Zcash Protocol Specification
Version 2018.0-beta-14

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

March 11, 2018

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

This specification defines the Zcash consensus protocol and explains its differences from Zerocash and Bitcoin.

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

Contents

1 Introduction ........................................ 5
   1.1 Caution ........................................... 5
   1.2 High-level Overview ............................. 5

2 Notation ............................................. 7

3 Concepts ............................................. 8
   3.1 Payment Addresses and Keys .................... 8
   3.2 Notes ........................................... 9
      3.2.1 Note Plaintexts and Memo Fields ........ 9
   3.3 The Block Chain ................................ 10
   3.4 Transactions and Treestates .................... 10
   3.5 JoinSplit Transfers and Descriptions .......... 11
   3.6 Note Commitment Trees ......................... 11
   3.7 Nullifier Sets .................................. 12
   3.8 Block Subsidy and Founders’ Reward .......... 12
   3.9 Coinbase Transactions ......................... 12

† Zerocoin Electric Coin Company
4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

4.1.2 Pseudo Random Functions

4.1.3 Authenticated One-Time Symmetric Encryption

4.1.4 Key Agreement

4.1.5 Key Derivation

4.1.6 Signature

4.1.6.1 Signature with Re-Randomizable Keys

4.1.7 Commitment

4.1.8 Represented Group

4.1.9 Represented Pairing

4.1.10 Zero-Knowledge Proving System

4.2 Key Components

4.3 JoinSplit Descriptions

4.4 Sending Notes

4.4.1 Dummy Notes

4.5 Merkle path validity

4.6 Non-malleability

4.7 Balance

4.8 Note Commitments and Nullifiers

4.9 Zk-SNARK Statements

4.9.1 JoinSplit Statement

4.10 In-band secret distribution

4.10.1 Encryption

4.10.2 Decryption by a Recipient

5 Concrete Protocol

5.1 Caution

5.2 Integers, Bit Sequences, and Endianness

5.3 Constants

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

5.4.1.1 SHA-256 and SHA256Compress Hash Functions

5.4.1.2 BLAKE2 Hash Function

5.4.1.3 Merkle Tree Hash Function

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

5.4.1.5 Equihash Generator

5.4.2 Pseudo Random Functions
5.4.3 Authenticated One-Time Symmetric Encryption .................................................. 29
5.4.4 Key Agreement and Derivation ........................................................................... 30
  5.4.4.1 Key Agreement ................................................................................................. 30
  5.4.4.2 Key Derivation ................................................................................................. 30
5.4.5 JoinSplit Signature ............................................................................................ 30
5.4.6 Commitment schemes ....................................................................................... 31
  5.4.6.1 Note Commitments .......................................................................................... 31
5.4.7 Represented Groups and Pairings ...................................................................... 31
  5.4.7.1 BN-254 ........................................................................................................... 31
5.4.8 Zero-Knowledge Proving Systems ...................................................................... 32
  5.4.8.1 PHGR13 ......................................................................................................... 32
5.5 Note Plaintexts and Memo Fields ......................................................................... 33
5.6 Encodings of Addresses and Keys ......................................................................... 34
  5.6.1 Transparent Addresses ...................................................................................... 34
  5.6.2 Transparent Private Keys .................................................................................. 35
  5.6.3 Shielded Payment Addresses ............................................................................ 35
  5.6.4 Incoming Viewing Keys ..................................................................................... 35
  5.6.5 Spending Keys .................................................................................................... 36
5.7 zk-SNARK Parameters .......................................................................................... 36
6 Consensus Changes from Bitcoin .............................................................................. 37
  6.1 Encoding of Transactions ..................................................................................... 37
  6.2 Encoding of JoinSplit Descriptions ..................................................................... 39
  6.3 Block Header ......................................................................................................... 40
  6.4 Proof of Work ........................................................................................................ 41
    6.4.1 Equihash .......................................................................................................... 42
    6.4.2 Difficulty filter .................................................................................................. 43
    6.4.3 Difficulty adjustment ....................................................................................... 43
    6.4.4 nBits conversion ............................................................................................... 44
    6.4.5 Definition of Work ........................................................................................... 44
  6.5 Calculation of Block Subsidy and Founders’ Reward ........................................... 45
  6.6 Payment of Founders’ Reward .............................................................................. 45
  6.7 Changes to the Script System .............................................................................. 47
  6.8 Bitcoin Improvement Proposals .......................................................................... 47
7 Differences from the Zerocash paper ..................................................................... 47
  7.1 Transaction Structure ............................................................................................ 47
  7.2 Memo Fields ......................................................................................................... 47
  7.3 Unification of Mints and Pours ............................................................................. 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Faerie Gold attack and fix</td>
<td>48</td>
</tr>
<tr>
<td>7.5</td>
<td>Internal hash collision attack and fix</td>
<td>49</td>
</tr>
<tr>
<td>7.6</td>
<td>Changes to PRF inputs and truncation</td>
<td>49</td>
</tr>
<tr>
<td>7.7</td>
<td>In-band secret distribution</td>
<td>50</td>
</tr>
<tr>
<td>7.8</td>
<td>Omission in Zerocash security proof</td>
<td>51</td>
</tr>
<tr>
<td>7.9</td>
<td>Miscellaneous</td>
<td>52</td>
</tr>
</tbody>
</table>

8 Acknowledgements | 52 |

9 Change History | 52 |

10 References | 59 |
1 Introduction

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

Changes from the original Zerocash are explained in §7 ‘Differences from the Zerocash paper’ on p. 47, and highlighted in magenta throughout the document.

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, and SHOULD NOT 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;
- Consensus Changes from Bitcoin — how Zcash differs from Bitcoin at the consensus layer, including the Proof of Work;
- Differences from the Zerocash protocol — a summary of changes from the protocol in [BCG+2014].

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 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.
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, which specify an amount and a paying key. The paying key is part of a shielded payment address, which is a destination to which notes can be sent. As in Bitcoin, this is associated with a private key that can be used to spend notes sent to the address; in Zcash this is called a spending key.

To each note there is cryptographically associated a note commitment, and a nullifier (so that there is a 1:1 relation between notes, note commitments, and nullifiers). Computing the nullifier requires the associated private spending key. It is infeasible to correlate the note commitment with the corresponding nullifier without knowledge of at least this spending 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 [Bitc-Protocol]. It also contains a sequence of zero or more JoinSplit descriptions. Each of these describes a JoinSplit transfer, which takes in a transparent value and up to two input notes, and produces a transparent value and up to two output notes.

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). Each JoinSplit description also includes a computationally sound zk-SNARK proof, which proves that all of the following hold except with insignificant probability:

- The input and output values balance (individually for each JoinSplit transfer).
- For each input note of non-zero value, some revealed note commitment exists for that note.
- The prover knew the private spending keys of the input notes.
- The nullifiers and note commitments are computed correctly.
- The private spending keys of the input notes are cryptographically linked to a signature over the whole transaction, in such a way that the transaction cannot be modified by a party who did not know these private keys.
- Each output note is generated in such a way that it is infeasible to cause its nullifier to collide with the nullifier of any other note.

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

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

The basis of the privacy properties of Zcash is that when a note is spent, the spender only proves that some commitment for it had been revealed, without revealing which one. This implies that a spent note cannot be linked to the transaction in which it was created. That is, from an adversary’s point of view the set of possibilities for a given note input to a transaction—its note traceability set—includes all previous notes that the adversary does not control or know to have been spent. This contrasts with other proposals for private payment systems, such as CoinJoin [Bitc–CoinJoin] or CryptoNote [vanS2014], that are based on mixing of a limited number of transactions and that therefore have smaller note traceability sets.

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

---
1 In Zerocash [BCG+2014], notes were called “coins,” and nullifiers were called “serial numbers.”
2 JoinSplit transfers in Zcash generalize "Mint" and "Pour" transactions in Zerocash; see §7.1 ‘Transaction Structure’ on p. 47 for differences.
2 Notation

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

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

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

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

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

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

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

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

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

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

\(S \cup T\) means the set union of \(S\) and \(T\), or the type corresponding to it.

\(S \cap T\) means the set intersection of \(S\) and \(T\).

\(0x\) followed by a string of monospace hexadecimal digits means the corresponding integer converted from hexadecimal.

“\ldots” 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.

\(a..b\), used as a subscript, means the sequence of values with indices \(a\) through \(b\) inclusive. For example, \(^{a\ldots b}_{\text{new}}\) means the sequence \([a^\text{new}_{pk,1}, a^\text{new}_{pk,2}, \ldots a^\text{new}_{pk,N^\text{new}}]\). (For consistency with the notation in [BCG+2014] and in [BK2016], this specification uses 1-based indexing and inclusive ranges, notwithstanding the compelling arguments to the contrary made in [EWD-831].)

\([a..b]\) means the set or type of integers from \(a\) through \(b\) inclusive.

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

\(a \| b\) means the concatenation of sequences \(a\) then \(b\).

\(\text{concat}_S(S)\) 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.

\(\text{sorted}(S)\) 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. \(\mathbb{F}_n[z]\) means the ring of polynomials over \(z\) with coefficients in \(\mathbb{F}_n\).

\(a \cdot b\) means the product of multiplying \(a\) and \(b\). This may refer to multiplication of integers, rationals, or finite field elements according to context.
\( a^b \), for \( a \) an integer or finite field element and \( b \) an integer, means the result of raising \( a \) to the exponent \( b \).

\( a \mod q \), for \( a : \mathbb{N} \) and \( q : \mathbb{N}^+ \), means the remainder on dividing \( a \) by \( q \).

\( a \oplus b \) 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 bit sequences according to context.

\[
\sum_{i=1}^{N} a_i \text{ means the sum of } a_{1..N}. \quad \bigoplus_{i=1}^{N} a_i \text{ means the bitwise-exclusive-or of } a_{1..N}.
\]

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

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

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

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

The following integer constants will be instantiated in §5.3 ‘Constants’ on p. 26: MerkleDepth, \( \mathbb{N}^\text{old} \), \( \mathbb{N}^\text{new} \), \( \ell_{\text{Merkle}} \), \( \ell_{\text{Sig}}, \ell_{\text{PRF}}, \ell_{\text{r}}, \ell_{\text{Seed}}, \ell_{\text{ask}}, \ell_{\text{ϕ}}, \text{MAX\_MONEY}, \text{SlowStartInterval}, \text{HalvingInterval}, \text{MaxBlockSubsidy}, \text{NumFounderAddresses}, \text{PoWAveragingWindow}, \text{PoWLimit}, \text{PoWMedianBlockSpan}, \text{PoWDampingFactor}, \text{PoWTargetSpacing} \).

The bit sequence constant \( \text{Uncommitted} : \mathbb{B}^{[\ell_{\text{Merkle}}]} \) and rational constants \( \text{FoundersFraction}, \text{PoWMaxAdjustDown}, \) and \( \text{PoWMaxAdjustUp} \) will also be defined in that section.

## 3 Concepts

### 3.1 Payment Addresses and Keys

Users who wish to receive payments under this scheme first generate a random spending key \( a_{\text{sk}} \).

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

![Diagram](image)

The receiving key \( sk_{\text{enc}} \), the incoming viewing key \( ivk = (apk, sk_{\text{enc}}) \), and the shielded payment address \( addr_{pk} = (apk, pk_{\text{enc}}) \) are derived from \( a_{\text{sk}} \), as described in §4.2 ‘Key Components’ on p.19.

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

Note: It is conventional in cryptography to 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) need not be publically distributed; it has the same distribution as the shielded payment address itself. As mentioned above, limiting the distribution of the shielded payment address is important for some use cases. This also helps to reduce reliance of the overall protocol on the security of the cryptosystem used for note encryption (see §4.10 In-band secret distribution on p. 24), since an adversary would have to know pk_enc in order to exploit a hypothetical weakness in that cryptosystem.

3.2 Notes

A note (denoted n) is a tuple (apk, v, ρ, r). It represents that a value v is spendable by the recipient who holds the spending key aspk corresponding to apk, as described in the previous section.

A note is a tuple (apk, v, ρ, r), where:
- apk + \{f_{PRF}\} 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);
- ρ : \{f_{PRF}\} is used as input to PRF^nf_a to derive the nullifier of the note;
- r : COMM\textsuperscript{Sprout}.Trapdoor is a random commitment trapdoor as defined in §4.1.7 ‘Commitment’ on p. 17.

Let Note be the type of a note, i.e.

\text{Note} := \{f_{PRF}\} \times \{0 .. MAX_MONEY\} \times \{f_{PRF}\} \times COMM\textsuperscript{Sprout}.Trapdoor.

Creation of new notes is described in §4.4 ‘Sending Notes’ on p. 20. When notes are sent, only a commitment (see §4.1.7 ‘Commitment’ on p. 17) to the above values is disclosed publically, and added to a data structure called the note commitment tree. This allows the value and recipient to be kept private, while the commitment is used by the zero-knowledge proof when the note is spent, to check that it exists on the block chain.

A note commitment on a note \( n = (apk, v, \rho, r) \) is computed as

\text{NoteCommitment}(n) = COMM\textsuperscript{Sprout}(apk, v, \rho),

where COMM\textsuperscript{Sprout} is instantiated in §5.4.6.1 ‘Note Commitments’ on p. 31.

A nullifier (denoted nf) is derived from the \( \rho \) value of a note and the recipient’s spending key aspk, using a Pseudo Random Function (see §4.1.2 ‘Pseudo Random Functions’ on p. 13). This computation is described in §4.8 ‘Note Commitments’ on p. 22.

A note is spent by proving knowledge of \( \rho \) and aspk in zero knowledge while publically disclosing its nullifier nf, allowing nf to be used to prevent double-spending.

3.2.1 Note Plaintexts and Memo Fields

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

The note plaintexts in a JoinSplit description are encrypted to the respective transmission keys pk_enc, new. Each note plaintext (denoted np) consists of (v, \rho, r, memo).
memo represents a memo field associated with this note. The usage of the memo field is by agreement between the sender and recipient of the note.

Other fields are as defined in §3.2 Notes on p. 9.

The result of encryption forms part of a transmitted notes ciphertext (see §4.10 In-band secret distribution’ on p. 24 for further details).

3.3 The Block Chain

At a given point in time, each full validator is aware of a set of candidate blocks. These form a tree rooted at the genesis block, where each node in the tree refers to its parent via the hashPrevBlock block header field (see §6.3 Block Header’ on p. 40).

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

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

In order to choose the best valid block chain in its view of the overall block tree, a node sums the work, as defined in §6.4.5 Definition of Work’ on p. 44, of all blocks in each chain, and considers the valid block chain with greatest total work to be best. To break ties between leaf blocks, a node will prefer the block that it received first.

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

3.4 Transactions and Treestates

Each block contains one or more transactions.

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

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

To each transaction there is associated an initial treestate.

A treestate consists of:

- a note commitment tree (§3.6 Note Commitment Trees’ on p. 11);
- a nullifier set (§3.7 Nullifier Sets’ on p. 12).

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

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

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

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

JoinSplit descriptions also have interstitial input and output treestates, explained in the following section.
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. This kind of value transfer is the primary Zcash-specific operation performed by transactions.

A JoinSplit transfer spends \( N^{\text{old}} \) notes \( n_{1..N}^{\text{old}} \) and transparent input \( v_{\text{pub}}^{\text{old}} \), and creates \( N^{\text{new}} \) notes \( n_{1..N}^{\text{new}} \) and transparent output \( v_{\text{pub}}^{\text{new}} \). It is associated with an instance of a JoinSplit statement (§4.9.1 ‘JoinSplit Statement’ on p. 23), for which it provides a zk-SNARK proof.

Each transaction has a sequence of JoinSplit descriptions.

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

The anchor of each JoinSplit description in a transaction refers to a treestate. For the first JoinSplit description, this MUST be the output treestate of a previous block.

For each JoinSplit description in a transaction, an interstitial output treestate is constructed which adds the note commitments and nullifiers specified in that JoinSplit description to the input treestate referred to by its anchor. This interstitial output treestate is available for use as the anchor of subsequent JoinSplit descriptions in the same transaction.

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

Consensus rules:

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

3.6 Note Commitment Trees

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

A root of this tree is associated with each treestate, as described in §3.4 ‘Transactions and Treestates’ on p. 10.

