</em>’ on p. 73).</td>
</tr>
<tr>
<td>1202</td>
<td>encCiphertexts</td>
<td>char[601][N^new]</td>
<td>A sequence of ciphertext components for the encrypted output notes $C_{i}^{enc}$.</td>
</tr>
</tbody>
</table>

† BCTV14 proofs are used when the transaction version is 2 or 3, i.e. before *Sapling* activation.
‡ Groth16 proofs are used when the transaction version is $\geq 4$, i.e. after *Sapling* activation.

The ephemeralKey and encCiphertexts fields together form the transmitted notes ciphertext, which is computed as described in §4.16 ‘*In-band secret distribution (Sprout)*’ on p. 45.

Consensus rules applying to a *JoinSplit description* are given in §4.3 ‘*JoinSplit Descriptions*’ on p. 30.
7.3 Encoding of Spend Descriptions

Let LEBS2OSP be as defined in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p.50. Let repr\_J and q\_J be as defined in §5.4.8.3 ‘Jubjub’ on p.69.

An abstract Spend description, as described in §3.6 ‘Spend Transfers, Output Transfers, and their Descriptions’ on p.16, is encoded in a transaction as an instance of a SpendDescription type as follows:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>cv</td>
<td>char[32]</td>
<td>A value commitment to the value of the input note, LEBS2OSP_256(repr_J(cv)).</td>
</tr>
<tr>
<td>32</td>
<td>anchor</td>
<td>char[32]</td>
<td>A root of the Sapling note commitment tree at some block height in the past, LEBS2OSP_256(rt).</td>
</tr>
<tr>
<td>32</td>
<td>rk</td>
<td>char[32]</td>
<td>The randomized public key for spendAuthSig, LEBS2OSP_256(repr_J(rk)).</td>
</tr>
<tr>
<td>192</td>
<td>zkproof</td>
<td>char[192]</td>
<td>An encoding of the zk-SNARK proof (\pi_{ZKOutput}) (see §5.4.9.2 ‘Groth16’ on p.73).</td>
</tr>
<tr>
<td>64</td>
<td>spendAuthSig</td>
<td>char[64]</td>
<td>A signature authorizing this Spend.</td>
</tr>
</tbody>
</table>

Consensus rule: LEOS2IP\_256(\text{anchor}) MUST be less than q\_J.

Other consensus rules applying to a Spend description are given in §4.4 ‘Spend Descriptions’ on p.31.

7.4 Encoding of Output Descriptions

Let LEBS2OSP be as defined in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p.50. Let repr\_J and q\_J be as in §5.4.8.3 ‘Jubjub’ on p.69, and Extract\_J as in §5.4.8.5 ‘Group Hash into Jubjub’ on p.71.

An abstract Output description, described in §3.6 ‘Spend Transfers, Output Transfers, and their Descriptions’ on p.16, is encoded in a transaction as an instance of an OutputDescription type as follows:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>cv</td>
<td>char[32]</td>
<td>A value commitment to the value of the output note, LEBS2OSP_256(repr_J(cv)).</td>
</tr>
<tr>
<td>32</td>
<td>cmu</td>
<td>char[32]</td>
<td>The (u)-coordinate of the note commitment for the output note, LEBS2OSP_256(cm_u) where (cm_u = \text{Extract}_J(cm)).</td>
</tr>
<tr>
<td>32</td>
<td>ephemeralKey</td>
<td>char[32]</td>
<td>An encoding of an ephemeral Jubjub public key, LEBS2OSP_256(repr_J(epk)).</td>
</tr>
<tr>
<td>580</td>
<td>encCiphertext</td>
<td>char[580]</td>
<td>A ciphertext component for the encrypted output note, (C^\text{enc}).</td>
</tr>
<tr>
<td>80</td>
<td>outCiphertext</td>
<td>char[80]</td>
<td>A ciphertext component for the encrypted output note, (C^\text{out}).</td>
</tr>
<tr>
<td>192</td>
<td>zkproof</td>
<td>char[192]</td>
<td>An encoding of the zk-SNARK proof (\pi_{ZKOutput}) (see §5.4.9.2 ‘Groth16’ on p.73).</td>
</tr>
</tbody>
</table>

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.46.

Consensus rule: LEOS2IP\_256(cm\_u) MUST be less than q\_J.

Other consensus rules applying to an Output description are given in §4.5 ‘Output Descriptions’ on p.32.
### 7.5 Block Header

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

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>nVersion</td>
<td>int32</td>
<td>The block version number indicates which set of block validation rules to follow. The current and only defined block version number for Zcash is 4.</td>
</tr>
<tr>
<td>32</td>
<td>hashPrevBlock</td>
<td>char[32]</td>
<td>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.</td>
</tr>
<tr>
<td>32</td>
<td>hashMerkleRoot</td>
<td>char[32]</td>
<td>A SHA-256d hash in internal byte order. The merkle root is derived from the hashes of all transactions included in this block, ensuring that none of those transactions can be modified without modifying the header.</td>
</tr>
<tr>
<td>32</td>
<td>hashReserved / hashFinalSaplingRoot / hashLightClientRoot</td>
<td>char[32]</td>
<td>[Pre-Sapling] A reserved field which should be ignored. [Sapling and Blossom only. pre-Heartwood] The root LEBS2OSP&lt;sub&gt;256&lt;/sub&gt;(rt) of the Sapling note commitment tree corresponding to the final Sapling treestate of this block. [Heartwood onward] The hashChainHistoryRoot of this block.</td>
</tr>
<tr>
<td>4</td>
<td>nTime</td>
<td>uint32</td>
<td>The block timestamp is a Unix epoch time (UTC) when the miner started hashing the header (according to the miner).</td>
</tr>
<tr>
<td>4</td>
<td>nBits</td>
<td>uint32</td>
<td>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]</td>
</tr>
<tr>
<td>32</td>
<td>nNonce</td>
<td>char[32]</td>
<td>An arbitrary field that miners can change to modify the header hash in order to produce a hash less than or equal to the target threshold.</td>
</tr>
<tr>
<td>3</td>
<td>solutionSize</td>
<td>compactSize uint</td>
<td>The size of an Equihash solution in bytes (always 1344).</td>
</tr>
</tbody>
</table>

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

Let ThresholdBits be as defined in §7.6.3 ‘Difficulty adjustment’ on p. 89, and let PoWMedianBlockSpan be the constant defined in §5.3 ‘Constants’ on p. 51.

Define the median-time-past of a block to be the median (as defined in §7.6.3 ‘Difficulty adjustment’ on p. 89) of the nTime fields of the preceding PoWMedianBlockSpan blocks (or all preceding blocks if there are fewer than PoWMedianBlockSpan). The median-time-past of a genesis block is not defined.
Consensus rules:

- The block version number MUST be greater than or equal to 4.
- For a block at block height height, nBits MUST be equal to ThresholdBits(height).
- The block MUST pass the difficulty filter defined in §7.6.2 ‘Difficulty filter’ on p. 89.
- solution MUST represent a valid Equihash solution as defined in §7.6.1 ‘Equihash’ on p. 88.
- For each block other than the genesis block, nTime MUST be strictly greater than the median-time-past of that block.
- For each block at block height 2 or greater on the production network, or block height 653606 or greater on the test network, nTime MUST be less than or equal to the median-time-past of that block plus 90 · 60 seconds.
- The size of a block MUST be less than or equal to 2000000 bytes.
- [Sapling and Blossom only, pre-Heartwood] hashLightClientRoot MUST be LEBS2OSP_{256}(rt) where rt is the root of the Sapling note commitment tree for the final Sapling treestate of this block.
- [Heartwood onward] hashLightClientRoot MUST be set to the value of hashChainHistoryRoot for this block, as specified in [ZIP-221].
- The block height MUST be encoded as the first item in the coinbase transaction’s scriptSig, as specified in [BIP-34]. The format of the height is “serialized CScript” – the first byte is the number of bytes in the number, and the following bytes are the signed little-endian representation of the number.
- TODO: Other rules inherited from Bitcoin.

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

Notes:

- The semantics of blocks with block version number not equal to 4 is not currently defined. Miners MUST NOT create such blocks.
- The exclusion of blocks with block version number greater than 4 is not a consensus rule; such blocks may exist in the block chain and MUST be treated identically to version 4 blocks by full validators. Note that a future upgrade might use block version number either greater than or less than 4. It is likely that such an upgrade will change the block header and/or transaction format, and software that parses blocks SHOULD take this into account.
- The nVersion field is a signed integer. (It was specified as unsigned in a previous version of this specification.) A future upgrade might use negative values for this field, or otherwise change its interpretation.
- There is no relation between the values of the version field of a transaction, and the nVersion field of a block header.
- Like other serialized fields of type compactSize 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.
- The rule limiting nTime to be no later than 90 · 60 seconds after the median-time-past is a retrospective consensus change, applied as a soft fork in zcashd v2.1.1–1. It had not been violated by any block from the given block heights in the consensus block chains of either the production or test networks.
- There are no changes to the block version number or format for Overwinter.
Although the block version number does not change for Sapling, the previously reserved (and ignored) field hashReserved has been repurposed for hashFinalSaplingRoot. There are no other format changes.

There are no changes to the block version number or format for Blossom.

For Heartwood, the hashFinalSaplingRoot field is renamed to hashLightClientRoot. Once Heartwood activates, the meaning of this field changes according to [ZIP-221].

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

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

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

- The solution field encodes a valid Equihash solution according to § 7.6.1 ‘Equihash’ on p. 88.
- The block header satisfies the difficulty check according to § 7.6.2 ‘Difficulty filter’ on p. 89.

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 ≥ 3.

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

Equihash is based on a variation of the Generalized Birthday Problem [AR2017]: given a sequence X₁..N of n-bit strings, find 2^k distinct X_i, such that \( \bigoplus_{j=1}^{2^k} X_{i,j} = 0 \).

In Equihash, N = 2^{n \cdot k} + 1, and the sequence X₁..N is derived from the block header and a nonce.

Let powheader :=

<table>
<thead>
<tr>
<th>List Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit nVersion</td>
</tr>
<tr>
<td>256-bit hashReserved</td>
</tr>
</tbody>
</table>

For i ∈ {1..N}, let X_i = EquihashGen_{n,k} (powheader, i).

EquihashGen is instantiated in § 5.4.1.9 ‘Equihash Generator’ on p. 58.

Define I2BEBSH : (\ell : N) × \{0..2^\ell - 1\} → \mathbb{B}[\ell] as in § 5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 50.

A valid Equihash solution is then a sequence i : {1..N}^{2^k} that satisfies the following conditions:

**Generalized Birthday condition**

\[ \bigoplus_{j=1}^{2^k} X_{i,j} = 0. \]

**Algorithm Binding conditions**

- For all r ∈ {1..k-1}, for all w ∈ {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 ∈ {1..k}, for all w ∈ {0..2^{k-r} - 1} : \( i_{w \cdot 2^r+1}..w \cdot 2^r+2^r-1 \) < \( i_{w \cdot 2^r+2^r-1+1}..w \cdot 2^r+2^r \) lexicographically.
Notes:

- This does not include a difficulty condition, because here we are defining validity of an *Equihash* solution independent of difficulty.
- Previous versions of this specification incorrectly specified the range of \( r \) to be \( \{1..k-1\} \) for both parts of the algorithm binding condition. The implementation in zcashd was as intended.

An *Equihash* solution with \( n = 200 \) and \( k = 9 \) is encoded in the solution field of a *block header* as follows:

\[
\begin{array}{cccc}
I2BEBS\_21(i_1 - 1) & I2BEBS\_21(i_2 - 1) & \cdots & I2BEBS\_21(i_{512} - 1) \\
\end{array}
\]

Recall from § 5.2 'Integers, Bit Sequences, and Endianness' on p. 50 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>
<thead>
<tr>
<th>I2BEBS_21(68)</th>
<th>I2BEBS_21(41)</th>
<th>I2BEBS_21(2^{21} - 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 1 0 1 0</td>
<td>1 1 1 1 1 1 1 1</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>8-bit 0</td>
<td>8-bit 2</td>
<td>8-bit 32</td>
</tr>
<tr>
<td>8-bit 0</td>
<td>8-bit 0</td>
<td>8-bit 10</td>
</tr>
<tr>
<td>8-bit 127</td>
<td>8-bit 255</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

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

**Note:** \( I2BEBS \) 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. 91.

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. 89.

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

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

The constants \( PoWLimit \), \( PreBlossomHalvingInterval \), \( PoWAveragingWindow \), \( PoWMaxAdjustDown \), \( PoWMaxAdjustUp \), \( PoWDampingFactor \), \( PreBlossomPoWTargetSpacing \), and \( PostBlossomPoWTargetSpacing \) are specified in section § 5.3 ‘Constants’ on p. 51.

Let \( ToCompact \) and \( ToTarget \) be as defined in § 7.6.4 ‘nBits conversion’ on p. 91.

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. 86.
Define:

\[
\begin{align*}
\text{mean}(S) & := \frac{\sum_{i=1}^{\text{length}(S)} S_i}{\text{length}(S)} \\
\text{median}(S) & := \text{sorted}(S)_{\text{ceiling}((\text{length}(S)+1)/2)} \\
\text{bound}_{\text{upper/lower}}(x) & := \max(\text{lower}, \min(\text{upper}, x)) \\
\text{trunc}(x) & := \begin{cases} 
\text{floor}(x), & \text{if } x \geq 0 \\
-\text{floor}(-x), & \text{otherwise}
\end{cases}
\end{align*}
\]

\[\text{IsBlossomActivated}(\text{height} : \mathbb{N}) := (\text{height} \geq \text{BlossomActivationHeight})\]

\[\text{BlossomPoWTargetSpacingRatio} := \frac{\text{PreBlossomPoWTargetSpacing}}{\text{PostBlossomPoWTargetSpacing}}\]

\[\text{PostBlossomHalvingInterval} := \text{floor}(\text{PreBlossomHalvingInterval} \cdot \text{BlossomPoWTargetSpacingRatio})\]

\[\text{PoWTargetSpacing}(\text{height} : \mathbb{N}) := \begin{cases} 
\text{PreBlossomPoWTargetSpacing}, & \text{if not IsBlossomActivated}(\text{height}) \\
\text{PostBlossomPoWTargetSpacing}, & \text{otherwise}
\end{cases}\]

\[\text{AveragingWindowTimespan}(\text{height} : \mathbb{N}) := \text{PoWAveragingWindow} \cdot \text{PoWTargetSpacing}(\text{height})\]

\[\text{MinActualTimespan}(\text{height} : \mathbb{N}) := \text{floor}(\text{AveragingWindowTimespan}(\text{height}) \cdot (1 - \text{PoWMaxAdjustUp}))\]

\[\text{MaxActualTimespan}(\text{height} : \mathbb{N}) := \text{floor}(\text{AveragingWindowTimespan}(\text{height}) \cdot (1 + \text{PoWMaxAdjustDown}))\]

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

\[\text{ActualTimespan}(\text{height} : \mathbb{N}) := \text{MedianTime}(\text{height}) - \text{MedianTime}(\text{height} - \text{PoWAveragingWindow})\]

\[\text{ActualTimespanDamped}(\text{height} : \mathbb{N}) := \text{AveragingWindowTimespan}(\text{height} : \mathbb{N}) + \text{trunc}\left(\frac{\text{ActualTimespan}(\text{height}) - \text{AveragingWindowTimespan}(\text{height})}{\text{PoWDampingFactor}}\right)\]

\[\text{ActualTimespanBounded}(\text{height} : \mathbb{N}) := \text{bound}_{\text{MinActualTimespan}(\text{height}) \text{ to } \text{MaxActualTimespan}(\text{height})} \text{ActualTimespanDamped}(\text{height})\]

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

The target threshold for a given block height \text{height} is then calculated as:

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

\[\text{ThresholdBits}(\text{height} : \mathbb{N}) := \text{ToCompact}(\text{Threshold}(\text{height})).\]

Notes:

\[\text{•} \text{The convention used for the height parameters to the functions } \text{MedianTime}, \text{MeanTarget}, \text{ActualTimespan}, \text{ActualTimespanDamped}, \text{ActualTimespanBounded}, \text{Threshold}, \text{and ThresholdBits is that these functions use only information from blocks preceding the given block height.}\]

\[\text{•} \text{When the median function is applied to a sequence of even length (which only happens in the definition of } \text{MedianTime} \text{ during the first } \text{PoWAveragingWindow} - 1 \text{ blocks of the block chain}, \text{the element that begins the second half of the sequence is taken. This corresponds to the } \text{zcashd implementation}, \text{but was not specified correctly in versions of this specification prior to 2019.0-beta-40.}\]

On the test network from block height 299188 onward, the difficulty adjustment algorithm is changed to allow minimum-difficulty blocks, as described in [ZIP-205]. The Blossom network upgrade changes the minimum-difficulty time threshold to 6 times the block target spacing, as described in [ZIP-208]. These changes do not apply to the production network.
7.6.4 nBits conversion

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

\[
\begin{align*}
\text{size}(x) & := \text{ceiling} \left( \frac{\text{bitlength}(x)}{8} \right) \\
\text{mantissa}(x) & := \text{floor} \left( x \cdot 256^{3 - \text{size}(x)} \right) \\
\text{ToCompact}(x) & := \begin{cases} 
\text{mantissa}(x) + 2^{24} \cdot \text{size}(x), & \text{if } \text{mantissa}(x) < 2^{23} \\
\text{floor} \left( \frac{\text{mantissa}(x)}{256} \right) + 2^{24} \cdot (\text{size}(x) + 1), & \text{otherwise} 
\end{cases} \\
\text{ToTarget}(x) & := \begin{cases} 
0, & \text{if } x \cdot 2^{23} = 2^{23} \\
(x \cdot (2^{23} - 1)) \cdot 256^{\text{floor}(x/2^{24}) - 3}, & \text{otherwise.} 
\end{cases}
\end{align*}
\]

7.6.5 Definition of Work

As explained in § 3.3 ‘The Block Chain’ on p. 15, a node chooses the “best” block chain visible to it by finding the chain of valid blocks with the greatest total work.

Let \text{ToTarget} be as defined in § 7.6.4 ‘nBits conversion’ on p. 91.

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

7.7 Calculation of Block Subsidy and Founders’ Reward

§ 3.9 ‘Block Subsidy and Founders’ Reward’ on p. 18 defines the block subsidy, miner subsidy, and Founders’ Reward. Their amounts in zatoshi are calculated from the block height using the formulae below.

