Zecnero Protocol Specification

Version 2026.0.3-draft [NU6-based]

The Zecnero Developers

September 19, 2026

Abstract. Zecnero is a decentralized anonymous payment system derived from Zcash. It keeps the transparent payment scheme inherited from Bitcoin and the Sapling and Orchard shielded payment schemes, which are secured by zero-knowledge succinct non-interactive arguments of knowledge. It replaces the Equihash proof of work with RandomX-Zecnero, an instance of RandomX whose cache derivation uses its own salt, so that general-purpose processors are the most cost-effective hardware for producing blocks. Difficulty is set by an absolutely scheduled exponential algorithm. Issuance halves every 1 680 000 blocks and reaches zero after 30 halvings, for a total of slightly less than 21 000 000 ZMR. During the first halving period, three percent of each block subsidy is paid to a founder script that is published before launch. There is no other allocation.

This specification defines the Zecnero consensus protocol at launch. It is self-contained for every rule in which Zecnero differs from Zcash, and for the structure of the protocol as a whole. The internals of cryptographic primitives that Zecnero uses without modification are specified by reference to the Zcash Protocol Specification. It is a work in progress.

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

Contents

1 Introduction

Zecnero is an implementation of the Decentralized Anonymous Payment scheme Zerocash [BCGGMTV2014], as refined by Zcash in [Zcash-Protocol]. It bridges the transparent payment scheme of Bitcoin [Nakamoto2008] with shielded payment schemes that are secured by zk-SNARKs. The cryptographic core of Zecnero is the Sapling and Orchard protocols as deployed in Zcash at network upgrade NU6. The consensus changes relative to Zcash concern proof of work, difficulty adjustment, issuance, network identity, and chain selection.

In this document, technical terms for concepts that play an important rôle in Zecnero are written in *slanted text*. Italics are used for emphasis. The symbol § precedes section numbers in cross-references. Where a section of this document relies on a definition in the Zcash Protocol Specification, the reference is written [Zcash-Protocol, §n], meaning section *n* of [Zcash-Protocol] at the version listed in §14.

The key words MUST, MUST NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in [RFC-2119] when they appear in all capitals. These words also appear in lower case as ordinary English words, without their normative meanings.

The name "Sapling" refers to the Sapling shielded protocol and its value pool. The name "Orchard" refers to the Orchard shielded protocol and its value pool. The Sprout shielded protocol of Zcash does not exist in Zecnero; the reasons are given in §9.5.

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 description of the protocol in terms of ideal cryptographic components;
- Concrete Protocol: how the functions and encodings of the abstract protocol are instantiated, including the proof of work and difficulty adjustment in full;
- Network Upgrades: how the Zecnero protocol is changed after launch;
- Consensus Changes from Zcash: a complete list of the places where Zecnero departs from Zcash;
- Chain Selection: the rules nodes use to choose between competing chains;
- Rationale, Launch Procedure, and Security Considerations: non-normative explanation of design choices;
- Appendices: a reference implementation of the difficulty algorithm, the complete issuance schedule, and worked examples.

1.1 Caution

Zecnero security depends on consensus. A program that interacts with the Zecnero network and diverges from consensus has its security weakened or destroyed. The cause of the divergence does not matter. It may be a bug in the program, an error in this document implemented as written, or correct software meeting unexpected behaviour elsewhere on the network. Users of the software lose funds regardless of which of these occurred.

A specification of intended behaviour is nevertheless essential for security analysis, understanding of the protocol, and maintenance of Zecnero software. Errors in this document should be reported in the issue tracker of the Zecnero specification repository. Security-relevant errors should be reported privately according to the security policy of the reference implementation.

This is a draft. Values marked *[unassigned]* MUST NOT be used on any public network.

1.2 High-level Overview

The following overview summarizes the ideas behind the protocol for readers familiar with block chain based cryptocurrencies such as Bitcoin. It is not part of the normative specification and is imprecise in places.

All value in Zecnero belongs to a *chain value pool*. The chain value pools are the *transparent pool*, the *Sapling pool*, and the *Orchard pool*. Transparent value is transferred essentially as in Bitcoin and has the same

privacy properties. Value in a *shielded pool* is carried by *notes*. A note specifies an amount and, indirectly, a *shielded payment address*. The holder of the corresponding *spending key* can spend the note.

