4.1.5 Key Derivation ................................................................. 12
4.1.6 Signatures ................................................................. 12
4.1.7 Commitment ............................................................... 13
4.1.8 Zero-Knowledge Proving System .................................. 13
4.2 Key Components ............................................................ 14
4.3 JoinSplit Descriptions ..................................................... 14
4.4 Sending Notes ............................................................... 15
  4.4.1 Dummy Notes ............................................................. 16
4.5 Merkle path validity ......................................................... 16
4.6 Non-malleability ............................................................. 17
4.7 Balance ........................................................................ 17
4.8 Note Commitments and Nullifiers .................................... 17
4.9 JoinSplit Statement ......................................................... 17
4.10 In-band secret distribution ............................................... 18
  4.10.1 Encryption ............................................................... 19
  4.10.2 Decryption by a Recipient .......................................... 19
5 Concrete Protocol ............................................................. 20
  5.1 Caution .................................................................... 20
  5.2 Integers, Bit Sequences, and Endianness .......................... 20
  5.3 Constants ................................................................. 20
  5.4 Concrete Cryptographic Functions .................................. 21
    5.4.1 Merkle Tree Hash Function ..................................... 21
    5.4.2 \textit{h}_{\textit{sig}} Hash Function ....................................... 21
    5.4.3 Equihash Generator ................................................. 21
    5.4.4 Pseudo Random Functions ........................................ 22
    5.4.5 Authenticated One-Time Symmetric Encryption .............. 22
    5.4.6 Key Agreement ....................................................... 23
    5.4.7 Key Derivation ....................................................... 23
    5.4.8 Signatures ............................................................. 23
    5.4.9 Commitment .......................................................... 24
  5.5 Note Plaintexts and Memo Fields ................................... 24
  5.6 Encodings of Addresses and Keys ................................... 24
    5.6.1 Transparent Payment Addresses ................................. 25
    5.6.2 Transparent Private Keys ......................................... 25
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. 35, and highlighted in magenta throughout the document.

Technical terms for concepts that play an important role in Zcash are written in slanted text. Italicics 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>. While the production Zcash network has yet to be launched, please feel free to do so in public even if you believe the mistake may indicate a security weakness.

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 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.

---

1 In Zerocash [BCG+2014], notes were called “coins”, and nullifiers were called “serial numbers”.
To each note there is cryptographically associated a note commitment, and a nullifier\(^1\) (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 [Bitcoin–Protocol]. It also contains a sequence of zero or more JoinSplit descriptions. Each of these describes a JoinSplit transfer\(^2\) 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 negligible 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 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 viewing key. This facility is used to communicate encrypted output notes on the block chain to their intended recipient, who can use the viewing 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 [Bitcoin–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.

2 Notation

The notation B means the type of bit values, i.e. \{0, 1\}.

The notation N means the set of nonnegative integers. \(N^+\) means the set of positive integers. \(Q\) means the set of rationals.

The notation \(x : T\) 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.

\(^1\) JoinSplit transfers in Zcash generalize "Mint" and "Pour" transactions in Zerocash; see §7.1 ‘Transaction Structure’ on p. 35 for the differences.
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 \( x \xleftarrow{R} f(s) \).

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

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

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

The notation \( 0x \) followed by a string of \textbf{boldface} hexadecimal digits means the corresponding integer converted from hexadecimal.

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

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

The notation \{\( a..b \)\} means the set or type of integers from \( a \) through \( b \) inclusive.

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

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

The notation \( \text{concat}_{B}(S) \) means the sequence of bits obtained by concatenating the elements of \( S \) viewed as bit sequences. If the elements of \( S \) are byte sequences, they are converted to bit sequences with the \textit{most significant} bit of each byte first.

The notation \( \mathbb{F}_n \) 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 \( \mathbb{F}_n \).

The notation \( a \cdot b \) means the result of multiplying \( a \) and \( b \). This may refer to multiplication of integers, rationals, or finite field elements according to context.

The notation \( a^b \), for \( a \) an integer or finite field element and \( b \) an integer, means the result of raising \( a \) to the exponent \( b \).

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

The notation \( a \oplus b \) means the bitwise exclusive-or of \( a \) and \( b \), defined either on integers or bit sequences according to context.

The notation \( \sum_{i=1}^{N} a_i \) means the sum of \( a_{1..N} \). \( \bigoplus_{i=1}^{N} a_i \) means the bitwise exclusive-or of \( a_{1..N} \).

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

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

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

The following integer constants will be instantiated in §5.3 ‘\textit{Constants}’ on p. 20: \( d_{\text{Merkle}}, N_{\text{old}}, N_{\text{new}}, \ell_{\text{Merkle}}, \ell_{\text{Sig}}, \ell_{\text{PRF}}, \ell_{r}, \ell_{\text{Seed}}, \ell_{a_{\text{sk}}}, \ell_{q}, \text{ MAX\_MONEY, SlowStartInterval, HalvingInterval, MaxBlockSubsidy, NumFounderAddresses.} \)
bit sequence constant Uncommitted : $B^{(\text{Univ})}$ and the rational constant FoundersFraction : $\mathbb{Q}$ will also be defined in that section.

# 3 Concepts

## 3.1 Payment Addresses and Keys

A key tuple $(a_{sk}, s_{kenc}, addr_{pk})$ is generated by users who wish to receive payments under this scheme. The viewing key $s_{kenc}$ and the payment address $addr_{pk} = (a_{pk}, p_{kenc})$ are derived from the spending key $a_{sk}$.

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 of key components](image)

The composition of payment addresses, 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 payment address or viewing key from a spending key.

Users can accept payment from multiple parties with a single payment address $addr_{pk}$ 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 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 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 ($p_{kenc}$) need not be publically distributed; it has the same distribution as the payment address itself. As mentioned above, limiting the distribution of the 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. 18), since an adversary would have to know $p_{kenc}$ in order to exploit a hypothetical weakness in that cryptosystem.

## 3.2 Notes

A note (denoted $n$) is a tuple $(a_{pk}, v, \rho, r)$. It represents that a value $v$ is spendable by the recipient who holds the spending key $a_{sk}$ corresponding to $a_{pk}$, as described in the previous section.

- $a_{pk} : B^{(\text{Univ})}$ is the paying key of the recipient;
- $v : \{0..\text{MAX\_MONEY}\}$ is an integer representing the value of the note in zatoshi ($1 \text{ ZEC} = 10^8 \text{ zatoshi}$);
- $\rho : B^{(|r|)}$ is used as input to $\text{PRF}_{a_{sk}}$ to derive the nullifier of the note.
Commitment
In-band secret distribution
Note Commitment Tree
Sending Notes

Pseudo Random Functions

· \( r : \mathbb{B}^{[\ell]} \) is a random bit sequence used as a commitment trapdoor as defined in §4.1.7 ‘Commitment’ on p. 13.

Let Note be the type of a note, i.e. \( \mathbb{B}^{[\ell_{\text{prev}}]} \times \{0.., \text{MAX\_MONEY} \} \times \mathbb{B}^{[\ell_{\text{prev}}]} \times \mathbb{B}^{[\ell]} \).

Creation of new notes is described in §4.4 ‘Sending Notes’ on p. 15. When notes are sent, only a commitment (see §4.1.7 ‘Commitment’ on p. 13) to the above values is disclosed publically. 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.

The note commitment is computed as \( \text{NoteCommitment}(n) = \text{COMM}_{r}(a_{pk}, v, \rho) \), where \( \text{COMM} \) is instantiated in §5.4.9 ‘Commitment’ on p. 24.

A nullifier (denoted nf) is derived from the \( \rho \) component of a note and the recipient’s spending key, using a Pseudo Random Function (see §4.1.2 ‘Pseudo Random Functions’ on p. 10). Specifically it is derived as \( \text{PRF}_{a_{sk}}(\rho) \) where \( \text{PRF} \) is instantiated in §5.4.4 ‘Pseudo Random Functions’ on p. 22.

A note is spent by proving knowledge of \( \rho \) and \( a_{sk} \) 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 \( \text{pk}^{\text{new}}_{\text{enc}} \), \( N_{\text{new}} \), and the result forms part of a transmitted notes ciphertext (see §4.10 ‘In-band secret distribution’ on p. 18 for further details).

Each note plaintext (denoted np) consists of \((v, \rho, r, \text{memo})\).

The first three of these fields are as defined earlier.

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.

3.3 Transactions, Blocks, and the Block Chain

At a given point in time, the block chain view of each full node consists of a sequence of one or more valid blocks. Each block consists of a sequence of one or more transactions. To each transaction there is associated an initial treestate, which consists of a note commitment tree (§3.5 ‘Note Commitment Tree’ on p. 9), nullifier set (§3.6 ‘Nullifier Set’ on p. 10), and data structures associated with Bitcoin such as the UTXO (Unspent Transaction Output) set.

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.

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 nullifier set.

In a given node’s block chain view, 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.
TODO: JoinSplit descriptions also have input and output treestates.

We rely on Bitcoin-style consensus for full nodes to eventually converge on their views of valid blocks, and therefore of the sequence of treestates in those blocks.

### 3.4 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; it uses, but should not be confused with, the JoinSplit statement used for the zk-SNARK proof and verification.

A JoinSplit transfer spends $N^{\text{old}}$ notes $v^{\text{old}}_{1..N^{\text{old}}}$ and transparent input $v^{\text{old}}_{\text{pub}}$, and creates $N^{\text{new}}$ notes $v^{\text{new}}_{1..N^{\text{new}}}$ and transparent output $v^{\text{new}}_{\text{pub}}$.

Each transaction is associated with a sequence of JoinSplit descriptions.

The input and output values of each JoinSplit transfer MUST balance exactly. The total $v^{\text{new}}_{\text{pub}}$ value adds to, and the total $v^{\text{old}}_{\text{pub}}$ value subtracts from the transparent value pool of the containing transaction.