Let the constants \text{SlowStartInterval}, \text{PreBlossomHalvingInterval}, \text{PostBlossomHalvingInterval}, \text{BlossomActivationHeight}, \text{MaxBlockSubsidy}, and \text{FoundersFraction} be as defined in § 5.3 ‘Constants’ on p. 51.

\[
\begin{align*}
\text{SlowStartShift} & := \frac{\text{SlowStartInterval}}{2} \\
\text{SlowStartRate} & := \frac{\text{MaxBlockSubsidy}}{\text{SlowStartInterval}} \\
\text{Halving}(\text{height} : \mathbb{N}) & := \begin{cases} 
\text{floor} \left( \frac{\text{height} - \text{SlowStartShift}}{\text{PreBlossomHalvingInterval}} \right), & \text{if not IsBlossomActivated(\text{height})} \\
\text{floor} \left( \frac{\text{BlossomActivationHeight} - \text{SlowStartShift}}{\text{PreBlossomHalvingInterval}} \right) + \frac{\text{height} - \text{BlossomActivationHeight}}{\text{PostBlossomHalvingInterval}}, & \text{otherwise} 
\end{cases} \\
\text{BlockSubsidy}(\text{height} : \mathbb{N}) & := \begin{cases} 
\text{SlowStartRate} \cdot \text{height}, & \text{if } \text{height} < \frac{\text{SlowStartInterval}}{2} \\
\text{SlowStartRate} \cdot (\text{height} + 1), & \text{if } \frac{\text{SlowStartInterval}}{2} \leq \text{height} \text{ and } \text{height} < \frac{\text{SlowStartInterval}}{2} \\
\text{floor} \left( \frac{\text{MaxBlockSubsidy}}{2^{\text{Halving}(\text{height})}} \right), & \text{if } \text{SlowStartInterval} \leq \text{height} \text{ and not IsBlossomActivated(\text{height})} \\
\text{floor} \left( \frac{\text{MaxBlockSubsidy}}{\text{BlossomPoWTargetSpacingRatio} \cdot 2^{\text{Halving}(\text{height})}} \right), & \text{otherwise} 
\end{cases} \\
\text{FoundersReward}(\text{height} : \mathbb{N}) & := \begin{cases} 
\text{BlockSubsidy}(\text{height}) \cdot \text{FoundersFraction}, & \text{if } \text{Halving}(\text{height}) < 1 \\
0, & \text{otherwise} 
\end{cases} \\
\text{MinerSubsidy}(\text{height} : \mathbb{N}) & := \text{BlockSubsidy}(\text{height}) - \text{FoundersReward}(\text{height}).
\end{align*}
\]
7.8 Payment of Founders’ Reward

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

For the production network, FounderAddressList_1...NumFounderAddresses is:

```
[ "t3Vz2v22yK5zwL0cKEdg16YV4FfNeeElzg9ojd", "t3C1WaUCa3jm3hDbb5JbnJ2krapVoxSo3", ".f3qvkxzzrNaemKQMwHyrJisdM2xqvDTR", "t3TG9R272CSTK44AnUPi6qenNaHe2eCpulFy", ".f3SpkkPQFfpuHRs5p5r3v8P5gk05n9vmx", "t3Xt4oQPvRagwpvBqkgAVijqT34oV5WRS", ".f3ayBk24wKXYw0hvFZUSPlkSTK7XNg", "t3adsJXQqua217xkbR87Nzep3kmzS24NGB", "t35k4AlYuS88yDrdkF5eGd5jd95zovQsz8t", "f3R7nasc5nHxVkwVa3p2fHrsk7eHy1cR6rK", "f3Ut4KqZ2SMFPb67PU5LuqYc12q36KpXQ", "t3ZnAvngujeCSyHm1wVtx3a19n9DA6gn", ".f3Br9b3e3Sey1m64S9uxxvAHJQKgLQqRoBDG", "f3cVzRKNNjxVzXMABHqee6m6x8kKFRndh18Kd", ".f3YcojuXfSpwy7rbnUSXeFwZqNtstope04", "f3LbC1Lr6c6brbFtUSN5WkgyVrZcZumTara", ".f3VhHwaw7rimy67yUt4LZGCWa2J6eGhVh1", "f3eF9X6X2dSo7MCvTffZExWVYqzVRgLNe", ".f3esCnwWcyc819QqYtyBhTmqXZ94AwK3X", "f3M4j7nHE2e27yLsuQjPjuVek81W3vBj", ".f3GwQgCd7FYN0bBdpVnrwLAWpqZLxrvr", "f3LTweoeWPbmdkUD3NBWqk4W3kzahFBnV", ".f3P5KkX97gyXF8ysajjPiruOEX4yf6z3TJq", "f3f3T3mCwpzEmpD35vK62JgqFfQ74dV9C9", ".f3Rqonuzzafkxf7156ZAv4i1imRSEn14j", "f3jJ5ZjYsyxdtvWrnMrwQJaqCj4JjgqX", ".f3Pnb7gFgPQbpZuCj5LSt7rwFem1MTLZ9vjkFhp", "f3Y9FNF126JUTa4CCoaETLmB08Ks1E5e6", "f3anRLLLs28xjcpjZe2Wfy3Cy7Cvs7TwBc", "f3QGEarv5KyAHK87rQu2BWDLXei1unBm", "f3bykhX1TuFRgXmYBrJa2STxRFLG79r", ".f3Aa4nM4TDpa87d1rY1Bd2yHybgegagMr", "f3Ye1Aa6eUjXyFLv2z5tU1f13nKg3qMqy", "f3gy1UwttP2PmdvHDevTCwUb6atLt21QQP", "f3DPwnepy6GypqU1ycopgzGr3Y91UWbHdy4z", ".f3QzRZHPdh2wQo1q6Qs277krKuuoWfWp4dV", "f3enhaCRx1i2D7e8ePomV0Ks7wp7NUM9F3", ".f3Sklg77176nFt12wMvtx5D7yNxb2gJY", "f3LQtHudo7ZzhvddKrv4moanuAHRvf4AOF", ".f3nCbdUBgvyctSntD9n3LxJvFPvBFB", "f3dkoJoU2Ejhe28oH8Y4tKvVEdU1m1FsEx", ".f3AH68iw1Wn0fd08c192l1gYtpjk3n697ME", "f3MEXDF9ws136kp3QudQ6dby32Mw2bITEA", ".f3WDisP1k34yWmp7tqZa0d8AS38YKT3", "f3PSn5TMMAAvq7Ue036YctFzRzp1hizF3M", ".f3R375vmBlnrEnL6wFpJpPbLnxSUQsKmFp", "f3pCm376EsVgkThbsu2NekktJeg92mvYyoN" ]
```

For the test network, FounderAddressList_1...NumFounderAddresses is:

```
[ "t3UnZuxUs8mBcRyPezva363EYXyEpHoKyi", "t3N9PHwK9xjxyGy9iin1u3aeekJqFaeT543", ".f3NGQjYMqfHmdDrhuGv4wZdssSaK7xZ", "f3Enhj7hvQq9JwU5cgjUBxntT2a9UNfhy", "f3BkgdCVIRz7Tu4x2BqeeQFrpqspxBbo", "f3EaMuPIQk1htQp05UEXJF42CAFkJQKXC9", ".f3F9dQ4c63JdyhrnfPzyT75Mcq12", "f3LPirmeMY3c481Bza6xOcvoovftBnc", "f36fCsoXvU9p5e93C8V4yqDB43ESfxt", "f32k4IfNdEr6cDyxtXEdf9xuloKR7DCv", ".f3DyWYBxKXHvdmHiiVJzutaQGQoMoRjL", "f3C3KfF919Xf4C89zwGw4dQLLqzqjpoUq", ".f3Tmtz5rKsCprpyUnw0tp9MueUSQna", "f3AEREsdw01F8EQYsCscjkggbmgKKeuk", ".f3Yf4wKc32Ft4lj4jezOOPrLr9Lh5UT", "f32k5d1h68r5truxlKhpoxyYxHyWghDn", "f3VEn3KlhySGyD3mdwEuTcaQOQwv9w", "f32F8oxuqdnMq2szExvHvJh5rWHRHegc", ".f3BS7mr3afef4Axmkvd1eFyXvrbm2Og", "f3sFus0LwDvZulrYoHrEzxaB9qy4qJbnl", ".f3S3xUN8rTrg5dO1BtACQC5qJprtrp8GJc6", "f3V51gznsoJSkRL74fb9yTb2v8FCxq2Fh", ".f2yfsTslidJe4mJewIPw4xzjAfkydidr1bA", "f3BeyGlekmqpyHn8UPF5kqahrY7d6N1Le", ".f2NtqRStz2HtJCnF3tdU0yA9AercPmcK", "f32GWSZ7JzoosyFpfPTXkFhp5UaixyJxQGBa", ".f2RpfkxzyLevR05W39aIdeqXMBy6buuAK3v", "f32jzjQquMtXTGNS7k7wyk5keUROBVfYihd", ".f3AEfCef72ieTnsXKmgK2bZNckiwvZe3oPNL", "f3NsN23ZFSfn2jwvmwSBsWfsvETigLrDj", ".f3CCQpcyVcXusQdpdqUgefPEI4HsFvCm", "f3JabDUBK8taqYfQ3DqvkHdpXhVg", ".f2FG2sWZdcCyS89zyKmngv2YloKcmCyryn", "f32DUD8a21lFtEfn4o2VLP5mNBgof1y3yuy9t", ".f2UijVs3R3rHCgPAKJuX8WWQc1C9Xh5/qQhEv", "f32TBAnaELYHUn816SXYsXz5Lmy7kdbAuuT", ".f2Tu3Cyph3t3J30Mv3Cgh7UX09qEYQNC", "f3NysJSzJL7WNL43EBH5B3exRh6j27nccSy", ".f2KXJjVvrYrVxXs8eazb9ykegy4QxQsn9", "f32JYyTH31cevILZjaE4acuwVbo6qjTnp", ".f2QWv4pS9a2PMhlHzE7ipydmurmR4AZ", "f32NDtJPPwosKpyFHPJnfic5cpGvAU58G4a", "f3zy9HdbwQuadNgEVJ5EWHZqEq9e0pkmVrtp", "f3Ez9KMBVJLauccrKxHNKhKivDkkXczj", ".f2D5y7j5xfpXiJbGrDqBqKFGq2mFN8f0m8cX", "f32UVw2r1PtaTuiybpkV3FsdGUXeJzdZyyt" ]
```
Note: For the test network only, the addresses from index 4 onward have been changed from what was implemented at launch. This reflects an upgrade on the test network, starting from block height 53127. [Zcash-Issue2113]

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

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

Define:

\[
\text{FounderAddressChangeInterval} := \text{ceiling} \left( \frac{\text{SlowStartShift} + \text{PreBlossomHalvingInterval}}{\text{NumFounderAddresses}} \right)
\]

\[
\text{FounderAddressAdjustedHeight}(\text{height} : N) :=
\begin{cases}
\text{height}, & \text{if not IsBlossomActivated(\text{height}),} \\
\text{BlossomActivationHeight} + \text{floor}\left( \frac{\text{height} - \text{BlossomActivationHeight}}{\text{BlossomPoWTargetSpacingRatio}} \right), & \text{otherwise}
\end{cases}
\]

\[
\text{FounderAddressIndex}(\text{height} : N) := 1 + \text{floor}\left( \frac{\text{FounderAddressAdjustedHeight}(\text{height})}{\text{FounderAddressChangeInterval}} \right)
\]

\[
\text{FoundersRewardLastBlockHeight} := \max\{\{\text{height} : N \mid \text{Halving(\text{height}) < 1}\}\}
\]

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

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

**Notes:**

- No Founders’ Reward is required to be paid for \text{height} > \text{FoundersRewardLastBlockHeight} (i.e. after the first halving), or for \text{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 \text{height} \in \{1..\text{FoundersRewardLastBlockHeight}\} to have other outputs, possibly to the same address, that do not meet the criterion in the above consensus rule, as long as at least one output meets it.
- The assertion \text{FounderAddressIndex(FoundersRewardLastBlockHeight) \leq NumFounderAddresses} holds, ensuring that the Founders’ Reward address index remains in range for the whole period in which the Founders’ Reward is paid.

**Non-normative notes:**

- [Blossom onward] \text{FoundersRewardLastBlockHeight} = 1046399.
- Blossom is not intended to change the total Founders’ Reward or the effective period over which it is paid.

### 7.9 Changes to the Script System

The \text{OP_CODESEPARATOR} opcode has been disabled. This opcode also no longer affects the calculation of \text{SIGHASH} transaction hashes.

### 7.10 Bitcoin Improvement Proposals

In general, Bitcoin Improvement Proposals (BIPs) do not apply to Zcash unless otherwise specified in this section. All of the BIPs referenced below should be interpreted by replacing "BTC", or "bitcoin" used as a currency unit, with "ZEC"; and "satoshi" with "zatoshi".
The following BIPs apply, otherwise unchanged, to Zcash: [BIP-11], [BIP-14], [BIP-31], [BIP-35], [BIP-37], [BIP-61].

The following BIPs apply starting from the Zcash genesis block, i.e. any activation rules or exceptions for particular blocks in the Bitcoin block chain are to be ignored: [BIP-16], [BIP-30], [BIP-65], [BIP-66]. [BIP-34] applies to all blocks other than the Zcash genesis block (for which the "height in coinbase" was inadvertently omitted).

[BIP-13] applies with the changes to address version bytes described in § 5.6.1 ‘Transparent Addresses’ on p. 75.

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

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

8 Differences from the Zerocash paper

8.1 Transaction Structure

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

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

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

Computation of SIGHASH transaction hashes, as described in § 4.9 ‘SIGHASH Transaction Hashing’ on p. 36, was changed to clean up handling of an error case for SIGHASH_SINGLE, to remove the special treatment of OP_CODESEPARATOR, and to include Zcash-specific fields in the hash [ZIP-76].

8.2 Memo Fields

Zcash adds a memo field sent from the creator of a JoinSplit description to the recipient of each output note. This feature is described in more detail in § 5.5 ‘Encodings of Note Plaintexts and Memo Fields’ on p. 73.

8.3 Unification of Mints and Pours

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

- a "Mint" transaction takes value from transparent UTXOs as input and produces a new shielded note as output.
- a "Pour" transaction takes up to \( N^{\text{old}} \) shielded notes as input, and produces up to \( N^{\text{new}} \) shielded notes and a transparent UTXO as output.
Only "Pour" transactions included a zk-SNARK proof.

[Pre-Sapling] In Zcash, the sequence of operations added to a transaction (see §8.1 ‘Transaction Structure’ on p. 94) consists only of JoinSplit transfers. A JoinSplit transfer is a Pour operation generalized to take a transparent UTXO as input, allowing JoinSplit transfers to subsume the functionality of Mints. An advantage of this is that a Zcash transaction that takes input from an UTXO can produce up to $N^\text{new}$ output notes, improving the indistinguishability properties of the protocol. A related change conceals the input arity of the JoinSplit transfer: an unused (zero-value) input is indistinguishable from an input that takes value from a note.

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

[Sapling onward] In Sapling, there are still no "Mint" transactions. Instead of JoinSplit transfers, there are Spend transfers and Output transfers. These make use of Pedersen value commitments to represent the shielded values that are transferred. Because these commitments are additively homomorphic, it is possible to check that all Spend transfers and Output transfers balance; see §4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38 for detail.

This reduces the granularity of the circuit, allowing a substantial performance improvement (orthogonal to other Sapling circuit improvements) when the numbers of shielded inputs and outputs are significantly different. This comes at the cost of revealing the exact number of shielded inputs and outputs, but dummy (zero-valued) outputs are still possible.

### 8.4 Faerie Gold attack and fix

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

An adversary can use this to mislead a note recipient, by sending two notes both of which are verified as valid by Receive (as defined in [BCGGM2014, Figure 2]), but only one of which can be spent.

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

This attack does not violate the security definitions given in [BCGGM2014]. The issue could be framed as a problem either with the definition of Completeness, or the definition of Balance:

- The Completeness property asserts that a validly received note can be spent provided that its nullifier does not appear on the ledger. This does not take into account the possibility that distinct notes, which are validly received, could have the same nullifier. That is, the security definition depends on a protocol detail – nullifiers – that is not part of the intended abstract security property, and that could be implemented incorrectly.

- The Balance property only asserts that an adversary cannot obtain more funds than they have minted or received via payments. It does not prevent an adversary from causing others’ funds to decrease. In a Faerie Gold attack, an adversary can cause spending of a note to reduce (to zero) the effective value of another note for which the adversary does not know the spending key, which violates an intuitive conception of global balance.

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

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

[Sprout] Instead, Zcash enforces that an adversary must choose distinct values for each $\rho$, by making use of the fact that all of the nullifiers in JoinSplit descriptions that appear in a valid block chain must be distinct. This is true
regardless of whether the nullifiers corresponded to real or dummy notes (see §4.7.1 ‘Dummy Notes (Sprout)’ on p. 34). The nullifiers are used as input to hSigCRH to derive a public value hSig which uniquely identifies the transaction, as described in §4.3.2 ‘JoinSplit Descriptions’ on p. 30. (hSig 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 hSig using PRFₜₚ. The correct construction of ρ for each output note is enforced by §4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42 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 hSigCRH and PRFₜₚ 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 PRFₜₚ(ϕ) (or PRFₜₚ(pk) for Sapling), this is only possible if the adversary finds a collision across both inputs on PRFₜₚ (or PRFₜₚ(pk)), which is assumed to be infeasible — see §4.12 ‘Pseudo Random Functions’ on p. 19.

**[Sprout]** Crucially, “nullifier integrity” is enforced whether or not the enforceMerklePathₜ flag is set for an input note (§4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42). 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, ρ = cm + [pos] J, where J is a generator independent of the generators used in NoteCommitₜₚ. 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 COMMₜₑₚ and COMMₜₑ is a computationally binding commitment to its inputs apk, v, and ρ. However, the instantiation of COMMₜₑₚ and COMMₜₑ 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 apk 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 2₆₄, to find distinct pairs (apk, ρ) and (apk’, ρ’) 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 themselves [HW2016].

Zcash uses a simpler construction with a single hash evaluation for the commitment: SHA-256 for Sprout, and PedersenHash for Sapling. The motivation for the nested construction in Zerocash was to allow Mint transactions to be publicly verified without requiring a zk-SNARK proof ([BCGGMVT2014, section 1.3, under step 3]). Since Zcash combines “Mint” and “Spend” transactions into generalized JoinSplit transfers (for Sprout), or Spend transfers and Output transfers (for Sapling), and each transfer always uses a zk-SNARK proof, Zcash does not require the
These notes PRFs. The format of inputs to the 256 bits are used for the PRF key to 252 bits. This is preferable to requiring reasoning about truncation, and 252 bits is quite sufficient for security of these cryptovalues.

[Sprout] Note: Sprout note commitments are not statistically hiding. So for Sprout notes, Zcash does not support the “everlasting anonymity” property described in [BCGGMTV2014, section 8.1], even when used as described in that section. While it is possible to define a statistically hiding, computationally binding commitment scheme for this use at a 128-bit security level, the overhead of doing so within the JoinSplit statement was not considered to justify the benefits.

[Sapling onward] In Sapling, Pedersen commitments are used instead of SHA256Compress. These commitments are statistically hiding, and so “everlasting anonymity” is supported for Sapling notes under the same conditions as in Zerocash (by the protocol, not necessarily by zcasha). Note that diversified payment addresses can be linked if the Discrete Logarithm Problem on the Jubjub curve can be broken.

8.6 Changes to PRF inputs and truncation

The format of inputs to the PRFs instantiated in § 5.4.2 ‘Pseudo Random Functions’ on p. 58 has changed relative to Zerocash. There is also a requirement for another PRF, PRF′, which must be domain-separated from the others.

In the Zerocash protocol, ρ_i^{old} is truncated from 256 to 254 bits in the input to PRF_{sn} (which corresponds to PRF_{nf} in Zcash). Also, h_{Sig} is truncated from 256 to 253 bits in the input to PRF_{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 H and \mathcal{D}_2, it is stated that in \mathcal{D}_2, “for each i \in \{1,2\}, sn_i := \text{PRF}_{a_{sk}}(\rho) for a random (and not previously used) \rho”. It is also argued that “the calls to \text{PRF}_{a_{pk}} are each by definition unique”. The latter assertion depends on the fact that \rho is “not previously used”. However, the argument is incorrect because the truncated input to \text{PRF}_{a_{pk}}, i.e. |\rho|_{254}, may repeat even if \rho does not.