Each note has a *note commitment*. When the transaction that creates a note is mined, its commitment is appended to a Merkle tree of note commitments at a fixed *note position*. Each note also has a *nullifier*. Computing the nullifier requires the *nullifier deriving key* associated with the recipient's address. Without that key, a note commitment cannot feasibly be linked to its nullifier. A note is unspent at a given point in the chain if its commitment has been revealed before that point and its nullifier has not.

A transaction can contain transparent inputs and outputs, *Spend descriptions* and *Output descriptions* for the Sapling pool, and *Action descriptions* for the Orchard pool. Spending a shielded note reveals its nullifier. Creating one reveals its commitment. Zero-knowledge proofs attached to the transaction show, among other things, that each spent note's commitment exists in the tree, that the spender knew the relevant keys, that nullifiers and commitments were computed correctly, and that value commitments match the notes. Value commitments are homomorphic, so the balance of a transaction is checked outside the proofs by a *binding signature*. Nodes check separately that no nullifier appears twice in the chain.

Output notes are encrypted to their recipients in the transaction. A recipient scans the chain with an *incoming viewing key* and decrypts notes addressed to them. A *full viewing key* additionally recognizes outgoing notes. When a note is spent, the spender proves only that some earlier commitment corresponds to it, without saying which. The set of notes that could be the input to a fully shielded transaction therefore includes every earlier note not known to have been spent.

Blocks are produced by proof of work. A miner searches for a block header whose RandomX-Zecnero hash, read as an integer, does not exceed a target. The target for each block is computed from the genesis time, the height of the chain, and the timestamp of the previous block, so that blocks arrive on average every 75 seconds. Each block pays its miner a subsidy that halves every 1 680 000 blocks, together with the fees of the transactions it contains. During the first halving period each block also pays a fixed fraction of its subsidy to a published founder script.

2 Notation

$\mathbb{B}$ means the type of bit values, that is $\{0, 1\}$. $\mathbb{BY}$ means the type of byte values, that is $\{0\,..\,255\}$.

$\mathbb{N}$ means the type of nonnegative integers, $\mathbb{N}^+$ the type of positive integers, and $\mathbb{Z}$ the type of integers.

$x : T$ specifies that x has type T. A cartesian product type is written $S \times T$ and a function type $S \to T$.

$T^{[\ell]}$, where T is a type and ℓ is an integer, means the type of sequences of length ℓ with elements in T. In particular $\mathbb{B}^{[\ell]}$ is the type of ℓ-bit sequences and $\mathbb{BY}^{[k]}$ the type of k-byte sequences. $\mathbb{BY}^{[\mathbb{N}]}$ is the type of byte sequences of any length.

$\{a\,..\,b\}$ means the set of integers from a to b inclusive.

`0x` followed by hexadecimal digits in monospace means the corresponding integer. $[\texttt{0x00}]^\ell$ means the sequence of ℓ zero bytes. "`abc`" means the byte sequence of the given US-ASCII string, here $[\texttt{0x61}, \texttt{0x62}, \texttt{0x63}]$.

$a \parallel b$ means the concatenation of sequences a and b. $\mathsf{length}(S)$ is the number of elements of S. $\mathsf{truncate}_k(S)$ is the sequence of the first k elements of S.

$\perp$ is a distinguished value indicating a failed check, an unavailable value, or an exceptional case.

$\mathsf{floor}(x)$ is the largest integer not greater than x, and $\mathsf{ceiling}(x)$ the smallest integer not less than x. Where an algorithm requires division of integers rounded toward zero, it writes $\mathsf{trunc}(a/b)$.

$a \bmod q$, for $a : \mathbb{Z}$ and $q : \mathbb{N}^+$, is the remainder in $\{0\,..\,q-1\}$ on dividing a by q.

$a \gg n$ and $a \ll n$, for $a : \mathbb{Z}$ and $n : \mathbb{N}$, are the arithmetic right shift and left shift of a by n bits. For negative a, $a \gg n = \mathsf{floor}(a/2^n)$.

$\mathbb{F}_n$ is the finite field with n elements.