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

Each full validator maintains a nullifier set logically associated with each treestate. As valid transactions containing JoinSplit transfers are processed, the nullifiers revealed in JoinSplit descriptions are inserted into the nullifier set associated with the new treestate.

Nullifiers are enforced to be unique within a valid block chain, in order to prevent double-spends.

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

3.8 Block Subsidy and Founders’ Reward

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. 10.

These calculations are described in §6.5 ‘Calculation of Block Subsidy and Founders’ Reward’ on p. 45.

3.9 Coinbase Transactions

The first transaction in a block must be a coinbase transaction, which should collect and spend any miner subsidy and transaction fees paid by transactions included in this block. The coinbase transaction must also pay the Founders’ Reward as described in §6.6 ‘Payment of Founders’ Reward’ on p. 45.

4 Abstract Protocol

4.1 Abstract Cryptographic Schemes

4.1.1 Hash Functions

MerkleCRH : \( \mathbb{B}^{[\ell_{\text{Merkle}}]} \times \mathbb{B}^{[\ell_{\text{Merkle}}]} \rightarrow \mathbb{B}^{[\ell_{\text{Merkle}}]} \) is a collision-resistant hash function used in §4.5 ‘Merkle path validity’ on p. 21. It is instantiated in §5.4.1.3 ‘Merkle Tree Hash Function’ on p. 28.

hSigCRH : \( \mathbb{B}^{[\ell_{\text{Seed}}]} \times \mathbb{B}^{[\ell_{\text{PRF}}][N^{\text{old}}]} \times \text{JoinSplitSig.Public} \rightarrow \mathbb{B}^{[\ell_{\text{Sig}}]} \) is a collision-resistant hash function used in §4.3 ‘JoinSplit Descriptions’ on p. 19. It is instantiated in §5.4.1.4 ‘hSig Hash Function’ on p. 28.

EquihashGen : \( (n \rightarrow \mathbb{N}^+) \times \mathbb{N}^+ \times \mathbb{B}^{[n]} \times \mathbb{N}^+ \rightarrow \mathbb{B}^{[n]} \) is another hash function, used in §6.4.1 ‘Equihash’ on p. 42 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 §5.4.1.5 ‘Equihash Generator’ on p. 28.
4.1.2 Pseudo Random Functions

PRF\textsubscript{x} is a Pseudo Random Function keyed by x.

Let \( \ell_{\text{ask}}, \ell_{\text{ϕ}}, \ell_{\text{hSig}}, \text{ and } \ell_{\text{PRF}} \) be as defined in §5.3 ‘Constants’ on p. 26.

Four independent \( \text{PRF} \) are needed in our protocol:

\[
\text{PRF}^{\text{addr}} : \mathbb{B}^{\ell_{\text{ask}}} \times \{0, \ldots, 255\} \rightarrow \mathbb{B}^{\ell_{\text{PRF}}}
\]
\[
\text{PRF}^{\text{nf}} : \mathbb{B}^{\ell_{\text{ask}}} \times \mathbb{B}^{\ell_{\text{PRF}}} \rightarrow \mathbb{B}^{\ell_{\text{PRF}}}
\]
\[
\text{PRF}^{\text{pk}} : \mathbb{B}^{\ell_{\text{ask}}} \times \{1, \ldots, N^\text{old}\} \times \mathbb{B}^{\ell_{\text{hSig}}} \rightarrow \mathbb{B}^{\ell_{\text{PRF}}}
\]
\[
\text{PRF}^{\text{ρ}} : \mathbb{B}^{\ell_{\text{ϕ}}} \times \{1, \ldots, N^\text{new}\} \times \mathbb{B}^{\ell_{\text{hSig}}} \rightarrow \mathbb{B}^{\ell_{\text{PRF}}}
\]

These are used in §4.9.1 ‘JoinSplit Statement’ on p. 23; \( \text{PRF}^{\text{addr}} \) is also used to derive a shielded payment address from a spending key in §4.2 ‘Key Components’ on p. 19.

They are instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 29.

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 \( \text{PRF}^{\text{nf}}, \text{PRF}^{\text{addr}}, \text{and } \text{PRF}^{\text{ρ}} \) be collision-resistant across all \( x \) — i.e. finding \((x, y) \neq (x', y')\) such that \( \text{PRF}^{\text{nf}}(y) = \text{PRF}^{\text{nf}}(y') \) should not be feasible, and similarly for \( \text{PRF}^{\text{addr}} \) and \( \text{PRF}^{\text{ρ}} \).

Note: \( \text{PRF}^{\text{nf}} \) was called \( \text{PRF}^{\text{sn}} \) in Zerocash [BCG+2014].

4.1.3 Authenticated One-Time Symmetric Encryption

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


Sym.Decrypt : Sym.K \times Sym.C \rightarrow Sym.P \cup \{⊥\} is the corresponding decryption algorithm, such that for any \( K \in \text{Sym.K} \) and \( P \in \text{Sym.P} \), \( \text{Sym.Decrypt}_K(\text{Sym.Encrypt}_K(P)) = P \). \( ⊥ \) is used to represent the decryption of an invalid ciphertext.

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

4.1.4 Key Agreement

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

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

Let KA.FormatPrivate : \mathbb{B}^{\ell_{\text{PRF}}} \rightarrow KA.Private be a function that converts a bit string of length \( \ell_{\text{PRF}} \) to a KA private key.
Let $KA.\text{DerivePublic} : KA.\text{Private} \times KA.\text{Public} \rightarrow KA.\text{Public}$ be a function that derives the KA public key corresponding to a given KA private key and base point.

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

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

**Note:** The range of $KA.\text{DerivePublic}$ may be a strict subset of $KA.\text{Public}$.

**Security requirements:**

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

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

### 4.1.5 Key Derivation

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

Let $KDF : \{1..N^{\text{new}}\} \times \mathbb{B}^{[h_{\text{Sig}}]} \times KA.\text{SharedSecret} \times KA.\text{Public} \times KA.\text{Public} \rightarrow \text{Sym.K}$ be a Key Derivation Function suitable for use with KA, deriving keys for Sym.Encrypt.

**Security requirement:** In addition to adaptive security of the key agreement and KDF, the following security property is required:

Let $g := KA.\text{Base}$.

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

Let $pk^{j}_{\text{enc}} := KA.\text{DerivePublic}(sk^{j}_{\text{enc}}, g)$.

An adversary can adaptively query a function $Q : \{1..2\} \times \mathbb{B}^{[h_{\text{Sig}}]} \rightarrow KA.\text{Public} \times \text{Sym.K}_{1..N^{\text{new}}}$ where $Q_{j}(h_{\text{Sig}})$ is defined as follows:

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

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

If the adversary’s advantage is insignificant, then the asymmetric encryption scheme constructed from KA, KDF and Sym in §4.10 ‘In-band secret distribution’ on p. 24 will be key-private as defined in [BBDP2001].

**Note:** The given definition only requires ciphertexts to be indistinguishable between transmission keys that are outputs of $KA.\text{DerivePublic}$ (which includes all keys generated as in §4.2 ‘Key Components’ on p. 19). If a transmission key not in that range is used, it may be distinguishable. This is not considered to be a significant security weakness.
4.1.6 Signature

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

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

Zcash uses two 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. 30), which is used to sign transactions that contain at least one $\text{JoinSplit}$ description.

The following defines only the security properties needed for $\text{JoinSplitSig}$.

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

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

Notes:
- A fresh signature key pair is generated for each transaction containing a $\text{JoinSplit}$ description. Since each key pair is only used for one signature (see §4.6 ‘Non-malleability’ on p. 22), a one-time signature scheme would suffice for $\text{JoinSplitSig}$. This is also the reason why only security against non-adaptive chosen message attack is needed. In fact the instantiation of $\text{JoinSplitSig}$ uses a scheme designed for security under adaptive attack even when multiple signatures are signed under the same key.
- SU-CMA security requires it to be infeasible for the adversary, not knowing the private key, to forge a distinct signature on a previously seen message. That is, $\text{JoinSplit signatures}$ are intended to be nonmalleable in the sense of [BIP-62].

4.1.6.1 Signature with Re-Randomizable Keys

A signature scheme with re-randomizable keys $\text{Sig}$ is a signature scheme that additionally defines:
- a type of randomizers $\text{Sig}.\text{Random}$;
- a public key randomization algorithm $\text{Sig}.\text{RandomizePublic} : \text{Sig}.\text{Public} \times \text{Sig}.\text{Random} \rightarrow \text{Sig}.\text{Public}$;
- a private key randomization algorithm $\text{Sig}.\text{RandomizePrivate} : \text{Sig}.\text{Private} \times \text{Sig}.\text{Random} \rightarrow \text{Sig}.\text{Private}$.
• a distinguished "identity" randomizer \( \text{Sig.Id} : \text{Sig.Random} \)

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

• \((\text{Sig.RandomizePublic}(\text{pk}, r), \text{Sig.RandomizePrivate}(\text{sk}, r))\) is also a valid \( \text{Sig} \) key pair for any \( r : \text{Sig.Random} \);

• \( \text{Sig.RandomizePrivate}(\cdot, r) : \text{Sig.Private} \to \text{Sig.Private} \) is injective and easily invertible for any \( r : \text{Sig.Random} \);

• For any key pair \((\text{pk}, \text{sk})\) returned by \( \text{Sig.Gen}() \), the distribution of

\[
(\text{Sig.RandomizePublic}(\text{pk}, r), \text{Sig.RandomizePrivate}(\text{sk}, r)) : r \stackrel{R}{\leftarrow} \text{Sig.Random}
\]

is identical to the distribution of \( \text{Sig.Gen}() \).

• \((\text{Sig.RandomizePublic}(\text{pk}, \text{Sig.Id}), \text{Sig.RandomizePrivate}(\text{sk}, \text{Sig.Id})) = (\text{pk}, \text{sk}) \).

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 just called "Unforgeability under Re-randomized Keys" in [FKMSSS2016, Definition 8]). Unlike the case for JoinSplitSig, we require security under adaptive chosen message attack with multiple messages signed using a given key. (Although each note uses a different re-randomized key pair, the same original key pair can be re-randomized for multiple notes, and also it can happen that multiple transactions spending the same note are revealed to an adversary.)

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

Let \( O : \text{Sig.Private} \times \text{Sig.Message} \times \text{Sig.Random} \to \text{Sig.Signature} \) be a generator of signing oracles.

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

\[
O_{sk} := \text{var } Q \leftarrow \{\} \text{ in } (m : \text{Sig.Message}, r : \text{Sig.Random}) \mapsto
\begin{align*}
&\text{let } \sigma = \text{Sig.Sign}_{\text{Sig.RandomizePrivate}(sk, r)}(m) \\
&Q \leftarrow Q \cup \{(m, \sigma)\} \\
&\text{return } \sigma : \text{Sig.Signature}.
\end{align*}
\]

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

**Notes:**

• The requirement for \( \text{Sig.Id} \) simplifies the definition of SURK-CMA by removing the need for two oracles (since the oracle for original keys, called \( O_1 \) in [FKMSSS2016], is a special case of the oracle for randomized keys).

• The fact that \((\text{Sig.RandomizePublic}(pk, r), \text{Sig.RandomizePrivate}(sk, r)) : r \stackrel{R}{\leftarrow} \text{Sig.Random} \) is identically distributed to \( \text{Sig.Gen}() \), implies that the combination of a re-randomized public key and signature(s) under that key do not reveal the key from which it was re-randomized.

• Since \( \text{Sig.RandomizePrivate}(\cdot, r) \) is injective and easily invertible, knowledge of \( \text{Sig.RandomizePrivate}(sk, r) \) and \( r \) implies knowledge of \( sk \).
4.1.7 Commitment

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

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

A commitment scheme \( \text{COMM} \) defines a type of inputs \( \text{COMM}.\text{Input} \), a type of commitments \( \text{COMM}.\text{Output} \), and a type of commitment trapdoors \( \text{COMM}.\text{Trapdoor} \).

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

Security requirements:

- **Computational hiding:** For all \( x, x' \) : \( \text{COMM}.\text{Input} \), the distributions \( \{ \text{COMM}_r(x) \mid r \stackrel{R}{\leftarrow} \text{COMM}.\text{Trapdoor} \} \) and \( \{ \text{COMM}_r(x') \mid r \stackrel{R}{\leftarrow} \text{COMM}.\text{Trapdoor} \} \) are computationally indistinguishable.

- **Computational binding:** It is infeasible to find \( x, x' \) : \( \text{COMM}.\text{Input} \) and \( r, r' \) : \( \text{COMM}.\text{Trapdoor} \) such that \( x \neq x' \) and \( \text{COMM}_r(x) = \text{COMM}_{r'}(x') \).

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

4.1.8 Represented Group

A represented group \( \mathbb{G} \) consists of:

- a subgroup order parameter \( r_{\mathbb{G}} : \mathbb{N}^+ \), which must be prime;
- a cofactor parameter \( h_{\mathbb{G}} : \mathbb{N}^+ \);
- a group \( \mathbb{G} \) of order \( h_{\mathbb{G}} \cdot r_{\mathbb{G}} \), written additively with operation \( + : \mathbb{G} \times \mathbb{G} \to \mathbb{G} \), and additive identity \( O_{\mathbb{G}} \);
- a generator \( P_{\mathbb{G}} \) of the subgroup of \( \mathbb{G} \) of order \( r_{\mathbb{G}} \);
- a bit-length parameter \( \ell_{\mathbb{G}} : \mathbb{N} \);
- a representation function \( \text{repr}_{\mathbb{G}} : \mathbb{G} \to \mathbb{B}^{[\ell_{\mathbb{G}}]} \);
- an abstraction function \( \text{abst}_{\mathbb{G}} : \mathbb{B}^{[\ell_{\mathbb{G}}]} \to \mathbb{G} \cup \{\bot\} \);

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

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

For \( \mathbb{G} : \mathbb{G} \) and \( k : \mathbb{N} \) (or \( k : \mathbb{F}_{r_{\mathbb{G}}} \)) we write \([k] \mathbb{G}\) for \( \sum_{i=1}^{k} \mathbb{G} \).
4.1.9 Represented Pairing

A represented pairing $\mathcal{P}$ consists of:

- a group order parameter $r_p: \mathbb{N}^+$ which must be prime;
- two represented groups $\mathbb{P}_{1, 2}$, both of order $r_p$;
- a group $\mathbb{P}_T$ of order $r_p$, written multiplicatively with operation $\cdot: \mathbb{P}_T \times \mathbb{P}_T \to \mathbb{P}_T$ and multiplicative identity $1_p$;
- a pairing function $\hat{e}_p: \mathbb{P}_1 \times \mathbb{P}_2 \to \mathbb{P}_T$ satisfying:
  - (Bilinearity) for all $a, b \in \mathbb{F}^*$, $P \in \mathbb{P}_1$, and $Q \in \mathbb{P}_2$, $\hat{e}_p([a] P, [b] Q) = \hat{e}_p(P, Q)^{a \cdot b}$, and
  - (Nondegeneracy) there does not exist $P \in \mathbb{P}_1 \setminus \mathcal{O}_{\mathcal{P}_1}$ such that for all $Q \in \mathbb{P}_2$, $\hat{e}_p(P, Q) = 1_p$:

4.1.10 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 $\mathcal{ZK}$ defines:

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

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

Security requirements:

- **Completeness**: An honestly generated proof will convince a verifier: for any $(x, w) \in \mathcal{ZK}\text{.SatisfyingInputs}$, if $\mathcal{ZK}.\text{Prove}_{pk}(x, w)$ outputs $\pi$, then $\mathcal{ZK}.\text{Verify}_{vk}(x, \pi) = 1$.