- In the same argument, it is stated that “with overwhelming probability, h_{Sig} is unique”. In fact what is required to be unique is the truncated input to PRF_{pk}, i.e. |h_{Sig}|_{253} = \text{CRH}(pk_{sig})_{253}. In practice this value will be unique under a plausible assumption on CRH provided that pk_{sig} is chosen randomly, but no formal argument for this is presented.

Note that \rho is truncated in the input to PRF_{sn} but not in the input to COMM_{rcm}, which further complicates the analysis.

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

[Sprout] For resistance to Faerie Gold attacks as described in § 8.4 ‘Faerie Gold attack and fix’ on p. 95, Zcash depends on collision resistance of h_{SigCRH} and PRF′ (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 h_{Sig} to the uses of \rho.

[Sprout] Since the PRFs are instantiated using SHA256Compress which has an input block size of 512 bits (of which 256 bits are used for the PRF input and 4 bits are used for domain separation), it was necessary to reduce the size of the PRF key to 252 bits. The key is set to a_{pk} in the case of PRF_{addr}, PRF_{nf}, and PRF_{pk}, and to \varphi (which does not exist in Zerocash) for PRF′, 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.

97


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 [MAEA2010].

- Although the **Zerocash** paper states that ECIES satisfies key privacy (as defined in [BBDP2001]), it is not clear that this holds for all curve parameters and key distributions. For example, if a group of non-prime order is used, the distribution of ciphertexts could be distinguishable depending on the order of the points representing the ephemeral and recipient public keys. Public key validity is also a concern. Curve25519 (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** both has the ephemeral and recipient public key encodings—which are unambiguous for Curve25519—and also $h_{Sig}$ and a nonce as described below, as input to the KDF. For **Sapling**, it is only possible to include the ephemeral public key encoding, but this is sufficient to retain the original security properties of DHAES. Note that being able to break the Elliptic Curve Diffie-Hellman Problem on Curve25519 or Jubjub (without breaking AEAD_CHACHA20_POLY1305 as an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the transmitted note(s) ciphertext unless $pk_{enc}$ is known or guessed.

- **Sprout** The KDF also takes a public seed $h_{Sig}$ as input. This can be modeled as using a different “randomness extractor” for each JoinSplit transfer, which limits degradation of security with the number of JoinSplit transfers. This facilitates security analysis as explained in [DGKM2011] — see section 7 of that paper for a security proof that can be applied to this construction under the assumption that single-block BLAKE2b-256 is a “weak PRF”. Note that $h_{Sig}$ is authenticated, by the zk-SNARK proof, as having been chosen with knowledge of $a_{sid,1..N_{out}}$, 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 $h_{Sig}$, but the binding signature and spend authorization signatures prevent such modifications.)

- **Sprout** The scheme used by **Sprout** includes an optimization that reuses the same ephemeral key (with different nonces) for the two ciphertexts encrypted in each JoinSplit description.
The security proofs of [ABR1999] can be adapted straightforwardly to the resulting scheme. Although DHAES as defined in that paper does not pass the recipient public key or a public seed to the hash function $H$, this does not impair the proof because we can consider $H$ to be the specialization of our KDF to a given recipient key and seed. (Passing the recipient public key to the KDF could in principle compromise key privacy, but not confidentiality of encryption.) [Sprout] It is necessary to adapt the “HDH independence” assumptions and the proof slightly to take into account that the ephemeral key is reused for two encryptions.

Note that the 256-bit key for AEAD_CHACHA20_POLY1305 maintains a high concrete security level even under attacks using parallel hardware [Bernstein2005] in the multi-user setting [Zaverucha2012]. This is especially necessary because the privacy of Zcash transactions may need to be maintained far into the future, and upgrading the encryption algorithm would not prevent a future adversary from attempting to decrypt ciphertexts encrypted before the upgrade. Other cryptovalues that could be attacked to break the privacy of transactions are also sufficiently long to resist parallel brute force in the multi-user setting: for Sprout, $a_{sk}$ is 252 bits, and $sk_{enc}$ is no shorter than $a_{sk}$.

### 8.8 Omission in Zerocash security proof

The abstract Zerocash protocol requires $PRF^{addr}$ 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 $PRF^{addr}$ such that $a_{sk}^1$ and $a_{sk}^2$ are distinct spending keys for the same $a_{pk}$. Because the note commitment is to $a_{sk}$, but the nullifier is computed from $a_{sk}$ (and $p$), the adversary is able to double-spend the note, once with each $a_{sk}$. This is not detected because each Spend reveals a different nullifier. The JoinSplit statements are still valid because they can only check that the $a_{sk}$ in the witness is some preimage of the $a_{pk}$ used in the note commitment.

The error is in the proof of Balance in [BCGGMTV2014, Appendix D.3]. For the “$A$ violates Condition I” case, the proof says:

“(i) If $cm_{1}^{old} = cm_{2}^{old}$, then the fact that $sn_{1}^{old} \neq sn_{2}^{old}$ implies that the witness $a$ contains two distinct openings of $cm_{1}^{old}$ (the first opening contains $(a_{sk,1}, p_{1}^{old})$, while the second opening contains $(a_{sk,2}, p_{2}^{old})$). This violates the binding property of the commitment scheme COMM.”

In fact the openings do not contain $a_{sk,1}^{old}$; they contain $a_{pk,i}^{old}$ (in Sprout $cm_{i}^{old}$ opens directly to $(a_{pk,i}, v_{i}^{old}, p_{i}^{old})$, and in Zerocash it opens to $(v_{i}^{old}, COMM_{s}(a_{pk,i}, p_{i}^{old}))$.)

A similar error occurs in the argument for the “$A$ violates Condition II” case.

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

The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of $PRF^{addr}$ (on both its arguments) to argue that distinctness of $a_{sk,1}^{old}$ and $a_{sk,2}^{old}$, together with constraint 1(b) of the JoinSplit statement (see §4.15.1 ‘JoinSplit Statement (Sprout)’ on p.42), implies distinctness of $a_{pk,1}^{old}$ and $a_{pk,2}^{old}$; therefore distinct openings of the note commitment when Condition I or II is violated.

### 8.9 Miscellaneous

- The paper defines a note as $\langle(a_{pk}, pk_{enc}, v, p, rcm, s, cm)\rangle$, whereas this specification defines a Sprout note as $(a_{pk}, v, p, rcm)$. The instantiation of $COMM_{s}$ in section 5.1 of the paper did not actually use $s$, and neither does the new instantiation of NoteCommit in Sprout. $pk_{enc}$ is also not needed as part of a note: it is not an input to NoteCommit, nor is it constrained by the Zerocash POUR statement or the Zcash JoinSplit statement. $cm$ can be computed from the other fields. (The definition of notes for Sapling is different again.)

- The length of proof encodings given in the paper is 288 bytes. [Sprout] This differs from the 296 bytes specified in §5.4.9.1 ‘BCTV14’ on p.72, 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 \texttt{libsnark} used by \texttt{Zcash} uses this standard encoding rather than the less efficient (uncompressed) one used by upstream \texttt{libsnark}. In \texttt{Sapling}, a customized encoding is used for BLS12-381 points in Groth16 proofs to minimize length.

- The range of monetary values differs. In \texttt{Zcash} this range is \( \{0..\text{MAX\_MONEY}\} \), while in \texttt{Zerocash} it is \( \{0..2^{\ell\text{\_value}} - 1\} \). (The \texttt{JoinSplit} statement still only directly enforces that the sum of amounts in a given \texttt{JoinSplit transfer} is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

# Acknowledgements

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

The designers of the \texttt{Zcash} protocol are the \texttt{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 \texttt{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 \texttt{Zerocash} and \texttt{Zcash} protocol designs; in addition to the preceding, this includes Mike Perry, isis agora lovecruft, Leif Ryge, Andrew Miller, Ben Blaxill, Samantha Hulsey, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, John Tromp, Paige Peterson, Jack Gavigan, j1777, Alison Stevenson, Maureen Walsh, Filippo Valsorda, Zaki Manian, Tracy Hu, Brian Warner, Mary Maller, Michael Dixon, Andrew Poelstra, Eirik Ogilvie-Wigley, Benjamin Winston, Kobi Gurkan, Weikeng Chen, Henry De Valence, Deirdre Connolly, Chelsea Komlo, Zancas Wilcox, and no doubt others. We would also like to thank the designers and developers of \texttt{Bitcoin}.

\texttt{Zcash} has benefited from security audits performed by NCC Group, Coinspect, Least Authority, Mary Maller, Kudelski Security, QED-it, and Trail of Bits.

The Faerie Gold attack was found by Zooko Wilcox; subsequent analysis of variations on the attack was performed by Daira Hopwood and Sean Bowe. The internal hash collision attack was found by Taylor Hornby. The error in the \texttt{Zerocash} proof of Balance relating to \texttt{collision resistance} of \texttt{PRF\_addr} was found by Daira Hopwood. The errors in the proof of \texttt{Ledger Indistinguishability} mentioned in §8.6 ‘\texttt{Changes to PRF inputs and truncation}’ on p. 97 were also found by Daira Hopwood.

The 2015 Soundness vulnerability in BCTV14 [Parno2015] was found by Bryan Parno. An additional condition needed to resist this attack was documented by Ariel Gabizon [Gabizon2019, section 3]. The 2019 Soundness vulnerability in BCTV14 [Gabizon2019] was found by Ariel Gabizon.

The design of \texttt{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.

The observation in §5.4.1.6 ‘\texttt{DiversifyHash Hash Function}’ on p. 55 that diversified payment address unlinkability can be proven in the same way as key privacy for ElGamal, is due to Mary Maller.

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 \texttt{Zcash} —including the use of zero-knowledge proofs to resolve the tension between privacy and auditability, Merkle trees over note commitments (using Pedersen hashes as in \texttt{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 \texttt{Zcash} 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].
10 Change History

2020.1.5  2020-06-02
  • Reference [ZIP-173] instead of BIP 173.
  • Mark more index entries as definitions.

2020.1.4  2020-05-27
  • Reference [BIP-32] and [ZIP-32] when describing keys and their encodings.
  • Network Upgrade 4 has been given the name Canopy.
  • Improve LaTeX portability of this specification.

2020.1.3  2020-04-22
  • Correct a wording error transposing transparent inputs and transparent outputs in §4.11 ‘Balance (Sprout)’ on p. 37.
  • Minor wording clarifications.

2020.1.2  2020-03-20
  • The implementation of Sprout Ed25519 signature verification in zcashd differed from what was specified in §5.4.5 ‘JoinSplit Signature’ on p. 61. The specification has been changed to match the implementation.
  • Add consensus rules for Heartwood.
  • Remove “pvc” Makefile targets.
  • Make the Heartwood specification the default.
  • Add macros and Makefile support for building the Canopy specification.

2020.1.1  2020-02-13
  • Resolve conflicts in the specification of memo fields by deferring to [ZIP-302].

2020.1.0  2020-02-06
  • Specify a retrospective soft fork implemented in zcashd v2.1.1-1 that limits the nTime field of a block relative to its median-time-past.
  • Correct the definition of median-time-past for the first PoWMedianBlockSpan blocks in a block chain.
  • Add acknowledgements to Henry de Valence, Deirdre Connolly, Chelsea Komlo, and Zancas Wilcox.
  • Add an acknowledgement to Trail of Bits for their security audit.
  • Change indices in the incremental Merkle tree diagram to be zero-based.
  • Use the term ‘monomorphism’ for an injective homomorphism, in the context of a signature scheme with key monomorphism.
2019.0.9  2019-12-27

- No changes to **Sprout** or **Sapling**.
- Specify the height at which **Blossom** activated.
- Add **Blossom** to §6 ‘**Network Upgrades**’ on p.79.
- Add a non-normative note giving the explicit value of FoundersRewardLastBlockHeight.
- Clarify the effect of **Blossom** on **SIGHASH transaction hashes**.
- **Makefile** updates for **Heartwood**.

2019.0.8  2019-09-24

- No changes to **Sprout**.
- Fix a typo in the generator $P_{S_1}$ in §5.4.8.2 ‘**BLS12-381**’ on p.68 found by magrady.
- Clarify the type of $v^{\text{new}}$ in §4.6.2 ‘**Sending Notes (Sapling)**’ on p.33.

2019.0.7  2019-09-24

- Fix a discrepancy in the number of constraints for **BLAKE2s** found by QED-it.
- Fix an error in the expression for $\Delta$ in §A.3.3.9 ‘**Pedersen hash**’ on p.138, and add acknowledgement to Kobi Gurkan.
- Fix a typo in §4.8 ‘**Merkle Path Validity**’ on p.35 and add acknowledgement to Weikeng Chen.
- Update references to ZIPs and to the Electric Coin Company blog.
- **Makefile** improvements to suppress unneeded output.

2019.0.6  2019-08-23

- No changes to **Sprout** or **Sapling**.
- Replace dummy **Blossom** activation height with the testnet height, and a reference to ZIP 206.

2019.0.5  2019-08-23

- Note the change to the minimum-difficulty threshold time on the test network for **Blossom**.
- Correct the packing of $nf_{\text{old}}$ into input elements in §A.4 ‘**The Sapling Spend circuit**’ on p.145.
- Add an epigraph from [Carroll1876] to the start of §5.4.8.3 ‘Jubjub’ on p.69.
- Clarify how the constant $c$ in §5.4.1.7 ‘**Pedersen Hash Function**’ on p.56 is obtained.
- Add a footnote that zcash uses [ZIP-32] **extended spending keys** instead of the derivation from $sk$ in §3.1 ‘**Payment Addresses and Keys**’ on p.12.
- Remove “optimized” **Makefile** targets (which actually produced a larger PDF, with TeXLive 2019).
- Remove “html” **Makefile** targets.
- Make the **Blossom** spec the default.

2019.0.4  2019-07-23

- Clicking on a section heading now shows section labels.
- Add a **List of Theorems and Lemmata**.
- Changes needed to support TeXLive 2019.
2019.0.3 2019-07-08

• Experimental support for building using LuaTEX and XeTEX.
• Add an Index.

2019.0.2 2019-06-18

• Correct a misstatement in the security argument in §4.12 ‘Balance and Binding Signature (Sapling)’ on p. 38: binding for a commitment scheme does not imply that the commitment determines its randomness. The rest of the security argument did not depend on this; it is simpler to rely on knowledge soundness of the Spend and Output proofs.
• Give a definition for complete twisted Edwards elliptic curves in §5.4.8.3 ‘Jubjub’ on p. 69.
• Clarify that Theorem 5.4.2 on p. 57 depends on the parameters of the Jubjub curve.
• Ensure that this document builds correctly and without missing characters on recent versions of TEXLive.
• Update the Makefile to use Ghostscript for PDF optimization.
• Ensure that hyperlinks are preserved, and available as “Destination names” in URL fragments and links from other PDF documents.

2019.0.1 2019-05-20

• No changes to Sprout or Sapling.
• Minor fix to the list of integer constants in §2 ‘Notation’ on p. 9.
• Use IsBlossomActivated in the definition of FounderAddressAdjustedHeight for consistency.

2019.0.0 2019-05-01

• Fix a specification error in the Founders’ Reward calculation during the slow start period.
• Correct an inconsistency in difficulty adjustment between the spec and zcashd implementation for the first PoWAveragingWindow − 1 blocks of the block chain. This inconsistency was pointed out by NCC Group in their Blossom specification audit.
• Revert changes for funding streams from Withdrawn ZIP 207.


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

2019.0-beta-38 2019-04-18

• Update the following sections to match the current draft of ZIP-208:
  – §7.6.3 ‘Difficulty adjustment’ on p. 89
  – §7.7 ‘Calculation of Block Subsidy and Founders’ Reward’ on p. 91
• Specify funding streams, along with the draft funding streams defined in the current draft of ZIP 207.
• Update the following sections to match the current draft of ZIP 207:
  – §3.9 ‘Block Subsidy and Founders’ Reward’ on p. 18
  – §3.10 ‘Coinbase Transactions’ on p. 18
  – §7.7 ‘Calculation of Block Subsidy and Founders’ Reward’ on p. 91
- §7.8 ‘Payment of Founders’ Reward’ on p. 92

- Correct the generators $P_{S_1}$ and $P_{S_2}$ for BLS12-381.
- Update README.rst to include Makefile targets for Blossom.
- Makefile updates:
  - Fix a typo for the pvcblossom target.
  - Update the pinned git hashes for sam2p and pdfsizeopt.

2019.0-beta-37 2019–02–22

- The rule that miners **SHOULD NOT** mine blocks that chain to other blocks with a block version number greater than 4, has been removed. This is because such blocks (mined nonconformantly) exist in the current consensus chain on the production Zcash network.
- Clarify that Equihash is based on a variation of the Generalized Birthday Problem, and cite [AR2017].
- Update reference [BGG2017] (previously [BGG2016]).
- Clarify which transaction fields are added by Overwinter and Sapling.
- Correct the rule about when a transaction is permitted to have no transparent inputs.
- Explain the differences between the system in [Groth2016] and what we refer to as Groth16.
- Reference Mary Maller’s security proof for Groth16 [Maller2018].
- Correct [BGM2018] to [BGM2017].
- Fix a typo in § B.2 ‘Groth16 batch verification’ on p. 149 and clarify the costs of Groth16 batch verification.
- Add macros and Makefile support for building the Blossom specification.

2019.0-beta-36 2019–02–09

- Correct isis agora lovecruf’ts name.

2019.0-beta-35 2019–02–08

- Cite [Gabizon2019] and acknowledge Ariel Gabizon.
- Correct [SBB2019] to [SWB2019].
- The [Gabizon2019] vulnerability affected Soundness of BCTV14 as well as Knowledge Soundness.
- Specify the difficulty adjustment change that occurred on the test network at block height 299188.
- Add Eirik Ogilvie-Wigley and Benjamin Winston to acknowledgements.
- Rename zk-SNARK Parameters sections to be named according to the proving system (BCTV14 or Groth16), not the shielded protocol construction (Sprout or Sapling).
- In §6 ‘Network Upgrades’ on p. 79, say when Sapling activated.

2019.0-beta-34 2019–02–05

- Disclose a security vulnerability in BCTV14 that affected Sprout before activation of the Sapling network upgrade (see §5.4.9.1 ‘BCTV14’ on p. 72).
- Rename PHGR13 to BCTV2014.
- Rename reference [BCTV2015] to [BCTV2014a], and [BCTV2014] to [BCTV2014b].
2018.0-beta-33  2018-11-14

- No changes to Sprout.
- Add § A.5 ‘The Sapling Output circuit’ on p. 147.
- Change the description of window lookup in § A.3.3.7 ‘Fixed-base Affine-ctEdwards scalar multiplication’ on p. 136 to match sapling-crypto.
- Describe 2-bit window lookup with conditional negation in § A.3.3.9 ‘Pedersen hash’ on p. 138.
- 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 $H^*$ used to derive the nonce $r$ in RedDSA.Sign, from $T \| M$ to $T \| \vk \| M$. 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. 141.