$\mathsf{I2LEOSP}_\ell(k)$, for ℓ a multiple of 8 and $k : \{0\,..\,2^\ell - 1\}$, is the $\ell/8$-byte little-endian encoding of k. $\mathsf{LEOS2IP}_\ell(S)$ is its inverse. Big-endian variants are written with BE in place of LE.

$\mathsf{SHA\text{-}256d}(x)$ means $\mathsf{SHA\text{-}256}(\mathsf{SHA\text{-}256}(x))$ [NIST2015].

Integer constants used throughout the document are defined in §5.2. Names of constants are written in sans-serif, for example BlockTargetSpacing.

3 Concepts

3.1 Payment Addresses and Keys

A user who wishes to receive shielded payments must have a *shielded payment address*, generated from a *spending key*.

Sapling. A Sapling *expanded spending key* consists of a *spend authorizing key* ask, a *nullifier private key* nsk, and an *outgoing viewing key* ovk. From these are derived a *proof authorizing key* $(\mathsf{ak}, \mathsf{nsk})$, a *full viewing key* $(\mathsf{ak}, \mathsf{nk}, \mathsf{ovk})$, an *incoming viewing key* ivk, and any number of *diversified payment addresses* $(\mathsf{d}, \mathsf{pk_d})$. The derivation is given in §4.4.1.

Orchard. An Orchard spending key sk determines a spend authorizing key ask and a full viewing key $(\mathsf{ak}, \mathsf{nk}, \mathsf{rivk})$. From the full viewing key are derived an incoming viewing key (a *diversifier key* dk and a private key ivk), an outgoing viewing key ovk, and diversified payment addresses $(\mathsf{d}, \mathsf{pk_d})$. The derivation is given in §4.4.2.

The consensus protocol does not depend on how spending keys are generated. Wallets SHOULD derive keys hierarchically according to [ZIP-32], using the Zecnero coin type of §5.13.

Payments to one shielded payment address from several senders are not linked on the chain, even for the senders. Two senders who compare the address they paid can see that it is the same. A recipient who wants to prevent this gives each sender a different diversified address. All diversified addresses derived from one incoming viewing key are recognized by a single scan of the chain.

Non-normative note: The same spending key produces the same shielded payment addresses on Zecnero and on Zcash. A user who publishes one address for both networks allows observers to link their activity on the two networks. Wallets SHOULD use a distinct account or coin type for each network.

3.2 Shielded Pools and Notes

A *note* represents value held in a shielded pool. Each note belongs to exactly one of the Sapling pool and the Orchard pool, and is spendable by the holder of the spending key for the shielded payment address it names.

Let $\mathsf{MAX\_MONEY}$, ℓ_{d}, and ℓ_{value} be as defined in §5.2.

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

- $\mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the diversifier of the recipient's shielded payment address;
- $\mathsf{pk_d}$ is the diversified transmission key of that address, a point of prime order on the Jubjub curve;
- $\mathsf{v} : \{0 .. \mathsf{MAX\_MONEY}\}$ is the value of the note in nero;
- rcm is the trapdoor of the note commitment.

An *Orchard-pool note* is a tuple $(\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$, where d, v, and rcm are as above, $\mathsf{pk_d}$ is a point on the Pallas curve, $\rho : \mathbb{F}_{q_{\mathbb{P}}}$ is the nullifier of the note spent in the same Action, and $\psi : \mathbb{F}_{q_{\mathbb{P}}}$ is additional randomness used in deriving the nullifier.

3.2.1 Note Plaintexts and Memo Fields

A note is transmitted on the chain in encrypted form. The *note plaintext* of a Sapling-pool or Orchard-pool note is

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

The note's rcm, and for Orchard its ψ, are derived from rseed as specified in [ZIP-212]. The memo field carries data agreed between sender and recipient. Its RECOMMENDED interpretation is given in [ZIP-302].

Consensus rule: The $\mathsf{leadByte}$ of every note plaintext MUST be $\mathtt{0x02}$.

Non-normative note: Zcash accepts lead byte `0x01` for Sapling notes created before a grace period following Canopy. Zecnero activates Canopy at height 1 and has no such period.

3.2.2 Note Commitments

When a note is created, only a commitment to its contents is published. The commitment is appended to the note commitment tree of its pool. When the note is later spent, the zk-SNARK proof shows that the commitment is in the tree without identifying it.