TODO: Describe the interaction of transparent value flows with the JoinSplit description’s $v^{\text{old}}_{\text{pub}}$ and $v^{\text{new}}_{\text{pub}}$.

The anchor of each JoinSplit description in a transaction must refer to either some earlier block’s final treestate, or to the output treestate of any prior JoinSplit description in the same transaction.

These conditions act as constraints on the blocks that a full node will accept into its block chain view.

### 3.5 Note Commitment Tree

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.

Blocks in the block chain are associated (by all nodes) with the root of this tree after all of its constituent JoinSplit descriptions’ note commitments have been entered into the note commitment tree associated with the previous block. TODO: Make this more precise.

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

![Diagram of Note Commitment Tree](image-url)
3.6 Nullifier Set

Each full node maintains a nullifier set alongside the note commitment tree and UTXO set. As valid transactions containing JoinSplit transfers are processed, the nullifiers revealed in JoinSplit descriptions are inserted into this nullifier set.

If a JoinSplit description reveals a nullifier that already exists in the full node’s block chain view, the containing transaction will be rejected, since it would otherwise result in a double-spend.

3.7 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 amount of the block subsidy and miner subsidy depends on the 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.

The calculations of the block subsidy, miner subsidy, and Founders’ Reward for a given block height are given in §6.5 ‘Calculation of Block Subsidy and Founders’ Reward’ on p. 33.

3.8 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 ‘Coinbase outputs’ on p. 34.

4 Abstract Protocol

4.1 Abstract Cryptographic Functions

4.1.1 Hash Functions

MerkleCRH : \( B^{[\ell_{\text{Merkle}}]} \times B^{[\ell_{\text{Merkle}}]} \rightarrow B^{[\ell_{\text{Merkle}}]} \) is a collision-resistant hash function used in §4.5 ‘Merkle path validity’ on p. 16. It is instantiated in §5.4.1 ‘Merkle Tree Hash Function’ on p. 21.

hSigCRH : \( B^{[\ell_{\text{Seed}}]} \times B^{[\ell_{\text{PRF}}]} \times \text{JoinSplitSig.Public} \rightarrow B^{[\ell_{\text{hSig}}]} \) is a collision-resistant hash function used in §4.3 ‘JoinSplit Descriptions’ on p. 14. It is instantiated in §5.4.2 ‘hSig Hash Function’ on p. 21.

EquihashGen : \( (n : \mathbb{N}^+) \times \mathbb{N}^+ \times B^{[8 \cdot 3]} \times \mathbb{N}^+ \rightarrow B^{[n]} \) is another hash function, used in §6.4.1 ‘Equihash’ on p. 32 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 §5.4.3 ‘Equihash Generator’ on p. 21.

4.1.2 Pseudo Random Functions

\( \text{PRF}_x \) is a Pseudo Random Function keyed by \( x \). Four independent \( \text{PRF}_x \) are needed in our protocol:

\[
\begin{align*}
\text{PRF}_{\text{addr}} & : B^{[\ell_{\text{addr}}]} \times \{0..255\} \rightarrow B^{[\ell_{\text{PRF}}]} \\
\text{PRF}_{\text{nf}} & : B^{[\ell_{\text{addr}}]} \times B^{[\ell_{\text{PRF}}]} \rightarrow B^{[\ell_{\text{PRF}}]} \\
\text{PRF}_{\text{PK}} & : B^{[\ell_{\text{addr}}]} \times \{1..N_{\text{addr}}\} \times B^{[\ell_{\text{hSig}}]} \rightarrow B^{[\ell_{\text{hSig}}]} \\
\text{PRF} & : B^{[\ell_{\text{addr}}]} \times \{1..N_{\text{new}}\} \times B^{[\ell_{\text{hSig}}]} \rightarrow B^{[\ell_{\text{hSig}}]}
\end{align*}
\]

These are used in §4.9 ‘JoinSplit Statement’ on p. 17; \( \text{PRF}_{\text{addr}} \) is also used to derive a payment address from a
spending key in §4.2 ‘Key Components’ on p.14. They are instantiated in §5.4.4 ‘Pseudo Random Functions’ on p.22.

Security requirement: In addition to being Pseudo Random Functions, it is required that PRF_{nf}^x, PRF_{addr}^x, and PRF_{ρ}^x be collision-resistant across all x — i.e. it should not be feasible to find (x, y) ≠ (x', y') such that PRF_{nf}^x(y) = PRF_{nf}^{x'}(y'), and similarly for PRF_{addr} and PRF_{ρ}.

Note: PRF_{nf} was called PRF_{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.


Security requirement: Sym must be one-time (INT-CTXT ∧ 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-CTXT 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 : B^{ℓ_{PRF}} → KA.Private be a function that converts a bit string of length ℓ_{PRF} to a KA private key.

Let KA.DerivePublic : KA.Private → KA.Public be a function that derives the KA public key corresponding to a given KA private key.

Let KA.Agree : KA.Private × KA.Public → KA.SharedSecret be the agreement function.

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

Security requirements:

- KA.FormatPrivate must preserve sufficient entropy from its input to be used as a secure KA private key.
- The key agreement and the KDF defined in the next section must together satisfy a suitable adaptive security assumption along the lines of [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 \( \text{KDF} \) be a Key Derivation Function suitable for use with \( \text{KA} \), deriving keys for \( \text{Sym} \). Encrypt.

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

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

Let \( \text{pk}^j_{\text{enc}} := \text{KA}.\text{DerivePublic}(\text{sk}^j_{\text{enc}}) \).

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

1. Choose \( \text{esk} \) uniformly at random from \( \text{KA}.\text{Private} \).
2. Let \( \text{epk} := \text{KA}.\text{DerivePublic}(\text{esk}) \).
3. For \( i \in \{1..N^{\text{new}}\} \), let \( K_i := \text{KDF}(i, h_{\text{Sig}}, \text{KA}.\text{Agree}(\text{esk}, \text{pk}^j_{\text{enc}}, \text{epk}, \text{pk}^j_{\text{enc}})) \).
4. Return \( (\text{epk}, K_1,..N^{\text{new}}) \).

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 negligible, then the asymmetric encryption scheme constructed from \( \text{KA}, \text{KDF} \) and \( \text{Sym} \) in §4.10 ‘In-band secret distribution’ on p.18 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 \( \text{KA}.\text{DerivePublic} \) (which includes all keys generated as in §4.2 ‘Key Components’ on p.14). 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 Signatures

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 \( (\text{sk}, \text{vk}) \xrightarrow{R} \text{Sig}.\text{Gen}() \), and any \( m : \text{Sig}.\text{Message} \) and \( s : \text{Sig}.\text{Signature} \xrightarrow{R} \text{Sig}.\text{Sign}_{\text{sk}}(m) \), \( \text{Sig}.\text{Verify}_{\text{vk}}(m, s) = 1 \).
Zcash uses two signature schemes, one used for signatures that can be verified by script operations such as OP_CHECKSIG and OP_CHECKMULTISIG as in Bitcoin, and one called JoinSplitSig which is used to sign transactions that contain at least one JoinSplit description. The latter is instantiated in §5.4.8 ‘Signatures’ on p. 23. The following defines only the security properties needed for JoinSplitSig.

**Security requirement:** 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.

**Notes:**
- Since a fresh key pair is generated for every transaction containing a JoinSplit description and is only used for one signature (see §4.6 ‘Non-malleability’ on p.17), a one-time signature scheme would suffice for JoinSplitSig. This is also the reason why only security against non-adaptive chosen message attack is needed. In fact the instantiation of JoinSplitSig uses a scheme designed for security under adaptive attack even when multiple signatures are signed under the same key.
- SU-CMA security requires it to be infeasible for the adversary, not knowing the private key, to forge a distinct signature on a previously seen message. That is, JoinSplit signatures are intended to be non-malleable in the sense of [BIP-62].

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 COMM defines a type of inputs COMM.Input, a type of commitments COMM.Output, and a type of commitment trapdoors COMM.Trapdoor.

Let COMM : COMM.Trapdoor × COMM.Input → COMM.Output be a function satisfying the security requirements of computational hiding and computational binding, as defined in TODO: need reference.

4.1.8 Zero-Knowledge Proving System

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

A preprocessing zk-SNARK instance ZK defines:

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

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

Security requirements:

- **Completeness**: An honestly generated proof will convince a verifier: for any \((x, w) \in ZK.SatisfyingInputs\), if \(ZK.Prove_{pk}(x, w)\) outputs \(\pi\), then \(ZK.Verify_{vk}(x, \pi) = 1\).
- **Proof of Knowledge**: For any adversary \(A\) able to find an \(x \in ZK.PrimaryInput\) and proof \(\pi \in ZK.Proof\) such that \(ZK.Verify_{vk}(x, \pi) = 1\), there is an efficient extractor \(E_A\) such that if \(E_A(vk, pk)\) returns \(w\), then the probability that \((x, w) \notin ZK.SatisfyingInputs\) is negligible.
- **Statistical Zero Knowledge**: An honestly generated proof is statistical zero knowledge. TODO: Full definition.

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

The Proof of Knowledge 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 \((x, w) \in ZK.SatisfyingInputs\). (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 §5.7 ‘Zero-Knowledge Proving System’ on p. 26. \(ZK.JoinSplit\) refers to this proving system specialized to the JoinSplit statement given in §4.9 ‘JoinSplit Statement’ on p. 17. In this case we omit the key subscripts on \(ZK.JoinSplit:\)\(Verify\) and \(ZK.JoinSplit:\)\(Prove\), taking them to be the particular proving key and verifying key defined by the JoinSplit parameters in §5.8 ‘JoinSplit Parameters’ on p. 28.

### 4.2 Key Components

Let \(KA\) be a key agreement scheme, instantiated in §5.4.6 ‘Key Agreement’ on p. 23.