2018.0-beta-31  2018-09-30

- No changes to Sprout.
- Correct some uses of $r_j$ that should have been $r_2$ or $q$.
- Correct uses of LEOS2IP$_\ell$ in RedDSA.Verify and RedDSA.BatchVerify to ensure that $\ell$ is a multiple of 8 as required.
- Minor changes to avoid clashing notation for Edwards curves $E_{Edwards(a,d)}$, Montgomery curves $E_{Mont(A,B)}$, and extractors $E_A$.
- Correct a use of $J$ that should have been $M$ in the proof of Theorem A.3.4 on p. 134, and make a minor tweak to the theorem statement ($k_2 \neq \pm k_1$ instead of $k_1 \neq \pm k_2$) to make the contradiction derived by the proof clearer.
- Clarify notation in the proof of Theorem A.3.3 on p. 134.
- Address some of the findings of the QED-it report:
  - Improved cross-referencing in § 5.4.1.7 ‘Pedersen Hash Function’ on p. 56.
  - Clarify the notes concerning domain separation of prefixes in § 5.4.1.3 ‘MerkleCRH$_{Sapling}$ Hash Function’ on p. 54 and § 5.4.7.2 ‘Windowed Pedersen commitments’ on p. 65.
- Correct the statement and proof of Theorem A.3.2 on p. 134.
- 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 [BGM2017] for the Sapling zk-SNARK parameters.
- Add a reference to the ristretto_bulletproofs design notes [Dalek-notes] for the synthetic blinding factor technique.
Ensure that the constraint costs in § A.3.3.1 ‘Checking that Affine-ctEdwards coordinates are on the curve’ on p. 133 and § A.3.3.6 ‘Affine-ctEdwards nonsmall-order check’ on p. 136 accurately reflect the implementation in sapling-crypto.

Minor correction to the non-normative note in § A.3.2.2 ‘Range check’ on p. 131.


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 § A.3.13 ‘Spend Authorization Signature’ on p. 40 may securely be delegated.

Add a non-normative note to § A.3.3.8 ‘Variable-base Affine-ctEdwards scalar multiplication’ on p. 137.

Clarify that when validating a Groth16 proof, it is necessary to perform a subgroup check for πA and πC, as well as for πB.

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-29 2018-08-15

- No changes to Sprout.
- Finish § A.3.2.2 ‘Range check’ on p. 131.
- Change § A.3.7 ‘BLAKE2s hashes’ on p. 142 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. 142.
- Minor corrections to § A.3.3.8 ‘Variable-base Affine-ctEdwards scalar multiplication’ on p. 137.

2018.0-beta-27 2018-08-12

- Notational changes:
  - Use a superscript \((r)\) to mark the subgroup order, instead of a subscript.
  - Use \(G_1^{(r)}\) for the set of \(r\)-order points in \(G\).
  - Mark the subgroup order in pairing groups, e.g. use \(G_1^{(r)}\) instead of \(G_1\).
  - Make the bit-representation indicator \(\star\) an affix instead of a superscript.

- Clarify that when validating a Groth16 proof, it is necessary to perform a subgroup check for \(\pi_A\) and \(\pi_C\), as well as for \(\pi_B\).
- Correct the description of Groth16 batch verification to explicitly take account of how verification depends on primary inputs.
- Add Charles Rackoff, Rafail Ostrovsky, and Amit Sahai to the acknowledgements section for their work on zero-knowledge proofs.

2018.0-beta-26 2018-08-05

- No changes to Sprout.
- Add § B.2 ‘Groth16 batch verification’ on p. 149.
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.148.

2018.0-beta-22  2018-07-18

- No changes to Sprout.
- Update § 6 'Network Upgrades’ on p.79 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.139.
- Add a note about redundancy in the nonsmall-order checking of rk.
- Clarify the use of $cv^{new}$ and $cm^{new}$, and the selection of outgoing viewing key, in sending Sapling notes.
- Delete the description of optimizations for the affine twisted Edwards nonsmall-order check, since the Sapling circuit does not use them. Also clarify that some other optimizations are not used.

2018.0-beta-21  2018-06-22

- Remove the consensus rule "If $n_{JoinSplit} > 0$, the transaction MUST NOT use SIGHASH types other than SIGHASH_ALL," which was never implemented.
- Add section on signature hashing.
- Briefly describe the changes to computation of SIGHASH transaction hashes in Sprout.
- Clarify that interstitial treestates form a tree for each transaction containing JoinSplit descriptions.
- Correct the description of P2PKH addresses in §5.6.1 ‘Transparent Addresses’ on p.75 — they use a hash of a compressed, not an uncompressed ECDSA key representation.
- Clarify the wording of the caveat about the claimed security of shielded transactions.
- Correct the definition of set difference ($S \setminus T$).
- Add a note concerning malleability of zk-SNARK proofs.
- Clarify attribution of the Zcash protocol design.
- Acknowledge Alex Biryukov and Dmitry Khovratovich as the designers of Equihash.
• Acknowledge Shafi Goldwasser, Silvio Micali, Oded Goldreich, Rosario Gennaro, Bryan Parno, Jon Howell, Craig Gentry, Mariana Raykova, and Jens Groth for their work on zero-knowledge proving systems.

• Acknowledge Tomas Sander and Amnon Ta-Shma for [ST1999].

• Acknowledge Kudelski Security’s audit.

• Use the more precise subgroup types $G^r$ and $J^r$ in preference to $G$ and $J$ where applicable.

• Change the types of auxiliary inputs to the Spend statement and Output statement, to be more faithful to the implementation.

• Rename the cm field of an Output description to $cmu$, reflecting the fact that it is a Jubjub curve $u$-coordinate.

• Add explicit consensus rules that the anchor field of a Spend description and the $cmu$ field 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 $\{-\text{MAX\_MONEY}, \ldots, \text{MAX\_MONEY}\}$.

• Enforce stronger constraints on the types of key components $pk_d, ak$, and $nk$.

• Correct the conformance rule for fOverwintered (it must not be set before Overwinter has activated, not before Sapling has activated).

• Correct 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 GroupHash$^r$, FindGroupHash$^r$, Extract$^r$, PRFexpand, PRFock, and CRHivk.

• Instantiate PRFock 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-specific types where appropriate.

• Improve cross-referencing.

• Clarify the use of BCTV14 vs Groth16 proofs in JoinSplit statements.

• Clarify that the $\sqrt{a}$ notation refers to the positive square root. (This matters for the conversion in §A.3.3.3 ‘ctEdwards ↔ Montgomery conversion’ on p. 133.)

• 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 GroupHash$^r$.

• 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 PRFexpand 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. 128 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. 38.
• Correct a type error in § 5.4.8.5 ‘Group Hash into Jubjub’ on p. 71.
• Correct a type error in RedDSA.Sign in § 5.4.6 ‘RedDSA and RedJubjub’ on p. 62.
• Ensure $G$ is defined in § 5.4.6.1 ‘Spend Authorization Signature’ on p. 64.
• Make the public key prefix part of the input to the hash function in RedDSA, not part of the message.
• Correct the statement about FindGroupHash$^{(r)}_J$ never returning ⊥.
• Correct an error in the computation of generators for Pedersen hashes.
• Change the order in which NoteCommit$^{Sapling}$ commits to its inputs, to match the sapling-crypto implementation.
• Fail Sapling key generation if ivk = 0. (This has negligible probability.)
• Change the notation $H^*$ to $H^\oplus$ in § 5.4.6 ‘RedDSA and RedJubjub’ on p. 62, to avoid confusion with the $^*$ 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 $\{0..2^{\text{value}}-1\}$, the security argument depends on a sum of commitments being binding on $\{-\frac{r_{i-1}}{2}..\frac{r_i-1}{2}\}$.
• Fix the loss of tightness in the use of $\text{PRF}_{\text{nfSapling}}$ by specifying the keyspace more precisely.
• Correct type ambiguities for $\rho$.
• Specify the representation of $i$ in group $G_2$ 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 \( R \) in an Ed25519 signature is checked to not be of small order; this is not the same as checking that it is of prime order \( \ell \).
- Specify support for [BIP-111] (the NODE_BLOOM service bit) in network protocol version 170004.
- Give references [Vercauter2009] and [AKLGL2010] for the optimal ate pairing.
- Define \( K^\text{Sprout}_A \). DerivePublic for Curve25519.
- Caveat the claim about *note traceability set* in §1.2 ‘High-level Overview’ on p. 8 and link to [Peterson2017] and [Quesnelle2017].
- Do not require a generator as part of the specification of a *represented group*; instead, define it in the *represented pairing* or scheme using the group.
- Refactor the abstract definition of a *signature scheme* to allow derivation of verifying keys independent of key pair generation.
- Correct the explanation in §1.2 ‘High-level Overview’ on p. 8 to apply to Sprout.
- Add the definition of a *private key* to *public key* homomorphism for *signature schemes*.
- Remove the output index as an input to \( K^\text{Sprout}_A \).
- Allow dummy Sprout input *notes*.
- Specify RedDSA and RedJubjub.
- Specify *binding signatures* and *spend authorization signatures*.
- Specify the randomness beacon.
- Add *Output ciphertexts* and ock.
- Define DefaultDiversifier.
- Change the *Spend circuit* and *Output circuit* specifications to remove unintended differences from sapling-crypto.
- Use \( h_3 \) 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. 134 (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 MerkleDepth\textsuperscript{Sapling} from 29 to 32.
• Updates to Sapling construction, changing how the nullifier is computed and separating it from the randomized Spend verifying key (rk).
• Clarify conversions between bit and byte sequences for sk, \(\text{repr}_J(ak)\) and \(\text{repr}_J(nk)\).
• 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 \(\text{FindGroupHash}^{(r)}\) 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 (\text{ask}, nsk) are derived from the spending key sk to ensure they are on the full range of \(\mathbb{F}_{r_J}\).
• Change PRF\textsuperscript{nr} to produce output computationally indistinguishable from uniform on \(\mathbb{F}_{r_J}\).
• Change Uncommitted\textsuperscript{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 PRF\textsuperscript{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. 50.
- Refer to Overwinter and Sapling just as “upgrades” in the abstract, not as the next “minor version” and “major version”.
- PRF\`w 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. 134) 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 \( \text{Extract}_{j(r)} \).
- Specify GroupHash\(^{(r)}\).  
- Specify Pedersen hashes.

2018.0-beta-8  2018-02-08

- No changes to Sprout.
- Add instantiation of CRH\(^{ivk}\).
- 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 zcashd was as intended.)
- Specify that 0xF6 followed by 511 zero bytes encodes an empty memo field.
- Reference security definitions for Pseudo Random Functions and Pseudo Random Generators.
- Rename clamp to bound and ActualTimespanClamped to ActualTimespanBounded in the difficulty adjustment algorithm, to avoid a name collision with Curve25519 scalar “clamping”.
- Change uses of the term full node to full validator. A full node by definition participates in the peer-to-peer network, whereas a full validator just needs a copy of the block chain from somewhere. The latter is what was meant.
- Add an explanation of how Sapling prevents Faerie Gold and roadblock attacks.
- Sapling work in progress.

2018.0-beta-6  2018-01-31

- No changes to Sprout.
- Sapling work in progress, mainly on Appendix §A ‘Circuit Design’ on p. 128.

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

2017.0-beta-2.9  2017-12-17

- Refer to 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 πB.
- Initial version of draft Sapling protocol specification.
2017.0-beta-2.7 2017-07-10

• Fix an off-by-one error in the specification of the Equihash algorithm binding condition. (The implementation in zcashd was as intended.)
• Correct the types and consensus rules for transaction version numbers and block version numbers. (Again, the implementation in zcashd was as intended.)
• Clarify the computation of $h_i$ 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. 99.
• Correct the order of arguments to COMM in §5.4.7.1 ‘Sprout Note Commitments’ on p. 65.
• Correct a statement about indistinguishability of JoinSplit descriptions.
• Change the Founders’ Reward addresses, for the test network only, to reflect the hard-fork upgrade described in [Zcash-Issue2113].

2017.0-beta-2.4 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 $(apk, \rho)$ rather than just on $\rho$.
• Rename enforce_i to enforceMerklePath_i.

2017.0-beta-2.3 2017-02-12

• Specify the security requirements on the $SHA256Compress$ function in order for the scheme in §5.4.71 ‘Sprout Note Commitments’ on p. 65 to be a secure commitment.
• Specify $G_2$ more precisely.
• Explain the use of interstitital treestates in chained JoinSplit transfers.

2017.0-beta-2.2 2017-02-11

• Give definitions of computational binding and computational hiding for commitment schemes.
• Give a definition of statistical zero knowledge.
• Reference the white paper on MPC parameter generation [BGG2017].

2017.0-beta-2.1 2017-02-06

• $\ell_{Merkle}$ is a bit length, not a byte length.
• Specify the maximum block size.
2017.0-beta-2 2017-02-04

- Add abstract and keywords.
- Fix a typo in the definition of nullifier integrity.
- Make the description of block chains more consistent with upstream Bitcoin documentation (referring to “best” chains rather than using the concept of a block chain view).
- Define how nodes select a best valid block chain.

2016.0-beta-1.13 2017-01-20

- Specify the difficulty adjustment algorithm.
- Clarify some definitions of fields in a block header.
- Define PRFaddr in § 4.2.1 ‘Sprout Key Components’ on p. 28.

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 zk-SNARK proof.
- Note that due to an oversight, the Zcash genesis block does not follow [BIP-34].

2016.0-beta-1.10 2016-10-30

- Update reference to the Equihash paper [BK2016]. (The newer version has no algorithmic changes, but the section discussing potential ASIC implementations is substantially expanded.)
- Clarify the discussion of proof size in “Differences from the Zerocash paper”.

2016.0-beta-1.9 2016-10-28

- Add Founders’ Reward addresses for 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 Zcash.
- Specify that OP_CODESEPARATOR has been disabled, and no longer affects SIGHASH transaction hashes.
- Change the representation type of vpub_old and vpub_new to uint64. (This is not a consensus change because the type of vpub_old and vpub_new was already specified to be {0..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 BLAKE2b-ℓ is different from truncated BLAKE2b-512.
- Clarify a note about SU-CMA security for signatures.
- Add a note about PRFnf corresponding to PRFsn in Zerocash.
- Add a paragraph about key length in §8.7 ‘In-band secret distribution’ on p. 98.
- 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[ℓ]” 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-ℓ.
- 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 fields to 512 bytes.
- Switch the JoinSplit signature scheme to Ed25519, with consequent changes to the computation of $h_{Sig}$.
- Fix the lead bytes in shielded payment address and spending key encodings to match the implemented protocol.
- Add a consensus rule about the ranges of $v_{old}^\text{pub}$ and $v_{new}^\text{pub}$.
- Clarify cryptographic security requirements and added definitions relating to the in-band secret distribution.
- Add various citations: the “Fixing Vulnerabilities in the Zcash Protocol” and “Why Equihash?” blog posts, several crypto papers for security definitions, the Bitcoin whitepaper, the CryptoNote whitepaper, and several references to Bitcoin documentation.
- Reference the extended version of the Zerocash paper rather than the Oakland proceedings version.
- Add JoinSplit transfers to the Concepts section.
- Add a section on Coinbase Transactions.
- Add acknowledgements for Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, and Jake Tarren.
- Fix a Makefile compatibility problem with the escaping behaviour of echo.
- Switch to biber for the bibliography generation, and add backreferences.
- Make the date format in references more consistent.
- Add visited dates to all URLs in references.
- Terminology changes.

2016.0-alpha-3.1  2016-05-20

- Change main font to Quattrocento.

2016.0-alpha-3  2016-05-09

- Change version numbering convention (no other changes).

2.0-alpha-3  2016-05-06

- Allow anchoring to any previous output treestate in the same transaction, rather than just the immediately preceding output treestate.
- Add change history.
2.0-alpha-2 2016-04-21

- Change from truncated BLAKE2b-512 to BLAKE2b-256.
- Clarify endianness, and that uses of BLAKE2b are unkeyed.
- Minor correction to what SIGHASH types cover.
- Add “as intended for the Zcash release of summer 2016” to title page.
- Require PRF_{addr} to be collision-resistant (see §8.8 ‘Omission in Zerocash security proof’ on p. 99).
- Add specification of path computation for the incremental Merkle tree.
- Add a note in §4.15.1 ‘JoinSplit Statement (Sprout)’ on p. 42 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


[DigiByte-PoW] DigiByte Core Developers. DigiSpeed 4.0.0 source code, functions GetNextWorkRequiredV3/4 in src/main.cpp as of commit 178e1348a67d9624db328062397fde0de03fe388 in src/main.cpp#L1587 (visited on 2017-01-20) (↑ p89).


# 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. 43 and § 4.15.3 ‘Output Statement (**Sapling**)’ on p. 44 respectively. It also adds a Groth16 circuit for the **JoinSplit** statement described in § 4.15.1 ‘JoinSplit Statement (**Sprout**)’ on p. 42.

At the next lower level, each circuit is defined in terms of a quadratic constraint program (specifying a Rank 1 Constraint System), as detailed in this section. In the BCTV14 or Groth16 proving systems, this program is translated to a Quadratic Arithmetic Program [BCTV2014a, section 2.3] [WCBTV2015]. The circuit descriptions given here are necessary to compute witness elements for each circuit, as well as the proving and verification keys.

Let $\mathbb{F}_r$ be the finite field over which Jubjub is defined, as given in § 5.4.8.3 ‘Jubjub’ on p. 69.

A quadratic constraint program consists of a set of constraints over variables in $\mathbb{F}_r$, each of the form:

$$(A) \cdot (B) = (C)$$

where $(A), (B)$, and $(C)$ are linear combinations of variables and constants in $\mathbb{F}_r$.

Here $\cdot$ and $\cdot$ both represent multiplication in the field $\mathbb{F}_r$, but we use $\cdot$ for multiplications corresponding to gates of the circuit, and $\cdot$ for multiplications by constants in the terms of a linear combination. $\cdot$ should not be confused with $\times$ which is defined as cartesian product in § 2 ‘Notation’ on p. 9.

### A.2 Elliptic curve background

The **Sapling** circuits make use of a complete twisted Edwards elliptic curve (“ctEdwards curve”) Jubjub, defined in § 5.4.8.3 ‘Jubjub’ on p. 69, and also a Montgomery elliptic curve $M$ that is birationally equivalent to Jubjub. Following the notation in [BL2017] we use $(u, v)$ for affine coordinates on the ctEdwards curve, and $(x, y)$ for affine coordinates on the Montgomery curve.

A point $P$ is normally represented by two $\mathbb{F}_r$ variables, which we name as $(P^u, P^v)$ for an affine-ctEdwards point, for instance.

The implementations of scalar multiplication require the scalar to be represented as a bit sequence. We therefore allow the notation $[k \star] P$ meaning $[\text{LEBS2IP}_{\text{length}(k \star)}(k \star)] P$. There will be no ambiguity because variables representing bit sequences are named with a $\star$ suffix.

The Montgomery curve $M$ has parameters $A_M = 40962$ and $B_M = 1$. We use an affine representation of this curve with the formula:

$$B_M \cdot y^2 = x^3 + A_M \cdot x^2 + x$$

Usually, elliptic curve arithmetic over prime fields is implemented using some form of projective coordinates, in order to reduce the number of expensive inversions required. In the circuit, it turns out that a division can be implemented at the same cost as a multiplication, i.e. one constraint. Therefore it is beneficial to use affine coordinates for both curves.

We define the following types representing affine-ctEdwards and affine-Montgomery coordinates respectively:

$$\text{AffineCtEdwardsJubjub} := (u : \mathbb{F}_r) \times (v : \mathbb{F}_r) : a_2 \cdot u^2 + v^2 = 1 + d_1 \cdot u^2 \cdot v^2$$

$$\text{AffineMontJubjub} := (x : \mathbb{F}_r) \times (y : \mathbb{F}_r) : B_M \cdot y^2 = x^3 + A_M \cdot x^2 + x$$
We also define a type representing compressed, *not necessarily valid*, CtEdwards coordinates:

\[
\text{CompressedCtEdwardsJubjub} := (\tilde{u} : B) \times (v : \mathbb{F}_p)
\]

See §5.4.8.3 ‘Jubjub’ on p. 69 for how this type is represented as a byte sequence in external encodings.