For a Sapling-pool note, with $\mathsf{g_d} = \mathsf{DiversifyHash}^{\mathsf{Sapling}}(\mathsf{d})$,

$$\mathsf{NoteCommitment}^{\mathsf{Sapling}}(\mathsf{n}) = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}\big(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}\big),$$

or $\bot$ if $\mathsf{g_d} = \bot$. It is represented on the chain by the u-coordinate of the resulting Jubjub point.

For an Orchard-pool note, with $\mathsf{g_d} = \mathsf{DiversifyHash}^{\mathsf{Orchard}}(\mathsf{d})$,

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

It is represented on the chain by the x-coordinate of the resulting Pallas point.

3.2.3 Nullifiers

The *nullifier* of a note is a value unique to that note, revealed when it is spent. A transaction that would reveal a nullifier already present in the chain is invalid, which prevents double spending.

A Sapling-pool nullifier is derived from the note's ρ value, which depends on the note commitment and its position in the tree, and from the recipient's nullifier deriving key nk. An Orchard-pool nullifier is derived from ρ, ψ, nk, and the note commitment. The computations are given in §4.8.

3.3 The Block Chain

Each full validator is aware of a set of candidate blocks. They form a tree rooted at the *genesis block*, in which each block refers to its parent by the hashPrevBlock field of its header. A path from the root consisting of valid blocks is a *valid block chain*.

Each block has a *block height*. The genesis block has height 0, and each subsequent block has the height of its parent plus one. Implementations MUST support heights up to $2^{31} - 1$.

To choose its *best valid block chain*, a node sums the work (§5.5.3) of the blocks in each valid block chain it knows, and prefers the chain with the greatest total, subject to the rules of §8. Between leaf blocks of equal total work, a node prefers the one it received first.

A full validator SHOULD obtain candidate blocks from several independent peers.

3.4 Transactions and Treestates

Each block contains one or more transactions. Each transaction has a *transaction ID* and a *wtxid*, computed as specified in [ZIP-244] and [ZIP-239].

Transparent inputs to a transaction add value to its *transparent transaction value pool* and transparent outputs remove value from it. The net effect of the transaction's Sapling and Orchard transfers is given by its valueBalanceSapling and valueBalanceOrchard fields, which add to the transparent transaction value pool when positive. As in Bitcoin, the value remaining in the transparent transaction value pool of a non-coinbase transaction is the fee, and the sum of fees in a block is an implicit input to the block's coinbase transaction.

Consensus rules:

- The value remaining in the transparent transaction value pool of a non-coinbase transaction MUST be nonnegative.
- The value remaining in the transparent transaction value pool of a coinbase transaction MUST be zero.

Each transaction has an input and an output *treestate* for each shielded pool. A treestate consists of a note commitment tree (§3.7) and a nullifier set (§3.8). Treestates are chained as follows:

- the input treestate of the first block is empty;
- the input treestate of the first transaction of a block is the final treestate of the preceding block;
- the input treestate of each later transaction in a block is the output treestate of the transaction before it;
- the final treestate of a block is the output treestate of its last transaction.

An *anchor* is the root of a note commitment tree. It identifies a treestate.

3.5 Spend Transfers, Output Transfers, and their Descriptions

A Sapling *Spend transfer* spends one note and is described by a *Spend description*. A Sapling *Output transfer* creates one note and is described by an *Output description*. Each carries a homomorphic Pedersen *value commitment* to the value of its note, and a Groth16 proof for the corresponding statement.

Balance is enforced by summing the value commitments of spends, subtracting those of outputs, and requiring a *Sapling binding signature* under the resulting key over the transaction's signature hash (§4.9).

Consensus rules:

- The Spend transfers and Output transfers of a transaction MUST be consistent with its valueBalanceSapling as specified in §4.9.
- The anchorSapling field of a transaction with at least one Spend description MUST be the final Sapling treestate root of some earlier block in the chain.

3.6 Action Transfers and their Descriptions

An Orchard *Action transfer* optionally spends one note and optionally creates one note. It is described by an *Action description*, which carries one value commitment to the net value of the Action. The Halo 2 proofs of all Actions in a transaction are aggregated into a single field of the transaction.