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

- \(a_{pk}, sk_{enc}\) and \(pk_{enc}\) are derived from \(a_{sk}\) as follows:
  - \(a_{pk} := PRF_{addr}(0)\)
  - \(sk_{enc} := KA.FormatPrivate(PR_{addr}(1))\)
  - \(pk_{enc} := KA.DerivePublic(sk_{enc})\)

### 4.3 JoinSplit Descriptions

A JoinSplit transfer, as specified in §3.4 ‘JoinSplit Transfers and Descriptions’ on p. 9, 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.

Each JoinSplit description consists of \((v_{pub}^{\text{old}}, v_{pub}^{\text{new}}, rt, n_{\text{old}}, cm_{\text{new}}, epk, \text{randomSeed}, h_{\text{old}}, \pi, \text{JoinSplit}, C_{\text{enc}}^{\text{new}})\) where

- \(v_{pub}^{\text{old}} : \{0..\text{MAX\_MONEY}\}\) is the value that the JoinSplit transfer removes from the transparent value pool;
In-band secret distribution

JoinSplitPubKey is an anchor, as defined in §3.3 ‘Transactions, Blocks, and the Block Chain’ on p. 8, for the output treestate of either a previous block, or a previous JoinSplit transfer in this transaction.

nf\text{old}_{1..N_{\text{old}}} : B^{[f_{\text{old}}]}[N_{\text{old}}] is the sequence of nullifiers for the input notes;

cm\text{new}_{1..N_{\text{new}}} : COMM.Output\text{[new]} is the sequence of note commitments for the output notes;

e_{\text{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. 18);

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

h_{1..N_{\text{old}}} : B^{[f_{\text{old}}]}[N_{\text{old}}] is a sequence of tags that bind h_{\text{Sig}} to each a_{\text{vk}} of the input notes;

\pi_{\text{JoinSplit}} : ZK_{\text{JoinSplit witnessed by the previous JoinSplit statement}};

C_{\text{enc}}_{1..N_{\text{new}}} : Sym.C^{[\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 randomSeed, nf\text{old}_{1..N_{\text{old}}}, and the joinSplitPubKey of the containing transaction:

h_{\text{Sig}} := h_{\text{Sig}}CRH(\text{randomSeed, } nf\text{old}_{1..N_{\text{old}}}, \text{ joinSplitPubKey}).

h_{\text{Sig}}CRH is instantiated in §5.4.2 ‘h_{\text{Sig}} Hash Function’ on p. 21.

Consensus rules:

- Elements of a JoinSplit description MUST have the types given above (for example: 0 ≤ v\text{old}_{\text{pub}} ≤ \text{MAX\_MONEY} and 0 ≤ v\text{new}_{\text{pub}} ≤ \text{MAX\_MONEY}).
- Either v\text{old}_{\text{pub}} or v\text{new}_{\text{pub}} MUST be zero.
- The proof \pi_{\text{JoinSplit}} MUST be valid given a primary input formed from the other fields and h_{\text{Sig}}. I.e. it must be the case that ZK_{\text{JoinSplit witnessed by the previous JoinSplit statement}}(rt, nf\text{old}_{1..N_{\text{old}}}, cm\text{new}_{1..N_{\text{new}}}, v\text{old}_{\text{pub}}, v\text{new}_{\text{pub}}, h_{\text{Sig}}, h_{1..N_{\text{old}}}), \pi_{\text{JoinSplit}} = 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:

(joinSplitPrivKey, joinSplitPubKey) \leftarrow \text{JoinSplitSig.Gen().}

For each JoinSplit description, the sender chooses randomSeed uniformly at random on B^{[f_{\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 ϕ uniformly at random on B^{[f_{\text{ϕ}}]}. Then it creates each output note with index i : {1..N_{\text{new}}} as follows:

- Choose r_{i}^{\text{new}} uniformly at random on B^{[r_{i}]}.
- Compute \rho_{i}^{\text{new}} := PRF_{\text{ϕ}}(i, h_{\text{Sig}}).
- Encrypt the note to the recipient transmission key pk_{\text{enc},i}^{\text{new}}, as described in §4.10 ‘In-band secret distribution’ on p. 18, 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. 17), and signs it with the private JoinSplit signing key:

\[
\text{joinSplitSig} \leftarrow \text{JoinSplitSig.Sign}_{\text{JoinSplitPrivKey}}(\text{dataToBeSigned})
\]

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

4.4.1 Dummy Notes

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

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

- Generate a new random spending key \(a_{\text{sk},i}^{\text{old}}\), and derive its paying key \(a_{\text{pk},i}^{\text{old}}\).  
- Set \(v_{\text{old}}^{\text{old}}i = 0\).  
- Choose \(\rho_{\text{old}}^{\text{old}}i\) uniformly at random on \(B^{[\ell_{\text{PRF}}]}\).  
- Choose \(r_{\text{old}}^{\text{old}}i\) uniformly at random on \(B^{[\ell_{r}]}\).  
- Compute \(nf_{\text{old}}^{\text{old}}i \equiv \text{PRF}_{a_{\text{sk},i}^{\text{old}}}(\rho_{\text{old}}^{\text{old}}i)\).  
- 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 \(\text{enforce}_i\) to 0.

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

4.5 Merkle path validity

The depth of the note commitment tree is \(d_{\text{Merkle}}\) (defined in §5.3 ‘Constants’ on p. 20).

Each node in the incremental Merkle tree is associated with a hash value, which is a byte 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 \(d_{\text{Merkle}}\) are called leaf nodes. When a note commitment is added to the tree, it occupies the leaf node hash value \(M_{d_{\text{Merkle}}}^i\) 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 \(\text{NoteCommitment}(n) = \text{Uncommitted}\).

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

\[
M_h^i := \text{MerkleCRH}(M_{2i}^{h+1}, M_{2i+1}^{h+1}).
\]

A path from leaf node \(M_{d_{\text{Merkle}}}^i\) in the incremental Merkle tree is the sequence

\[M_{\text{sibling}(h,i)}^h\text{ for } h \text{ from } d_{\text{Merkle}}\text{ down to } 1\]

where

\[
\text{sibling}(h,i) = \text{floor}(\frac{i}{2^{d_{\text{Merkle}}-h}}) + 1
\]

Given such a path, it is possible to verify that leaf node \(M_{d_{\text{Merkle}}}^i\) 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. 28. 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^new and v^old and to the other JoinSplit descriptions in the same transaction, an ephemeral JoinSplit 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 §5.4.8 ‘Signatures’ on p. 23.

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

The condition enforced by the JoinSplit statement specified in §4.9 ‘Non-malleability’ on p. 18 ensures that a holder of all of sk_{old,..}, for each JoinSplit description 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^old takes value from the transparent value pool and v^new adds value to the transparent value pool. As a result, v^old is treated like an output value, whereas v^new is treated like an input value.

Note: Unlike original Zerocash [BCG+2014], Zcash does not have a distinction between Mint and Pour operations. The addition of v^old to a JoinSplit description subsumes the functionality of both Mint and Pour. Also, JoinSplit descriptions are indistinguishable regardless of the number of real input notes.

As stated in §4.3 ‘JoinSplit Descriptions’ on p. 14, either v^old or v^new MUST be zero. No generality is lost because, if a transaction in which both v^old and v^new were nonzero were allowed, it could be replaced by an equivalent one in which min(v^old, v^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 blockchain, 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 block chain view and mempool. A transaction is not valid if it attempts to add a nullifier to the nullifier set that already exists in the set.

4.9 JoinSplit Statement

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

\[
\begin{align*}
(rt \in \mathbb{B}^{[\#\text{of} \text{ inputs}]}, n_{1..\text{N}_{\text{old}}} \in \mathbb{B}^{[\#\text{of} \text{ inputs}]}, \text{cm}_{1..\text{N}_{\text{new}}} \in \text{COMM. Output}^{[\text{N}_{\text{new}]}, v^\text{old} \in \{0, 2^{64} - 1\}, v^\text{new} \in \{0, 2^{64} - 1\},
\end{align*}
\]

and

\[
\begin{align*}
\text{h}_{\text{Sig}} \in \mathbb{B}^{[\#\text{Sig}], h_{1..\text{N}_{\text{old}}} \in \mathbb{B}^{[\#\text{Sig}]},}\n\end{align*}
\]
the prover knows an auxiliary input:

\[(\text{path}_{i,\ldots,N_{\text{old}}} : B[[\text{Merkle}]_i d_{\text{Merkle}}][N_{\text{old}}^{-1}], n^{\text{old}}_{i,\ldots,N_{\text{old}}} : \text{Note}^{\text{old}}, a^{\text{old}}_{sk,i,\ldots,N_{\text{old}}} : B[i_{\text{old}}][N_{\text{old}}], n^{\new}_{i,\ldots,N_{\text{new}}} : \text{Note}^{\text{old}}, \]

\[\varphi : B[i_{\text{old}}], \text{enforce}_{i,\ldots,N_{\text{old}}} : B[i_{\text{old}}])\],

where:

- for each \(i \in \{1..N_{\text{old}}\}\): \(n_i^{\text{old}} = (a_{pk,i}^{\text{old}}, v_i^{\text{old}}, p_i^{\text{old}}, r_i^{\text{old}})\);
- for each \(i \in \{1..N_{\text{new}}\}\): \(n_i^{\new} = (a_{pk,i}^{\new}, v_i^{\new}, p_i^{\new}, r_i^{\new})\)

such that the following conditions hold:

**Merkle path validity**  for each \(i \in \{1..N_{\text{old}}\}\) \(\text{enforce}_{i} = 1\): path\(_i\) must be a valid path of depth \(d_{\text{Merkle}}\), as defined in §4.5 ‘Merkle path validity’ on p. 16, from NoteCommitment\(_i(n_i^{\text{old}})\) 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].