We use *affine-Montgomery* arithmetic in parts of the circuit because it is more efficient, in terms of the number of constraints, than *affine-cTEdwards* arithmetic.

An important consideration when using Montgomery arithmetic is that the addition formula is not complete, that is, there are cases where it produces the wrong answer. We must ensure that these cases do not arise.

We will need the theorem below about \(y\)-coordinates of points on *Montgomery curves*.

**Fact:** \(A_M^2 - 4\) is a nonsquare in \(\mathbb{F}_p\).

**Theorem A.2.1.** \((0,0)\) is the only point with \(y = 0\) on certain Montgomery curves.

Let \(P = (x, y)\) be a point other than \((0,0)\) on a Montgomery curve \(E_{\text{Mont}(A,B)}\) over \(\mathbb{F}_p\), such that \(A^2 - 4\) is a nonsquare in \(\mathbb{F}_p\). 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 \(\mathbb{F}_p\) 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 $\mathbb{F}_{r_S}$ have a multiplicative inverse, the assertion $a \neq 0$ can be implemented by witnessing the inverse, $a_{\text{inv}} = a^{-1} \pmod{r_S}$:

$$ (a_{\text{inv}}) \times (a) = (1) $$

This technique comes from [SVPBABW2012, Appendix D.1].

**Non-normative note:** A global optimization allows to use a single inverse computation outside the circuit for any number of nonzero constraints. Suppose that we have $n$ variables (or linear combinations) that are supposed to be nonzero: $a_0...n-1$. Multiply these together (using $n-1$ constraints) to give $a^* = \prod_{i=0}^{n-1} a_i$; then, constrain $a^*$ to be nonzero. This works because the product $a^*$ is nonzero if and only if all of $a_0...n-1$ are nonzero. However, the Sapling circuit does not use this optimization.

A.3.1.5  Exclusive-or constraints

An exclusive-or operation $a \oplus b = c$, where $a, b : \mathbb{B}$ 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 $r_S$

Let $n : \mathbb{N}^+$ be a constant. The operation of converting a field element, $a : \mathbb{F}_{r_S}$, to a sequence of boolean variables $b_0...n-1 : \mathbb{B}^n$ such that $a = \sum_{i=0}^{n-1} b_i \cdot 2^i \pmod{r_S}$, 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 $a \pmod{r_S} = \left( \sum_{i=0}^{n-1} b_i \cdot 2^i \right) \pmod{r_S} = \left( \sum_{i=0}^{n-1} b_i \cdot (2^i \pmod{r_S}) \right) \pmod{r_S}$. 

130
This can be implemented in one constraint:

\[
\left( \sum_{i=0}^{n-1} b_i \cdot (2^i \mod r_S) \right) \times (1) = (a)
\]

**Notes:**

- The bit length \( n \) is not limited by the field element size.
- Since the constraint has only a trivial multiplication, it is possible to eliminate it by merging it into the boolean constraint of one of the output bits, expressing that bit as a linear combination of the others and \( a \). However, this optimization requires substitutions that would interfere with the modularity of the circuit implementation (for a saving of only one constraint per unpacking operation), and so we do not use it for the Sapling circuit.
- In the case \( n = 255 \), for \( a < 2^{255} - r_S \) there are two possible representations of \( a \in F_{r_S} \) as a sequence of 255 bits, corresponding to \( \text{I2LEBSP}_{255}(a) \) and \( \text{I2LEBSP}_{255}(a + r_S) \). This is a potential hazard, but it may or may not be necessary to force use of the canonical representation \( \text{I2LEBSP}_{255}(a) \), depending on the context in which the [un]packing operation is used. We therefore do not consider this to be part of the [un]packing operation itself.

### A.3.2.2 Range check

Let \( n : \mathbb{N}^+ \) be a constant, and let \( a = \sum_{i=0}^{n-1} a_i \cdot 2^i : \mathbb{N} \). Suppose we want to constrain \( a \leq c \) for some constant \( c = \sum_{i=0}^{n-1} c_i \cdot 2^i : \mathbb{N} \).

Without loss of generality we can assume that \( c_{n-1} = 1 \), because if it were not then we would decrease \( n \) accordingly.

Note that since \( a \) and \( c \) are provided in binary representation, their bit length \( n \) is not limited by the field element size. We do not assume that the bits \( a_0 \ldots n-1 \) are already boolean-constrained.

Define \( \Pi_m = \prod_{i=m}^{n-1} (c_i = 0 \lor a_i = 1) \) for \( m \in \{0 \ldots n-1\} \). Notice that for any \( m < n - 1 \) such that \( c_m = 0 \), we have \( \Pi_m = \Pi_{m+1} \), and so it is only necessary to allocate separate variables for the \( \Pi_m \) such that \( m < n - 1 \) and \( c_m = 1 \). Furthermore if \( c_{n-2} \ldots 0 \) has \( t > 0 \) trailing 1 bits, then we do not need to allocate variables for \( \Pi_0 \ldots t-1 \) 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. 129.

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 \( c_{n-2} \ldots 0 \).
Theorem A.3.1. Correctness of a constraint system for range checks.

Assume $c_{0..n-1} : \mathbb{B}^[n]$ and $c_{n-1} = 1$. Define $A_m := \sum_{i=m}^{n-1} a_i \cdot 2^i$ and $C_m := \sum_{i=m}^{n-1} e_i \cdot 2^i$. For any $m \in \{0..n-1\}$, $A_m \leq C_m$ iff the restriction of the above constraint system to $i \in \{m..n-1\}$ is satisfied. Furthermore the system at least boolean-constrains $a_0..a_{n-1}$.

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 $a_0..a_{n-1}$ are at least boolean-constrained.

To prove the rest of the theorem we proceed by induction on decreasing $m$, i.e. taking successively longer prefixes of the big-endian binary representations of $a$ and $c$.

Base case $m = n-1$: since $c_{n-1} = 1$, the constraint system has just one boolean constraint on $a_{n-1}$, which fulfils the theorem since $A_{n-1} \leq C_{n-1}$ is always satisfied.

Inductive case $m < n-1$:

- If $A_{m+1} > C_{m+1}$, then by the inductive hypothesis the constraint system must fail, which fulfils the theorem regardless of the value of $a_m$.

- If $A_{m+1} \leq C_{m+1}$, then by the inductive hypothesis the constraint system restricted to $i \in \{m+1..n-1\}$ succeeds. We have $\Pi_{m+1} = \prod_{i=m+1}^{n-1} (c_i = 0 \lor a_i = 1) = \prod_{i=m+1}^{n-1} (a_i \geq c_i)$.

  - If $A_{m+1} = C_{m+1}$, then $a_i = c_i$ for all $i \in \{m+1..n-1\}$ and so $\Pi_{m+1} = 1$. Also $A_m \leq C_m$ iff $a_m \leq c_m$.

  - If $A_{m+1} < C_{m+1}$, then it cannot be the case that $a_i \geq c_i$ for all $i \in \{m+1..n-1\}$, so $\Pi_{m+1} = 0$. This implies that the constraint on $a_m$ is always equivalent to a boolean constraint, which fulfils the theorem because $A_m \leq C_m$ must be true regardless of the value of $a_m$.

This covers all cases. \(\square\)

Correctness of the full constraint system follows by taking $m = 0$ in the above theorem.

The algorithm in §A.3.3.2 ‘ctEdwards [de]compression and validation’ on p.133 uses range checks with $c = r_3 - 1$ to validate ctEdwards compressed encodings. In that case $n = 255$ and $k = 132$, so the cost of each such range check is 387 constraints.

Non-normative note: It is possible to optimize the computation of $\Pi_{n-1}$ further. Notice that $\Pi_m$ is only used when $m$ is the index of the last bit of a run of 1 bits in $c$. So for each such run of 1 bits $c_m..c_{m+N-2}$ of length $N-1$, it is sufficient to compute an $N$-ary AND of $a_m..a_{m+N-2}$ and $\Pi_{m+N-1}$: $R = \prod_{i=0}^{N-1} X_i$. This can be computed in 3 constraints for any $N$; boolean-constrain the output $R$, and then add constraints

\[
\left(N - \sum_{i=0}^{N-1} X_i\right) \times (\text{inv}) = (1 - R) \quad \text{to enforce that} \quad \sum_{i=0}^{N-1} X_i \neq N \text{ when } R = 0;
\]

\[
\left(N - \sum_{i=0}^{N-1} X_i\right) \times (R) = (0) \quad \text{to enforce that} \quad \sum_{i=0}^{N-1} X_i = N \text{ when } R = 1.
\]

where $\text{inv}$ is witnessed as $\left(N - \sum_{i=0}^{N-1} X_i\right)^{-1}$ if $R = 0$ or is unconstrained otherwise. (Since $N < r_3$, the sums cannot overflow.)

In fact the last constraint is not needed in this context because it is sufficient to compute an upper bound on each $\Pi_m$ (i.e. it does not benefit a malicious prover to witness $R = 1$ when the result of the AND should be 0). So the cost of computing $\Pi$ variables for an arbitrarily long run of 1 bits can be reduced to 2 constraints. For example, for $c = r_3 - 1$ the overall cost would be reduced to $255 + 68 = 323$ constraints.

These optimizations are not used in Sapling.
A.3.3 Elliptic curve operations

A.3.3.1 Checking that Affine-ctEdwards coordinates are on the curve

To check that \((u, v)\) is a point on the ctEdwards curve, the Sapling circuit uses 4 constraints:

\[
\begin{align*}
(u) \times (u) &= (uu) \\
(v) \times (v) &= (vv) \\
(uu) \times (vv) &= (uuvv) \\
(a_J \cdot uu + vv) \times (1) &= (1 + d_J \cdot uuvv)
\end{align*}
\]

Non-normative note: The last two constraints can be combined into \((d_J \cdot uu) \times (vv) = (a_J \cdot uu + vv - 1)\). The Sapling circuit does not use this optimization.

A.3.3.2 ctEdwards [de]compression and validation

Define DecompressValidate : CompressedCtEdwardsJubjub \(\rightarrow\) AffineCtEdwardsJubjub as follows:

\[
\text{DecompressValidate}((\tilde{u}, v)) :
\]

// Prover supplies the u-coordinate.
Let \(u : E_{\tilde{g}}\).

// § A.3.3.1 ‘Checking that Affine-ctEdwards coordinates are on the curve’ on p. 133.
Check that \((u, v)\) is a point on the ctEdwards curve.

// § A.3.2.1 ‘Un/packing modulo \(r_S\)’ on p. 130.
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. 131.
Check that \(\sum_{i=0}^{254} u_i \cdot 2^i \leq r_S - 1\).

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. 131) for a total of 392 constraints. The cost of the range check includes boolean-constraining \(u_0 \ldots 254\).

The same quadratic constraint program is used for compression and decompression.

Non-normative note: The point-on-curve check could be omitted if \((u, v)\) were already known to be on the curve. However, the Sapling circuit never omits it; this provides a consistency check on the elliptic curve arithmetic.

A.3.3.3 ctEdwards \(\leftrightarrow\) Montgomery conversion

Define CtEdwardsToMont : AffineCtEdwardsJubjub \(\rightarrow\) AffineMontJubjub as follows:

\[
\text{CtEdwardsToMont}(u, v) = \left(\frac{1 + v}{1 - v}, \frac{\sqrt{-40964} \cdot (1 + v)}{(1 - v) \cdot u}\right) \quad [1 - v \neq 0 \text{ and } u \neq 0]
\]

Define MontToCtEdwards : AffineMontJubjub \(\rightarrow\) AffineCtEdwardsJubjub as follows:

\[
\text{MontToCtEdwards}(x, y) = \left(\frac{\sqrt{-40964} \cdot x}{y}, \frac{x - 1}{x + 1}\right) \quad [x + 1 \neq 0 \text{ and } y \neq 0]
\]

133
Either of these conversions can be implemented by the same quadratic constraint program:

\[
\begin{align*}
(y) \times (u) &= (\sqrt{-40964} \cdot x) \\
(x + 1) \times (v) &= (x - 1)
\end{align*}
\]

The above conversions should only be used if the input is guaranteed to be a point on the relevant curve. If that is the case, the theorems below enumerate all exceptional inputs that may violate the side-conditions.

**Theorem A.3.2.** Exceptional points (ctEdwards → Montgomery).

Let \((u, v)\) be an affine point on a ctEdwards curve \(E_{\text{ctEdwards}(a, d)}\). Then the only points with \(u = 0\) or \(1 - v = 0\) are \((0, 1) = \mathcal{O}_S\) 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 [BBLP2008, Definition 2.1]). By substituting \(u = 0\) we obtain \(v = \pm 1\), and by substituting \(v = 1\) and using \(a \neq d\) we obtain \(u = 0\). \(\square\)

**Theorem A.3.3.** Exceptional points (Montgomery → ctEdwards).

Let \((x, y)\) be an affine point on a Montgomery curve \(E_{\text{Mont}(A, B)}\) over \(\mathbb{F}_r\), with parameters \(A\) and \(B\) such that \(A^2 - 4\) is a nonsquare in \(\mathbb{F}_r\), that is birationally equivalent to a ctEdwards curve. Then \(x + 1 \neq 0\), and the only point \((x, y)\) with \(y = 0\) is \((0, 0)\) of order 2.

**Proof.** That the only point with \(y = 0\) is \((0, 0)\) is proven by Theorem A.2.1 on p. 129.

If \(x + 1 = 0\), then substituting \(x = -1\) into the Montgomery curve equation gives \(B \cdot y^2 = x^3 + A \cdot x^2 + x = A - 2\). So in that case \(y^2 = (A - 2)/B\). The right-hand-side is equal to the parameter \(d\) of a particular ctEdwards curve birationally equivalent to the Montgomery curve (see [BL2017, section 4.3.5]). For all ctEdwards curves, \(d\) is nonsquare, so this equation has no solutions for \(y\), hence \(x + 1 \neq 0\). \(\square\)

(When the theorem is applied with \(E_{\text{Mont}(A, B)} = \mathbb{M}\) defined in §A.2 ‘Elliptic curve background’ on p.128, the ctEdwards curve referred to in the proof is an isomorphic rescaling of the Jubjub curve.)

### A.3.3.4 Affine-Montgomery arithmetic

The incomplete affine-Montgomery addition formulae given in [BL2017, section 4.3.2] are:

\[
\begin{align*}
x_3 &= B_M \cdot \lambda^2 - A_M - x_1 - x_2 \\
y_3 &= (x_1 - x_3) \cdot \lambda - y_1 \\
\end{align*}
\]

where \(\lambda = \begin{cases} 3 \cdot x_1^2 + 2 \cdot A_M \cdot x_1 + 1, & \text{if } x_1 = x_2 \\ 2 \cdot B_M \cdot y_1, & \text{otherwise.} \end{cases}\)

The following theorem helps to determine when these incomplete addition formulae can be safely used:

**Theorem A.3.4.** Distinct-\(x\) theorem.

Let \(Q\) be a point of odd-prime order \(s\) on a Montgomery curve \(\mathbb{M} = E_{\text{Mont}(A_M, B_M)}\) over \(\mathbb{F}_r\). Let \(k_{1, 2}\) be integers in \(\{-\frac{s+1}{2}, \frac{s-1}{2}\}\) \(\setminus\{0\}\). Let \(P_i = [k_i] Q = (x_i, y_i)\) for \(i \in \{1, 2\}\), with \(k_2 \neq \pm k_1\). Then the non-unified addition constraints

\[
\begin{align*}
(x_2 - x_1) \times (\lambda) &= (y_2 - y_1) \\
(B_M \cdot \lambda) \times (\lambda) &= (A_M + x_1 + x_2 + x_3) \\
(x_1 - x_3) \times (\lambda) &= (y_3 + y_1)
\end{align*}
\]

implement the affine-Montgomery addition \(P_1 + P_2 = (x_3, y_3)\) for all such \(P_1, P_2\).
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 \( k_{1..2} \neq 0 \) (mod \( s \)). 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 \( k : \{-\frac{k_1 - 1}{2} \ldots \frac{k_1 - 1}{2}\} \mapsto [k]Q : \mathbb{M} \) is injective and \( k_{1..2} \) are in \( \{1.. \frac{k_1 - 1}{2}\} \), then \( k_2 = \pm k_1 \) (contradiction). \( \square \)

The conditions of this theorem are called the distinct-\( x \) criterion.

In particular, if \( k_{1..2} \) are integers in \( \{1.. \frac{k_1 - 1}{2}\} \) 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:

- \((x) \times (x) = (xx)\)
- \((2 \cdot B_M \cdot y) \times (\lambda) = (3 \cdot xx + 2 \cdot A_M \cdot x + 1)\)
- \((B_M \cdot y) \times (\lambda) = (A_{bd} + 2 \cdot x + x_3)\)
- \((x - x_3) \times (\lambda) = (y_3 + y)\)

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. 129.

### A.3.3.5 Affine-ctEdwards arithmetic

Formulae for **affine-ctEdwards** addition are given in [BBLP2008, section 6]. With a change of variable names to match our convention, the formulae for \((u_1, v_1) + (u_2, v_2) = (u_3, v_3)\) are:

\[
\begin{align*}
  u_3 &= \frac{u_1 \cdot v_2 + v_1 \cdot u_2}{1 + d_j \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2} \\
  v_3 &= \frac{v_1 \cdot v_2 - a_j \cdot u_1 \cdot u_2}{1 - d_j \cdot u_1 \cdot u_2 \cdot v_1 \cdot v_2}
\end{align*}
\]

We use an optimized implementation found by Daira Hopwood making use of an observation by Bernstein and Lange in [BL2017, last paragraph of section 4.5.2]:

\[
\begin{align*}
  (u_1 + v_1) \times (v_2 - a_j \cdot u_2) &= (T) \\
  (u_1) \times (v_2) &= (A) \\
  (v_1) \times (u_2) &= (B) \\
  (d_j \cdot A) \times (B) &= (C) \\
  (1 + C) \times (u_3) &= (A + B) \\
  (1 - C) \times (v_3) &= (T - A + a_j \cdot B)
\end{align*}
\]

The correctness of this implementation can be seen by expanding \( T - A + a_j \cdot B \):

\[
T - A + a_j \cdot B = (u_1 + v_1) \cdot (v_2 - a_j \cdot u_2) - u_1 \cdot v_2 + a_j \cdot v_1 \cdot u_2 \\
= v_1 \cdot v_2 - a_j \cdot u_1 \cdot u_2 + u_1 \cdot v_2 - a_j \cdot v_1 \cdot u_2 - u_1 \cdot v_2 + a_j \cdot v_1 \cdot u_2 \\
= v_1 \cdot v_2 - a_j \cdot u_1 \cdot u_2
\]
The above addition formulae are "unified", that is, they can also be used for doubling. Affine-ctEdwards doubling
[2] \((u, v) = (u_3, v_3)\) can also be implemented slightly more efficiently as:

\[
\begin{align*}
(u + v) \times (v - a_j \cdot u) &= (T) \\
(u) \times (v) &= (A) \\
(d_j \cdot A) \times (A) &= (C) \\
(1 + C) \times (u_3) &= (2 \cdot A) \\
(1 - C) \times (v_3) &= (T + (a_j - 1) \cdot A)
\end{align*}
\]

This implementation is obtained by specializing the addition formulae to \((u, v) = (u_1, v_1) = (u_2, v_2)\) and observing
that \(u \cdot v = A = B\).