Consensus rules:

- The Action transfers of a transaction MUST be consistent with its valueBalanceOrchard as specified in §4.9.
- The anchorOrchard field of a transaction with at least one Action description MUST be the final Orchard treestate root of some earlier block in the chain.

Non-normative note: Neither Sapling nor Orchard needs interstitial treestates, because a transfer cannot spend an output of the same transaction.

3.7 Note Commitment Trees

A *note commitment tree* is an append-only incremental Merkle tree of fixed depth, one for each shielded pool. The Sapling tree uses a Pedersen-hash based compression function and the Orchard tree uses Sinsemilla [Zcash-Protocol, §5.4.1.3]. The index of a note's commitment among the leaves is its *note position*.

Consensus rules:

- A block MUST NOT add Sapling note commitments beyond the capacity of $2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}$ leaves.
- A block MUST NOT add Orchard note commitments beyond the capacity of $2^{\mathsf{MerkleDepth}^{\mathsf{Orchard}}}$ leaves.

3.8 Nullifier Sets

Each full validator maintains a nullifier set for each shielded pool.

Consensus rule: A nullifier MUST NOT repeat, either within a transaction or across transactions in a valid block chain. Sapling and Orchard nullifiers are considered disjoint even when their bit patterns are equal.

3.9 Block Subsidy and Founder Stream

As in Bitcoin, Zecnero creates currency when blocks are mined. The currency created by a block is its *block subsidy*. For blocks below FounderStreamEndHeight, the block subsidy is divided between the *miner subsidy* and the *founder stream*. From FounderStreamEndHeight onward, the whole block subsidy is miner subsidy. The miner of a block also receives the fees of its transactions.

The calculations are specified in §5.9.

3.10 Coinbase Transactions

A transaction with a single transparent input whose previous output reference is null is a *coinbase transaction*. Every block has exactly one coinbase transaction, as its first transaction. The coinbase transaction collects the miner subsidy and fees, and pays the founder stream where §5.9.2 requires it.

3.11 Mainnet and Testnet

The production Zecnero network, which supports the ZMR currency, is called *Mainnet*. A public test network called *Testnet* supports a currency with no intended value. Testnet uses the same consensus rules as Mainnet except where §5.13 lists a different parameter. Testnet MAY be reset at any time.

The smallest unit of currency on either network is the *nero*. One ZMR is 10^8 nero.

Block hashes in this document are written in RPC byte order, which is byte-reversed relative to the output of SHA-256d.

Mainnet genesis block: *[unassigned]*

Testnet genesis block: *[unassigned]*

Other networks using variants of the Zecnero protocol may exist and are not described by this specification.

4 Abstract Protocol

This part describes the protocol in terms of abstract cryptographic components with defined interfaces and security requirements. Their instantiations are given in §5. An instantiation that fails to meet the requirements of its abstraction is an error to be corrected whether or not it leads to a known attack, because the abstraction is what makes the protocol analysable.

Zecnero inherits every abstract scheme used by the Sapling and Orchard protocols from Zcash without change. §4.1 lists them with their rôles. The complete interface definitions and security requirements are those of [Zcash-Protocol, §4.1]. The abstractions that are new in Zecnero, the proof-of-work function and the difficulty adjustment function, are defined in full in §4.2 and §4.3.

4.1 Abstract Cryptographic Schemes

Scheme	Rôle in Zecnero
Merkle tree hash functions	Compression functions of the Sapling and Orchard note commitment trees. Collision resistant.
$\mathsf{CRH}^{\mathsf{ivk}}$	Derives a Sapling incoming viewing key from ak and nk. Collision resistant.
Mixing Pedersen hash	Derives the ρ of a Sapling note from its commitment and position.
$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$, $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$	Map a diversifier to a diversified base. Required to be unlinkable.
$\mathsf{PRF}^{\mathsf{expand}}$	Expands a spending key or seed into key components and note randomness.
$\mathsf{PRF}^{\mathsf{nfSapling}}$, $\mathsf{PRF}^{\mathsf{nfOrchard}}$	Derive nullifiers. Required to be collision resistant across keys.
$\mathsf{PRF}^{\mathsf{ockSapling}}$, $\mathsf{PRF}^{\mathsf{ockOrchard}}$	Derive the key that encrypts outgoing ciphertexts.