**Commitment Enforcement**  for each \(i \in \{1..N_{\text{old}}\}\), if \(v_i^{\text{old}} \neq 0\) then \(\text{enforce}_{i} = 1\).

\[\text{Balance} \quad v_{\text{pub}}^{\text{old}} + \sum_{i=1}^{N_{\text{old}}} v_i^{\text{old}} = v_{\text{pub}}^{\text{new}} + \sum_{i=1}^{N_{\text{new}}} v_i^{\text{new}} \in \{0..2^{64} - 1\} \]

**Nullifier integrity**  for each \(i \in \{1..N_{\text{new}}\}\): \(nf_i^{\text{old}} = \text{PRF}^{nf}_{a_{sk,i}^{\text{old}}}(p_i^{\text{old}})\).

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

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

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

**Commitment integrity**  for each \(i \in \{1..N_{\text{new}}\}\): \(cm_i^{\text{new}} = \text{NoteCommitment}(n_i^{\text{new}})\).

For details of the form and encoding of proofs, see §5.7 ‘Zero-Knowledge Proving System’ on p. 26.

### 4.10 In-band secret distribution

In order to transmit the secret \(v, p,\) and \(r\) (necessary for the recipient to later spend) and also a memo field to the recipient without requiring an out-of-band communication channel, the transmission key \(pk_{\text{enc}}\) is used to encrypt these secrets. The recipient’s possession of the associated key tuple \((a_{sk}, sk_{\text{enc}}, addr_{pk})\) is used to reconstruct the original note and memo field.

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

For both encryption and decryption,

- Let Sym be the encryption scheme instantiated in §5.4.5 ‘Authenticated One-Time Symmetric Encryption’ on p. 22.
Let KDF be the key derivation function instantiated in §5.4.7 ‘key derivation’ on p. 23.
Let KA be the key agreement scheme instantiated in §5.4.6 ‘key agreement’ on p. 23.
Let hSig be the value computed for this joinsplit description in §4.3 ‘joinsplit descriptions’ on p. 14.

4.10.1 Encryption

Let pk_{enc,1..N_{new}} be the transmission keys for the intended recipient addresses of each new note.
Let np_{1..N_{new}} be the note plaintexts as defined in §5.5 ‘Note plaintexts and memo fields’ on p. 24.

Then to encrypt:

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

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

Note: It is technically possible to replace C_{i}^{enc} for a given note with a random (and undecryptable) dummy ciphertext, relying instead on out-of-band transmission of the note to the recipient. In this case the ephemeral key MUST still be generated as a random public key (rather than a random bit string) 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 addr_{pk} = (apk, pk_{enc}) be the recipient’s payment address, and let sk_{enc} be the recipient’s viewing key.
Let cm_{1..N_{new}} be the note commitments of each output coin.

Then for each i ∈ {1..N_{new}}, the recipient will attempt to decrypt that ciphertext component as follows:

- Let sharedSecret_{i} := KA.Agree(sk_{enc}, epk).
- Let K_{i}^{enc} := KDF(i, hSig, sharedSecret_{i}, epk, pk_{enc,i}).
- Return DecryptNote(K_{i}^{enc}, C_{i}^{enc}, cm_{i}^{new}, apk).

DecryptNote(K_{i}^{enc}, C_{i}^{enc}, cm_{i}^{new}, apk) is defined as follows:

- Let P_{i}^{enc} := Sym.Decrypt_{K_{i}^{enc}}(C_{i}^{enc}).
- If P_{i}^{enc} = ⊥, return ⊥.
- Extract np_{i} = (v_{i}^{new}, r_{i}^{new}, memo_{i}) from P_{i}^{enc}.
- If NoteCommitment((apk, v_{i}^{new}, r_{i}^{new}, memo_{i})) ≠ cm_{i}^{new}, return ⊥, else return np_{i}.

To test whether a note is unspent in a particular block chain view also requires the spending key apk; the coin is unspent if and only if nf = PRF_{sk_{enc}}(ρ) is not in the nullifier set for that block chain view.
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 on a given block chain view, as transactions are added to that view. Also, blockchain reorganisations can cause the transaction in which a note was output to no longer be on the consensus blockchain.

See §7.7 ‘In-band secret distribution’ on p. 38 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. 37

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.

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 is given explicitly in each box, except for the case of a single bit, or for the notation $[0]^n$ which represents the sequence of $n$ 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:

\[
\begin{align*}
&d_{\text{Merkle}} : \mathbb{N} := 29 \\
&N_{\text{old}} : \mathbb{N} := 2 \\
&N_{\text{new}} : \mathbb{N} := 2 \\
&\ell_{\text{Merkle}} : \mathbb{N} := 256 \\
&\ell_{\text{hSig}} : \mathbb{N} := 256 \\
&\ell_{\text{PRF}} : \mathbb{N} := 256 \\
&\ell_r : \mathbb{N} := 256 \\
&\ell_{\text{Seed}} : \mathbb{N} := 256 \\
&\ell_{\text{a}} : \mathbb{N} := 252 \\
&\ell_q : \mathbb{N} := 252 \\
&\text{Uncommitted} : \mathbb{B}^{\ell_{\text{Merkle}}} := [0]^{\ell_{\text{Merkle}}}
\end{align*}
\]
MAX\_MONEY : N := 2.1 \cdot 10^{15} \text{ (zatoshi)}

SlowStartInterval : N := 20000

HalvingInterval : N := 840000

MaxBlockSubsidy : N := 1.25 \cdot 10^9 \text{ (zatoshi)}

NumFounderAddresses : N := 48

FoundersFraction : Q := \frac{1}{5}.

5.4 Concrete Cryptographic Functions

5.4.1 Merkle Tree Hash Function

MerkleCRH is used to hash incremental Merkle tree hash values. It is instantiated by the SHA-256 compression function, which takes a 512-bit block and produces a 256-bit hash. \[\text{NIST2015}\]

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

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

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

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

hSigCRH is used to compute the value \(h_{\text{Sig}}\) in §4.3 ‘JoinSplit Descriptions’ on p.14.

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

where

\[
\text{hSigInput} := \begin{array}{|c|c|c|}
\hline
256\text{-bit randomSeed} & 256\text{-bit n}^{\text{old}}_1 & \ldots & 256\text{-bit n}^{\text{old}}_{N^{\text{old}}} & 256\text{-bit joinSplitPubKey} \\
\hline
\end{array}
\]

BLAKE2b-256\((p, x)\) refers to unkeyed BLAKE2b-256 [ANWW2013] in sequential mode, with an output digest length of 32 bytes, 16-byte personalization string \(p\), and input \(x\). This is not the same as BLAKE2b-512 truncated to 256 bits, because the digest length is encoded in the parameter block.

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

5.4.3 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.32.

Let \(\text{powtag} := \begin{array}{|c|c|c|}
\hline
64\text{-bit “ZcashPow”} & 32\text{-bit } n & 32\text{-bit } k \\
\hline
\end{array}\)

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

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

\[
\cdot m := \text{floor}(\frac{312}{n})
\]
h := (i - 1 \mod m) \cdot n;
\quad T := \text{BLAKE2b-}(n \cdot m)(\text{powtag}, S \parallel \text{powcount}(\lfloor \frac{i - 1}{m} \rfloor)).

Indices of bits in T are 1-based.

BLAKE2b-ℓ(p, x) refers to unkeyed BLAKE2b-ℓ [ANWW2013] in sequential mode, with an output digest length of ℓ/8 bytes, 16-byte personalization string p, and input x. This is not the same as BLAKE2b-512 truncated to ℓ bits, because the digest length is encoded in the parameter block.

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 § 6.4.1 ‘Equihash’ on p. 32), the number of calls to BLAKE2b can be reduced by a factor of \floor{512 \cdot n} in the best case (which is a factor of 2 for n = 200).

5.4.4 Pseudo Random Functions

The four independent PRFs described in § 4.1.2 ‘Pseudo Random Functions’ on p. 10 are all instantiated using the SHA-256 compression function:

| PRF_{addr}(t) := SHA256Compress(1100, 252-bit x, 8-bit t, [0]^{248}) |
| PRF_{a_{sk}}(p) := SHA256Compress(1110, 252-bit a_{sk}, 256-bit p) |
| PRF_{a_{sk}}(i, h_{Sig}) := SHA256Compress(0100, 252-bit a_{sk}, 256-bit h_{Sig}) |
| PRF_{p_{\phi}}(i, h_{Sig}) := SHA256Compress(0110, 252-bit \phi, 256-bit h_{Sig}) |

Security requirements:

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

Note: The first four bits –i.e. the most significant four bits of the first byte– are used to 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 § 5.4.9 ‘Commitment’ on p. 24. (The specific bit patterns chosen here are motivated by the possibility of future extensions that either increase N_{old} and/or N_{new} to 3, or that add an additional bit to a_{sk} to encode a new key type, or that require an additional PRF.)

5.4.5 Authenticated One-Time Symmetric Encryption

Let Sym.K := B^{[256]}, Sym.P := B^{[8 \cdot N]}, and Sym.C := B^{[8 \cdot N]}.


Similarly, let Sym.Decrypt_{\phi}(C) be AEAD_CHACHA20_POLY1305 decryption of ciphertext C \in Sym.C, with empty “associated data”, all-zero nonce [0]^{96}, and 256-bit key K \in Sym.K. The result is either the plaintext byte sequence, or ⊥ indicating failure to decrypt.
5.4.6 Key Agreement

The key agreement scheme specified in §4.1.4 ‘Key Agreement’ on p. 11 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 \(9\) be the public byte sequence representing the Curve25519 base point.

Let clamp\(_{\text{curve25519}}(x)\) 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\(_{\text{curve25519}}(x)\).