### A.3.3.6 Affine-ctEdwards nonsmall-order check

In order to avoid small-subgroup attacks, we check that certain points used in the circuit are not of small order. In
practice the Sapling circuit uses this in combination with a check that the coordinates are on the curve (§ A.3.3.1
'Checking that Affine-ctEdwards coordinates are on the curve' on p. 133), so we combine the two operations.

The Jubjub curve has a large prime-order subgroup with a cofactor of 8. To check for a point \(P\) of order 8 or less,
the Sapling circuit doubles three times (as in § A.3.3.5 'Affine-ctEdwards arithmetic' on p. 135) and checks that the
resulting \(u\)-coordinate is not 0 (as in § A.3.1.4 'Nonzero constraints' on p. 130).

On a ctEdwards curve, only the zero point \(O_J\), 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 \(O_J\).

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 \(O_J\) and the point of order 2.
- It is possible to reduce the cost to 8 constraints by eliminating the redundant constraint in the curve point
  check (as mentioned in § A.3.3.1 'Checking that Affine-ctEdwards coordinates are on the curve' on p. 133):
  merging the first doubling with the curve point check; and then optimizing the second doubling based on the
  fact that we only need to check whether the resulting \(u\)-coordinate is zero. The Sapling circuit does not use
  these optimizations.

### A.3.3.7 Fixed-base Affine-ctEdwards scalar multiplication

If the base point \(B\) is fixed for a given scalar multiplication \([k] B\), we can fully precompute window tables for each
window position.

It is most efficient to use 3-bit fixed windows. Since the length of \(r_J\) is 252 bits, we need 84 windows.

Express \(k\) in base 8, i.e. \(k = \sum_{i=0}^{83} k_i \cdot 8^i\).

Then \([k] B = \sum_{i=0}^{83} w(B, i, k_i)\), where \(w(B, i, k_i) = [k_i \cdot 8^i] B\).

We precompute all of \(w(B, i, s)\) for \(i \in \{0..83\}, s \in \{0..7\}\).
To look up a given window entry \( w(B, i, s) = (u_s, v_s) \), where \( s = 4 \cdot s_2 + 2 \cdot s_1 + s_0 \), we use:

\[
\begin{align*}
(s_1) & \times (s_2) = (s_b) \\
(s_0) & \times (-u_0 \cdot s_b + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_b - u_2 \cdot s_1 + u_4 \cdot s_b - u_4 \cdot s_2 - u_6 \cdot s_b \\
& \quad + u_1 \cdot s_b - u_1 \cdot s_2 - u_1 \cdot s_1 + u_3 \cdot s_b + u_3 \cdot s_1 - u_5 \cdot s_b + u_5 \cdot s_2 + u_7 \cdot s_b) = \\
& (u_s - u_0 \cdot s_b + u_0 \cdot s_2 + u_0 \cdot s_1 - u_0 + u_2 \cdot s_b - u_2 \cdot s_1 + u_4 \cdot s_b - u_4 \cdot s_2 - u_6 \cdot s_b) \\
(s_0) & \times (-v_0 \cdot s_b + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_b - v_2 \cdot s_1 + v_4 \cdot s_b - v_4 \cdot s_2 - v_6 \cdot s_b \\
& \quad + v_1 \cdot s_b - v_1 \cdot s_2 - v_1 \cdot s_1 + v_3 \cdot s_b + v_3 \cdot s_1 - v_5 \cdot s_b + v_5 \cdot s_2 + v_7 \cdot s_b) = \\
& (v_s - v_0 \cdot s_b + v_0 \cdot s_2 + v_0 \cdot s_1 - v_0 + v_2 \cdot s_b - v_2 \cdot s_1 + v_4 \cdot s_b - v_4 \cdot s_2 - v_6 \cdot s_b)
\end{align*}
\]

For a full-length (252-bit) scalar this costs 3 constraints for each of 84 window lookups, plus 6 constraints for each of 83 \textit{ctEdwards} additions (as in §A.3.3.5 ‘\textit{Affine-ctEdwards arithmetic}’ on p.135), for a total of 750 constraints.

Fixed-base scalar multiplication is also used in two places with shorter scalars:

- §A.3.6 ‘\textit{Homomorphic Pedersen Commitment}’ on p.141 uses a 64-bit scalar for the \( v \) input to \textit{ValueCommit}, requiring 22 windows at a cost of \( 3 \cdot 22 - 1 + 6 \cdot 21 = 191 \) constraints;

- §A.3.3.10 ‘\textit{Mixing Pedersen hash}’ on p.140 uses a 32-bit scalar for the \( pos \) input to \textit{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.

\textbf{Non-normative notes:}

- It would be more efficient to use arithmetic on the \textit{Montgomery curve}, as in §A.3.3.9 ‘\textit{Pedersen hash}’ on p.138. However since there are only three instances of fixed-base scalar multiplication in the \textit{Spend circuit} and two in the \textit{Output circuit}, the additional complexity was not considered justified for \textit{Sapling}.

- For the multiplications with 64-bit and 32-bit scalars, the scalar is padded to a multiple of 3 bits with zeros. This causes the computation of \( s_b \) in the lookup for the most significant window to be optimized out, which is where the ‘\(-1\)’ comes from in the above cost calculations. No further optimization is done for this lookup.

\textbf{A.3.3.8 Variable-base Affine-ctEdwards scalar multiplication}

When the base point \( B \) is not fixed, the method in the preceding section cannot be used. Instead we use a naive double-and-add method. Given \( k = \sum_{i=0}^{256} k_i \cdot 2^i \), we calculate \( R = [k] B \) using:

\[
\text{// Base}_i = [2^i] B \\
\text{let Base}_0 = B \\
\text{let Acc}_0^u = k_0 \ ? \ Base_0^u : 0 \\
\text{let Acc}_0^v = k_0 \ ? \ Base_0^v : 1 \\
\text{for } i \text{ from 1 up to 250:} \\
\text{let Base}_i = [2] \text{ Base}_{i-1} \\
\text{\quad // select Base}_i \text{ or } O_i \text{ depending on the bit } k_i \\
\text{let Addend}_i^u = k_i \ ? \ Base_i^u : 0 \\
\text{let Addend}_i^v = k_i \ ? \ Base_i^v : 1 \\
\text{let Acc}_i = Acc_{i-1} + Addend_i \\
\text{let } R = Acc_{250}.
\]

\textit{\footnote{A Pedersen commitment uses fixed-base scalar multiplication as a subcomponent.}}
This costs 5 constraints for each of 250 ctEdwards doublings, 6 constraints for each of 250 ctEdwards additions, and 2 constraints for each of 251 point selections, for a total of 3252 constraints.

Non-normative note: It would be more efficient to use 2-bit fixed windows, and/or to use arithmetic on the Montgomery curve in a similar way to §A.3.3.9 ‘Pedersen hash’ on p.138. However since there are only two instances of variable-base scalar multiplication in the Spend circuit and one in the Output circuit, the additional complexity was not considered justified for Sapling.

A.3.3.9 Pedersen hash

The specification of the Pedersen hashes used in Sapling is given in §5.4.1.7 ‘Pedersen Hash Function’ on p. 56. 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. MerkleDepthSapling 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 NoteCommitSapling. 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 efficient lookup of the window entry for each chunk than if the set \{1 .. 8\} had been used, because a point can be conditionally negated using only a single constraint.

Next, we optimize the cost of point addition by allowing as many additions as possible to be performed on the Montgomery curve. An incomplete Montgomery addition costs 3 constraints, in comparison with a ctEdwards addition which costs 6 constraints.

However, we cannot do all additions on the Montgomery curve because the Montgomery addition is incomplete. In order to be able to prove that exceptional cases do not occur, we need to ensure that the distinct-x criterion from §A.3.3.4 ‘Affine-Montgomery arithmetic’ on p. 134 is met. This requires splitting the input into segments (each using an independent generator), calculating an intermediate result for each segment, and then converting to the ctEdwards curve and summing the intermediate results using ctEdwards addition.

Abstracting away the changes of curve, this calculation can be written as:

\[ \text{PedersenHashToPoint}(D, M) = \sum_{j=1}^{N} [(M_j)] T_j^D \]

where \((\cdot)\) and \(T_j^D\) are defined as in §5.4.1.7 ‘Pedersen Hash Function’ on p. 56.

We have to prove that:

- the Montgomery-to-ctEdwards conversions can be implemented without exceptional cases;
- the distinct-x criterion is met for all Montgomery additions within a segment.

The proof of Theorem 5.4.1 on p. 57 showed that all indices of addition inputs are in the range \(\{-\frac{r_j-1}{2}..\frac{r_j-1}{2}\} \{0\}\).

Because the \(T_j^D\) (which are outputs of GroupHashSapling\((\cdot)\)) are all of prime order, and \(\langle M_j \rangle \not\equiv 0 \pmod{r_j}\), it is guaranteed that all of the terms \([(M_j)] T_j^D\) to be converted to ctEdwards form are of prime order. From Theorem A.3.3 on p. 134, we can infer that the conversions will not encounter exceptional cases.

We also need to show that the indices of addition inputs are all distinct disregarding sign.
Theorem A.3.5. Concerning addition inputs in the Pedersen circuit.

For all disjoint nonempty subsets \( S \) and \( S' \) of \( \{1..c\} \), all \( m \in \mathbb{B}^{[3]}[c] \), and all \( \Theta \in \{-1,1\} \):

\[
\sum_{j \in S} \text{enc}(m_j) \cdot 2^{4(j-1)} \neq \Theta \cdot \sum_{j \in S'} \text{enc}(m_{j'}) \cdot 2^{4(j'-1)}.
\]

Proof. Suppose for a contradiction that \( S, S', m, \Theta \) is a counterexample. Taking the multiplication by \( \Theta \) on the right hand side inside the summation, we have:

\[
\sum_{j \in S} \text{enc}(m_j) \cdot 2^{4(j-1)} = \Theta \cdot \sum_{j \in S'} \text{enc}(m_{j'}) \cdot 2^{4(j'-1)}.
\]

Define \( \text{enc}' : \{-1,1\} \times \mathbb{B}^{[3]} \to \{0..8\} \setminus \{4\} \) as \( \text{enc}'(m_i) := 4 + \Theta \cdot \text{enc}(m_i) \).

Let \( \Delta = 4 \cdot \sum_{i=1}^{c} 2^{4(i-1)} \) as in the proof of Theorem 5.4.1 on p. 57. By adding \( \Delta \) to both sides, we get

\[
\sum_{j \in S} \text{enc}'(m_j) \cdot 2^{4(j-1)} + \sum_{j \in \{1..c\}\setminus S} 4 \cdot 2^{4(j-1)} = \sum_{j \in S'} \text{enc}'(m_{j'}) \cdot 2^{4(j'-1)} + \sum_{j' \in \{1..c\}\setminus S'} 4 \cdot 2^{4(j'-1)}
\]

where all of the \( \text{enc}'(m_j) \) and \( \text{enc}'(m_{j'}) \) are in \( \{0..8\} \setminus \{4\} \).

Each term on the left and on the right affects the single hex digit indexed by \( j \) and \( j' \) respectively. Since \( S \) and \( S' \) are disjoint subsets of \( \{1..c\} \) and \( S \) is nonempty, \( S \cap (\{1..c\} \setminus S') \) is nonempty. Therefore the left hand side has at least one hex digit not equal to 4 such that the corresponding right hand side digit is 4: contradiction.

This implies that the terms in the Montgomery addition—as well as any intermediate results formed from adding a distinct subset of terms—have distinct indices disregarding sign, hence distinct \( x \)-coordinates by Theorem A.3.4 on p. 134. (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 \( s_h = s_0 \& s_1 \):

\[
(s_0) \times (s_1) = (s_h)
\]

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 \( s_h \):

\[
\begin{align*}
\text{let } x_S &= x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_h \\
\text{let } y_R &= y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_h
\end{align*}
\]

We implement the conditional negation as \( (2 \cdot y_R) \times (s_2) = (y_R - y_3) \). After substitution of \( y_R \) this becomes:

\[
\begin{align*}
(2 \cdot (y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_h)) \times (s_2) &= \\
(y_1 + (y_2 - y_1) \cdot s_0 + (y_3 - y_1) \cdot s_1 + (y_4 + y_1 - y_2 - y_3) \cdot s_h - y_3)
\end{align*}
\]
Then we substitute $x_S$ into the Montgomery addition constraints from §A.3.3.4 ‘Affine-Montgomery arithmetic’ on p.134, as follows:

$$
(x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_k - x_A) \cdot (\lambda) = (y_S - y_A)
$$

$$(B_{65} \cdot \lambda) \cdot (\lambda) = (A_{65} + x_A + x_1 + (x_2 - x_1) \cdot s_0 + (x_3 - x_1) \cdot s_1 + (x_4 + x_1 - x_2 - x_3) \cdot s_k + x_C)
$$

$$(x_A - x_C) \cdot (\lambda) = (y_C + y_A)
$$

(In the sapling-crypto implementation, linear combinations are first-class values, so these substitutions do not need to be done "by hand").

For the first addition in each segment, both sides are looked up and substituted into the Montgomery addition, so the first lookup takes only 2 constraints.

When these hashes are used in the circuit, the first 6 bits of the input are fixed. For example, in the Merkle tree hashes they represent the layer number. This would allow a precomputation for the first two windows, but that optimization is not done in Sapling.

The cost of a Pedersen hash over $\ell$ bits (where $\ell$ includes the fixed bits) is as follows. The number of chunks is $c = \lceil \frac{\ell}{3} \rceil$ and the number of segments is $n = \lceil \frac{\ell}{63} \rceil$.

The cost is then:

- $2 \cdot c$ constraints for the lookups;
- $3 \cdot (c - n)$ constraints for incomplete additions on the Montgomery curve;
- $2 \cdot n$ constraints for Montgomery-to-ctEdwards conversions;
- $6 \cdot (n - 1)$ constraints for ctEdwards additions;

for a total of $5 \cdot c + 5 \cdot n - 6$ constraints. This does not include the cost of boolean-constraining inputs.

In particular,

- for the Merkle tree hashes $\ell = 516$, so $c = 172$, $n = 3$, and the cost is 869 constraints;
- when a Pedersen hash is used to implement part of a Pedersen commitment for NoteCommitSapling (§5.4.7.2 ‘Windowed Pedersen commitments’ on p.65), $\ell = 6 + \ell_{value} + 2 \cdot \ell_J = 582$, $c = 194$, 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 $\rho$ from $cm$ and $pos$ in §4.14 ‘Note Commitments and Nullifiers’ on p.41. It takes as input a Pedersen commitment $P$, and hashes it with another input $x$.

Let $J$ be as defined in §5.4.1.8 ‘Mixing Pedersen Hash Function’ on p.57.

We define MixingPedersenHash : $\{0 \ldots r_J - 1\} \times J \rightarrow J$ by:

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

This costs 92 constraints for a scalar multiplication (§A.3.3.7 ‘Fixed-base Affine-ctEdwards scalar multiplication’ on p.136), and 6 constraints for a ctEdwards addition (§A.3.3.5 ‘Affine-ctEdwards arithmetic’ on p.135), 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. 35, 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 \( r_S \)’ on p. 130; that is, it is not necessary to ensure that the left or right inputs to the hash represent integers in the range \( \{0 \ldots r_S - 1\} \). Since the root of the Merkle tree is calculated outside the circuit using the canonical representations, and since the Pedersen hashes are collision-resistant on arbitrary bit-sequence inputs, an attempt by an adversarial prover to use a non-canonical input would result in the wrong root being calculated, and the overall path check would fail.

For each layer, the cost is \( 1 + 2 \cdot 255 \) boolean constraints, \( 2 \) constraints for the conditional swap (implemented as two selection constraints), and 869 constraints for the Merkle hash (§A.3.3.9 ‘Pedersen hash’ on p. 138), 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. 138, and adding a randomized point:

\[
\text{WindowedPedersenCommit}_r(s) = \text{PedersenHashToPoint}(“Zcash_PH”, s) + [r] \text{FindGroupHash}^{f(r)}(“Zcash_PH”, “r”)
\]

This can be implemented in:

- \( 5 \cdot c + 5 \cdot n - 6 \) constraints for the Pedersen hash applied to \( \ell = 6 + \text{length}(s) \) bits, where \( c = \text{ceiling} \left( \frac{\ell}{3} \right) \) and \( n = \text{ceiling} \left( \frac{\ell}{3 \cdot 63} \right) \);
- 750 constraints for the fixed-base scalar multiplication;
- 6 constraints for the final \( \text{ctEdwards} \) addition.

When WindowedPedersenCommit is used to instantiate \( \text{NoteCommit}^{\text{Sapling}} \), the cost of the Pedersen hash is 984 constraints as calculated in §A.3.3.9 ‘Pedersen hash’ on p. 138, 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 \( \text{ValueCommit} \).
In order to support this property, we also define *homomorphic Pedersen commitments* as follows:

\[
\text{HomomorphicPedersenCommit}_{rcv}(D, v) = [v] \text{FindGroupHash}^{(r)}((D, "v") + \text{[rcv]} \text{FindGroupHash}^{(r)}(D, "r"))
\]

In the case that we need for ValueCommit, \( v \) has 64 bits\(^8\). This value is given as a bit representation, which does not need to be constrained equal to an integer.

**ValueCommit** can be implemented in:

- 750 constraints for the 252-bit fixed-base multiplication by \( rcv \);
- 191 constraints for the 64-bit fixed-base multiplication by \( v \);
- 6 constraints for the ctEdwards addition

for a total cost of 947 constraints. This does not include the cost to boolean-constrain the input \( v \) or randomness \( rcv \).

### A.3.7 BLAKE2s hashes

BLAKE2s is defined in [ANWW2013]. Its main subcomponent is a "\( G \) function", defined as follows:

\[
G : \{0..9\} \times \{0..2^{32} - 1\}^{[4]} \rightarrow \{0..2^{32} - 1\}^{[4]}
\]

\[
G(a, b, c, d, x, y) = (a'', b'', c'', d'') \quad \text{where}
\]

\[
a' = (a + b + x) \mod 2^{32}
\]

\[
d' = (d \oplus a') \gg 16
\]

\[
c' = (c + d') \mod 2^{32}
\]

\[
b' = (b \oplus c') \gg 12
\]

\[
a'' = (a' + b' + y) \mod 2^{32}
\]

\[
d'' = (d' \oplus a'') \gg 8
\]

\[
c'' = (c' + d'') \mod 2^{32}
\]

\[
b'' = (b' \oplus c'') \gg 7
\]

The following table is used to determine which message words the \( x \) and \( y \) arguments to \( G \) are selected from:

\[
\sigma_0 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
\]

\[
\sigma_1 = [14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3]
\]

\[
\sigma_2 = [11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4]
\]

\[
\sigma_3 = [7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8]
\]

\[
\sigma_4 = [9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13]
\]

\[
\sigma_5 = [2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9]
\]

\[
\sigma_6 = [12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11]
\]

\[
\sigma_7 = [13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10]
\]

\[
\sigma_8 = [6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5]
\]

\[
\sigma_9 = [10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0]
\]

\(^8\) It would be sufficient to use 51 bits, which accommodates the range \( \{0..\text{MAX\_MONEY}\} \), but the Sapling circuit uses 64.
The Initialization Vector is defined as:

\[ IV : \{ 0..2^{32} - 1 \}^8 : = [ 0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A, 0x510E527F, 0xB9B05688C, 0x1F83D9AB, 0x5BE0CD19 ] \]

The full hash function applied to an 8-byte personalization string and a single 64-byte block, in sequential mode with 32-byte output, can be expressed as follows.