- **Knowledge Soundness**: For any adversary $\mathcal{A}$ able to find an $x \in \mathcal{ZK}.\text{PrimaryInput}$ and proof $\pi \in \mathcal{ZK}.\text{Proof}$ such that $\mathcal{ZK}.\text{Verify}_{vk}(x, \pi) = 1$, there is an efficient extractor $E_\mathcal{A}$ such that if $E_\mathcal{A}(vk, pk)$ returns $w$, then the probability that $(x, w) \notin \mathcal{ZK}\text{.SatisfyingInputs}$ 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:

$$\Pr\left[(x, w) \in \mathcal{ZK}\text{.SatisfyingInputs} \middle| D(\pi) = 1\right] \quad \text{and} \quad \Pr\left[(x, w) \in \mathcal{ZK}\text{.SatisfyingInputs} \middle| D(\pi) = 1\right] \quad \text{and} \quad \Pr\left[(x, w) \in \mathcal{ZK}\text{.SatisfyingInputs} \middle| D(\pi) = 1\right]$$
These definitions are derived from those in [BCTV2014, Appendix C], adapted to state concrete security for a fixed circuit, rather than asymptotic security for arbitrary circuits. (\(ZK.Prove\) corresponds to \(P\), \(ZK.Verify\) corresponds to \(V\), and \(ZK.SatisfyingInputs\) corresponds to \(R_C\) 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 existing an auxiliary input \(w\) such that \((x, w) \in ZK.SatisfyingInputs\).

It is possible to replay proofs, but informally, a proof for a given \((x, w)\) gives no information that helps to find a proof for other \((x, w)\).

The proving system is instantiated in §5.4.8.1 ‘PHGR13’ on p. 32. ZKJoinSplit refers to this proving system with the BN-254 pairing, specialized to the JoinSplit statement given in §4.9.1 ‘JoinSplit Statement’ on p. 23. In this case we omit the key subscripts on ZKJoinSplit.Prove and ZKJoinSplit.Verify, taking them to be the particular proving key and verifying key defined by the JoinSplit parameters in §5.7 ‘zk-SNARK Parameters’ on p. 36.

### 4.2 Key Components

Let \(PRF^{addr}\) be a Pseudo Random Function, instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 29.

Let \(KA\) be a key agreement scheme, instantiated in §5.4.4.1 ‘Key Agreement’ on p. 30.

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

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

\[
a_{pk} := PRF^{addr}_{a_{sk}}(0) \\
sk_{enc} := KA.FormatPrivate(PRF^{addr}_{a_{sk}}(1)) \\
pk_{enc} := KA.DerivePublic(sk_{enc}, KA.Base).
\]

### 4.3 JoinSplit Descriptions

A JoinSplit transfer, as specified in §3.5 ‘JoinSplit Transfers and Descriptions’ on p. 11, is encoded in transactions as a JoinSplit description.

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

A JoinSplit description consists of \(\langle v_{old_{pub}}, v_{new_{pub}}, rt, n_{old_{n}}, cm_{new_{n}}^{\text{new}}, epk, \text{randomSeed}, h_{1..N_{old}}, \pi_{ZKJoinSplit}, c_{enc_{1..N_{new}}}^{\text{new}} \rangle\)

where

- \(v_{old_{pub}} : \{0 .. \text{MAX\_MONEY}\}\) is the value that the JoinSplit transfer removes from the transparent value pool;
- \(v_{new_{pub}} : \{0 .. \text{MAX\_MONEY}\}\) is the value that the JoinSplit transfer inserts into the transparent value pool;
- \(rt : B^{\ell_{\text{Merkle}}}\) is an anchor, as defined in §3.3 ‘The Block Chain’ on p. 10, for the output treestate of either a previous block, or a previous JoinSplit transfer in this transaction;
- \(n_{old_{n}} : B^{\ell_{\text{PRF}}}\) is the sequence of nullifiers for the input notes;
- \(cm_{new_{new}}^{\text{new}} : \text{COMM}^\text{Sprout}.Output_{\text{new}}\) is the sequence of note commitments for the output notes;
- \(epk : KA.Public\) is a key agreement public key, used to derive the key for encryption of the transmitted notes ciphertext (§4.10 ‘In-band secret distribution’ on p. 24);
- \(\text{randomSeed} : B^{\ell_{\text{Seed}}}\) is a seed that must be chosen independently at random for each JoinSplit description;
• \(h_{1..N^{\text{old}}} : \mathbb{B}^{[\ell_{\text{PSG}} | N^{\text{old}}]}\) is a sequence of tags that bind \(h_{\text{Sig}}\) to each \(a_k\) of the input notes;

• \(\pi_{\text{ZKJoinSplit}} : \text{ZKJoinSplit.Proof}\) is the zero-knowledge proof for the JoinSplit statement;

• \(C_{\text{enc}}^{1..N^{\text{new}}} : \text{Sym.C}^{[N^{\text{new}}]}\) is a sequence of ciphertext components for the encrypted output notes.

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

The value \(h_{\text{Sig}}\) is also computed from \(\text{randomSeed}, n_{1..N^{\text{old}}}, \) and the joinSplitPubKey of the containing transaction:

\[
h_{\text{Sig}} := h_{\text{Sig}} \text{CRH}(\text{randomSeed}, n_{1..N^{\text{old}}}, \text{joinSplitPubKey})
\]

\(h_{\text{Sig}} \text{CRH}\) is instantiated in §5.4.1.4 ‘\(h_{\text{Sig}}\) Hash Function’ on p. 28.

**Consensus rules:**

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

- Either \(v_{\text{old}}^{\text{pub}}\) or \(v_{\text{new}}^{\text{pub}}\) MUST be zero.

- The proof \(\pi_{\text{ZKJoinSplit}}\) MUST be valid given a primary input formed from the other fields and \(h_{\text{Sig}}\). I.e. it must be the case that \(\text{ZKJoinSplit.Verify}((rt, n_{1..N^{\text{old}}}, cm_{1..N^{\text{new}}}, v_{\text{old}}^{\text{pub}}, v_{\text{new}}^{\text{pub}}, h_{\text{Sig}}, h_{1..N^{\text{old}}})), \pi_{\text{ZKJoinSplit}}) = 1\).  

### 4.4 Sending Notes

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

\[
(\text{joinSplitPrivKey}, \text{joinSplitPubKey}) \xleftarrow{\text{R}} \text{JoinSplitSig.Gen}().
\]

For each JoinSplit description, the sender chooses \(\text{randomSeed}\) uniformly at random on \(\mathbb{B}^{[\ell_{\text{Seed}}]}\), 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 \(q\) uniformly at random on \(\mathbb{B}^{[\ell_{q}]}\). Then it creates each output note with index \(i : \{1..N^{\text{new}}\}\) as follows:

- Choose \(r_{i}^{\text{new}}\) uniformly at random on \(\mathbb{B}^{[\ell_{r}]}\).

- Compute \(\rho_{i}^{\text{new}} = \text{PRF}_{q}(i, h_{\text{Sig}})\).

- Encrypt the note to the recipient transmission key \(\text{pk}_{\text{enc},i}\), as described in §4.10 ‘In-band secret distribution’ on p. 24, giving the ciphertext component \(C_{i}^{\text{enc}}\).

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

After generating all of the JoinSplit descriptions, the sender obtains the dataToBeSigned (§4.6 ‘Non-malleability’ on p. 22), and signs it with the private JoinSplit signing key:

\[
\text{joinSplitSig} \xleftarrow{\text{R}} \text{JoinSplitSig.Sign}_{\text{joinSplitPrivKey}}(\text{dataToBeSigned})
\]

Then the encoded transaction including joinSplitSig is submitted to the network.
4.4.1 Dummy Notes

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

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

- Generate a new random spending key a^old_\text{sk},i and derive its paying key a^old_\text{pk},i.
- Set v^old_i := 0.
- Choose ρ^old_i uniformly at random on \mathbb{B}^{ℓ_{\text{PRF}}}.
- Choose r^old_i uniformly at random on \mathbb{B}^{ℓ_{r}}.
- Compute nf^old_i := \text{PRF}_{a^\text{old_\text{sk}},i}(ρ^old_i).
- Construct a dummy path \text{path}_i for use in the auxiliary input to the JoinSplit statement (this will not be checked).
- When generating the JoinSplit proof, set enforceMerklePath_i to 0.

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

4.5 Merkle path validity

The depth of the note commitment tree is MerkleDepth (defined in §5.3 ‘Constants’ on p. 26).

Each node in the incremental Merkle tree is associated with a hash value, which is a bit sequence. The layer numbered h, counting from layer 0 at the root, has 2^h nodes with indices 0 to 2^h − 1 inclusive.

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

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

As-yet unused leaf nodes are associated with a distinguished hash value Uncommitted. It is assumed to be infeasible to find a preimage note n such that NoteCommitment(n) = Uncommitted.

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

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

where

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

Given such a 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.
4.6 Non-malleability

Bitcoin defines several SIGHASH types that cover various parts of a transaction. In Zcash, all of these SIGHASH types are extended to cover the Zcash-specific fields nJoinSplit, vJoinSplit, and (if present) joinSplitPubKey, described in §6.1 ‘Encoding of Transactions’ on p. 37. They do not cover the field joinSplitSig.

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

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

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

JoinSplitSig is instantiated in §5.4.5 ‘JoinSplit Signature’ on p. 30.

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.19, and let \text{PRF}^{pk} be as defined in §4.1.2 ‘Pseudo Random Functions’ on p.13.

For each i ∈ {1...N\textsubscript{old}}, the creator of a JoinSplit description calculates h\textsubscript{i} = \text{PRF}^{pk}_{\text{sk}_{i}}(i, h\textsubscript{Sig}).

The correctness of h\subscript{i, N\textsubscript{old}} is enforced by the JoinSplit statement given in §4.9.1 ‘Non-malleability’ on p. 24. This ensures that a holder of all of the \text{sk}_{i} for every JoinSplit description in the transaction has authorized the use of the private signing key corresponding to joinSplitPubKey to sign this transaction.

4.7 Balance

A JoinSplit transfer can be seen, from the perspective of the transaction, as an input and an output simultaneously. v\textsubscript{old} takes value from the transparent value pool and v\textsubscript{new} adds value to the transparent value pool. As a result, v\textsubscript{old} is treated like an output value, whereas v\textsubscript{new} is treated like an input value.

Unlike original Zerocash [BCG+2014], Zcash does not have a distinction between Mint and Pour operations. The addition of v\textsubscript{old} to a JoinSplit description subsumes the functionality of both Mint and Pour. Also, a difference in the number of real input notes does not by itself cause two JoinSplit descriptions to be distinguishable.

As stated in §4.3 ‘JoinSplit Descriptions’ on p.19, either v\textsubscript{old} or v\textsubscript{new} MUST be zero. No generality is lost because, if a transaction in which both v\textsubscript{old} and v\textsubscript{new} were nonzero were allowed, it could be replaced by an equivalent one in which min(v\textsubscript{old}, v\textsubscript{new}) is subtracted from both of these values. This restriction helps to avoid unnecessary distinctions between transactions according to client implementation.

4.8 Note Commitments and Nullifiers

A transaction that contains one or more JoinSplit descriptions, when entered into the block chain, appends to the note commitment tree with all constituent note commitments. All of the constituent nullifiers are also entered into the nullifier set of the associated treesate. A transaction is not valid if it attempts to add a nullifier to the nullifier set that already exists in the set.

Each note has a ρ component.

Let \text{PRF}^{nf} be as instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 29.
The nullifier of a note is derived as PRF_{a_{sk}}(\rho).

4.9 Zk-SNARK Statements

4.9.1 JoinSplit Statement

A valid instance of \( \pi_{ZKJoinSplit} \) assures that given a primary input:

\[
(rt : B^{[\text{Merkle}]}, \ n_{1..N_{old}} : B^{[\text{PRF}] \times [N_{old}]}),
\]

\[
c_{1..N_{new}} : \text{COMM}^{\text{Sprout}.\text{Output}}^{[N_{new}]}, \ v_{pub}^{old} : \{0..2^{64} - 1\}, \ v_{pub}^{new} : \{0..2^{64} - 1\}, \ h_{Sig} : B^{[\text{Ask}]}.
\]

the prover knows an auxiliary input:

\[
(\text{path}_{1..N_{old}} : B^{[\text{Merkle}] \times \{0..2^{\text{MerkleDepth}} - 1\} \times [N_{old}]}), \ n_{1..N_{old}} : \text{Note}^{[N_{old}]}, \ a_{sk,1..N_{old}} : B^{[\text{Ask}] \times [N_{old}]}, \ n_{1..N_{new}} : \text{Note}^{[N_{new}]}, \ \phi : B^{[\text{Ask}]}.
\]

\[
enforceMerklePath_{1..N_{old}} : B^{[N_{old}]}.
\]

where:

for each \( i \in \{1..N_{old}\} \):

\[
n_{i}^{old} = (a_{pk,i}^{old}, v_{i}^{old}, p_{i}^{old}, f_{i}^{old});
\]

for each \( i \in \{1..N_{new}\} \):

\[
n_{i}^{new} = (a_{pk,i}^{new}, v_{i}^{new}, p_{i}^{new}, f_{i}^{new})
\]

such that the following conditions hold:

**Merkle path validity** for each \( i \in \{1..N_{old}\} \ | enforceMerklePath_{i} = 1 \): path_{i} must be a valid path of depth given by MerkleDepth, as defined in §4.5 ‘Merkle path validity’ on p. 21, from NoteCommitment(n_{i}^{old}) to note commitment tree root rt.

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

**Merkle path enforcement** for each \( i \in \{1..N_{old}\} \), if \( v_{i}^{old} \neq 0 \) then enforceMerklePath_{i} = 1.

\[
\text{Balance} \quad v_{pub}^{old} + \sum_{i=1}^{N_{old}} v_{i}^{old} = v_{pub}^{new} + \sum_{i=1}^{N_{new}} v_{i}^{new} \in \{0..2^{64} - 1\}.
\]
Nullifier integrity for each \(i \in \{1..N^{\text{old}}\}: n^{\text{old}}_i = \text{PRF}_{\sigma^{\text{old}}}_{\text{a}_{\text{sk},i}}(\rho^{\text{old}}_i).

Spend authority for each \(i \in \{1..N^{\text{old}}\}: a_{\text{pk},i}^{\text{old}} = \text{PRF}_{\sigma^{\text{old}}}_{\text{a}_{\text{sk},i}}(0).

Non-malleability for each \(i \in \{1..N^{\text{old}}\}: h_i = \text{PRF}_{\sigma_{\text{sk},i}}(\text{pk}_{\text{a}_{\text{old}},i}(i, h_{\text{Sig}})).

Uniqueness of \(\rho^{\text{new}}_i\) for each \(i \in \{1..N^{\text{new}}\}: \rho^{\text{new}}_i = \text{PRF}_{\sigma_{\text{sk},i}}(\text{ϕ}_{\text{r}_{\text{new}},i}(i, h_{\text{Sig}})).

Note commitment integrity for each \(i \in \{1..N^{\text{new}}\}: \text{cm}^{\text{new}}_i = \text{COMM}_{\text{Sprout}}(n^{\text{new}}_i).

For details of the form and encoding of proofs, see §5.4.8.1 ‘PHGR13’ on p. 32.

### 4.10 In-band secret distribution

The secrets that need to be transmitted to a recipient of funds in order for them to later spend, are v, p, r. A memo field (§3.2.1 ‘Note Plaintexts and Memo Fields’ on p. 9) is also transmitted.

In order to transmit these secrets securely to a recipient without requiring an out-of-band communication channel, the transmission key \(\text{pk}_{\text{enc}}\) is used to encrypt them. The recipient’s possession of the associated incoming viewing key \(\text{ivk}\) is used to reconstruct the original note and memo field.

All of the resulting ciphertexts are combined to form a transmitted notes ciphertext.

Let \(\text{Sym}\) be the encryption scheme instantiated in §5.4.3 ‘Authenticated One-Time Symmetric Encryption’ on p. 29.

For both encryption and decryption:

- Let KDF be the Key Derivation Function instantiated in §5.4.4.2 ‘Key Derivation’ on p. 30.
- Let KA be the key agreement scheme instantiated in §5.4.4.1 ‘Key Agreement’ on p. 30.
- Let \(h_{\text{Sig}}\) be the value computed for this JoinSplit description in §4.3 ‘JoinSplit Descriptions’ on p. 19.

#### 4.10.1 Encryption

Let \(\text{pk}_{\text{enc},1..N^{\text{new}}}\) be the transmission keys for the intended recipient addresses of each new note.