Scheme	Rôle in Zecnero
$\mathsf{PRP}^{\mathsf{d}}$	Generates Orchard diversifiers from a diversifier key and index.
Sym	Authenticated one-time symmetric encryption of note plaintexts. One-time (INT-CTXT $\wedge$ IND-CPA) secure.
$\mathsf{KA}^{\mathsf{Sapling}}$, $\mathsf{KA}^{\mathsf{Orchard}}$	Key agreement between sender and recipient of a note.
$\mathsf{KDF}^{\mathsf{Sapling}}$, $\mathsf{KDF}^{\mathsf{Orchard}}$	Derive Sym keys from key agreement output. Together with KA and Sym, must give an IND-CCA2 secure, key-private encryption scheme.
Transparent signature	Validated by script operations. ECDSA on secp256k1, as in Bitcoin.
SpendAuthSig	Signs authorization of Sapling spends and Orchard Actions. A signature scheme with re-randomizable keys, SURK-CMA secure.
BindingSig	Enforces balance of shielded transfers. A signature scheme with key monomorphism, SU-CMA secure.
$\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathsf{NoteCommit}^{\mathsf{Orchard}}$	Note commitments. Computationally hiding and binding.
$\mathsf{ValueCommit}^{\mathsf{Sapling}}$, $\mathsf{ValueCommit}^{\mathsf{Orchard}}$	Homomorphic value commitments.
$\mathsf{Commit}^{\mathsf{ivk}}$	Derives an Orchard incoming viewing key.
Represented groups	Jubjub for Sapling, Pallas and Vesta for Orchard.
Zero-knowledge proving systems	Groth16 over BLS12-381 for Sapling, Halo 2 over Pallas and Vesta for Orchard.

4.2 Proof-of-Work Function

An *asymmetric keyed proof-of-work function* PoW consists of:

- a key type $\mathsf{PoW.Key} = \mathbb{B}\mathbb{Y}^{[32]}$;
- an input type $\mathsf{PoW.Input} = \mathbb{B}\mathbb{Y}^{[140]}$;
- a hash function $\mathsf{PoW.Hash} : \mathsf{PoW.Key} \times \mathsf{PoW.Input} \to \mathbb{B}\mathbb{Y}^{[32]}$;
- a key schedule $\mathsf{SeedHeight} : \mathbb{N} \to \mathbb{N}$, with $\mathsf{SeedHeight}(h) < h$ for all $h > 0$.

The key used for a block at height $h > 0$ is the block hash of its ancestor at height $\mathsf{SeedHeight}(h)$. The genesis block uses a fixed key.

Security requirements:

- **Hardness.** For any key k and target T, finding an input x such that $\mathsf{LEOS2IP}_{256}(\mathsf{PoW.Hash}(k, x)) \leq T$ is expected to require about $2^{256}/(T+1)$ evaluations of PoW.Hash by any method.
- **Progress freedom.** The probability of success of each evaluation is independent of previous evaluations, so that a miner's expected share of blocks is proportional to its share of evaluations.
- **Hardware neutrality.** Evaluating PoW.Hash should cost no less, per unit of energy and per unit of capital, on general-purpose processors than on other hardware available at comparable cost.
- **Verifiability.** A single evaluation must be feasible for every full validator, in both time and memory.

Non-normative note: Hardware neutrality is an engineering goal rather than a property that can be proven. It is argued for RandomX in §9.1.

4.3 Difficulty Adjustment Function

A *difficulty adjustment function* maps the state of a valid block chain whose tip is at height h to the target that a block at height $h+1$ must satisfy. Zecnero uses an *absolutely scheduled* function: its output depends only on constants, on the genesis block, and on the height and timestamp of the tip. It does not depend on the timestamps or targets of intermediate blocks.

4.4 Key Components

4.4.1 Sapling Key Components

Let ℓ_{sk} and ℓ_{d} be as defined in §5.2. Let $\mathcal{G}^{\mathsf{Sapling}}$ and $\mathcal{H}^{\mathsf{Sapling}}$ be the Jubjub generators of [Zcash-Protocol, §5.4.7.1] and [Zcash-Protocol, §4.2.2].