Define BLAKE2s-256 : \( (p : \mathbb{B}^{[8]} ) \times (x : \mathbb{B}^{[64]} ) \rightarrow \mathbb{B}^{[32]} \) as:

let \( PB : \mathbb{B}^{[32]} = [ 32, 0, 1, 1 ] || [ 0x00 ]^{20} || p \)

let \( t_0, t_1, f_0, f_1 : \{ 0..2^{32} - 1 \}^{[4]} = [ 0, 0, 0, 0xFFFFFFFF ] \)

let \( h : \{ 0..2^{32} - 1 \}^{[8]} = [ \text{LEOS2IP}_{32}(PB_{4..4+i+3}) \oplus IV, f \text{or } i \text{ from } 0 \text{ up to } 7 ] \)

let \( v : \{ 0..2^{32} - 1 \}^{[16]} = h || [ IV_0, IV_1, IV_2, IV_3, t_0 \oplus IV_4, t_1 \oplus IV_5, f_0 \oplus IV_6, f_1 \oplus IV_7 ] \)

let \( m : \{ 0..2^{32} - 1 \}^{[16]} = [ \text{LEOS2IP}_{32}(x_{4..4+i+3}) \text{ for } i \text{ from } 0 \text{ up to } 15 ] \)

for \( r \) from 0 up to 9:

set \((v_0, v_4, v_8, v_{12}) := G(v_0, v_4, v_8, v_{12}, m_{\sigma_{r,0}}, m_{\sigma_{r,1}})\)

set \((v_1, v_5, v_9, v_{13}) := G(v_1, v_5, v_9, v_{13}, m_{\sigma_{r,2}}, m_{\sigma_{r,3}})\)

set \((v_2, v_6, v_{10}, v_{14}) := G(v_2, v_6, v_{10}, v_{14}, m_{\sigma_{r,4}}, m_{\sigma_{r,5}})\)

set \((v_3, v_7, v_{11}, v_{15}) := G(v_3, v_7, v_{11}, v_{15}, m_{\sigma_{r,6}}, m_{\sigma_{r,7}})\)

set \((v_0, v_5, v_{10}, v_{15}) := G(v_0, v_5, v_{10}, v_{15}, m_{\sigma_{r,8}}, m_{\sigma_{r,9}})\)

set \((v_1, v_6, v_{11}, v_{12}) := G(v_1, v_6, v_{11}, v_{12}, m_{\sigma_{r,10}}, m_{\sigma_{r,11}})\)

set \((v_2, v_7, v_8, v_{13}) := G(v_2, v_7, v_8, v_{13}, m_{\sigma_{r,12}}, m_{\sigma_{r,13}})\)

set \((v_3, v_4, v_9, v_{14}) := G(v_3, v_4, v_9, v_{14}, m_{\sigma_{r,14}}, m_{\sigma_{r,15}})\)

return \( \text{LEBS2OSP}_{256}(\text{concat}_8([ I2LEBSP_{32}(h_i \oplus v_i \oplus v_{i+8}) \text{ for } i \text{ from } 0 \text{ up to } 7 ])) \)

In practice the message and output will be expressed as bit sequences. In the Sapling circuit, the personalization string will be constant for each use.

Each 32-bit exclusive-or is implemented in 32 constraints, one for each bit position \( a \oplus b = c \) as in §A.3.1.5 'Exclusive-or constraints' on p.130.

Additions not involving a message word, i.e. \( (a + b) \mod 2^{32} = c \), are implemented using 33 constraints and a 33-bit equality check: constrain 33 boolean variables \( c_{0..32} \), and then check \( \sum_{i=0}^{i=31} (a_i + b_i) \cdot 2^i = \sum_{i=0}^{i=32} c_i \cdot 2^i \).

Additions involving a message word, i.e. \( (a + b + m) \mod 2^{32} = c \), are implemented using 34 constraints and a 34-bit equality check: constrain 34 boolean variables \( c_{0..33} \), and then check \( \sum_{i=0}^{i=31} (a_i + b_i + m_i) \cdot 2^i = \sum_{i=0}^{i=33} c_i \cdot 2^i \).

For each addition, only \( c_{0..31} \) are used subsequently.

The equality checks are batched; as many sets of 33 or 34 boolean variables as will fit in a \( \mathbb{F}_2 \) field element are equated together using one constraint. This allows 7 such checks per constraint.

Each \( G \) evaluation requires 262 constraints:

- \( 4 \cdot 32 = 128 \) constraints for \( \oplus \) operations;
- \( 2 \cdot 33 = 66 \) constraints for 32-bit additions not involving message words (excluding equality checks);
- \( 2 \cdot 34 = 68 \) constraints for 32-bit additions involving message words (excluding equality checks).
The overall cost is 21006 constraints:

- \(10 \cdot 8 \cdot 262 - 4 \cdot 2 \cdot 32 = 20704\) constraints for 80 \(G\) evaluations, excluding equality checks (the deduction of \(4 \cdot 2 \cdot 32\) is because \(v\) is constant at the start of the first round, so in the first four calls to \(G\), the parameters \(b\) and \(d\) are constant, eliminating the constraints for the first two XORs in those four calls to \(G\));
- \(\text{ceiling}\left(\frac{10 \cdot 8 \cdot 4}{7}\right) = 46\) 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 \(\mathbb{F}_p\) elements to represent single bits. However Pedersen hashes do not have the necessary cryptographic properties for the two cases where the Spend circuit uses BLAKE2s. While it might be possible to use variants of functions with low circuit cost such as MiMC [AGRRT2017], it was felt that they had not yet received sufficient cryptanalytic attention to confidently use them for Sapling.
A.4 The Sapling Spend circuit

The **Sapling Spend** statement is defined in §4.15.2 ‘Spend Statement (Sapling)’ on p.43.

The primary input is

\[
(rt : \mathbb{F}_{\text{MerkleSapling}}), \\
\text{cv}^{old} : \text{ValueCommit}.\text{Output}, \\
\text{nf}^{old} : \mathbb{F}_{\text{ProofSapling}}, \\
rk : \text{SpendAuthSig}.\text{Public}),
\]

which is encoded as 8 \(\mathbb{F}_{\mathbb{Z}}\) elements (starting with the fixed element 1 required by Groth16):

\[
[1, U(rk), V(rk), U(\text{cv}^{old}), V(\text{cv}^{old}), \text{LEBS2IP}_{\text{MerkleSapling}}(rt), \text{LEBS2IP}_{254}(\text{nf}^{old}), \text{LEBS2IP}_{2}(\text{nf}^{old})]
\]

The auxiliary input is

\[
(\text{path} : \mathbb{F}_{\text{MerkleDepthSapling}}, \\
\text{pos} : \{0..2^{\text{MerkleDepthSapling}} - 1\}, \\
\text{gd} : \mathbb{J}, \\
\text{pkd} : \mathbb{J}, \\
\text{v}^{old} : \{0..2^{\ell_{value}} - 1\}, \\
\text{rcv}^{old} : \{0..2^{\ell_{scalar}} - 1\}, \\
\text{cm}^{old} : \mathbb{J}, \\
\text{rcm}^{old} : \{0..2^{\ell_{scalar}} - 1\}, \\
\alpha : \{0..2^{\ell_{scalar}} - 1\}, \\
ak : \text{SpendAuthSig}.\text{Public}, \\
\text{nsk} : \{0..2^{\ell_{scalar}} - 1\}).
\]

\text{ValueCommit}.\text{Output} and \text{SpendAuthSig}.\text{Public} are of type \(\mathbb{J}\), so we have \text{cv}^{old}, \text{cm}^{old}, \text{rk}, \text{gd}, \text{pkd}, \text{and ak} that represent Jubjub curve points. However,

- \text{cv}^{old} will be constrained to an output of ValueCommit;
- \text{cm}^{old} will be constrained to an output of Note Commit^{Sapling};
- \text{rk} will be constrained to \([\alpha]G + ak;
- \text{pkd} will be constrained to \([ivk]gd

so \text{cv}^{old}, \text{cm}^{old}, \text{rk}, \text{and pkd} do not need to be explicitly checked to be on the curve.

In addition, \text{nk\textsterisk} and \text{p\textsterisk} used in Nullifier integrity are compressed representations of Jubjub curve points. TODO: explain why these are implemented as §A.3.3.2 ‘ctEdwards [de]compression and validation’ on p.133 even though the statement spec doesn’t explicitly say to do validation.

Therefore we have \text{gd}, \text{ak}, \text{nk}, and \text{p} that need to be constrained to valid Jubjub curve points as described in §A.3.3.2 ‘ctEdwards [de]compression and validation’ on p.133.
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:

<table>
<thead>
<tr>
<th>Check</th>
<th>Implements</th>
<th>Cost</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_k$ is on the curve (TODO: FIXME also decompressed below)</td>
<td>$a_k :$ SpendAuthSig.Public</td>
<td>4</td>
<td>§A.3.3.1 on p. 133</td>
</tr>
<tr>
<td>$a_k$ is not small order</td>
<td>$\alpha^* : \mathbb{G}$</td>
<td>16</td>
<td>§A.3.3.6 on p. 136</td>
</tr>
<tr>
<td>$\alpha^* : \mathbb{G}$</td>
<td>$\alpha : {0 \ldots 2^{\text{scale} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p. 129</td>
</tr>
<tr>
<td>$\alpha' = [\alpha^*] \mathbb{G}$</td>
<td>Spend authority</td>
<td>750</td>
<td>§A.3.3.7 on p. 136</td>
</tr>
<tr>
<td>$r_k = \alpha' + a_k$</td>
<td></td>
<td>6</td>
<td>§A.3.3.5 on p. 135</td>
</tr>
<tr>
<td>inputize $r_k$ TODO: not ccted decompress validate =&gt; wrong count</td>
<td>$r_k :$ SpendAuthSig.Public</td>
<td>392</td>
<td>§A.3.3.2 on p. 133</td>
</tr>
<tr>
<td>$n_{sk}^* : \mathbb{G}$</td>
<td>$n_{sk} : {0 \ldots 2^{\text{scale} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p. 129</td>
</tr>
<tr>
<td>$n_k = [n_{sk}^*] \mathcal{H}$</td>
<td>Nullifier integrity</td>
<td>750</td>
<td>§A.3.3.7 on p. 136</td>
</tr>
<tr>
<td>$a_k^* = \text{repr}_j(a_k : \mathbb{J})$</td>
<td>Diversified address integrity</td>
<td>392</td>
<td>§A.3.3.2 on p. 133</td>
</tr>
<tr>
<td>$n_{sk}^* = \text{repr}_j(n_k)$ TODO: spec doesn't say to validate $n_k$ since it's calculated</td>
<td>Nullifier integrity</td>
<td>392</td>
<td>§A.3.3.2 on p. 133</td>
</tr>
<tr>
<td>$iv_{k^*} = \text{I2LEBSP}_{251}(\text{CRH}^{iv_k}(a_k, n_k))$</td>
<td>Diversified address integrity</td>
<td>21006</td>
<td>§A.3.7 on p. 142</td>
</tr>
<tr>
<td>$g_d$ is on the curve</td>
<td>$g_d : \mathbb{J}$</td>
<td>4</td>
<td>§A.3.3.1 on p. 133</td>
</tr>
<tr>
<td>$g_d$ is not small order</td>
<td>$\alpha^* : \mathbb{G}$</td>
<td>16</td>
<td>§A.3.3.6 on p. 136</td>
</tr>
<tr>
<td>$p_{kd} = [iv_{k^*}] g_d$</td>
<td>Diversified address integrity</td>
<td>3252</td>
<td>§A.3.3.8 on p. 137</td>
</tr>
<tr>
<td>$v^{\text{old}}^* : \mathbb{G}$</td>
<td>$v^{\text{old}} : {0 \ldots 2^{64} - 1}$</td>
<td>64</td>
<td>§A.3.1.1 on p. 129</td>
</tr>
<tr>
<td>$rcv^{\text{old}}^* : \mathbb{G}$</td>
<td>$rcv : {0 \ldots 2^{\text{scale} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p. 129</td>
</tr>
<tr>
<td>$cv = \text{ValueCommit}_{rcv}(v^{\text{old}})$</td>
<td>Value commitment integrity</td>
<td>947</td>
<td>§A.3.6 on p. 141</td>
</tr>
<tr>
<td>inputize $cv$</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>$rcm^{\text{old}}^* : \mathbb{G}$</td>
<td>$rcm : {0 \ldots 2^{\text{scale} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p. 129</td>
</tr>
<tr>
<td>$cm = \text{NoteCommit}<em>{cm}(g_d, pk</em>{d}, v^{\text{old}})$</td>
<td>Note commitment integrity</td>
<td>1740</td>
<td>§A.3.5 on p. 141</td>
</tr>
<tr>
<td>$cm_u = \text{Extract}_j(r)(cm)$</td>
<td>Merkle path validity</td>
<td>0</td>
<td>§A.3.4 on p. 141</td>
</tr>
<tr>
<td>$r_t'$ is the root of a Merkle tree with leaf $cm_u$ and authentication path $(\text{path}, \text{pos}^*)$</td>
<td></td>
<td>32 • 1380</td>
<td>§A.3.4 on p. 141</td>
</tr>
<tr>
<td>$\text{pos}^{\text{old}}^* = \text{I2LEBSP}<em>{\text{MerkleDepth}</em>{\text{Sapling}}}(\text{pos})$</td>
<td></td>
<td>1</td>
<td>§A.3.2.1 on p. 130</td>
</tr>
<tr>
<td>if $v^{\text{old}} \neq 0$ then $r_t' = rt$</td>
<td></td>
<td>1</td>
<td>§A.3.1.2 on p. 130</td>
</tr>
<tr>
<td>inputize $rt$</td>
<td></td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>$\rho = $ MixingPedersenHash$(cm^{\text{old}}, \text{pos})$</td>
<td>Nullifier integrity</td>
<td>98</td>
<td>§A.3.3.10 on p. 140</td>
</tr>
<tr>
<td>$p^* = \text{repr}_j(\rho)$ TODO: spec doesn't say to validate $\rho$ since it's calculated</td>
<td>Nullifier integrity</td>
<td>392</td>
<td>§A.3.3.2 on p. 133</td>
</tr>
<tr>
<td>$n^{\text{old}}^* = \text{PRF}^{\text{Sapling}}<em>{n</em>{sk^<em>}}(p^</em>)$</td>
<td></td>
<td>21006</td>
<td>§A.3.7 on p. 142</td>
</tr>
<tr>
<td>pack $n_{sk}^{\text{old}}^<em><em>{0..253}$ and $n</em>{sk}^{\text{old}}^</em>_{254..255}$ into two $\mathbb{F}_p$ inputs</td>
<td>input encoding</td>
<td>2</td>
<td>§A.3.2.1 on p. 130</td>
</tr>
</tbody>
</table>

† This is implemented by taking the output of BLAKE2s-256 as a bit sequence and dropping the most significant 5 bits, not by converting to an integer and back to a bit sequence as literally specified.
A.5 The Sapling Output circuit

The Sapling Output statement is defined in §4.15.3 ‘Output Statement (Sapling)’ on p. 44.

The primary input is
\[
(cv^{new} : \text{ValueCommit.Output}, \\
\text{cm}_u : \mathbb{B}^{|\text{MerkleSapling}|}, \\
\text{epk} : \mathcal{J})
\]

which is encoded as 6 \(\mathbb{F}_{r_3}\) elements (starting with the fixed element 1 required by Groth16):
\[
[1, U(cv^{new}), V(cv^{new}), U(\text{epk}), V(\text{epk}), \text{LEBS2IP}_{\text{MerkleSapling}}(\text{cm}_u)]
\]

The auxiliary input is
\[
(g_d : \mathcal{J}, \\
\text{pk} \cdot d : \mathbb{B}^{[6]}, \\
v^{new} : \{0..2^\ell_{\text{value}} - 1\}, \\
rcv^{new} : \{0..2^\ell_{\text{scalar}} - 1\}, \\
rcm^{new} : \{0..2^\ell_{\text{scalar}} - 1\}, \\
\text{esk} : \{0..2^\ell_{\text{scalar}} - 1\})
\]

ValueCommit.Output is of type \(\mathcal{J}\), so we have \(cv^{new}\), \(\text{epk}\), and \(g_d\) that represent Jubjub curve points. However,

- \(cv^{new}\) will be constrained to an output of ValueCommit;
- \(\text{epk}\) will be constrained to \([\text{esk}] g_d\)

so \(cv^{new}\) and \(\text{epk}\) do not need to be explicitly checked to be on the curve.

Therefore we have only \(g_d\) that needs to be constrained to a valid Jubjub curve point as described in §A.3.3.2 ‘ctEdwards [de]compression and validation’ on p. 133.

Note: \(\text{pk} \cdot d\) 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:

<table>
<thead>
<tr>
<th>Check</th>
<th>Implements</th>
<th>Cost</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v^{\text{old}} \in \mathbb{B}^{64}$</td>
<td>$v^{\text{old}} : {0..2^{64} - 1}$</td>
<td>64</td>
<td>§A.3.1.1 on p.129</td>
</tr>
<tr>
<td>$\text{rcv}^* : \mathbb{B}^{l_{\text{scalar}}}$</td>
<td>$\text{rcv} : {0..2^{l_{\text{scalar}} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p.129</td>
</tr>
<tr>
<td>$c_{\text{v}} = \text{ValueCommit}_{\text{rcv}}(v^{\text{old}})$</td>
<td>Value commitment integrity</td>
<td>947</td>
<td>§A.3.6 on p.141</td>
</tr>
<tr>
<td>inputize $c_{\text{v}}$</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$g^*_{d} = \text{repr}_G(g_d : J)$</td>
<td>Note commitment integrity</td>
<td>392</td>
<td>§A.3.3.2 on p.133</td>
</tr>
<tr>
<td>$g_d$ is not small order</td>
<td>Small order checks</td>
<td>16</td>
<td>§A.3.3.6 on p.136</td>
</tr>
<tr>
<td>$\text{esk}^* : \mathbb{B}^{l_{\text{scalar}}}$</td>
<td>$\text{esk} : {0..2^{l_{\text{scalar}} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p.129</td>
</tr>
<tr>
<td>$\text{epk} = [\text{esk}^*] g_d$</td>
<td>Ephemeral public key integrity</td>
<td>3252</td>
<td>§A.3.3.8 on p.137</td>
</tr>
<tr>
<td>inputize $\text{epk}$</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p_{k_d}^* : \mathbb{B}^{l_{\text{bit}}}$</td>
<td>$p_{k_d}^* : \mathbb{B}^{l_{\text{bit}}}$</td>
<td>256</td>
<td>§A.3.1.1 on p.129</td>
</tr>
<tr>
<td>$\text{rcm}^* : \mathbb{B}^{l_{\text{scalar}}}$</td>
<td>$\text{rcm} : {0..2^{l_{\text{scalar}} - 1}}$</td>
<td>252</td>
<td>§A.3.1.1 on p.129</td>
</tr>
<tr>
<td>$c_{\text{m}} = \text{NoteCommit}<em>{\text{rcm}}(g_d, p</em>{k_d}^*, v^{\text{old}})$</td>
<td>Note commitment integrity</td>
<td>1740</td>
<td>§A.3.5 on p.141</td>
</tr>
<tr>
<td>pack inputs</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The implementation represents $\text{esk}^*$, $p_{k_d}^*$, $\text{rcm}^*$, $\text{rcv}^*$, and $v^{\text{old}}$ as bit sequences rather than integers.