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

5.4.7 Key Derivation

The Key Derivation Function specified in §4.1.5 ‘Key Derivation’ on p. 12 is instantiated using BLAKE2b-256 as follows:

\[
\text{KDF}(i, h_{\text{Sig}}, \text{sharedSecret}, \text{epk}, p_{\text{enc}}^{\text{new}}) := \text{BLAKE2b-256}(\text{kdfTag}, \text{kdfInput})
\]

where:

\[
\begin{array}{c}
\text{kdfTag} := \begin{bmatrix} 64\text{-bit "ZcashKDF"} & 8\text{-bit } i - 1 & [0]^{56} \end{bmatrix} \\
\text{kdfInput} := \begin{bmatrix} 256\text{-bit } h_{\text{Sig}} & 256\text{-bit sharedSecret} & 256\text{-bit epk} & 256\text{-bit } p_{\text{enc}}^{\text{new}} \end{bmatrix}
\end{array}
\]

BLAKE2b-256\((p, x)\) refers to unkeyed BLAKE2b-256 [ANWW2013] in sequential mode, with an output digest length of 32 bytes, 16-byte personalization string \(p\), and input \(x\). This is not the same as BLAKE2b-512 truncated to 256 bits, because the digest length is encoded in the parameter block.

5.4.8 Signatures

JoinSplitSig is specified in §4.1.6 ‘Signatures’ on p. 12.

It is instantiated as Ed25519 [BDL+2012], with the additional requirement that \(S\) (the integer represented by \(S\)) must be less than the prime \(\ell = 2^{256} + 2^{255} + 2^{254} + \cdots + 2^0 + 2^{255} - 277423177737235355851937790883648493\), otherwise the signature is considered invalid. Ed25519 is defined as using SHA-512 internally.

The encoding of a signature is:

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

where \(R\) and \(S\) are as defined in [BDL+2012].

The encoding of a public key is as defined in [BDL+2012].
5.4.9 Commitment

The commitment scheme $\text{COMM}$ specified in §4.1.7 ‘Commitment’ on p. 13 is instantiated using SHA-256 as follows:

$$\text{COMM}_r(v, a_{\text{pk}}, \rho) := \text{SHA256}(10100000 \ 256\text{-bit } a_{\text{pk}} \ 64\text{-bit } v \ 256\text{-bit } \rho \ 256\text{-bit } r).$$

**Note:** The leading byte of the SHA256 input is 0xB0.

TODO: Security requirements on SHA-256.

5.5 Note Plaintexts and Memo Fields

Transmitted notes are stored on the blockchain in encrypted form, together with a note commitment $\text{cm}$.

The note plaintexts associated with a JoinSplit description are encrypted to the respective transmission keys $pk_{\text{new}}^{1..N_{\text{new}}}$, and the result forms part of a transmitted notes ciphertext (see §4.10 ‘In-band secret distribution’ on p. 18 for further details).

Each note plaintext (denoted $np$) consists of $(v, \rho, r, \text{memo})$.

The first three of these fields are as defined earlier. memo is a 512-byte memo field associated with this note.

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

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

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

In the latter case, the contents of the memo field **SHOULD NOT** be displayed. A start byte of 0xF5 is reserved for use by automated software by private agreement. A start byte of 0xF6 or greater is reserved for use in future Zcash protocol extensions.

The encoding of a note plaintext consists of, in order:

- A byte, 0x00, indicating this version of the encoding of a note plaintext.
- 8 bytes specifying $v$.
- 32 bytes specifying $\rho$.
- 32 bytes specifying $r$.
- 512 bytes specifying memo.

5.6 Encodings of Addresses and Keys

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

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

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

The raw encoding of a P2SH address consists of:

<table>
<thead>
<tr>
<th>8-bit 0x1C</th>
<th>8-bit 0xBD</th>
<th>160-bit script hash</th>
</tr>
</thead>
</table>

- 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.)
- 160 bits specifying a script hash [Bitcoin-P2SH].

The raw encoding of a P2PKH address consists of:

<table>
<thead>
<tr>
<th>8-bit 0x1C</th>
<th>8-bit 0xB8</th>
<th>160-bit public key hash</th>
</tr>
</thead>
</table>

- 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.)
- 160 bits 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 [Bitcoin-Base58], for both the production and test networks.

5.6.3 Shielded Payment Addresses

A payment address consists of \( a_{pk} \) and \( pk_{enc} \). \( a_{pk} \) is a SHA-256 compression output. \( pk_{enc} \) is a Bern2006 public key, for use with the encryption scheme defined in §4.10 ‘In-band secret distribution’ on p.18.

The raw encoding of a payment address consists of:

| 8-bit 0x16 | 8-bit 0x9A | 256-bit \( a_{pk} \) | 256-bit \( pk_{enc} \) |
Two bytes \([0x16, 0x9A]\), indicating this version of the raw encoding of a Zcash payment address on the production network. (Addresses on the test network use \([0x16, 0xB6]\) instead.)

- 256 bits specifying \(a_{pk}\).
- 256 bits specifying \(p_{enc}\), 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 Spending Keys

A spending key consists of \(a_{sk}\), which is a sequence of 252 bits.

The raw encoding of a spending key consists of, in order:

\[
\begin{array}{c|c|c}
8\text{-bit} & 0xAB & 8\text{-bit} & 0x36 & [0]^4 & 252\text{-bit} \ a_{sk}
\end{array}
\]

- 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.)
- 4 zero padding bits.
- 252 bits specifying \(a_{sk}\).

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

**Notes:**

- If an implementation represents \(a_{sk}\) internally as a sequence of 32 bytes with the 4 bits of zero padding intact, it will be in the correct form for use as an input to PRF\(^{add}\), 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 Zero-Knowledge Proving System

Zcash uses zk-SNARKs generated by its fork of libsnark [libsnark-fork] with the proving system described in [BCTV2015], which is a refinement of the systems in [PGHR2013] and [BCGTV2013].

The pairing implementation is ALT\_BN128.

Let \(q = 2^{18888242871839275222246405745257275088696311157297823662689037894645226208583}\).

Let \(r = 2^{1888824287183927522224640574525727508854836440041603433698204186575808495617}\).

Let \(b = 3\).

\((q \text{ and } r \text{ are prime}).\)

The pairing is of type \(G_1 \times G_2 \rightarrow G_T\), where:

- \(G_1\) is a Barreto–Naehrig curve over \(\mathbb{F}_q\) with equation \(y^2 = x^3 + b\). This curve has embedding degree 12 with respect to \(r\).
\( \mathcal{G}_2 \) is the subgroup of order \( r \) in the twisted Barreto–Naehrig curve over \( \mathbb{F}_{q^2} \) with equation \( y^2 = x^3 + \frac{b}{x^2} \). We represent elements of \( \mathbb{F}_{q^2} \) as polynomials \( a_1 \cdot t + a_0 : \mathbb{F}_q[t] \), modulo the irreducible polynomial \( t^2 + 1 \).

\( \mathcal{G}_T \) is \( \mu_r \), the subgroup of \( r \)th roots of unity in \( \mathbb{F}_{q^4} \).

Let \( \mathcal{P}_1 : \mathcal{G}_1 = (1, 2) \).

Let \( \mathcal{P}_2 : \mathcal{G}_2 = (11559732032986387107991004021392285812681821192530917403154291805634 \cdot t + 1085704699992305713954457076223282948137075635957851806999051993285655852781, 408236787586343368133220340314543556831685132759340120810574107621412093531 \cdot t + 849563923123431417604973247489272438418190587263600148770280649306958101930) \).

\( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are generators of \( \mathcal{G}_1 \) and \( \mathcal{G}_2 \) respectively.

A proof consists of a tuple \((\pi_A : \mathcal{G}_1, \pi_A' : \mathcal{G}_1, \pi_B : \mathcal{G}_2, \pi_B' : \mathcal{G}_1, \pi_C : \mathcal{G}_1, \pi_C' : \mathcal{G}_1, \pi_K : \mathcal{G}_1, \pi_H : \mathcal{G}_1)\). 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 arithmetic circuit verifying the JoinSplit statement, or its expression as a Rank 1 Constraint System, are not specified here. In practice it will be necessary to use the specific proving and verification keys generated for the Zcash production block chain (see §5.8 ‘JoinSplit Parameters’ on p. 28), and a proving system implementation that is interoperable with the Zcash fork of libsnark, to ensure compatibility.

### 5.7.1 Encoding of Points

Define \( \text{I2OSP} : (k : \mathbb{N}) \times \{0..256^k - 1\} \rightarrow \{0..255\}^{[k]} \) such that \( \text{I2OSP}_\ell(n) \) is the sequence of \( \ell \) bytes representing \( n \) in big-endian order.

For a point \( P : \mathcal{G}_1 = (x_P, y_P) \):

- The field elements \( x_P \) and \( y_P : \mathbb{F}_q \) are represented as integers \( x \) and \( y : \{0..q - 1\} \).
- Let \( \tilde{y} = y \mod 2 \).
- \( P \) is encoded as \([0 0 0 0 \ 0 \ 0 \ 1 \text{\ 1-bit } \tilde{y} \text{\ 256-bit I2OSP}_{32}(x)\]).

For a point \( P : \mathcal{G}_2 = (x_P, y_P) \):

- A field element \( w : \mathbb{F}_{q^2} \) is represented as a polynomial \( a_{w,1} \cdot t + a_{w,0} : \mathbb{F}_q[t] \) modulo \( t^2 + 1 \). Define \( \text{FE2IP} : \mathbb{F}_{q^2} \rightarrow \{0..q^2 - 1\} \) such that \( \text{FE2IP}(w) = a_{w,1} \cdot q + a_{w,0} \).
- Let \( x = \text{FE2IP}(x_P), y = \text{FE2IP}(y_P) \), and \( y' = \text{FE2IP}(-y_P) \).
- Let \( \tilde{y} = \begin{cases} 1, & \text{if } y > y' \\ 0, & \text{otherwise} \end{cases} \)
- \( P \) is encoded as \([0 0 0 0 \ 1 \ 0 \ 1 \text{\ 1-bit } \tilde{y} \text{\ 512-bit I2OSP}_{64}(x)\]).