Let \(\text{np}_{1..N^{\text{new}}}\) be the note plaintexts as defined in §5.5 ‘Note Plaintexts and Memo Fields’ on p. 33.

Then to encrypt:

- Generate a new KA (public, private) key pair (epk, esk).
- For \(i \in \{1..N^{\text{new}}\},\)
  - Let \(P_i^{\text{enc}}\) be the raw encoding of \(\text{np}_i\).
  - Let \(\text{sharedSecret}_i := \text{KA}.\text{Agree}(\text{esk}, \text{pk}_{\text{enc},i}^{\text{new}})\).
  - Let \(K_i^{\text{enc}} := \text{KDF}(i, h_{\text{Sig}}, \text{sharedSecret}_i, \text{epk}, \text{pk}_{\text{enc},i}^{\text{new}})\).
  - Let \(C_i^{\text{enc}} := \text{Sym}.\text{Encrypt}_{K_i^{\text{enc}}}(P_i^{\text{enc}})\).

The resulting transmitted notes ciphertext is \((\text{epk}, C_{1..N^{\text{new}}}^{\text{enc}})\).
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 JoinSplit descriptions. This mode of operation raises further security considerations, for example of how to validate a note received out-of-band, which are not addressed in this document.

4.10.2 Decryption by a Recipient

Let $ivk = (a_{pk}, sk_{\text{enc}})$ be the recipient’s incoming viewing key, and let $pk_{\text{enc}}$ be the corresponding transmission key derived from $sk_{\text{enc}}$ as specified in §4.2 ‘Key Components’ on p.19.

Let $cm_{1..N}^{\text{new}}$ be the note commitments of each output coin.

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

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

$$\text{let } K_{i}^{\text{enc}} = \text{KDF}(i, h_{\text{Sig}}, \text{sharedSecret}_i, epk, pk_{\text{enc}})$$

$$\text{return DecryptNote}(K_{i}^{\text{enc}}, C_{i}^{\text{enc}}, cm_{i}^{\text{new}}, a_{pk}).$$

DecryptNote($K_{i}^{\text{enc}}, C_{i}^{\text{enc}}, cm_{i}^{\text{new}}, a_{pk})$ is defined as follows:

$$\text{let } P_{i}^{\text{enc}} = \text{Sym.Decrypt}_{K_{i}^{\text{enc}}}(C_{i}^{\text{enc}})$$

$$\text{if } P_{i}^{\text{enc}} = \bot, \text{ return } \bot$$

$$\text{extract } np_i = (v_{i}^{\text{new}}, \rho_{i}^{\text{new}}, r_{i}^{\text{new}}, \text{memo}_i) \text{ from } P_{i}^{\text{enc}}$$

$$\text{if } \text{COMM}_{\text{Sprout}}(a_{pk}, v_{i}^{\text{new}}, \rho_{i}^{\text{new}}, r_{i}^{\text{new}}) \neq cm_{i}^{\text{new}}, \text{ return } \bot, \text{ else return } np_i.$$ 

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

**Notes:**

- The decryption algorithm corresponds to step 3 (b) i. and ii. (first bullet point) of the Receive algorithm shown in [BCG+2014, Figure 2].
- A note can change from being unspent to spent as a node’s view of the best block chain is extended by new transactions. Also, block chain reorganizations can cause a node to switch to a different best block chain that does not contain the transaction in which a note was output.

See §7.7 ‘In-band secret distribution’ on p.50 for further discussion of the security and engineering rationale behind this encryption scheme.

5 Concrete Protocol

5.1 Caution

TODO: Explain the kind of things that can go wrong with linkage between abstract and concrete protocol. E.g. §7.5 ‘Internal hash collision attack and fix’ on p.49.
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:

- $I2LEBS\ell(N) \times \{0..2^{\ell}-1\} \rightarrow \mathbb{B}[\ell]$, such that $I2LEBS\ell(x)$ is the sequence of $\ell$ bits representing $x$ in little-endian order;
- $I2BEBSP\ell(N) \times \{0..2^{\ell}-1\} \rightarrow \mathbb{B}[\ell]$, such that $I2BEBSP\ell(x)$ is the sequence of $\ell$ bits representing $x$ in big-endian order.
- $LEOS2IP(k)(S)$ is the integer represented in little-endian order by the byte sequence $S$ of length $k$.
- $LEBS2OSP(\ell) \rightarrow \mathbb{B}[\lceil\ceiling(\ell/8)\rceil]$ defined as follows: pad the input on the right with $8 \cdot \ceiling(\ell/8) - \ell$ zero bits so that its length is a multiple of 8 bits. Then convert each group of 8 bits to a byte value with the least significant bit first, and concatenate the resulting bytes in the same order as the groups.

In bit layout diagrams, each box of the diagram represents a sequence of bits. Diagrams are read from left-to-right, with lines read from top-to-bottom; the breaking of boxes across lines has no significance. The bit length $\ell$ is given explicitly in each box, except for the case of a single bit, or for the notation $\{0\}^{\ell}$ representing the sequence of $\ell$ zero bits.

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

5.3 Constants

Define:

- $\text{MerkleDepth} : N := 29$
- $N^\text{old} : N := 2$
- $N^\text{new} : N := 2$
- $\ell^\text{Merkle} : N := 256$
- $\ell^\text{Sig} : N := 256$
- $\ell^\text{PRF} : N := 256$
- $\ell_r : N := 256$
- $\ell^\text{Seed} : N := 256$
- $\ell^\text{Ask} : N := 252$
- $\ell^\text{q} : N := 252$
- $\text{Uncommitted} : \mathbb{B}[\ell^\text{Merkle}] := \{0\}^{\ell^\text{Merkle}}$
- $\text{MAX\_MONEY} : N := 2.1 \cdot 10^{15} \text{ (zatoshi)}$
- $\text{SlowStartInterval} : N := 20000$
- $\text{HalvingInterval} : N := 840000$
- $\text{MaxBlockSubsidy} : N := 1.25 \cdot 10^9 \text{ (zatoshi)}$
- $\text{NumFounderAddresses} : N := 48$
- $\text{FoundersFraction} : Q := \frac{1}{5}$
PoWLimit : $\mathbb{N} := \begin{cases} 2^{243} - 1, & \text{for the production network} \\ 2^{251} - 1, & \text{for the test network} \end{cases}$

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

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

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

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

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

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

5.4 Concrete Cryptographic Schemes

5.4.1 Hash Functions

5.4.1.1 SHA-256 and SHA256Compress Hash Functions

SHA-256 is defined by [NIST2015].

Zcash uses the full SHA-256 hash function to instantiate NoteCommitment.

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

It also uses the \textit{SHA-256 compression function}, SHA256Compress. This operates on a single 512-bit block and excludes the padding step specified in [NIST2015, section 5.1]; i.e. 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.

Zcash uses SHA256Compress to instantiate several \textit{Pseudo Random Functions} and MerkleCRH.

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

TODO: Specify bit order.

5.4.1.2 BLAKE2 Hash Function

BLAKE2 is defined by [ANWW2013]. Zcash uses only the BLAKE2b variant.

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

BLAKE2b is used to instantiate hSigCRH, EquihashGen, and KDF.

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

Note: BLAKE2b-$\ell$ is not the same as BLAKE2b-512 truncated to $\ell$ bits, because the digest length is encoded in the parameter block.
5.4.1.3  Merkle Tree Hash Function

MerkleCRH is used to hash incremental Merkle tree hash values.
Let SHA256Compress be as specified in §5.4.1.1 ‘SHA-256 and SHA256Compress Hash Functions’ on p. 27.
MerkleCRH : \( B^{\ell_{\text{Merkle}}} \times B^{\ell_{\text{Merkle}}} \rightarrow B^{\ell_{\text{Merkle}}} \) is defined as follows:

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

Note: SHA256Compress is not the same as the SHA-256 function, which hashes arbitrary-length byte sequences.

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

5.4.1.4  hSig Hash Function

hSigCRH is used to compute the value \( h_{\text{Sig}} \) in §4.3 ‘JoinSplit Descriptions’ on p. 19.

\[
\text{hSigCRH} (\text{randomSeed}, n_{\text{old}}^{1..N_{\text{old}}}, \text{joinSplitPubKey}) := \text{BLAKE2b-256} (\text{“ZcashComputehSig”}, \text{hSigInput})
\]

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

BLAKE2b-256\( (p, x) \) is defined in §5.4.1.2 ‘BLAKE2 Hash Function’ on p. 27.

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

5.4.1.5  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 §6.4.1 ‘Equihash’ on p. 42.

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

Let powcount(\( g \)) := \( \begin{array}{c}
\text{32-bit } g
\end{array} \).

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

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

Indices of bits in \( T \) are 1-based.
BLAKE2b-\( \ell (p, x) \) is defined in §5.4.1.2 ‘BLAKE2 Hash Function’ on p. 27.
Security requirement: BLAKE2b-ℓ(powtag, x) must generate output that is sufficiently unpredictable to avoid short-cuts to the Equihash solution process. It would suffice to model it as a random oracle.

Note: When EquihashGen is evaluated for sequential indices, as in the Equihash solving process (§6.4.1 ‘Equihash’ on p. 42), the number of calls to BLAKE2b can be reduced by a factor of floor \((\frac{512}{n})\) in the best case (which is a factor of 2 for \(n = 200\)).

5.4.2 Pseudo Random Functions

PRF\textsuperscript{addr}, PRF\textsuperscript{nf}, PRF\textsuperscript{pk}, and PRF\textsuperscript{ρ}, described in §4.1.2 ‘Pseudo Random Functions’ on p. 13, are all instantiated using the SHA-256 compression function defined in §5.4.1.1 ‘SHA-256 and SHA256Compress Hash Functions’ on p. 27:

Security requirements:

\begin{itemize}
  \item The SHA-256 compression function must be collision-resistant.
  \item The SHA-256 compression function must be a PRF when keyed by the bits corresponding to \(x\), \(a_{sk}\) or \(ψ\) in the above diagrams, with input in the remaining bits.
\end{itemize}

Note: The first four bits –i.e. the most significant four bits of the first byte– are used to distinguish different uses of SHA256Compress, ensuring that the functions are independent. In addition to the inputs shown here, the bits 1011 in this position are used to distinguish uses of the full SHA-256 hash function — see §5.4.6.1 ‘Note Commitments’ on p. 31.

(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 \(a_{sk}\) to encode a new key type, or that would have required an additional PRF.)

5.4.3 Authenticated One-Time Symmetric Encryption

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

Let \(\text{Sym}.Encrypt_{K}(P)\) 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 \(\text{Sym}.Decrypt_{K}(C)\) 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 \(⊥\) 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 Key Agreement

The key agreement scheme specified in §4.1.4 ‘Key Agreement’ on p. 13 is instantiated using Curve25519 [Bern2006] as follows.

Let KA.Public and KA.SharedSecret be the type of Curve25519 public keys (i.e. a sequence of 32 bytes), and let KA.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 [Bern2006, section 2].

Let KA.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 [Bern2006, section 3]: “clear bits 0, 1, 2 of the first byte, clear bit 7 of the last byte, and set bit 6 of the last byte.” Here the bits of a byte are numbered such that bit b has numeric weight $2^b$.

Define KA.FormatPrivate(x) := clamp_{Curve25519}(x).

Define KA.Agree(n, q) := Curve25519(n, q).

5.4.4.2 Key Derivation

The Key Derivation Function specified in §4.1.5 ‘Key Derivation’ on p.14 is instantiated using BLAKE2b-256 as follows:

$$KDF(i, h_{Sig}, sharedSecret, epk, p_{k_{enc},i}^\text{new}) := \text{BLAKE2b-256}(kdftag, kdfinput)$$

where:

kdftag := 64-bit "ZcashKDF" 8-bit $i−1$ [0]$^{56}$

kdfinput := 256-bit $h_{Sig}$ 256-bit sharedSecret 256-bit epk 256-bit $p_{k_{enc},i}^\text{new}$

BLAKE2b-256($p, x$) is defined in §5.4.1.2 ‘BLAKE2 Hash Function’ on p.27.

5.4.5 JoinSplit Signature

JoinSplitSig is specified in §4.1.6 ‘Signature’ on p.15.

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

- S MUST represent an integer less than the prime $\ell = 2^{252} + 277423177773723535851937790883648493$;
- R MUST represent a point of order $\ell$ on the Ed25519 curve;

If these requirements are not met then the signature is considered invalid. Note that it is not required that the encoding of the y-coordinate in R is less than $2^{255} − 19$.

Ed25519 is defined as using SHA-512 internally.

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

30
The encoding of a signature is:

| 256-bit $R$ | 256-bit $S$ |

where $R$ and $S$ are as defined in [BDLSY2012].

The encoding of a public key is as defined in [BDLSY2012].

5.4.6 Commitment schemes

5.4.6.1 Note Commitments

The commitment scheme $COMM^Sprout$ specified in §4.1.7 ‘Commitment’ on p. 17 is instantiated using SHA-256 as follows:

$$COMM^Sprout_{Sprout}(a_{pk}, v, \rho) := \text{SHA-256}(10100000256\text{-bit } a_{pk} 64\text{-bit } v 256\text{-bit } \rho 256\text{-bit } r).$$

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

Security requirements:

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

5.4.7 Represented Groups and Pairings

5.4.7.1 BN-254

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

Let $q_G := 2188824287183927522224640574525727508869631157297823662689037894645226208583$.

Let $r_G := 2188824287183927522224640574525727508854836400416034343698204186575808495617$.

Let $b_G := 3$.

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

Let $G_1$ be the group of points on a Barreto–Naehrig 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 $G_2$ be the subgroup of order $r$ in the sextic twist $E_{G_2}$ of $G_1$ over $\mathbb{F}_{q_G^2}$ with equation $y^2 = x^3 + \frac{b_G}{\xi}$, where $\xi : \mathbb{F}_{q_G^2}$.

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

Let $G_T$ be the subgroup of $r_G^{12}$ roots of unity in $\mathbb{F}_{q_G^2}^*$.

Let $e_G$ be the optimized ate pairing of type $G_1 \times G_2 \rightarrow G_T$.

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

Let $P_{G_1} : G_1^* := (1, 2)$. 31
Let \( \mathcal{P}_{G_2} : G_2^* := \{11559732032986387107991004021392285783925812861821192530917403151452391805634 \cdot t + 1085704699902305713594457076232829481370756359758180899051992832855852781, 40823678758634368133220340314543556831685132759340128105741076214120093531 \cdot t + 849563923123431417604973247489272438418190587263600148770280649306958101930\).\)

\( \mathcal{P}_{G_1} \) and \( \mathcal{P}_{G_2} \) are generators of \( G_1 \) and \( G_2 \) respectively.

Define \( \text{I2BEBSP} : (\ell : N) \times \{0..2^\ell - 1\} \rightarrow \mathbb{B}[\ell] \) as in §5.2 'Integers, Bit Sequences, and Endianness' on p. 26.

For a point \( P : G_1^* = (x_P, y_P) \):
- The field elements \( x_P \) and \( y_P : \mathbb{F}_q \) are represented as integers \( x \) and \( y : \{0..q-1\} \).
- Let \( \hat{y} = y \mod 2 \).
- \( P \) is encoded as \( \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 1 \end{array} \) 1-bit \( \hat{y} \) 256-bit \( \text{I2BEBSP}_{256}(x) \).

For a point \( P : G_2^* = (x_P, y_P) \):
- Define \( \text{FE2IP} : \mathbb{F}_q[l] / (t^2 + 1) \rightarrow \{0..q_0^2 - 1\} \) such that \( \text{FE2IP}(a_{w,1} \cdot t + a_{w,0}) = a_{w,1} \cdot q + a_{w,0} \).
- Let \( x = \text{FE2IP}(x_P), y = \text{FE2IP}(y_P), \) and \( y' = \text{FE2IP}(-y_P) \).
- Let \( \hat{y} = \begin{cases} 1, \text{ if } y > y' \\ 0, \text{ otherwise.} \end{cases} \)
- \( P \) is encoded as \( \begin{array}{cccccc} 0 & 0 & 0 & 1 & 1 \end{array} \) 1-bit \( \hat{y} \) 512-bit \( \text{I2BEBSP}_{512}(x) \).