A Sapling spending key is $\mathsf{sk} : \mathbb{B}^{[\ell_{\mathsf{sk}}]}$. It is expanded as follows:

$$\begin{aligned}\mathsf{ask} &= \mathsf{ToScalar}^{\mathsf{Sapling}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x00}])\big),\\ \mathsf{nsk} &= \mathsf{ToScalar}^{\mathsf{Sapling}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x01}])\big),\\ \mathsf{ovk} &= \mathsf{truncate}_{32}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x02}])\big).\end{aligned}$$

Then $\mathsf{ak} = [\mathsf{ask}]\,\mathcal{G}^{\mathsf{Sapling}}$, $\mathsf{nk} = [\mathsf{nsk}]\,\mathcal{H}^{\mathsf{Sapling}}$, and $\mathsf{ivk} = \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{ak}), \mathsf{repr}_{\mathbb{J}}(\mathsf{nk}))$. If $\mathsf{ivk} = 0$, the key is discarded and a new sk is chosen.

A diversified payment address is obtained by choosing $\mathsf{d} : \mathbb{B}^{[\ell_{\mathsf{d}}]}$ such that $\mathsf{g_d} = \mathsf{DiversifyHash}^{\mathsf{Sapling}}(\mathsf{d}) \neq \bot$, and setting $\mathsf{pk_d} = [\mathsf{ivk}]\,\mathsf{g_d}$.

4.4.2 Orchard Key Components

An Orchard spending key is $\mathsf{sk} : \mathbb{BY}^{[32]}$. The key components are:

$$\begin{aligned}\mathsf{ask} &= \mathsf{ToScalar}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x06}])\big),\\ \mathsf{nk} &= \mathsf{ToBase}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x07}])\big),\\ \mathsf{rivk} &= \mathsf{ToScalar}^{\mathsf{Orchard}}\big(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{sk}}([\mathtt{0x08}])\big).\end{aligned}$$

If $\mathsf{ask} = 0$ the key is discarded. The point $\mathsf{ak}^{\mathbb{P}} = [\mathsf{ask}]\,\mathcal{G}^{\mathsf{Orchard}}$ is required to have a y-coordinate whose low bit is zero; if it does not, ask is negated. Then $\mathsf{ak} = \mathsf{Extract}_{\mathbb{P}}(\mathsf{ak}^{\mathbb{P}})$, and

$$\mathsf{ivk} = \mathsf{Commit}^{\mathsf{ivk}}_{\mathsf{rivk}}(\mathsf{ak}, \mathsf{nk}),$$

which is rejected if it is 0 or $\bot$. The diversifier key and outgoing viewing key are the two 32-byte halves of $\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rivk}}([\mathtt{0x82}] \,\|\, \mathsf{I2LEOSP}_{256}(\mathsf{ak}) \,\|\, \mathsf{I2LEOSP}_{256}(\mathsf{nk}))$, in the order dk, ovk.

The diversifier with index j is $\mathsf{d}_j = \mathsf{PRP}^{\mathsf{d}}_{\mathsf{dk}}(\mathsf{I2LEOSP}_{88}(j))$, and the corresponding address is

$$\big(\mathsf{d}_j,\ [\mathsf{ivk}]\,\mathsf{DiversifyHash}^{\mathsf{Orchard}}(\mathsf{d}_j)\big).$$

Every diversifier yields a valid Orchard address.

The normative definitions, including the edge cases above, are [Zcash-Protocol, §4.2.2] and [Zcash-Protocol, §4.2.3].

4.5 Spend Descriptions

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

- cv is the value commitment to the value of the input note;
- $\mathsf{rt}^{\mathsf{Sapling}}$ is an anchor, shared by all Spend descriptions of a version 5 transaction;
- nf is the nullifier of the input note;
- rk is a randomized validating key for spendAuthSig;
- π_{ZKSpend} is a Groth16 proof of the Spend statement [Zcash-Protocol, §4.18.2];
- spendAuthSig is a spend authorization signature over the transaction's signature hash.

Consensus rules:

- cv and rk MUST NOT be of small order.
- spendAuthSig MUST be a valid signature under rk on the signature hash of §4.10.
- The proof MUST be valid for the primary input $(\mathsf{cv}, \mathsf{rt}^{\mathsf{Sapling}}, \mathsf{nf}, \mathsf{rk})$.