**Non-normative notes:**

- The use of big-endian byte order is different from the encoding of most other integers in this protocol. The above encodings are consistent with the definition of EC2OSP for compressed curve points in [IEEE2004, section 5.5.6.2]. The LSB compressed form (i.e. EC2OSP-XL) is used for points on \( \mathcal{G}_1 \), and the SORT compressed form (i.e. EC2OSP-XS) for points on \( \mathcal{G}_2 \).
- Testing \( y > y' \) for the compression of \( \mathcal{G}_2 \) points is equivalent to testing whether \((a_{y,1}, a_{y,0}) > (a_{-y,1}, a_{-y,0})\) in lexicographic order.
5.7.2 Encoding of Zero-Knowledge Proofs

A 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 $\{0..q-1\}$ or (in the case of $\pi_B$) $\{0..q^2-1\}$;
- the encoding represents a point on the relevant curve.

5.8 JoinSplit Parameters

For the testnet in release v0.11.2.z9 and later, the SHA-256 hashes of the proving key and verifying key for the JoinSplit statement, encoded in libsnark format, are:

- 226913bbdc48b70834f8e044d194dd61c8e15329f67cde6014f4e5ac11a82ab z9-proving.key
- 4c151c562fce2cde055ac0a0f8bd9454eb69e6a0db9a8443b58b770ec29b37f5 z9-verifying.key

The Zcash production block chain will use parameters obtained by a multi-party computation, which has yet to be performed.

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>uint32_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 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 changes relative to Bitcoin version 1 transactions as described in [Bitcoin-Format] are:

- The transaction version number can be either 1 or 2. A version 1 transaction is equivalent to a version 2 transaction with nJoinSplit = 0. Software that parses blocks MUST NOT assume, when an encoded block starts with an version field representing a value other than 1 or 2 (e.g. future versions potentially introduced by hard forks), that it will be parseable according to this format.
- The nJoinSplit, vJoinSplit, joinSplitPubKey, and joinSplitSig fields have been added.

Software that creates transactions SHOULD use version 1 for transactions with no JoinSplit descriptions.

**Note:** 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.

### 6.2 Encoding of JoinSplit Descriptions

An abstract JoinSplit description, as described in §3.4, ‘JoinSplit Transfers and Descriptions’ on p. 9, 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_{pub}^{old}) that the JoinSplit 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_{pub}^{new}) that the JoinSplit transfer inserts into the transparent value pool.</td>
</tr>
<tr>
<td>32</td>
<td>anchor</td>
<td>char[32]</td>
<td>A merkle root (rt) of the note commitment tree at some block height in the past, or the merkle root produced by a previous JoinSplit transfer in this transaction.</td>
</tr>
<tr>
<td>64</td>
<td>nullifiers</td>
<td>char[32][(N_{old})]</td>
<td>A sequence of nullifiers of the input notes (n_{1...N_{old}}^{old}).</td>
</tr>
<tr>
<td>64</td>
<td>commitments</td>
<td>char[32][(N_{new})]</td>
<td>A sequence of note commitments for the output notes (cm_{1...N_{new}}^{new}).</td>
</tr>
<tr>
<td>32</td>
<td>ephemeralKey</td>
<td>char[32]</td>
<td>A Curve25519 public key (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 JoinSplit description.</td>
</tr>
<tr>
<td>64</td>
<td>vmacs</td>
<td>char[32][(N_{old})]</td>
<td>A sequence of message authentication tags (h_{1...N_{old}}) that bind (h_{Sig}^{a_{sk}}) to each (a_{sk}) of the JoinSplit description.</td>
</tr>
<tr>
<td>296</td>
<td>zkproof</td>
<td>char[296]</td>
<td>An encoding of the zero-knowledge proof (\pi_{JoinSplit}) (see §5.7.2 ‘Encoding of Zero-Knowledge Proofs’ on p. 28).</td>
</tr>
<tr>
<td>1202</td>
<td>encCiphertexts</td>
<td>char[601][(N_{new})]</td>
<td>A sequence of ciphertext components for the encrypted output notes, (C_{1...N_{new}}^{enc}).</td>
</tr>
</tbody>
</table>

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

### 6.3 Block Headers

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>uint32_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 when the miner started hashing the header (according to the miner). This MUST be greater than or equal to the median time of the previous 11 blocks. A full node MUST NOT accept blocks with headers more than two hours in the future according to its clock.</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. Bitcoin-nBits</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 below 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>
<tr>
<td>1344</td>
<td>solution</td>
<td>char[1344]</td>
<td>The Equihash solution, which MUST be valid according to §6.4.1 ‘Equihash’ on p. 32.</td>
</tr>
</tbody>
</table>

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

- The block version number MUST be 4. Previous versions are not supported. Software that parses blocks MUST NOT assume, when an encoded block starts with an nVersion field representing a value other than 4 (e.g. future versions potentially introduced by hard forks), that it will be parseable according to this format.
- The hashReserved, solutionSize, and solution fields have been added.
- The type of the nNonce field has changed from uint32_t to char[32].

Notes:

- 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.

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. 32.
- The block header satisfies the difficulty check according to §6.4.2 ‘Difficulty filter’ on p. 33.

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, $N = 2^{n+1}$, and the sequence $X_1..N$ is derived from the block header and a nonce:

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

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

EquihashGen is instantiated in §5.4.3 ‘Equihash Generator’ on p. 21.

Define $\text{I2BSP} : (u : \mathbb{N}) \times \{0..2^u - 1\} \rightarrow \mathbb{B}^u$ such that $\text{I2BSP}_u(x)$ is the sequence of $u$ bits representing $x$ in big-endian order.

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 $w \in \{0..2^{k-r}-1\}$:

- $\bigoplus_{j=1}^{2^r} X_{i_{w\cdot2^r+j}}$ has $\frac{n-r}{k+1}$ leading zeroes; and
- $i_{w\cdot2^r+1..w\cdot2^r+2^r-1} < i_{w\cdot2^r+2^r-1+1..w\cdot2^r+2^r}$ lexicographically.

Note: This does not include a difficulty condition, because here we are defining validity of an Equihash solution independent of difficulty.

An Equihash solution with $n = 200$ and $k = 9$ is encoded in the solution field of a block header as follows:
Recall from §5.2 ‘Integers, Bit Sequences, and Endianness’ on p. 20 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 $[69, 42, 21]$, then the corresponding bit array is:

<table>
<thead>
<tr>
<th>68</th>
<th>41</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit 0</td>
<td>8-bit 2</td>
<td>8-bit 32</td>
</tr>
<tr>
<td>8-bit 0</td>
<td>8-bit 10</td>
<td>8-bit 127</td>
</tr>
<tr>
<td>8-bit 255</td>
<td>8-bit 55</td>
<td></td>
</tr>
</tbody>
</table>

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

Note: I2BSP is big-endian, while integer field encodings in powheader and in the instantiation of EquihashGen are little-endian. The rationale for this is that little-endian serialization of block headers is consistent with Bitcoin, but using little-endian ordering of bits in the solution encoding would require bit-reversal (as opposed to only shifting).

6.4.2 Difficulty filter

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

6.4.3 Difficulty adjustment

Zcash uses a difficulty adjustment algorithm based on DigiShield v3/v4, with simplifications and altered parameters, to adjust difficulty to target the desired 2.5-minute block time. Unlike Bitcoin, the difficulty adjustment occurs after every block.

TODO: Describe the algorithm.

6.5 Calculation of Block Subsidy and Founders’ Reward

§3.7 ‘Block Subsidy and Founders’ Reward’ on p. 10 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. 20.

SlowStartShift := \frac{\text{SlowStartInterval}}{2}

SlowStartRate := \frac{\text{MaxBlockSubsidy}}{\text{SlowStartInterval}}

Halving(height) := \floor{\frac{\text{height} - \text{SlowStartShift}}{\text{HalvingInterval}}}

BlockSubsidy(height) :=
\begin{cases} 
\text{SlowStartRate} \cdot \text{height}, & \text{if } \text{height} < \frac{\text{SlowStartInterval}}{2} \\
\text{SlowStartRate} \cdot (\text{height} + 1), & \text{if } \frac{\text{SlowStartInterval}}{2} \leq \text{height} < \text{SlowStartInterval} \\
\floor{\frac{\text{MaxBlockSubsidy}}{2 \cdot \text{Halving(height)}}}, & \text{otherwise}
\end{cases}

FoundersReward(height) :=
\begin{cases} 
\text{BlockSubsidy(height)} \cdot \text{FoundersFraction}, & \text{if } \text{height} < \text{SlowStartShift} + \text{HalvingInterval} \\
0, & \text{otherwise}
\end{cases}

MinerSubsidy(height) := \text{BlockSubsidy(height)} - \text{FoundersReward(height)}.
6.6 Coinbase outputs