**Non-normative notes:**
- The use of big-endian order by \( \text{I2BEBSP} \) is different from the encoding of most other integers in this protocol. The encodings for \( G_1^* \) are consistent with the definition of \( \text{EC2OSP} \) for compressed curve points in \([\text{IEEE2004, section 5.5.6.2}]\). The LSB compressed form (i.e. \( \text{EC2OSP-XL} \)) is used for points in \( G_1^* \), and the \( \text{SORT} \) compressed form (i.e. \( \text{EC2OSP-XS} \)) for points in \( G_2^* \).
- The points at infinity \( \mathcal{O}_{G_1,2} \) never occur in proofs and have no defined encodings in this protocol.
- Testing \( y > y' \) for the compression of \( G_2^* \) points is equivalent to testing whether \((a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})\) in lexicographic order.
- Algorithms for decompressing points from the above encodings are given in \([\text{IEEE2000, Appendix A.12.8}]\) for \( G_1^* \), and \([\text{IEEE2004, Appendix A.12.11}]\) for \( G_2^* \).
- A rational point \( P \neq \mathcal{O}_{G_2} \) on the curve \( E_{G_2} \) can be verified to be of order \( r_{G_2} \), and therefore in \( G_2^* \), by checking that \( r_{G} \cdot P = \mathcal{O}_{G_2} \).

When computing square roots in \( \mathbb{F}_{q_2} \) 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 Zero-Knowledge Proving Systems

#### 5.4.8.1 PHGR13

**Zcash** uses **zk-SNARKs** generated by its fork of **libsark** [libsark-fork] with the *proving system* described in [BCTV2015], which is a refinement of the systems in [PHGR2013] and [BCGT2013].

A proof consists of a tuple \((\pi_A : G_1^*, \pi_A' : G_1^*, \pi_B : G_2^*, \pi_B' : G_1^*, \pi_C : G_1^*, \pi_C' : G_1^*, \pi_K : G_1^*, \pi_H : G_1^*)\). It is computed using the parameters above as described in [BCTV2015, Appendix B].
Note: Many details of the proving system are beyond the scope of this protocol document. For example, the quadratic arithmetic program verifying the JoinSplit statement, or its expression as a Rank 1 Constraint System, are not specified in this document. In practice it will be necessary to use the specific proving and verification keys generated for the Zcash production block chain (see §5.7 ‘zk-SNARK Parameters’ on p. 36), and a proving system implementation that is interoperable with the Zcash fork of libsnark, to ensure compatibility.

Encoding of PHGR13 Proofs  A PHGR13 proof is encoded by concatenating the encodings of its elements:

- 264-bit \( \pi_A \)
- 264-bit \( \pi_A' \)
- 520-bit \( \pi_B \)
- 264-bit \( \pi_B' \)
- 264-bit \( \pi_C \)
- 264-bit \( \pi_C' \)
- 264-bit \( \pi_K \)
- 264-bit \( \pi_H \)

The resulting proof size is 296 bytes.

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

- the lead byte is of the required form;
- the remaining bytes encode a big-endian representation of an integer in \( \{0..q_3 - 1\} \) or (in the case of \( \pi_B \)) \( \{0..q_3^2 - 1\} \);
- the encoding represents a point in \( G_1^* \) or (in the case of \( \pi_B \)) \( G_2^* \), including checking that it is of order \( r_G \) in the latter case.

5.5 Note Plaintexts and Memo Fields

As explained in §3.2.1 ‘Note Plaintexts and Memo Fields’ on p. 9, 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 note plaintext (denoted \( np \)) consists of \((v, \rho, r, \text{memo})\).

The usage of the memo field is by agreement between the sender and recipient of the note. The memo field SHOULD be encoded either as:

- a UTF-8 human-readable string [Unicode], padded by appending zero bytes; or
- an arbitrary sequence of 512 bytes starting with a byte value of 0xF5 or greater, which is therefore not a valid UTF-8 string.

In the former case, wallet software is expected to strip any trailing zero bytes and then display the resulting UTF-8 string to the recipient user, where applicable. Incorrect UTF-8-encoded byte sequences should be displayed as replacement characters (U+FFFD).

In the latter case, the contents of the memo field SHOULD NOT be displayed. A start byte of 0xF5 is reserved for use by automated software by private agreement. A start byte of 0xF6 followed by 511 0x00 bytes means “no memo”. A start byte of 0xF6 followed by anything else, or a start byte of 0xF7 or greater, are reserved for use in future Zcash protocol extensions.

Other fields are as defined in §3.2 ‘Notes’ on p. 9.

The encoding of a note plaintext consists of:

- 8-bit 0x00
- 64-bit \( v \)
- 256-bit \( \rho \)
- 256-bit \( r \)
- memo (512 bytes)

- A byte, 0x00, indicating this version of the encoding of a note plaintext.
5.6 Encodings of Addresses and Keys

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

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

SHA-256 compression outputs are always represented as sequences of 32 bytes.

The language consisting of the following encoding possibilities is prefix-free.

5.6.1 Transparent Addresses

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

The raw encoding of a P2SH address consists of:

- Two bytes \([0x1C, 0xBD]\), indicating this version of the raw encoding of a P2SH address on the production network. (Addresses on the test network use \([0x1C, 0xBA]\) instead.)
- 20 bytes specifying a script hash [Bitc-P2SH].

The raw encoding of a P2PKH address consists of:

- Two bytes \([0x1C, 0xB8]\), indicating this version of the raw encoding of a P2PKH address on the production network. (Addresses on the test network use \([0x1D, 0x25]\) instead.)
- 20 bytes specifying a public key hash, which is a RIPEMD-160 hash [RIPEMD160] of a SHA-256 hash [NIST2015] of an uncompressed ECDSA key encoding.

Notes:

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

These are encoded in the same way as in Bitcoin [Bitc-Base58], for both the production and test networks.

5.6.3 Shielded Payment Addresses

A shielded payment address consists of $a_{pk} \cdot B^r_{\ell PRF}$ and $pk_{enc} \cdot KA$. Public.

$a_{pk}$ is a SHA-256 compression output. $pk_{enc}$ is a KA.Public key (see §5.4.4.1 ‘Key Agreement’ on p. 30), for use with the encryption scheme defined in §4.10 ‘In-band secret distribution’ on p. 24. These components are derived from a spending key as described in §4.2 ‘Key Components’ on p. 19.

The raw encoding of a 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 $pk_{enc}$</th>
</tr>
</thead>
</table>

- Two bytes [0x16, 0x9A], indicating this version of the raw encoding of a Zcash shielded payment address on the production network. (Addresses on the test network use [0x16, 0xB6] instead.)
- 32 bytes specifying $a_{pk}$.
- 32 bytes specifying $pk_{enc}$, using the normal encoding of a Curve25519 public key [Bern2006].

**Note:** For addresses on the production network, the lead bytes and encoded length cause the first two characters of the Base58Check encoding to be fixed as “zc”. For the test network, the first two characters are fixed as “zt”.

5.6.4 Incoming Viewing Keys

An incoming viewing key consists of $a_{pk} \cdot B^r_{\ell PRF}$ and $sk_{enc} \cdot KA$. Private.

$a_{pk}$ is a SHA-256 compression output. $sk_{enc}$ is a KA.Private key (see §5.4.4.1 ‘Key Agreement’ on p. 30), for use with the encryption scheme defined in §4.10 ‘In-band secret distribution’ on p. 24. These components are derived from a spending key as described in §4.2 ‘Key Components’ on p. 19.

The raw encoding of an 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 $sk_{enc}$</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 $sk_{enc}$, using the normal encoding of a Curve25519 private key [Bern2006].

$sk_{enc}$ MUST be “clamped” using KA.FormatPrivate as specified in §4.2 ‘Key Components’ on p. 19. That is, a decoded incoming viewing key MUST be considered invalid if $sk_{enc} \neq KA$.FormatPrivate($sk_{enc}$). (KA.FormatPrivate is defined in §5.4.4.1 ‘Key Agreement’ on p. 30.)
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”.

5.6.5 Spending Keys

A spending key consists of $a_k$, which is a sequence of 252 bits (see §4.2 ‘Key Components’ on p. 19). The raw encoding of a spending key consists of:

- 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 $a_k$.

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

Notes:

- If an implementation represents $a_k$ 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”.

5.7 zk-SNARK Parameters

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

```
8bc20a7f013b2b58970cdd2e7ea028975c88ae7ceeb259a5344a16bc2c0eef7 sprout-proving.key
4bd498dae0a0acfd8e98dc3063380d17d9c08dd0918ead18172bd0ae2fc5df82 sprout-verifying.key
```

These parameters were obtained by a multi-party computation described in [GitHub-mpc] and [BGG2016].
6 Consensus Changes from Bitcoin

6.1 Encoding of Transactions

The Zcash transaction format is as follows:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>version</td>
<td>int32_t</td>
<td>Transaction version number; either 1 or 2.</td>
</tr>
<tr>
<td>Varies</td>
<td>tx_in_count</td>
<td>compactSize uint</td>
<td>Number of transparent inputs in this transaction.</td>
</tr>
<tr>
<td>Varies</td>
<td>tx_in</td>
<td>tx_in</td>
<td>Transparent inputs, encoded as in Bitcoin.</td>
</tr>
<tr>
<td>Varies</td>
<td>tx_out_count</td>
<td>compactSize uint</td>
<td>Number of transparent outputs in this transaction.</td>
</tr>
<tr>
<td>Varies</td>
<td>tx_out</td>
<td>tx_out</td>
<td>Transparent outputs, encoded as in Bitcoin.</td>
</tr>
<tr>
<td>4</td>
<td>lock_time</td>
<td>uint32_t</td>
<td>A Unix epoch time (UTC) or block number, encoded as in Bitcoin.</td>
</tr>
<tr>
<td>Varies†</td>
<td>nJoinSplit</td>
<td>compactSize uint</td>
<td>The number of JoinSplit descriptions in vJoinSplit.</td>
</tr>
<tr>
<td>32†</td>
<td>joinSplitPubKey</td>
<td>char[32]</td>
<td>An encoding of a JoinSplitSig public verification key.</td>
</tr>
<tr>
<td>64†</td>
<td>joinSplitSig</td>
<td>char[64]</td>
<td>A signature on a prefix of the transaction encoding, to be verified using joinSplitPubKey.</td>
</tr>
</tbody>
</table>

† The nJoinSplit and vJoinSplit fields are present if and only if version > 1.
‡ The joinSplitPubKey and joinSplitSig fields are present if and only if version > 1 and nJoinSplit > 0.
The encoding of joinSplitPubKey and the data to be signed are specified in §4.6 ‘Non-malleability’ on p. 22.

Consensus rules:

- The transaction version number MUST be greater than or equal to 1.
- If version = 1 or nJoinSplit = 0, then tx_in_count MUST NOT be 0.
- A transaction with one or more inputs from coinbase transactions MUST have no transparent outputs (i.e. tx_out_count MUST be 0).
- If nJoinSplit > 0, then joinSplitSig MUST represent a valid signature over dataToBeSigned as defined in §4.6 ‘Non-malleability’ on p. 22.
- If nJoinSplit > 0, then joinSplitPubKey MUST represent a valid Ed25519 public key encoding as specified in §5.4.5 ‘JoinSplit Signature’ on p. 30.
• The encoded size of the transaction **MUST** be less than or equal to 100000 bytes.

• A coinbase transaction **MUST NOT** have any JoinSplit descriptions.

• A transaction **MUST NOT** spend an output of a coinbase transaction (necessarily a transparent output) from a block less than 100 blocks prior to the spend.

• TODO: Other rules inherited from Bitcoin.

**Notes:**

• The semantics of transactions with transaction version number not equal to either 1 or 2 is not currently defined. Miners **MUST NOT** create blocks containing such transactions.

• The exclusion of transactions with transaction version number greater than 2 is not a consensus rule. Such transactions may exist in the block chain and **MUST** be treated identically to version 2 transactions.

• Note that a future hard fork might use any transaction version number. It is likely that a hard fork that changes the transaction version number will also change the transaction format, and software that parses transactions **SHOULD** take this into account.

• The version field is a signed integer. (It was incorrectly specified as unsigned in a previous version of this specification.) A future hard fork might use negative values for this field, or otherwise change its interpretation.

• A transaction version number of 2 does not have the same meaning as in Bitcoin, where it is associated with support for OP_CHECKSEQUENCEVERIFY as specified in [BIP-68]. Zcash was forked from Bitcoin v0.11.2 and does not currently support BIP 68, or the related BIPs 9, 112 and 113.

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

• **Transaction version 0** is not supported.

• A version 1 transaction is equivalent to a version 2 transaction with nJoinSplit = 0.

• The nJoinSplit, vJoinSplit, joinSplitPubKey, and joinSplitSig fields have been added.

• In Zcash it is permitted for a transaction to have no transparent inputs provided that nJoinSplit > 0.

• A consensus rule limiting transaction size has been added. In Bitcoin there is a corresponding standard rule but no consensus rule.

Software that creates transactions **SHOULD** use version 1 for transactions with no JoinSplit descriptions.
6.2 Encoding of JoinSplit Descriptions

An abstract *JoinSplit description*, as described in §3.5 ‘JoinSplit Transfers and Descriptions’ on p. 11, 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_t</td>
<td>A value $v_{\text{pub old}}$ that the <em>JoinSplit</em> transfer removes from the transparent value pool.</td>
</tr>
<tr>
<td>8</td>
<td>vpub_new</td>
<td>uint64_t</td>
<td>A value $v_{\text{pub new}}$ that the <em>JoinSplit</em> transfer inserts into the transparent value pool.</td>
</tr>
<tr>
<td>32</td>
<td>anchor</td>
<td>char[32]</td>
<td>A root $rt$ of the 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][Nold]</td>
<td>A sequence of <em>nullifiers</em> of the input notes $n_{1\ldots N}^{\text{old}}$.</td>
</tr>
<tr>
<td>64</td>
<td>commitments</td>
<td>char[32][Nnew]</td>
<td>A sequence of <em>note commitments</em> for the output notes $c_{1\ldots N}^{\text{new}}$.</td>
</tr>
<tr>
<td>32</td>
<td>ephemeralKey</td>
<td>char[32]</td>
<td>A Curve25519 public key $\text{epk}$.</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 description</em>.</td>
</tr>
<tr>
<td>64</td>
<td>vmacs</td>
<td>char[32][Nold]</td>
<td>A sequence of message authentication tags $h_{1\ldots N}^{\text{old}}$ that bind $h_{\text{sig}}$ to each $a_{\text{sk}}$ of the <em>JoinSplit description</em>.</td>
</tr>
<tr>
<td>296</td>
<td>zkproof</td>
<td>char[296]</td>
<td>An encoding of the zero-knowledge proof $\pi_{\text{ZKJoinSplit}}$ (see §5.4.8.1 ‘PHGR13’ on p. 32).</td>
</tr>
<tr>
<td>1202</td>
<td>encCiphertexts</td>
<td>char[601][Nnew]</td>
<td>A sequence of ciphertext components for the encrypted output notes, $C_{1\ldots N}^{\text{enc}}$.</td>
</tr>
</tbody>
</table>

The *vmacs* field encodes $h_{1\ldots N}^{\text{old}}$ which are computed as described in §4.6 ‘Non-malleability’ on p. 22.

The *ephemeralKey* and *encCiphertexts* fields together form the transmitted notes ciphertext, which is computed as described in §4.10 ‘In-band secret distribution’ on p. 24.

Consensus rules applying to a *JoinSplit description* are given in §4.3 ‘JoinSplit Descriptions’ on p. 19.
6.3 Block Header

The Zcash block header format is as follows:

<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_t</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</td>
<td>char[32]</td>
<td>A reserved field which should be ignored.</td>
</tr>
<tr>
<td>4</td>
<td>nTime</td>
<td>uint32_t</td>
<td>The block time 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_t</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.</td>
</tr>
<tr>
<td>32</td>
<td>nNonce</td>
<td>char[32]</td>
<td>An arbitrary field miners 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 §6.4.3 ‘Difficulty adjustment’ on p. 43, and let PoWMedianBlockSpan be the constant defined in §5.3 ‘Constants’ on p. 26.