### B Batching Optimizations

#### B.1 RedDSA batch verification

The reference verification algorithm for RedDSA signatures is defined in §5.4.6 ‘RedDSA and RedJubjub’ on p.62.

Let the RedDSA parameters $G$ (defining a subgroup $G^{(r)}$ of order $r_G$, a cofactor $h_G$, a group operation $+$, an additive identity $O_G$, a bit-length $\ell_G$, a representation function $\text{repr}_G$, and an abstraction function $\text{abst}_G$); $P_G : G$; $\ell_H : \mathbb{N}$; $H : \mathbb{B}^{N} \rightarrow \mathbb{B}^{\lceil \ell_H/8\rceil}$; and the derived hash function $H^\circ : \mathbb{B}^{N} \rightarrow \mathbb{F}_r[\mathbb{B}]$ 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 $N$ signature batch entries, each of which is a (public key, message, signature) triple.

Let $\text{LEOS2BSP}$, $\text{LEOS2IP}$, and $\text{LEBS2OSP}$ be as defined in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p.50.

Define RedDSA.BatchEntry := RedDSA.Public $\times$ RedDSA.Message $\times$ RedDSA.Signature.
Define RedDSA.BatchVerify : (entry_{0..N-1} : RedDSA.BatchEntry^{[N]}) → \mathbb{B} as:

For each \( j \in \{0..N-1\} \):
- Let \((vk_j, M_j, \sigma_j) = \text{entry}_j\).
- Let \( R_j \) be the first ceiling (\( \ell_G / 8 \)) bytes of \( \sigma_j \), and let \( S_j \) be the remaining ceiling (bitlength\( r_G / 8 \)) bytes.
- Let \( R_j = \text{abst}_G(\text{LEOS2BSP}_{\ell_G}(R_j)) \), and let \( S_j = \text{LEOS2IP}_{\text{bitlength}(S_j)}(S_j) \).
- Let \( vk_j = \text{LEBS2OSP}_{\ell_G}(\text{repg}_G(vk_j)) \).
- Let \( c_j = H^G(R_j || vk_j || M_j) \).

Choose random \( z_j : \mathbb{F}_r \to \{1..2^{128} - 1\} \).

Return 1 if
- for all \( j \in \{0..N-1\}, R_j \neq \bot \) and \( S_j < r_G \); and
- \([h_G]\) \((\sum_{j=0}^{N-1} (z_j \cdot S_j) \mod r_G)) P_G + \sum_{j=0}^{N-1} ([z_j] R_j + [z_j \cdot c_j \mod r_G]) \text{vk}_j = \mathcal{O}_G \).

otherwise 0.

The \( z_j \) values MUST be chosen independently of the signature batch entries.

The performance benefit of this approach arises partly from replacing the per-signature scalar multiplication of the base \( P_G \) with one such multiplication per batch, and partly from using an efficient algorithm for multiscalar multiplication such as Pippinger’s method [Bernstein2001] or the Bos-Coster method [deRooij1995], as explained in [BDLSY2012, section 5].

**Note:** Spend authorization signatures (§ 5.4.6.1 ‘Spend Authorization Signature’ on p. 64) and binding signatures (§ 5.4.6.2 ‘Binding Signature’ on p. 64) use different bases \( P_G \). It is straightforward to adapt the above procedure to handle multiple bases; there will be one \( \left[ \sum_j (z_j \cdot S_j) \mod r_G \right] P \) term for each base \( P \). 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. 73.

Let \( q, r, \beta, S^{(r)}_1, S^{(r)}_2, S^{(r)}_3, P_{\beta_1, \beta_2, \beta_3}, \mathcal{1}_G \), and \( \hat{e}_G \) be as defined in § 5.4.8.2 ‘BLS12-381’ on p. 68.

Define \( \text{MillerLoop}_G : S^{(r)}_1 \times S^{(r)}_2 \to S^{(r)}_3 \) and \( \text{FinalExp}_G : S^{(r)}_3 \to S^{(r)}_3 \) to be the Miller loop and final exponentiation respectively of the \( \hat{e}_G \) pairing computation, so that:

\[
\hat{e}_G(P, Q) = \text{FinalExp}_G(\text{MillerLoop}_G(P, Q))
\]

where \( \text{FinalExp}_G(R) = R^t \) for some fixed \( t \).

Define \( \text{Groth16}_G \).\text{Proof} := S^{(r)}_1 \times S^{(r)}_2 \times S^{(r)}_3 \).

A \( \text{Groth16}_G \) proof consists of a tuple \((\pi_1, \pi_2, \pi_3) : \text{Groth16}_G \).\text{Proof}.

Verification of a single \( \text{Groth16}_G \) proof against an instance encoded as \( a_0..\ell \in \mathbb{F}_r \) \([\ell + 1]\) requires checking the equation

\[
\hat{e}_G(\pi_A, \pi_B) = \hat{e}_G(\pi_C, \Delta) \cdot \hat{e}_G(\sum_{i=0}^{\ell} [a_i] \Psi_i, \Gamma) \cdot \mathcal{Y}
\]

where \( \Delta = \delta \beta P_{\beta_1}, \Gamma = \gamma P_{\beta_3}, \mathcal{Y} = \alpha \beta P_{\beta_2}, \) and \( \Psi_i = \left[ a_i \right] P_{\beta_4} \) for \( i \in \{0..\ell\} \) are elements of the verification key, as described (with slightly different notation) in [Groth2016, section 3.2].

149
This can be written as:
\[ \hat{\epsilon}_S(\pi_A, -\pi_B) \cdot \hat{\epsilon}_S(\pi_C, \Delta) \cdot \hat{\epsilon}_S\left(\sum_{i=0}^{\ell} [a_i] \Psi_i, \Gamma\right) \cdot Y = 1_S. \]

Raising to the power of random \( z \neq 0 \) gives:
\[ \hat{\epsilon}_S([z] \pi_A, -\pi_B) \cdot \hat{\epsilon}_S([z] \pi_C, \Delta) \cdot \hat{\epsilon}_S\left(\sum_{i=0}^{\ell} [z \cdot a_i] \Psi_i, \Gamma\right) \cdot Y^z = 1_S. \]

This justifies the following optimized procedure for performing faster verification of a batch of Groth16_S proofs. Implementations MAY use this procedure to determine whether all proofs in a batch are valid.

Define a type Groth16_S.BatchEntry := Groth16_S.Proof × Groth16_S.PrimaryInput representing proof batch entries.

Define Groth16_S.BatchVerify : (entry_0..N−1 : Groth16_S.BatchEntry[N]) → B as:

For each \( j \in \{0..N-1\} \):

Let \( ((\pi_j,A, \pi_j,B, \pi_j,C), a_j, 0..\ell) = \text{entry}_j \).

Choose random \( z_j \in \mathbb{F}_p^* \in \mathbb{F}_p \{1..2^{128} - 1\} \).

Let \( \text{Accum}_{AB} = \prod_{j=0}^{N-1} \text{MillerLoop}_S\left([z_j] \pi_j,A, -\pi_j,B\right). \)

Let \( \text{Accum}_\Delta = \sum_{j=0}^{N-1} [z_j] \pi_j,C. \)

Let \( \text{Accum}_{\Gamma,i} = \sum_{j=0}^{N-1} (z_j \cdot a_{j,i}) \mod r_S \) for \( i \in \{0..\ell\} \).

Let \( \text{Accum}_Y = \sum_{j=0}^{N-1} z_j \mod r_S. \)

Return 1 if

\[ \text{FinalExp}_S\left(\text{Accum}_{AB} \cdot \text{MillerLoop}_S(\text{Accum}_\Delta, \Delta) \cdot \text{MillerLoop}_S\left(\sum_{i=0}^{\ell} [\text{Accum}_{\Gamma,i}] \Psi_i, \Gamma\right)\right) \cdot Y^{\text{Accum}_Y} = 1_S, \]

otherwise 0.

The \( z_j \) values MUST be chosen independently of the proof batch entries.

The performance benefit of this approach arises from computing two of the three Miller loops, and the final exponentiation, 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. 43) and output proofs (of the statement in §4.15.3 ‘Output Statement (Sapling)’ on p. 44) use different verification keys, with different parameters \( \Delta, \Gamma, Y, \) and \( \Psi_0, \ell. \) It is straightforward to adapt the above procedure to handle multiple verification keys; the accumulator variables \( \text{Accum}_\Delta, \text{Accum}_{\Gamma,i}, \text{Accum}_Y \) are duplicated, with one term in the verification equation for each variable, while \( \text{Accum}_{AB} \) is shared.

Neglecting multiplications in \( S_\Gamma^{(r)} \) and \( \mathbb{F}_p \), 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_S.Proof, which requires three point decompressions and three subgroup checks (two for \( S_1^{(r),*} \) and one for \( S_2^{(r),*} \));
- for each successfully decoded proof: a Miller loop; and a 128-bit scalar multiplication by \( z_j \) in \( S_1^{(r)} \);
- for each verification key: two Miller loops; an exponentiation in \( S_\Gamma^{(r)} \); a multiscalar multiplication in \( S_1^{(r)} \) with \( N \) 128-bit scalars to compute \( \text{Accum}_\Delta \); and a multiscalar multiplication in \( S_1^{(r)} \) with \( \ell + 1 \) 255-bit scalars to compute \( \sum_{i=0}^{\ell} [\text{Accum}_{\Gamma,i}] \Psi_i; \)
- one final exponentiation.
List of Theorems and Lemmata

Theorem 5.4.1 The encoding function (•) is injective ................................................. 57
Theorem 5.4.2 Uncommitted Sapling is not in the range of PedersenHash ............................ 57
Lemma 5.4.3 Let P = (u, v) ∈ J(r). Then (u, −v) /∈ J(r) ...................................................... 70
Theorem 5.4.4  is injective on J(r) ......................................................................................... 71
Theorem A.2.1 (0, 0) is the only point with y = 0 on certain Montgomery curves ................. 129
Theorem A.3.1 Correctness of a constraint system for range checks ................................. 132
Theorem A.3.2 Exceptional points (ctEdwards → Montgomery) ...................................... 134
Theorem A.3.3 Exceptional points (Montgomery → ctEdwards) ....................................... 134
Theorem A.3.4 Distinct-x theorem ...................................................................................... 134
Theorem A.3.5 Concerning addition inputs in the Pedersen circuit .................................. 139

Index

account, 33
activation block height, 80, 83
ALL CAPS, 7
anchor, 15, 16, 17, 30, 31, 42, 43
authenticated one-time symmetric encryption, 20, 60
auxiliary input, 27, 34–36, 40, 42–45, 108
balancing value, 38, 109
BCTV14, 28, 31, 36, 40, 72, 81, 84, 100, 104, 108, 128
bellman, 73, 79
best valid block chain, 15, 46, 48, 115
bilateral consensus rule change, 80
binding (commitment scheme), 24
binding signature, 17, 22, 36, 38, 39, 64, 98, 110
binding signature scheme, 62, 64
Bitcoin, 1, 7, 8, 15, 17, 18, 21, 36, 37, 74, 75, 79, 81–83, 86–89, 94, 100, 115, 117
block, 15–18, 28, 30, 31, 37, 79, 80, 82, 83, 86, 87–91, 94, 101, 103, 104, 114, 115
block chain, 8, 9, 13, 14, 15, 16, 41, 46, 48–50, 72, 73, 78, 83, 87, 90, 91, 94, 96, 101, 103, 113, 115
block chain branch, 83
block chain reorganization, 46, 48, 80
block header, 15, 52, 86, 87–89, 91, 112, 115, 116
block height, 15, 18, 79–82, 84, 85, 87, 89–93, 104
block subsidy, 18, 79, 91, 116
block target spacing, 89, 90
block timestamp, 86
block version number, 86, 87–88, 90, 93, 102–104
BLS12-381, 28, 68, 73, 100, 109
BN-254, 28, 66, 69, 72
Bulletproofs, 40

Canopy, 79, 101
chunks (of a Pedersen hash input), 56
coinbase transaction, 18, 37, 82, 83, 87, 92, 93, 110, 112, 116
coins (in Zerocash), 8
collision resistance, 18, 19, 26, 53, 54, 56–59, 65, 96, 97, 99, 100, 112, 118, 141
commitment scheme, 24, 65, 66, 108, 138
commitment trapdoor, 13, 24, 40, 95, 108
complete twisted Edwards affine coordinates, 57, 128, 129, 135, 136
complete twisted Edwards compressed encoding, 70, 76, 77, 132
complete twisted Edwards elliptic curve, 11, 34, 62, 69, 70–71, 103, 128, 133, 134, 136, 138
consensus branch ID, 36, 37
consensus rule change, 80
nullifier private key,  12
nullifier set, 15, 18, 41, 46, 48

one-time (authenticated symmetric encryption),  20
one-time (signature scheme),  22
open (a commitment),  24
outgoing cipher key,  34, 47, 59
outgoing viewing key,  12, 29, 33, 34, 47, 48, 77, 82, 107, 108
Output ciphertext,  47, 59, 110
Output circuit,  79, 110, 137, 138
Output description,  8, 9, 14, 16, 17, 22, 25, 32, 33–34, 38, 39, 47, 48, 50, 73, 81, 82, 85, 98, 108, 110
Output statement,  17, 28, 32, 34, 39, 44, 73, 108, 110
Output transfer,  16, 17, 21, 32, 38, 81, 94–96

Overwinter,  1, 7, 36, 53, 79, 80, 82, 83, 87, 104, 107–109, 111–113

packing, 130
paying key,  13, 34
Pedersen commitment,  17, 39, 40, 56, 57, 65, 97, 111, 112, 137, 140
Pedersen hash,  26, 56, 57, 65, 69, 98, 109, 111, 112, 138, 140, 141
Pedersen value commitment,  17, 95
positioned note,  14, 41, 48, 96
primary input,  27, 31, 32, 40, 42–44, 106
private key,  8, 9, 20, 22, 23, 32, 40, 47, 60, 76, 98, 108, 110
proof authorizing key,  8, 12, 29, 40, 49
proof batch entry,  150
proving key (for a zk-SNARK),  27, 28, 78, 79
proving system (preprocessing zk-SNARK),  1, 7, 8, 17, 27, 28, 72, 73, 95, 105, 117
Pseudo Random Function,  14, 19, 28, 30, 53, 58, 59, 65, 97, 99, 113
Pseudo Random Permutation,  55
public key,  13, 20, 22, 23, 31, 32, 40, 44–47, 55, 60, 62, 64, 65, 70, 75, 76, 82, 84, 85, 98, 99, 108–110, 113, 148

Quadratic Arithmetic Program,  72, 73, 128
quadratic constraint program,  7, 72, 73, 113, 128, 130, 133, 134

randomized Spend verifying key,  111
Rank 1 Constraint System,  128
raw encoding,  45, 47, 74, 75–78
receiving key,  9, 12, 113

RECOMMENDED,  7, 30
represented group,  25, 62, 69, 110
represented pairing,  26, 66, 68, 110
represented subgroup,  25, 26
root (of a Merkle tree),  17, 35, 36, 84–87

segments (of a Pedersen hash input),  56
serial numbers (in Zerocash),  8
SHA-256,  52, 53, 59, 65, 78, 79, 96, 112
SHA-256d,  52, 86, 88, 89
SHA256Compress,  53, 58, 59, 65, 75, 76, 97–99, 112, 114
shielded,  8, 16, 33, 83, 94, 95
shielded input,  8, 16, 17, 35
shielded output,  8, 16, 17, 45, 46
shielded payment address,  8, 9, 12, 13, 19, 34, 35, 40, 47, 49, 50, 54, 55, 74, 75, 76, 79, 85, 98, 115, 117
shielded transfer,  8
SHOULD,  7, 33, 34, 39, 56, 72, 74, 83, 87
SHOULD NOT,  7, 74, 104
SIGHASH algorithm,  36, 37
SIGHASH transaction hash,  32, 36, 39, 40, 53, 81, 93, 94, 102, 107, 115
SIGHASH type,  36, 37, 39, 40, 107, 118
signature batch entry,  148, 149
signature scheme,  21, 22, 23, 53, 61, 62, 64, 110
signature scheme with key monomorphism,  23, 65, 101
signature scheme with re-randomizable keys,  22, 28, 40, 64
slanted text,  7
spend authorization address key,  40
spend authorization private key,  40
spend authorization randomizer,  40, 106
spend authorization signature,  14, 31, 32, 36, 38, 40, 98, 106, 110
spend authorization signature scheme,  62, 64
Spend authorizing key,  12, 29, 59
Spend circuit,  32, 79, 110, 137, 138, 144
Spend description,  8, 16, 17, 18, 22, 25, 31, 32, 38–41, 50, 73, 81, 82, 85, 108, 112
Spend transfer,  16, 17–18, 21, 31, 36, 38, 81, 95, 96
spending key,  8, 9, 12, 13–14, 19, 28, 29, 34, 40, 41, 46, 49, 74–76, 78, 95, 96, 99, 111, 115, 117
Sprout, 7, 8, 12–20, 24, 28, 33–38, 41, 45, 46, 49, 51, 53, 73–76, 78, 79, 84, 95–99, 101–113
statement, 17, 27, 40, 42, 99, 111, 128, 145, 147, 150
synthetic binding factor, 40

target threshold, 86, 89, 90, 91
transaction, 8, 9, 13, 15, 16–18, 21, 22, 30–34, 36–41, 46, 48–50, 80, 81, 82–87, 94–96, 104, 107, 112, 113, 117
transaction binding verification key, 38, 82
transaction fee, 18
transaction value pool (Sapling), 38
transaction value pool (transparent), 15, 16, 30, 37, 38, 84
transaction version number, 36, 73, 81, 82, 83–84, 114
transmission key, 9, 13, 14, 20, 33, 45, 46, 49, 73
transmitted note ciphertext (Sapling), 32, 34, 47, 48, 50, 85
transmitted notes ciphertext (Sprout), 31, 33, 45, 49, 84
transparent, 8, 15–17, 37, 75, 81–83, 92–95, 104, 115, 116
transparent address, 75
transparent input, 15, 36, 37, 101
transparent output, 15, 37, 101
treestate, 15, 16–18, 30, 31, 41, 86, 87, 107, 114, 117
Uniform Random String, 26, 108
unpacking, 130
unspent transaction output set, 17
valid block chain, 15, 18, 95, 96
valid Equihash solution, 87, 88
value commitment, 8, 17, 22, 31, 32, 38, 39, 47, 48, 85
value commitment scheme, 39
verifying key (for a zk-SNARK), 27, 28, 78, 79
version group ID, 82, 83
weak PRF, 98
windowed, 65
windowed Pedersen commitment, 141
zatoshi, 13, 38, 52, 91, 93
Zcash, 1, 7–9, 12, 16, 18, 21, 27, 28, 36, 37, 50, 52, 53, 64, 69, 71–76, 78, 79, 81, 83, 86, 88, 89, 93–97, 99, 100, 104, 107, 115, 118
zcashd, 12, 30, 62, 87, 89, 90, 97, 101–103, 113, 114
ZEC, 13
Zerocash, 1, 7, 8, 19, 37, 94–100, 113, 115–118
zk-SNARK circuit, 56, 69, 79
zk-SNARK proof, 14, 16, 17, 27, 31, 32, 40, 84, 85, 96, 98, 107, 115