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

For the production network, FounderAddressList...NumFounderAddresses is:

```plaintext
["t3Vz22wKvz52LcKEdg16Yy4FFneELIzg9qjd", "t3cL9AucCajm3LHDhb5jBnJ2VapVoXsop3", 
"t3fQvkrrzNaMcmmkQM4vHyrjfdDM4xQvDTR", "t3Tg2Z72TSk44AnUP16qenHa2eCP7uyF", 
"t3SpkcpQPFuREhP5e3v3P86fgko5m8Vkm", "t3xt4QMPagwpqkgAViQgST4vSWR6S", 
"t3ayBkZ4w6skXYnwOHZFUSsgXRktogTXngb", "t3adJBQuaa2u17nXrb8Y7mz3k3t82S4M", 
"t3K4yaLagSB8sDrdfAg6U5h952v88t", "t3YRnsc5nVehkiva3Zphf85K7eyh1Cr6k", 
"t3Ut4Ku2Z5MTPNe676PuL5qYCI1n98KpXg", "t3ZnCNaVgu6CyHmlwtrX3aiN98dSAQpbn", 
"t3f9bC9e3yIm64BS9x5wAHQUKLgqZQ6roBDg", "t3cwZfKHm2xVMAXHBBqewem6pXKFDkh18k", 
"t3YcoujXspWf7rNusGkXEWQzNaSt0peG", "t3BrLCLicg6rBrNrsTS3NWkgVyRrzCzumTR4", 
"t3VvWnWt3r0y67YtU4LZGKCwA2J3e6Hhvsh", "t3eF9X62sd5h7CvTjFZsZeVzWbVzquRLNe", 
"t3esCNNwmcyc89q9qTvB7qTmQnYX9Awk3", "t3m4Jn7hYbe2e717sLuQPjPVek8IbV3b", 
"t3GgwCxtD76CYNwBo8pjNvrrP2LXpQ2LXroi", "t3LTweoWEpmdkUD3NWbq4kWzaFhBwMu", 
"t3PS5KXX97gXYFsasGoPirRoQX84yf5Z3t3j", "t3f3T3nCwsoPzmD53vK62JqPfFq74vD8Vc9", 
"t3RmgovuzZafkF715s414im8n8En411j", "t3fJZ5jysYxVtDrwBewoMbvJaqCj4jJgK", 
"t3Pnbg7xI7P7GFBUoUJ5h65acphHgkpoJ", "t3w3KQDdxCjL5x7rfem1MTLr2wVJkufHF", 
"t3Y9FfNI26J7UtAcU4MoaETLboNo8518Be6M", "t3aNRRLLsL2yxcjHzeFvZwyc7cP7sTwBc", 
"t3DQeDvak5VvAAHKT8rQu2BDWdxEi1FumB", "t3Rbykhx1TFUgxrXmryJAr2eSTXRFKL7g9", 
"t3aaAwA4Tdp7a8d11VEIIdod2ybbegHgMaj", "t3YeIaAe6uEjxWFLv2ZuItu1fn3ykGzmqNy", 
"t3GilyUwFt2PbMdMrTcPwpHucDLdtL2iQg", "t3dPwneqPyqPuiCec overwhelm51uYH8D4z", 
"t3RZQXHDPph2uU461q827VK6ruWfuFp4d", "t3enHAcrxi1ZD7e8eqPvmOvnK7n79NFJ3", 
"t3PkkLg7T11TnF12nsSwBToxeD77yNnx2jGi", "t3LQXhDuspe7Zhhvbdry4vnaNoAhnCrf24fO", 
"t3fCndubucybbctnisD2a9q3LUXg77PVFB", "t3dKojUZ2Emx28h8vYtvKWEUJiu1MAFA", 
"t3AHkH1Nw10if6d8c9r1rZiagYpkJ3M679M", "t3MEXDF9wai63KwpqDd6b32w2bGtbE", 
"t3WDbfpf3k343YnmPqT7kAZoQsOqQ83K7y3f", "t3PSn5TMMMEw7EU36DycFezRzp1xibhfz", 
"t3R3Y5vbnLrEn5L6sWfP7PjBlnxSuoQsNkmfp", "t3Pcm37EsVtGTHbsu2NeKfJteEg92mvYyo", 
]```

For the test network, FounderAddressList...NumFounderAddresses is:

```plaintext
["t2UNzUux9wMbcryRPeZv363EYxyEpHokY", "t2N9PH9Wk9xjQjvG9iin1ua3eakqfAtE543", 
"t2NQGQyMhFndDgHvuyw4zdNdssaa6KXz2", "t27knmtq1kebeCWIq5T7w5n5pSzbdbM7BTv6", 
"t2GcBttkAKD2WThhA8yHgGc2dfvATkY2FUfH", "t2Q3vxwaD9LrdqE8x9Ddjr9p9U2qAgotK", 
"t2TTrffDwsy98fHw5ZPv9P9n4efxXmRi1wzu", "t2K36K4MwMdSmJCIw2ymWGBRqPmyQrn", 
"t2Z2LRgotwvZ3EecLltMiquq83cVo38tu8J", "t2AEJga88vTVawkDFJdyURyYHUtCquAZ15G1", 
"t2HSCmdqpg1tQKksuvPQwvawATpgf2rHkh", "t2HPQCPFUaUdp7JWHPhg5pPbX7itlJaInu", 
"t2Fzqvq89Y963J3Pn9b2a5yM6nq45bmHm", "t2HE7yZyqQudC5by42WSMdswn3qNVRj", 
"t2GCRI5Csk686eo5ne223smSns7w1YBgf", "t2KyiPr9Lzqt2w71w747X64KnukAMGL0M9N", 
"t2UxymadxsVyihmbq71syxw5dCqbiJ1SaJ", "t2AVeMy7dfmdCtbkq1KOg87B7FvcvEbB", 
"t26mL5h1wqD2sHhVePnYpl5M57jzxbfwK9", "t2DgwUNte7NxyPufxSb5XJXap3E4Wy5Kr", 
"t2U6fucn1lfC9S2eHyvlU3rk3X3vbnf7fs", "t284JhY8LGLM72X1tporSqwrqc3JeQhj7P", 
"t29egu4QpqKsLoLq5J5sG3V4mUMQUP6deN", "t29LQd9pD3BD6nuewF6mfcWu6HAPA38N", 
"t28GnsAMcxAlvy85xqaaddbDzaYttfev6w", "t2YV4Q4YqaiQKPlUFm6oZTw7n1Ljkn7G", 
"t2U2QZb1qJtu4uL6xmXNLB8sQu0VGR2", "t2QKrgr5PAn7nrwdGqseyHN9NFeueJUcbh", 
"t2AfSu6H6vbeJkpBpxzvtvJrupQKDQnwr", "t2CTRQVUqfd3CWMnKhFUnHqxLuyTmXh", 
"t2BMc9EqszUHnrU1KUXZHvPnRK4enRE", "t2L7nY1asKDU42GNSMdwDxbz81d0QWYt", 
"t2AgYsT35LRH378AE3ouz6xmKnhKTLXsC6n", "t285EAQXUVyi14NmdJvdJvQTrnv45GMRNP", 
"t2EpMRCDSB582DQM37nPULCPZnkj7Cmus", "t2BcmWxrFpeCeqpZpizSWKKNPMKX6PS7umY", 
"t2DNT7X6xvdGxuXh1Bm93Zsy019yariTV", "t2QJj8HeCwQ6hWQexKLXZLJgZp7THNUE2t", 
"t2QhdB1p9yCq4z85pSdQZrtYekvMrj", "t2E5cpLAey5VWFCuopeMQR2H2NHPid", 
"t2EVRgtzJfAyc8FZndvlui7jQu2Fde9h", "t2KoQDk3BSfadkuaWdLchWFUqamzW9RE4", 
"t2Fnr3QhyTuiejeJLeu6Pqi4W7ghRd4hJ", "t2BEAnBAosPdC9d1u56Nj6i6x4mNJB2Dp", 
"t2RtcKrLCgyPkm4a4ApFiyY92wM24rjgpr", "t2AusbStezZqBQpFgJ1K1XNZRZPS6MMAYk", 
"t2Urdy1ErfkveFuy6Z4BkhvYgZwdmvFA", "t2ADIn4RjCvMd4QXGALP4rFrqyvEHDE", 
]```

Each address representation in FounderAddressList denotes a transparent P2SH multisig address.
Let SlowStartShift be defined as in the previous section.

Define:

\[
\text{FounderAddressChangeInterval} := \text{ceiling} \left( \frac{\text{SlowStartShift} + \text{HalvingInterval}}{\text{NumFounderAddresses}} \right)
\]

\[
\text{FounderAddressIndex}(\text{height}) := 1 + \text{floor} \left( \frac{\text{height}}{\text{FounderAddressChangeInterval}} \right)
\]

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

**Consensus rule:** A coinbase transaction for block height \( \text{height} \in \{1..\text{SlowStartShift} + \text{HalvingInterval} - 1\} \) 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.

TODO: Coinbase maturity rule. TODO: Any tx with a coinbase input must have no transparent outputs (vout).

**Notes:**

- No Founders’ Reward is required to be paid for \( \text{height} \geq \text{SlowStartShift} + \text{HalvingInterval} \) (i.e. after the first halving), or for \( \text{height} = 0 \) (i.e. the genesis block).
- The Founders’ Reward addresses are not treated specially in any other way, and there can be other outputs to them, in coinbase transactions or otherwise. In particular, it is valid for a coinbase transaction with height \( \in \{1..\text{SlowStartShift} + \text{HalvingInterval} - 1\} \) to have other outputs, possibly to the same address, that do not meet the criterion in the above consensus rule, as long as at least one output meets it.

### 6.7 Changes to the Script System

The \texttt{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 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-34], [BIP-65], [BIP-66].

[BIP-13] applies with the changes to address version bytes described in §5.6.1 ‘Transparent Payment Addresses’ on p. 25.

### 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. 24.

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 $N$ old shielded notes as input, and produces up to $N$ new 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 (described in §7.1 ‘Transaction Structure’ on p. 35) 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 $N$ 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 view 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. 16).

The nullifiers are used as input to \( h_{\text{SigCRH}} \) to derive a public value \( h_{\text{Sig}} \) which uniquely identifies the transaction, as described in §4.3 ‘JoinSplit Descriptions’ on p. 14. \( h_{\text{Sig}} \) 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 \( \varphi \) and \( h_{\text{Sig}} \) using \( \text{PRF}_\varphi \). The correct construction of \( \rho \) for each output note is enforced by the JoinSplit statement (see §4.9 ‘Uniqueness of \( \rho_i^{\text{new}} \)’ on p. 18).