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 §6.4.2 ‘Difficulty filter’ on p. 43.
- solution MUST represent a valid Equihash solution as defined in §6.4.1 ‘Equihash’ on p. 42.
- nTime MUST be strictly greater than the median time of the previous PoWMedianBlockSpan blocks.
- The size of a block MUST be less than or equal to 2000000 bytes.
- TODO: Other rules inherited from Bitcoin.
In addition, a full validator **MUST NOT** accept blocks with nTime more than two hours in the future according to its clock. This is not strictly a consensus rule because it is nondeterministic, and clock time varies between nodes. Also note that a block that is rejected by this rule at a given point in time may later be accepted.

**Notes:**

- The semantics of blocks with block version number not equal to 4 is not currently defined. Miners **MUST NOT** create such blocks, and **SHOULD NOT** mine other blocks on top of them.

- The exclusion of blocks with block version number greater than 4 is not a consensus rule; such blocks may exist in the block chain and **MUST** be treated identically to version 4 blocks by full validators. Note that a future hard fork might use block version number either greater than or less than 4. It is likely that such a hard fork 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 incorrectly specified as unsigned in a previous version of this specification.) A future hard fork might use negative values for this field, or otherwise change its interpretation.

- There is no relation between the values of the version field of a transaction, and the nVersion field of a block header.

- Like other serialized fields of type `compactSize uint`, the solutionSize field **MUST** be encoded with the minimum number of bytes (3 in this case), and other encodings **MUST** be rejected. This is necessary to avoid a potential attack in which a miner could test several distinct encodings of each Equihash solution against the difficulty filter, rather than only the single intended encoding.

- As in Bitcoin, the nTime field **MUST** represent a time strictly greater than the median of the timestamps of the past PoWMedianBlockSpan blocks. The Bitcoin Developer Reference [Bitc-Block] was previously in error on this point, but has now been corrected.

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

- **Block versions** less than 4 are not supported.

- The hashReserved, solutionSize, and solution fields have been added.

- The type of the nNonce field has changed from `uint32_t` to `char[32]`.

- The maximum block size has been doubled to 200000 bytes.

### 6.4 Proof of Work

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

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

- The solution field encodes a valid Equihash solution according to §6.4.1 ‘Equihash’ on p. 42.

- The block header satisfies the difficulty check according to §6.4.2 ‘Difficulty filter’ on p. 43.
6.4.1 Equihash

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

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

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

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

Let $\text{powheader} := \begin{array}{c|c|c} 32$-bit nVersion & 256-bit hashPrevBlock & 256-bit hashMerkleRoot \\ \hline 256-bit hashReserved & 32-bit nTime & 32-bit nBits \\ \hline 256-bit nNonce & & \\ \end{array}$

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

EquihashGen is instantiated in §5.4.1.5 ‘Equihash Generator’ on p. 28.

Define $\text{I2BEBSP} : (u : N) \times \{0..2^u - 1\} \rightarrow \mathbb{B}^{|u|}$ as in §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 26.

A valid Equihash solution is then a sequence $i : \{1..N\}^{2^k}$ that satisfies the following conditions:

**Generalized Birthday condition** $\bigoplus_{j=1}^{2^k} X_{ij} = 0$.

**Algorithm Binding conditions**

- For all $r \in \{1..k-1\}$, for all $w \in \{0..2^{k-r} - 1\}$: $\bigoplus_{j=1}^{2^r} X_{w \cdot 2^{r+j}}$ has $\frac{n \cdot r}{k+1}$ leading zeros; and
- For all $r \in \{1..k\}$, for all $w \in \{0..2^{k-r} - 1\}$: $i_{w \cdot 2^{r+1} + \ldots + 2^{r+2} - 1} < i_{w \cdot 2^{r+2} + \ldots + 2^{r+3} + 1 \ldots - 2^{r+2} - 1}$ 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:

\[
\begin{array}{cccc}
\text{I2BEBSP}_{21}(i_1 - 1) & \text{I2BEBSP}_{21}(i_2 - 1) & \ldots & \text{I2BEBSP}_{21}(i_{512} - 1)
\end{array}
\]
Recall from §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 26 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>I2BEBSP(_{21}(68))</th>
<th>I2BEBSP(_{21}(41))</th>
<th>I2BEBSP(_{21}(2^{21} - 1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 000000000000</td>
<td>00000000 000000000000</td>
<td>11111111 111111111111</td>
</tr>
<tr>
<td>8-bit 0</td>
<td>8-bit 0</td>
<td>8-bit 255</td>
</tr>
<tr>
<td>8-bit 32</td>
<td>8-bit 10</td>
<td>8-bit 127</td>
</tr>
</tbody>
</table>

and so the first 7 bytes of \( \text{solution} \) would be \([0, 2, 32, 0, 10, 127, 255]\).

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

### 6.4.2 Difficulty filter

Let \( \text{ToTarget} \) be as defined in §6.4.4 ‘\( n\text{Bits conversion} \)’ on p. 44.

Difficulty is defined in terms of a target threshold, which is adjusted for each block according to the algorithm defined in §6.4.3 ‘Difficulty adjustment’ on p. 43.

The difficulty filter is unchanged from \( \text{Bitcoin} \), and is calculated using \( \text{SHA-256d} \) on the whole block header (including \( \text{solutionSize} \) and \( \text{solution} \)). 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 \( \text{ToTarget}(\text{nBits}) \).

### 6.4.3 Difficulty adjustment

\( \text{Zcash} \) uses a difficulty adjustment algorithm based on DigiShield v3/v4 [DigiByte-PoW], with simplifications and altered parameters, to adjust difficulty to target the desired 2.5-minute block time. Unlike \( \text{Bitcoin} \), the difficulty adjustment occurs after every block.

The constants \( \text{PoWLimit} \), \( \text{PoWAveragingWindow} \), \( \text{PoWMaxAdjustDown} \), \( \text{PoWMaxAdjustUp} \), \( \text{PoWDampingFactor} \), and \( \text{PoWTargetSpacing} \) are instantiated in §5.3 ‘Constants’ on p. 26.

Let \( \text{ToCompact} \) and \( \text{ToTarget} \) be as defined in §6.4.4 ‘\( n\text{Bits conversion} \)’ on p. 44.

Let \( \text{nTime}(\text{height}) \) be the value of the \( \text{nTime} \) field in the header of the block at block height \( \text{height} \).

Let \( \text{nBits}(\text{height}) \) be the value of the \( \text{nBits} \) field in the header of the block at block height \( \text{height} \).

*Block header* fields are specified in §6.3 ‘\( \text{Block Header} \)’ on p. 40.

Define:

\[
\text{mean}(S) := \left( \sum_{i=1}^{\text{length}(S)} S_i \right) / \text{length}(S).
\]

\[
\text{median}(S) := \text{sorted}(S)_{\text{ceiling}(\text{length}(S)/2)}
\]

\[
\text{bound}_{\text{upper}}(x) := \max(\text{lower}, \min(\text{upper}, x))
\]

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

\[
\text{AveragingWindowTimespan} := \text{PoWAveragingWindow} \cdot \text{PoWTargetSpacing}
\]

43
MinActualTimespan := floor(AveragingWindowTimespan \cdot \left(1 - \text{PowMaxAdjustUp}\right))
\text{MaxActualTimespan} := floor(AveragingWindowTimespan \cdot \left(1 + \text{PowMaxAdjustDown}\right))
\text{MedianTime}(height) := \text{median}\left[ n\text{Time}(i) \text{ for } i \text{ from } \max(0, height - \text{PowMedianBlockSpan}) \text{ up to } height - 1 \right]
\text{ActualTimespan}(height) := \text{MedianTime}(height) - \text{MedianTime}(height - \text{PowAveragingWindow})
\text{ActualTimespanDamped}(height) := \text{AveragingWindowTimespan} + \text{trunc}\left(\frac{\text{ActualTimespan}(height) - \text{AveragingWindowTimespan}}{\text{PowDampingFactor}}\right)
\text{ActualTimespanBounded}(height) := \text{bound}_{\min\text{ActualTimespan}}^{\max\text{ActualTimespan}}(\text{ActualTimespanDamped}(height))
\text{MeanTarget}(height) := \begin{cases} 
\text{PowLimit}, & \text{if } height \leq \text{PowAveragingWindow} \\
\text{mean}\left[ \text{ToTarget}(n\text{Bits}(i)) \text{ for } i \text{ from } height - \text{PowAveragingWindow} \text{ up to } height - 1 \right], & \text{otherwise}
\end{cases}
\text{Threshold}(height) := \begin{cases} 
\text{PowLimit}, & \text{if } height = 0 \\
\min(\text{PowLimit}, \text{floor}\left(\frac{\text{MeanTarget}(height)}{\text{AveragingWindowTimespan}}\right) \cdot \text{ActualTimespanBounded}(height)), & \text{otherwise}
\end{cases}
\text{ThresholdBits}(height) := \text{ToCompact}(\text{Threshold}(height)).

The \textit{target threshold} for a given \textit{block height} \( \text{height} \) is then calculated as:

\[
\text{Threshold}(height) := \begin{cases} 
\text{PowLimit}, & \text{if } height = 0 \\
\min(\text{PowLimit}, \text{floor}\left(\frac{\text{MeanTarget}(height)}{\text{AveragingWindowTimespan}}\right) \cdot \text{ActualTimespanBounded}(height)), & \text{otherwise}
\end{cases}
\]

\text{ThresholdBits}(height) := \text{ToCompact}(\text{Threshold}(height)).

\textbf{Note:} The convention used for the height parameters to \text{MedianTime}, \text{ActualTimespan}, \text{ActualTimespanDamped}, \text{ActualTimespanBounded}, \text{MeanTarget}, \text{Threshold}, and \text{ThresholdBits} is that these functions use only information from \textit{blocks preceding} the given \textit{block height}.

\subsection{6.4.4 \text{nBits} conversion}

Deterministic conversions between a \textit{target threshold} and a "compact" \textit{nBits} value are not fully defined in the Bitcoin documentation \cite{Bitc-nBits}, and so we define them here:

\[
\text{size}(x) := \text{ceiling}\left(\frac{\text{bitlength}(x)}{8}\right)
\]

\[
\text{mantissa}(x) := \text{floor}\left(x \cdot 256^{3 - \text{size}(x)}\right)
\]

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

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

\subsection{6.4.5 Definition of Work}

As explained in §3.3 \textit{‘The Block Chain'} on p.10, a node chooses the "best" \textit{block chain} visible to it by finding the chain of valid \textit{blocks} with the greatest total work.

Let \text{ToTarget} be as defined in §6.4.4 \textit{‘nBits conversion'} on p.44.

The work of a \textit{block} with value \( n\text{Bits} \) for the \textit{nBits} field in its \textit{block header} is defined as \( \text{floor}\left(\frac{2^{256}}{\text{ToTarget}(n\text{Bits}) + 1}\right) \).
### 6.5 Calculation of Block Subsidy and Founders' Reward

§ 3.8 ‘Block Subsidy and Founders’ Reward’ on p. 12 defines the block subsidy, miner subsidy, and Founders’ Reward. Their amounts in satoshi are calculated from the block height using the formulae below. The constants SlowStartInterval, HalvingInterval, MaxBlockSubsidy, and FoundersFraction are instantiated in §5.3 ‘Constants’ on p. 26.

- **SlowStartShift** \( : \mathbb{N} := \frac{\text{SlowStartInterval}}{2} \)
- **SlowStartRate** \( : \mathbb{N} := \frac{\text{MaxBlockSubsidy}}{\text{SlowStartInterval}} \)
- **Halving(\text{height})** := floor\( \left( \frac{\text{height} - \text{SlowStartShift}}{\text{HalvingInterval}} \right) \)
- **BlockSubsidy(\text{height})** := \[
\begin{cases}
\text{SlowStartRate} \cdot \text{height}, & \text{if height} < \frac{\text{SlowStartInterval}}{2} \\
\text{SlowStartRate} \cdot (\text{height} + 1), & \text{if } \frac{\text{SlowStartInterval}}{2} \leq \text{height} < \text{SlowStartInterval} \\
\text{floor} \left( \frac{\text{MaxBlockSubsidy}}{\text{Halving(\text{height})}} \right), & \text{otherwise}
\end{cases}
\]
- **FoundersReward(\text{height})** := \[
\text{blockSubsidy(\text{height})} \cdot \text{FoundersFraction}, \text{if height} < \text{SlowStartShift} + \text{HalvingInterval}
\]
- **MinerSubsidy(\text{height})** := \( \text{blockSubsidy(\text{height})} - \text{FoundersReward(\text{height})} \).

### 6.6 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...NumFounderAddresses is:

```
[ "t3Vz22vK5z2LcKE5g16yv4FFneEL1z9gOjd", "t3fqvkzrrNaMcamkMQwAyHRjfdQMe7vDTR", "t3Tg29ZT2CTSK44AnUpi6qNaNhtceC7Pyf", "t3SpkcPQfuRYHeP5vz3Pv86PkgOs9m9Kmx", "t3ayBkZ4w6kJxynzHOFUSgXRKtegXbXmg", "t3adJBusa21u7NhbrBTM3z3k3tSB34WGB", "t3KaLJagS5bySrfaG66uEhS9z5qzv8zT2", "t3RYnscb6nEhKiva3ZPf5rS7cyh1CtAR5", "t3Ut4kUqP25U6PFW7pB5ULuQyC12q36kPQX", "t3ZnCMagv6C6yHmIvWtrx3aIN9S3Q5pn", "t3fpC9cB3eSiy64BShxfATQKLGqQroBD", "t3CwZfKNN2v2XAHBQweemw6pxrKFhks3B", "t3YcoujXpspWy77rWUeGkxFEwzQnSpG4", "t3LvCLgcbr6nBrU5S5WkgyVrZcZumTR4", "t3VvHwa7r30y67Tu4LZKCGwa2J6eGhShi", "t3eF9X6x2S07HcvTjZfEzwrVrZquXRlNe", "t3esCNwmcycB19qFyTbYhTqmXYZ29Aw3K", "t3Mj7NhYe2e27ylSuuQPPjuVek81W3VbJ", "t3gWxDrC67YNoBpJnVrrWLaWxPqZLxrYY", "t3LTWeoxeWPbmdkUD3NWbqu4Wka2zPrZ", "t3P5KKX7gXyFsa5JlPuruQEX8yFg3T7q", "t3fJt3nCwesEpzmD5V2JgQfFg74dV8C9", "t3Ronzuq7afkF7156Z6A4v1i4iRNEn41Ij", "t3fJZ5jYsyxDtvNwBemovJAc6j4jgyG", "t3PbBQ77jP7FPGBUuz75h65aczpHgphg2OJ", "t3wKQDXcijL5x7rwFemf1MTL9zwVjKUFp", "t3Y9FNi26J7UTAC4moaEtlBlb08K51BeME", "t3a3NRLLeL2y8xjCpZzFwVz7CsTwBec", "t3QGDeaev5zvAAKHT8TeQo2BDLxelF1umBn", "t3rykhx1UFrgxMrByrAe2STxRKF7L7G", "t3aaW4aTdp7a8d1V7E1Bod2yh8egghHMaR", "t3YEia6e6uEjXwFV25vZtu1fn3yKgzMQnY", "t3Gly1Uut3hPbmdmDvTuCTPUCwDAtJL2iQP", "t3dPwneq6YqPuy1GcXbBzry91uyHxM4", "t3QZXHdPh2huv4I0e7276kRlWwF4pd", "t3enbACri1Z7de8ePomVgK7wv79N9F"J", "t3PKLg77ITfr112nswBwToXsd77ynb2gJY", "t3LQTUDEoe7ZhbvdrVr4vnaoNahCr2f40FH", "t3fncDBuyvcbCtsD29q3L6rxjyfPBBL", "t3dkojUU2Emj28nHV8TveVLEUau1M1Fe", "t3aK662W1oIfg6c19rzi4gjfpkI3n079M", "t3MEYDYP9w8i63KtpUpQd6y62MW2BtFr", "t3WDPflTk3437mNMpTqk2AeodQeQaB3K7Y", "t3PSn5TbMMAW7Eu36DYctFezRzpn1hZ", "t3R3Y5vBfEr6l86WfJpBjLnxSUQsmNmpFv", "t3Pcm737EsVgTBhsu2NekTJeG92mvYyO" ]
```
For the test network, FounderAddressList starts with:

```
[ "t2UNzUUx8mWBCRYPrezva363EYYyEpHoky", "t2N9PHWk9xjy9gYgiin1u3a3ekJqtE543", "t2N9QjYMqhFndDHgVYw4ZNdIsaA6K7x2", "t2E2Ng7hHVqq9Jw5jcvjBSxTM2a9USNfhy", "t2BkYdVChzvTJJUTx4yZB8qeeqDQqsPB8o", "t2J8q1xH1EuijG52MfExyyjYtN3VgyxHkD", "t2Crq5mdXm37kX6c8Z6tY6ye3t2CNBF", "t2EaMPUIQ1khqcpC5UEkF42CAFKJqXCKXC9", "t2F9dtdQc63JDyrhnpzYTJcSr5Mkkd1", "t2LPRmnmY3C481GzB6xUGcoovfyBtBC", "t26xfsoW9U9pe5o3C7YbD4QSeSxtp", "t2D3k4NdErd66YxtxEDft9xuLoK7CD70", "t2DWBXbKXmvdmsMiivNzutaQGmoRjRn", "t2C3k9F9iQXfB54B9zGw0dALQ54zqjpuGQ", "t2MnT5zu9HSKppRyUNw0p8MUueuSGBh", "t2AREsWdoW1F8EQYsCskjgbmSvkKeUk", "t2F4wKcK3ZFTLj4jezUkKwY9KBHm5UT", "t2K3fdV16R5tuRLphKyoYxyZbyWGbDNY", "t2VE3kIHYhGcYz3nDw6ESWtaCQHQwv9WC", "t2F8XouqDM6q6ZzEvXQXH1TjwZRHRwq8G", "t2BS7Mrbaef3f4xxmxvBiFVYXVrRBNz6Qj", "t2FuSwolzC5uFw2zUoRrRexzAb9qy4jbn", "t2SX3U90nT6g5D5b14TcS6Jr3rpr8J3C6", "t2v51g2NSoJkR7L4b9f7T2zbZvFwCq2zF", "t2FyTsljdm4jeWv4r4xj7FAKujidbr14R", "t2YEy BjEkmmpqHey8UN4kq8p favoredM76D11Le", "t2NQTtStZhtJECNFT3dUBLyaAEXPCcmkka", "t2GSWZJZoesXFpTWWKFnS5gujlyY2BU2", "t2RpfkzyrLRevGM3w9aWdqMX6bd8uuAK3v", "t2Jzj4qnuXtTSN7kY7y5k6eUBR6Kyovfhd", "t2AEaeF7c2ieTnsKXmkG2bZNCkiwvZ63oPN", "t2NNs3ZGFZ9sNj2wvmd8BSwSfWtETLglR8", "t2ECCPQVcxUCSSsQdpNqguEPE14HsVfCUn", "t2JabDukG8TaqvKYqfD3rqkdKHP6hwXV", "t2FGzw5Zdc8Cyc982KZmryGv160KcmYi9m", "t2DUD6z21FtEfn42oLp55Gnb6g313yj9", "t2UjVsd3zheHPAgkuX8QW2C19XhQ8eWvp", "t2TBUaEHLYHun6i6wXyZxn5Lmy7k5d2A5", "t27ZeuCyhP6eizzWDOc3bGHX7UC99QeAYnC", "t2NYsJSZtLwMLWE6M3BxRb6h27mNCesY", "t2KKJvvyrjVxSeazby9Kke4tq4gsKUm9", "t2J9YYtH3cve1LZjJa4Acuwo80q6jQjTnzp", "t2Qgw4s9paGpPMH15Gzy7cdmyuRFB42", "t2NTDP9mosKpyFPHJmfj5pGCVaU58XGqa", "t29PhDBWQ7nQ4Ejw3EJ6gh8Wv9EqupkmVtrR", "t2Ez9KM8VJLuArcxuEkNRAkhNvidKkzXcj", "t2D5y7J5fpAxajLbGrMBQkFg2mFN8fo3n6cX", "t2UV2wr1PTAuiybpvkF3SdGxUJeZdZzyt" ]
```

Note: For the test network only, the addresses from index 4 onward have been changed from what was implemented at launch. This reflects a hard fork on the test network, starting from block height 53127. [ZcashIssue-2113]

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

Let SlowStartShift be defined as in the previous section.

Define:

```
FounderAddressChangeInterval := ceiling( \frac{SlowStartShift + HalvingInterval}{NumFounderAddresses} )
```

```
FounderAddressIndex(height) := 1 + floor( \frac{height}{FounderAddressChangeInterval} )
```

Let RedeemScriptHash(hash) be the standard redeem script hash, as defined in [Bitc-Multisig], for the P2SH multisig address with Base58Check representation given by FounderAddressList[FounderAddressIndex(height)].

Consensus rule: A coinbase transaction for block height $h \in \{1..SlowStartShift + HalvingInterval − 1\}$ MUST include at least one output that pays exactly FoundersReward(height) zatoshi with a standard P2SH script of the form OP_HASH160 RedeemScriptHash(height) OP_EQUAL as its scriptPubKey.

Notes:

- No Founders’ Reward is required to be paid for height $\geq SlowStartShift + HalvingInterval$ (i.e. after the first halving), or for 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 height $h \in \{1..SlowStartShift + HalvingInterval\}$ 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.
6.7 Changes to the Script System

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

6.8 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. 34.

7 Differences from the Zerocash paper

7.1 Transaction Structure

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

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

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

TODO: Describe changes to signing.

7.2 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 ‘Note Plaintexts and Memo Fields’ on p. 33.

7.3 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 Nold shielded notes as input, and produces up to Nnew shielded notes and a transparent UTXO as output.
Only "Pour" transactions included a zk-SNARK proof.

In Zcash, the sequence of operations added to a transaction (see §7.1 ‘Transaction Structure’ on p. 47) 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.

### 7.4 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 \((a_{sk})\) and \(\rho\). The note commitment is derived from the recipient address component \(a_{pk}\), the value \(v\), and the commitment trapdoor \(r\), as well as \(\rho\). However nothing prevents creating multiple notes with different \(v\) and \(r\) (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 [BCG+2014, Figure 2]), but only one of which can be spent.

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

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

- The Completeness property asserts that a validly received note can be spent provided that its nullifier does not appear on the ledger. This does not take into account the possibility that distinct notes, which are validly received, could have the same nullifier. That is, the security definition depends on a protocol detail – nullifiers – that is not part of the intended abstract security property, and that could be implemented incorrectly.

- The Balance property only asserts that an adversary cannot obtain more funds than they have minted or received via payments. It does not prevent an adversary from causing others’ funds to decrease. In a Faerie Gold attack, an adversary can cause spending of a note to reduce (to zero) the effective value of another note for which the attacker does not know the spending key, which violates an intuitive conception of global balance.

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

It would be possible to address the attack by requiring that a recipient remember all of the \(\rho\) 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.

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 §4.4.1 ‘Dummy Notes’ on p. 21). The nullifiers are used as input to \(h_{\text{Sig}}\)CRH to derive a public value \(h_{\text{Sig}}\) which uniquely identifies the transaction, as described in §4.3 ‘JoinSplit Descriptions’ on p. 19. \(h_{\text{Sig}}\) was already used in Zerocash in a way that requires it to be unique in order to maintain indistinguishability of JoinSplit descriptions; adding the nullifiers to the input of the hash used to calculate it has the effect of making this uniqueness property robust even if the transaction creator is an adversary.

The \(\rho\) value for each output note is then derived from a random private seed \(\xi\) and \(h_{\text{Sig}}\) using PRF\(_{\xi}\). The correct construction of \(\rho\) for each output note is enforced by §4.9.1 ‘Uniqueness of \(\rho_i^{\text{new}}\)’ on p. 24 in the JoinSplit statement.
Now even if the creator of a JoinSplit description does not choose \( q \) randomly, uniqueness of nullifiers and collision resistance of both hSigCRH and PRF\( ^p \) will ensure that the derived \( \rho \) values are unique, at least for any two JoinSplit descriptions that get into a valid block chain. This is sufficient to prevent the Faerie Gold attack.

A variation on the attack attempts to cause the nullifier of a sent note to be repeated, without repeating \( \rho \). However, since the nullifier is computed as PRF\( _{a_{pk}}^nf \) (\( \rho \)), this is only possible if the adversary finds a collision (across both inputs) on PRF\( _{a_{pk}}^nf \), which is assumed to be infeasible — see §4.1.2 ‘Pseudo Random Functions’ on p. 13.

Crucially, "nullifier integrity" (§4.9.1 ‘Nullifier integrity’ on p. 24) is enforced whether or not the enforceMerklePath, flag is set for an input note. If this were not the case then an adversary could perform the attack by creating a zero-valued note with a repeated nullifier, since the nullifier does not depend on the value.

Nullifier integrity also prevents a “roadblock attack” in which the attacker sees a victim’s transaction, and is able to publish another transaction that is mined first and blocks the victim's transaction. This attack would be possible if the public value(s) used to enforce uniqueness of \( \rho \) could be chosen arbitrarily by the transaction creator: the victim’s transaction, rather than the attacker’s, would be considered to be repeating these values. In the chosen solution that uses nullifiers for these public values, they are enforced to be dependent on spending keys controlled by the original transaction creator (whether or not each input note is a dummy), and so a roadblock attack cannot be performed by another party who does not know these keys.

### 7.5 Internal hash collision attack and fix

The Zerocash security proof requires that the composition of COMM\( _{v} \) and COMM\( _{\rho} \) is a computationally binding commitment to its inputs \( a_{pk}, v \), and \( \rho \). However, the instantiation of COMM\( _{v} \) and COMM\( _{\rho} \) in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of \( a_{pk} \) and \( \rho \) is truncated to 128 bits (motivated by providing statistical hiding security). This allows an attacker, with a work factor on the order of \( 2^{64} \), to find distinct pairs \( (a_{pk}, \rho) \) and \( (a'_{pk}, \rho') \) with colliding outputs of the truncated hash, and therefore the same note commitment. This would have allowed such an attacker to break the Balance property by double-spending notes, potentially creating arbitrary amounts of currency for themselves [HW2016].

Zcash uses a simpler construction with a single SHA-256 evaluation for the commitment. The motivation for the nested construction in Zerocash was to allow Mint transactions to be publicly verified without requiring a zero-knowledge proof (as described under step 3 in [BCG+2014, section 1.3]). Since Zcash combines “Mint” and “Pour” transactions into generalized JoinSplit transfers, and each transfer always uses a zero-knowledge proof, it does not require the nesting. A side benefit is that this reduces the cost of computing the note commitments: it reduces the number of SHA256Compress evaluations needed to compute each note commitment from three to two, saving a total of four SHA256Compress evaluations in the JoinSplit statement.

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

### 7.6 Changes to PRF inputs and truncation

The format of inputs to the PRFs instantiated in §5.4.2 ‘Pseudo Random Functions’ on p. 29 has changed relative to Zerocash. There is also a requirement for another PRF, PRF\( ^p \), which must be domain-separated from the others.

In the Zerocash protocol, \( \rho_i^{old} \) is truncated from 256 to 254 bits in the input to PRF\( ^se \) (which corresponds to PRF\( ^nf \) in Zerocash). Also, hSig 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 [BCG+2014, Appendix D].
In more detail:

- In the argument relating $H$ and $D_2$, it is stated that in $D_2$, “for each $i \in \{1, 2\}$, $s_{n_i} := \text{PRF}_{a_{sk}}^r (\rho)$ for a random (and not previously used) $\rho$. It is also argued that “the calls to PRF$^r_{a_{sk}}$ are each by definition unique”. The latter assertion depends on the fact that $\rho$ is “not previously used”. However, the argument is incorrect because the truncated input to PRF$^r_{a_{sk}}$, i.e. $|\rho|_{254}$, may repeat even if $\rho$ does not.

- In the same argument, it is stated that “with overwhelming probability, $h_{\text{Sig}}$ is unique”. In fact what is required to be unique is the truncated input to PRF$^k$, i.e. $|h_{\text{Sig}}|_{253} = |\text{CRH}(pk_{\text{Sig}})|_{253}$. In practice this value will be unique under a plausible assumption on CRH provided that $pk_{\text{Sig}}$ is chosen randomly, but no formal argument for this is presented.

Note that $\rho$ is truncated in the input to PRF$^s$ but not in the input to COMM, 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, but not in the input to PRF$^s$. 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.

For resistance to Faerie Gold attacks as described in §7.4 ‘Faerie Gold attack and fix’ on p. 48, Zcash depends on collision resistance of $h_{\text{Sig}}$CRH (instantiated using BLAKE2b-256) and PRF$^p$ (instantiated using SHA256Compress). Collision resistance of a truncated hash does not follow from collision resistance of the original hash, even if the truncation is only by one bit. This motivated avoiding truncation along any path from the inputs to the computation of $h_{\text{Sig}}$ to the uses of $\rho$.

Since the PRFs are instantiated using SHA256Compress which has an input block size of 512 bits (of which 256 bits are used for the PRF input and 4 bits are used for domain separation), it was necessary to reduce the size of the PRF key to 252 bits. The key is set to $a_{sk}$ in the case of PRF$^{\text{addr}}$, PRF$^{\text{nf}}$, and PRF$^p$, and to $q$ (which does not exist in Zerocash) for PRF$^p$, 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.

### 7.7 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, and the authenticated encryption algorithm Aead_chacha20_poly1305. This scheme is still loosely based on ECIES, and on the crypto_box_seal scheme defined in libsodium [libsodium SEAL].

The motivations for this change were as follows:

- The Zerocash paper did not specify the curve to be used. We believe that Curve25519 has significant side-channel resistance, performance, implementation complexity, and robustness advantages over most other available curve choices, as explained in [Bern2006].

- ECIES permits many options, which were not specified. There are at least –counting conservatively– 576 possible combinations of options and algorithms over the four standards (ANSI X9.63, IEEE Std 1363a–2004, ISO/IEC 18033–2, and SEC 1) that define ECIES variants [MAEA2010].

- Although the Zerocash paper states that ECIES satisfies key privacy (as defined in [BBDP2001]), it is not clear that this holds for all curve parameters and key distributions. For example, if a group of non-prime order is used, the distribution of ciphertexts could be distinguishable depending on the order of the points representing the ephemeral and recipient public keys. Public key validity is also a concern. Curve25519 key agreement is defined in a way that avoids these concerns due to the curve structure and the “clamping” of private keys.

- Unlike the DHAES/DHIES proposal on which it is based [ABR1999], ECIES does not require a representation of the sender’s ephemeral public key to be included in the input to the KDF, which may impair the security properties of the scheme. (The Std 1363a–2004 version of ECIES [IEEE2004] has a “DHAES mode” that allows
this, but the representation of the key input is underspecified, leading to incompatible implementations.)
The scheme we use has both the ephemeral and recipient public key encodings—which are unambiguous for
Curve25519—and also \( h_{\text{Sig}} \) and a nonce as described below, as input to the KDF. Note that being able to break
the Elliptic Curve Diffie–Hellman Problem on Curve25519 (without breaking \text{AEAD\_CHACHA20\_POLY1305} as
an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the transmitted
notes ciphertext unless \( p_{\text{enc}} \) is known or guessed.

- The KDF also takes a public seed \( h_{\text{Sig}} \) as input. This can be modeled as using a different “randomness extrac-
tor” for each \text{JoinSplit transfer}, which limits degradation of security with the number of \text{JoinSplit transfers}. This
facilitates security analysis as explained in [DGKMW11]—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 \( h_{\text{Sig}} \) is authenticated, by the \text{zk\-SNARK proof}, as having been chosen with knowledge of \( a_{\text{sk,1..N\_old}} \), so
an adversary cannot modify it in a ciphertext from someone else’s transaction for use in a chosen-ciphertext
attack without detection.