Now even if the creator of a JoinSplit description does not choose \( \varphi \) randomly, uniqueness of nullifiers and collision resistance of both \( h_{\text{SigCRH}} \) and \( \text{PRF}_\varphi \) will ensure that the derived \( \rho \) values are unique, at least for any two JoinSplit descriptions that get into a valid block chain view. This is sufficient to prevent the Faerie Gold attack.

### 7.5 Internal hash collision attack and fix

The Zerocash security proof requires that the composition of \( \text{COMM}_r \) and \( \text{COMM}_s \) is a computationally binding commitment to its inputs \( a_{pk}, \varphi, v, \) and \( \rho \). However, the instantiation of \( \text{COMM}_r \) and \( \text{COMM}_s \) in section 5.1 of the paper did not meet the definition of a binding commitment at a 128-bit security level. Specifically, the internal hash of \( a_{pk} \) 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 values of \( \rho \) with colliding outputs of the truncated hash, and therefore the same note commitment. This would have allowed such an attacker to break the Balance property by double-spending notes, potentially creating arbitrary amounts of currency for themself [HW2016].

Zcash uses a simpler construction with a single SHA-256 evaluation for the commitment. The motivation for the nested construction in Zerocash was to allow Mint transactions to be publically 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 a generalized JoinSplit transfer which always uses a zero-knowledge proof, it does not require the nesting. A side benefit is that this 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 §5.4.4 ‘Pseudo Random Functions’ on p. 22 has changed relative to Zerocash. There is also a requirement for another PRF, PRF^pf, which must be domain-separated from the others.

In the Zerocash protocol, \( \rho_i^{\text{old}} \) is truncated from 256 to 254 bits in the input to PRF^sn (which corresponds to PRF^nf in Zerocash). Also, \( h_{\text{Sig}} \) is truncated from 256 to 253 bits in the input to PRF^pk. These truncations are not taken into account in the security proofs.

Both truncations affect the validity of the proof sketch for Lemma D.2 in the proof of Ledger Indistinguishability in [BCG+2014, Appendix D]. In more detail:

- In the argument relating \( \mathbf{H} \) and \( \mathbb{D}_2 \), it is stated that in \( \mathbb{D}_2 \), “for each \( i \in \{1, 2\} \), \( s_n := \text{PRF}^\text{sn} \_a_i (\rho) \) for a random (and not previously used) \( \rho \).” It is also argued that “the calls to \( \text{PRF}^\text{sn} \_a_i \) are each by definition unique”. The latter assertion depends on the fact that \( \rho \) is “not previously used”. However, the argument is incorrect because the truncated input to \( \text{PRF}^\text{sn} \_a_i \), 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 \( \text{PRF}^\text{pk} \), i.e. \( h_{\text{Sig}} |_{253} = | \text{CRH}(\text{pk}_{\text{sig}}) |_{253} \). In practice this value will be unique under a plausible assumption on \( \text{CRH} \) provided that \( \text{pk}_{\text{sig}} \) is chosen randomly, but no formal argument for this is presented.

Note that \( \rho \) is truncated in the input to PRF^sn but not in the input to COMM, 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^sn. In that case, it would be possible to violate balance by creating two notes for which \( \rho \) differs only in the truncated bits. These notes would have the same note commitment but different nullifiers, so it would be possible to spend the same value twice.

For resistance to Faerie Gold attacks as described in §7.4 ‘Faerie Gold attack and fix’ on p. 36, Zerocash depends on collision resistance of both \( h_{\text{Sig}} \text{CRH} \) and \( \text{PRF}^p \) (instantiated using BLAKE2b-256 and SHA256Compress respectively). Collision resistance of a truncated hash does not follow from collision resistance of the original hash, even if the truncation is only by one bit. This motivated avoiding truncation along any path from the inputs to the computation of \( h_{\text{Sig}} \) 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 \( a_{\text{sk}} \) in the case of \( \text{PRF}^\text{addr}, \text{PRF}^\text{nf}, \text{PRF}^\text{pk} \), and to \( v \) (which does not exist in Zerocash) for \( \text{PRF}^p \), 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 scheme based on Curve25519 key agreement, 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 because $p_{\text{enc}}$ is included in the KDF input, being able to break the Elliptic Curve Diffie–Hellman Problem on Curve25519 (without breaking AEAD_CHACHA20_POLY1305 as an authenticated encryption scheme or BLAKE2b-256 as a KDF) would not help to decrypt the transmitted 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 extractor” for each JoinSplit transfer, which limits degradation of security with the number of JoinSplit transfers. This facilitates security analysis as explained in [DGKM2011] — see section 7 of that paper for a security proof that can be applied to this construction under the assumption that single-block BLAKE2b-256 is a “weak PRF”. Note that $h_{\text{Sig}}$ is authenticated, by the ZK proof, as having been chosen with knowledge of $\alpha_{sk,1..N_{\text{inv}}}$, 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 Zcash includes an optimization that uses the same ephemeral key (with different nonces) for the two ciphertexts encrypted in each JoinSplit description.

The security proofs of [ABR1999] can be adapted straightforwardly to the resulting scheme. Although DHAES as defined in that paper does not pass the recipient public key or a public seed to the hash function $H$, this does not impair the proof because we can consider $H$ to be the specialization of our KDF to a given recipient key and seed. It is necessary to adapt the "HDH independence" assumptions and the proof slightly to take into account that the ephemeral key is reused for two encryptions.

Note that the 256-bit key for AEAD_CHACHA20_POLY1305 maintains a high concrete security level even under attacks using parallel hardware [Bern2005] in the multi-user setting [Zave2012]. This is especially necessary because the privacy of Zcash transactions may need to be maintained far into the future, and upgrading the encryption algorithm would not prevent a future adversary from attempting to decrypt ciphertexts encrypted before the upgrade. Other cryptovalues that could be attacked to break the privacy of transactions are also sufficiently long to resist parallel brute force in the multi-user setting: $\alpha_{sk}$ is 252 bits, and $s_{\text{enc}}$ is no shorter than $\alpha_{sk}$.

### 7.8 Omission in Zerocash security proof

The abstract Zerocash protocol requires PRF*adr only to be a PRF; it is not specified to be collision-resistant. This reveals a flaw in the proof of the Balance property.

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

The error is in the proof of Balance in [BCG+2014, Appendix D.3]. For the "A violates Condition 1" case, the proof says:

"(i) If $cm^1_{\text{old}} = cm^2_{\text{old}}$, then the fact that $sn^1_{\text{old}} \neq sn^2_{\text{old}}$ implies that the witness $a$ contains two distinct openings of $cm^1_{\text{old}}$ (the first opening contains $(a_{sk,1}^{old}, \rho_1^{old})$, while the second opening contains $(a_{sk,2}^{old}, \rho_2^{old})$). This violates the binding property of the commitment scheme COMM."
In fact the openings do not contain \( a_{sk,1}^{old} \); they contain \( a_{pk,1}^{old} \).

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

The flaw is not exploitable for the actual instantiations of \( \text{PRF}_{addr} \) in \textit{Zerocash} and \textit{Zcash}, which are collision-resistant assuming that SHA256Compress is.

The proof can be straightforwardly repaired. The intuition is that we can rely on collision resistance of \( \text{PRF}_{addr} \) (on both its arguments) to argue that distinctness of \( a_{sk,1}^{old} \) and \( a_{sk,2}^{old} \), together with constraint 1(b) of the \textit{JoinSplit statement} (see §4.9 ‘\textit{Spend authority}’ on p. 18), implies distinctness of \( a_{pk,1}^{old} \) and \( a_{pk,2}^{old} \), therefore distinct openings of the \textit{note commitment} when Condition I or II is violated.

7.9 Miscellaneous

- The paper defines a note as \((a_{pk}, pk_{enc}, v, \rho, r, s, cm)\), whereas this specification defines it as \((a_{pk}, v, \rho, r)\). The instantiation of \text{COMM}_s in section 5.1 of the paper did not actually use \( s \), and neither does the new instantiation of \text{COMM} in \textit{Zcash}. \( pk_{enc} \) is also not needed as part of a note: it is not an input to \text{COMM} nor is it constrained by the \textit{Zerocash} \textit{POUR statement} or the \textit{Zcash} \textit{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 §5.7.2 ‘\textit{Encoding of Zero-Knowledge Proofs}’ on p. 28, because the paper did not take into account the need to encode compressed \( y \)-coordinates. The fork of \textit{libsnark} used by \textit{Zcash} uses a different format to upstream \textit{libsnark}, in order to follow [IEEE2004].

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

8 Acknowledgements

The inventors of \textit{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 \textit{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, and no doubt others.

\textit{Zcash} has benefited from security audits performed by NCC Group and Coinspect.

The \textit{Faerie Gold attack} was found by Zooko Wilcox. The internal hash collision attack was found by Taylor Hornby. The error in the \textit{Zerocash} proof of Balance relating to collision-resistance of \( \text{PRF}_{addr} \) was found by Daira Hopwood. The errors in the proof of Ledger Indistinguishability mentioned in §7.6 ‘\textit{Changes to PRF inputs and truncation}’ on p. 38 were also found by Daira Hopwood.

9 Change history

2016.0-beta-1.9

- Add \textit{Founders’ Reward} addresses for the production network.

- Change “\textit{protected}” terminology to “\textit{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 vold_pub and vnew_pub 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 PRFnf corresponding to PRFsn in Zerocash.

• Add a paragraph about key length in §7.7 ‘In-band secret distribution’ on p. 38.

• 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[ℓ]” 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 Coin- spect security audits.
2016.0-beta-1.2
- Remove GeneralCRH in favour of specifying hSigCRH and EquihashGen directly in terms of BLAKE2b.
- 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 reorganisation 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 hSig.
- Fix the lead bytes in 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 Zcach 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 PRFaddr to be collision-resistant (see §7.8 ‘Omission in Zerocash security proof’ on p. 39).
- Add specification of path computation for the incremental Merkle tree.
- Add a note in §4.9 ‘Merkle path validity’ on p. 18 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) (↑ p5).