- The scheme used by \text{Zcash} includes an optimization that reuses the same ephemeral key (with different
nonces) for the two ciphertexts encrypted in each \text{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.
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 \text{AEAD\_CHACHA20\_POLY1305} maintains a high concrete security level even under at-
tacks using parallel hardware [Bern2005] in the multi-user setting [Zave2012]. This is especially necessary because
the privacy of \text{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 up-
grade. 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: \( a_{\text{sk}} \) is 252 bits, and \( s_{\text{enc}} \) is no shorter than \( a_{\text{sk}} \).

### 7.8 Omission in Zerocash security proof

The abstract \text{Zerocash} protocol requires \text{PRF}^{\text{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 \text{PRF}^{\text{addr}} such that \( a_{\text{sk}}^1 \) and \( a_{\text{sk}}^2 \) are distinct spending keys for the same
\( a_{\text{pk}} \). Because the note commitment is to \( a_{\text{sk}} \) but the nullifier is computed from \( a_{\text{sk}} \) (and \( p \)), the adversary is able to
to double-spend the note, once with each \( a_{\text{sk}} \). This is not detected because each spend reveals a different nullifier.
The \text{JoinSplit statements} are still valid because they can only check that the \( a_{\text{sk}} \) in the witness is some preimage of
the \( a_{\text{pk}} \) used in the note commitment.

The error is in the proof of Balance in [BCG+14, Appendix D.3]. For the “\( \mathcal{A} \) violates Condition I” case, the proof
says:

“(i) If \( \text{cm}_{\text{old}}^1 = \text{cm}_{\text{old}}^2 \), then the fact that \( \text{sn}_{\text{old}}^1 \neq \text{sn}_{\text{old}}^2 \) implies that the witness \( a \) contains two distinct openings of
\( \text{cm}_{\text{old}}^1 \) (the first opening contains \( (a_{\text{old}}^{\text{sk,1..N\_old}}, \rho_{\text{old}}^{\text{sk,1..N\_old}}) \), while the second opening contains \( (a_{\text{old}}^{\text{sk,2..N\_old}}, \rho_{\text{old}}^{\text{sk,2..N\_old}}) \)). This violates the
binding property of the commitment scheme \text{COMM}.”

In fact the openings do not contain \( a_{\text{old}}^{\text{sk,1..N\_old}} \); they contain \( a_{\text{old}}^{\text{sk,1..N\_old}} \). (In \text{Zcash} \( \text{cm}_{\text{old}}^1 \) opens directly to \( (a_{\text{old}}^{\text{pk,1..N\_old}}, v_{\text{old}}^{\text{pk,1..N\_old}}, \rho_{\text{old}}^{\text{pk,1..N\_old}}) \), and in
\text{Zerocash} it opens to \( (v_{\text{old}}^{\text{pk,1..N\_old}}, \text{COMM}_{\text{sk}}(a_{\text{old}}^{\text{pk,1..N\_old}}, \rho_{\text{old}}^{\text{pk,1..N\_old}})) \).

A similar error occurs in the argument for the “\( \mathcal{A} \) violates Condition II” case.

The flaw is not exploitable for the actual instantiations of \text{PRF}^{\text{addr}} in \text{Zerocash} and \text{Zcash}, which are collision-resistant assuming that \text{SHA256Compress} is.
The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of $\text{PRF}^{\text{addr}}$ (on both its arguments) to argue that distinctness of $\text{old}^{\text{sk}}_1$ and $\text{old}^{\text{sk}}_2$, together with constraint 1(b) of the $\text{JoinSplit statement}$ (see §4.9.1 ‘Spend authority’ on p. 24), implies distinctness of $\text{old}^{\text{pk}}_1$ and $\text{old}^{\text{pk}}_2$, therefore distinct openings of the note commitment when Condition I or II is violated.

7.9 Miscellaneous

- The paper defines a note as $((\text{a}^{\text{pk}}, \text{pk}_{\text{enc}}), v, \rho, r, s, \text{cm})$, whereas this specification defines it as $(\text{a}^{\text{pk}}, v, \rho, r)$. The instantiation of $\text{COMM}_m$ in section 5.1 of the paper did not actually use $s$, and neither does the new instantiation of $\text{COMM}^{\text{Sprout}}$ in Zcash. $\text{pk}_{\text{enc}}$ is also not needed as part of a note: it is not an input to $\text{COMM}^{\text{Sprout}}$ nor is it constrained by the $\text{Zerocash POUR statement}$ or the $\text{Zcash JoinSplit statement}$. cm can be computed from the other fields.

- The length of proof encodings given in the paper is 288 bytes. This differs from the 296 bytes specified in §5.4.8.1 ‘PHGR13’ on p. 32, 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 $F_q$ can be represented in 254 bits, we prefer to use the standard formats for points defined in [IEEE2004]. The fork of libsnark used by Zcash uses this standard encoding rather than the less efficient (uncompressed) one used by upstream libsnark.

- The range of monetary values differs. In Zcash, this range is $\{0 .. \text{MAX\_MONEY}\}$; in Zerocash it is $\{0 .. 2^{64} − 1\}$. (The $\text{JoinSplit statement}$ still only directly enforces that the sum of amounts in a given $\text{JoinSplit transfer}$ is in the latter range; this enforcement is technically redundant given that the Balance property holds.)

8 Acknowledgements

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

The authors would like to thank everyone with whom they have discussed the Zerocash protocol design: in addition to the inventors, this includes Mike Perry, Isis Lovecruft, Leif Ryge, Andrew Miller, Zooko Wilcox, Samantha Hulsey, Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, Jake Tarren, Solar Designer, Ling Ren, Alison Stevenson, John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, Jack Gavigan, Filippo Valsorda, Zaki Manian, George Tankersley, Tracy Hu, and no doubt others.

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

The Faerie Gold attack was found by Zooko Wilcox; subsequent analysis of variations on the attack was performed by Daira Hopwood and Sean Bowe. The internal hash collision attack was found by Taylor Hornby. The error in the Zerocash proof of Balance relating to collision-resistance of $\text{PRF}^{\text{addr}}$ was found by Daira Hopwood. The errors in the proof of Ledger Indistinguishability mentioned in §7.6 ‘Changes to PRF inputs and truncation’ on p. 49 were also found by Daira Hopwood.

9 Change History

2018.0-beta-14

- Only cosmetic changes to Sprout.

2018.0-beta-13

- Only cosmetic changes to Sprout.
2018.0-beta-12
  • No changes to Sprout.

2018.0-beta-11
  • No changes to Sprout.

2018.0-beta-10
  • Split the descriptions of SHA-256 and SHA256Compress into their own sections. Specify SHA256Compress more precisely.
  • Add Tracy Hu to acknowledgements.

2018.0-beta-9
  • Specify the coinbase maturity rule, and the rule that coinbase transactions cannot contain JoinSplit descriptions.

2018.0-beta-8
  • No changes to Sprout.

2018.0-beta-7
  • Specify the 100000-byte limit on transaction size. (The implementation in zcashd was as intended.)
  • Specify that 0xF6 followed by 511 zero bytes encodes an empty memo field.
  • Reference security definitions for Pseudo Random Functions.
  • Rename clamp to bound and ActualTimespanClamped to ActualTimespanBounded in the difficulty adjustment algorithm, to avoid a name collision with Curve25519 scalar “clamping”.
  • Change uses of the term full node to full validator. A full node by definition participates in the peer-to-peer network, whereas a full validator just needs a copy of the block chain from somewhere. The latter is what was meant.

2018.0-beta-6
  • No changes to Sprout.

2018.0-beta-5
  • Specify more precisely the requirements on Ed25519 public keys and signatures.
2018.0-beta-4

• No changes to Sprout.

2018.0-beta-3

• Explain how the chosen fix to Faerie Gold avoids a potential “roadblock” attack.

2017.0-beta-2.9

• Refer to $sk_{enc}$ as a receiving key rather than as a viewing key.
• Updates for incoming viewing key support.

2017.0-beta-2.8

• Correct the non-normative note describing how to check the order of $\pi_H$.

2017.0-beta-2.7

• Fix an off-by-one error in the specification of the Equihash algorithm binding condition. (The implementation in zcashd was as intended.)
• Correct the types and consensus rules for transaction version numbers and block version numbers. (Again, the implementation in zcashd was as intended.)
• Clarify the computation of $h_i$ in a JoinSplit statement.

2017.0-beta-2.6

• Be more precise when talking about curve points and pairing groups.

2017.0-beta-2.5

• Clarify the consensus rule preventing double-sends.
• Clarify what a note commitment opens to in §7.8 ‘Omission in Zerocash security proof’ on p. 51.
• Correct the order of arguments to $\text{COMM}$ in §5.4.6.1 ‘Note Commitments’ on p. 31.
• Correct a statement about indistinguishability of JoinSplit descriptions.
• Change the Founders’ Reward addresses, for the test network only, to reflect the hard fork described in [ZcashIssue-2113].

2017.0-beta-2.4

• Explain a variation on the Faerie Gold attack and why it is prevented.
• Generalize the description of the InternalH attack to include finding collisions on $(a_{pk}, \rho)$ rather than just on $\rho$.
• Rename $\text{enforce}_i$ to $\text{enforceMerklePath}_i$. 

54
2017.0-beta-2.3
  • Specify the security requirements on the SHA-256 compression function in order for the scheme in §5.4.6.1 ‘Note Commitments’ on p. 31 to be a secure commitment.
  • Specify $G_2$ more precisely.
  • Explain the use of interstitial treestates in chained JoinSplit transfers.

2017.0-beta-2.2
  • Give definitions of computational binding and computational hiding for commitment schemes.
  • Give a definition of statistical zero knowledge.
  • Reference the white paper on MPC parameter generation [BGG2016].

2017.0-beta-2.1
  • $\ell_{\text{Merkle}}$ is a bit length, not a byte length.
  • Specify the maximum block size.

2017.0-beta-2
  • Add abstract and keywords.
  • Fix a typo in the definition of nullifier integrity.
  • Make the description of block chains more consistent with upstream Bitcoin documentation (referring to “best” chains rather than using the concept of a block chain view).
  • Define how nodes select a best chain.

2016.0-beta-1.13
  • Specify the difficulty adjustment algorithm.
  • Clarify some definitions of fields in a block header.
  • Define PRF$^\text{addr}$ in §4.2 ‘Key Components’ on p. 19.

2016.0-beta-1.12
  • Update the hashes of proving and verifying keys for the final Sprout parameters.
  • Add cross references from shielded payment address and spending key encoding sections to where the key components are specified.
  • Add acknowledgements for Filippo Valsorda and Zaki Manian.

2016.0-beta-1.11
  • Specify a check on the order of $\pi_B$ in a zero-knowledge proof.
  • Note that due to an oversight, the Zcash genesis block does not follow [BIP-34].
2016.0-beta-1.10
  • Update reference to the Equihash paper [BK2016]. (The newer version has no algorithmic changes, but the section discussing potential ASIC implementations is substantially expanded.)
  • Clarify the discussion of proof size in “Differences from the Zerocash paper”.

2016.0-beta-1.9
  • Add Founders’ Reward addresses for the production network.
  • Change “protected” terminology to “shielded”.

2016.0-beta-1.8
  • Revise the lead bytes for transparent P2SH and P2PKH addresses, and reencode the testnet Founders’ Reward addresses.
  • Add a section on which BIPs apply to Zcash.
  • Specify that OP_CODESEPARATOR has been disabled, and no longer affects signature hashes.
  • Change the representation type of vpub_old and vpub_new to uint64_t. (This is not a consensus change because the type of v_pub_old and v_pub_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_t.

2016.0-beta-1.7
  • Clarify the consensus rule for payment of the Founders’ Reward, in response to an issue raised by the NCC audit.

2016.0-beta-1.6
  • Fix an error in the definition of the sortedness condition for Equihash: it is the sequences of indices that are sorted, not the sequences of hashes.
  • Correct the number of bytes in the encoding of solutionSize.
  • Update the section on encoding of transparent addresses. (The precise prefixes are not decided yet.)
  • Clarify why BLAKE2b-l is different from truncated BLAKE2b-512.
  • Clarify a note about SU-CMA security for signatures.
  • Add a note about PRF^nf corresponding to PRF^sn in Zerocash.
  • Add a paragraph about key length in §7.7 ‘In-band secret distribution’ on p. 50.
  • Add acknowledgements for John Tromp, Paige Peterson, Maureen Walsh, Jay Graber, and Jack Gavigan.

2016.0-beta-1.5
  • Update the Founders’ Reward address list.
  • Add some clarifications based on Eli Ben-Sasson’s review.
2016.0-beta-1.4
- Specify the *block subsidy*, *miner subsidy*, and the *Founders' Reward*.
- Specify *coinbase transaction* outputs to *Founders' Reward* addresses.
- Improve notation (for example “·” for multiplication and “$T^{[]}$” for sequence types) to avoid ambiguity.

2016.0-beta-1.3
- Correct the omission of *solutionSize* from the *block header* format.
- Document that *compactSize uint* encodings must be canonical.
- Add a note about conformance language in the introduction.
- Add acknowledgements for Solar Designer, Ling Ren and Alison Stevenson, and for the NCC Group and Coinspect security audits.

2016.0-beta-1.2
- Remove *GeneralCRH* in favour of specifying *hSigCRH* and *EquihashGen* directly in terms of *BLAKE2b-ℓ*.
- Correct the security requirement for *EquihashGen*.

2016.0-beta-1.1
- Add a specification of abstract signatures.
- Clarify what is signed in the “Sending Notes” section.
- Specify ZK parameter generation as a randomized algorithm, rather than as a distribution of parameters.

2016.0-beta-1
- Major reorganization to separate the abstract cryptographic protocol from the algorithm instantiations.
- Add type declarations.
- Add a “High-level Overview” section.
- Add a section specifying the *zero-knowledge proving system* and the encoding of proofs. Change the encoding of points in proofs to follow *IEEE Std 1363[a]*.
- Add a section on consensus changes from *Bitcoin*, and the specification of *Equihash*.
- Complete the “Differences from the *Zerocash* paper” section.
- Correct the Merkle tree depth to 29.
- Change the length of *memo fields* to 512 bytes.
- Switch the *JoinSplit signature* scheme to Ed25519, with consequent changes to the computation of $h_{Sig}$.
- 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 $v_{old pub}$ and $v_{new pub}$.
- Clarify cryptographic security requirements and added definitions relating to the in-band secret distribution.
- Add various citations: the “Fixing Vulnerabilities in the Zcash Protocol” and “Why Equihash?” blog posts, several crypto papers for security definitions, the *Bitcoin* whitepaper, the *CryptoNote* whitepaper, and several references to *Bitcoin* documentation.
• Reference the extended version of the Zerocash paper rather than the Oakland proceedings version.
• Add JoinSplit transfers to the Concepts section.
• Add a section on Coinbase Transactions.
• Add acknowledgements for Jack Grigg, Simon Liu, Ariel Gabizon, jl777, Ben Blaxill, Alex Balducci, and Jake Tarren.
• Fix a Makefile compatibility problem with the escaping behaviour of echo.
• Switch to biber for the bibliography generation, and add backreferences.
• Make the date format in references more consistent.
• Add visited dates to all URLs in references.
• Terminology changes.

2016.0-alpha-3.1
• Change main font to Quattrocento.

2016.0-alpha-3
• Change version numbering convention (no other changes).

2.0-alpha-3
• Allow anchoring to any previous output treestate in the same transaction, rather than just the immediately preceding output treestate.
• Add change history.

2.0-alpha-2
• Change from truncated BLAKE2b-512 to BLAKE2b-256.
• Clarify endianness, and that uses of BLAKE2b are unkeyed.
• Minor correction to what SIGHASH types cover.
• Add “as intended for the Zcash release of summer 2016” to title page.
• Require PRF^{addr} to be collision-resistant (see §7.8 ‘Omission in Zerocash security proof’ on p. 51).
• Add specification of path computation for the incremental Merkle tree.
• Add a note in §4.9.1 ‘Merkle path validity’ on p. 23 about how this condition corresponds to conditions in the Zerocash paper.
• Changes to terminology around keys.

2.0-alpha-1
• First version intended for public review.
10 References


Nicolas van Saberhagen. *CryptoNote v 2.0.* Date disputed. URL: [https://cryptonote.org/whitepaper.pdf](https://cryptonote.org/whitepaper.pdf) (visited on 2016-08-17) († p6).