4.6 Output Descriptions

An Output description is a tuple $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi_{\mathsf{ZKOutput}})$, where cv is the value commitment to the output note, cm_u is the u-coordinate of its note commitment, epk is an ephemeral key agreement public key, $\mathsf{C}^{\mathsf{enc}}$ and $\mathsf{C}^{\mathsf{out}}$ are the note and outgoing ciphertexts of §4.12, and π_{ZKOutput} is a Groth16 proof of the Output statement [Zcash-Protocol, §4.18.3].

Consensus rules:

- cv and epk MUST NOT be of small order.
- The proof MUST be valid for the primary input $(\mathsf{cv}, \mathsf{cm}_u, \mathsf{epk})$.

4.7 Action Descriptions

An Action description is a tuple $(\mathsf{cv}^{\mathsf{net}}, \mathsf{rt}^{\mathsf{Orchard}}, \mathsf{nf}, \mathsf{rk}, \mathsf{cm}_x, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \mathsf{enableSpends}, \mathsf{enableOutputs}, \pi)$, where $\mathsf{cv}^{\mathsf{net}}$ commits to the value of the spent note minus the value of the created note, $\mathsf{rt}^{\mathsf{Orchard}}$, enableSpends, and enableOutputs are shared by all Actions in the transaction, and π is a Halo 2 proof of the Action statement [Zcash-Protocol, §4.18.4].

Consensus rules:

- $\mathsf{cv}^{\mathsf{net}}$, rk, and epk MUST be valid encodings of Pallas points; rk and epk MUST NOT be the identity.
- The spend authorization signature of each Action MUST be valid under rk on the signature hash.
- The aggregated proof MUST be valid for the primary inputs of all Actions in the transaction, under the Action circuit of §6.1, and sizeProofsOrchard MUST be canonical for nActionsOrchard.
- In a coinbase transaction, enableSpends MUST be 0.

4.8 Computing ρ Values and Nullifiers

For a Sapling note with commitment cm at position pos,

$$\rho = \mathsf{MixingPedersenHash}(\mathsf{cm}, \mathsf{pos}), \qquad \mathsf{nf} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{repr}_{\mathbb{J}}(\mathsf{nk})}\big(\mathsf{repr}_{\mathbb{J}}(\rho)\big).$$

For an Orchard note $(\mathsf{d}, \mathsf{pk}_{\mathsf{d}}, \mathsf{v}, \rho, \psi, \mathsf{rcm})$ with commitment cm,

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

The ρ of each note created by an Action is the nullifier revealed by that Action, so every Orchard note has a unique ρ without reference to its tree position.

4.9 Balance and Binding Signatures

Let $\mathsf{v}^{\mathsf{balanceSapling}}$ and $\mathsf{v}^{\mathsf{balanceOrchard}}$ be the signed values of the transaction's valueBalanceSapling and valueBalanceOrchard fields.

For Sapling, the binding validating key is

$$\mathsf{bvk}^{\mathsf{Sapling}} = \Big(\textstyle\sum_i \mathsf{cv}_i^{\mathsf{spend}} - \sum_j \mathsf{cv}_j^{\mathsf{output}}\Big) - \mathsf{ValueCommit}_0^{\mathsf{Sapling}}\big(\mathsf{v}^{\mathsf{balanceSapling}}\big).$$

For Orchard,

$$\mathsf{bvk}^{\mathsf{Orchard}} = \Big(\textstyle\sum_i \mathsf{cv}^{\mathsf{net}}{}_i\Big) - \mathsf{ValueCommit}_0^{\mathsf{Orchard}}\big(\mathsf{v}^{\mathsf{balanceOrchard}}\big).$$

Consensus rule: If a transaction has any Spend or Output descriptions, bindingSigSapling MUST be a valid signature under $\mathsf{bvk}^{\mathsf{Sapling}}$ on its signature hash. If it has any Action descriptions, bindingSigOrchard MUST be a valid signature under $\mathsf{bvk}^{\mathsf{Orchard}}$ on its signature hash.

A valid binding signature shows that the signer knew the sum of the value commitment trapdoors, which is only possible if the committed values balance with the declared value balance.

4.10 Signature Hash

Transparent signatures, spend authorization signatures, and binding signatures are made over the signature digest of [ZIP-244]. The digest is computed with a BLAKE2b-256 personalization that includes the transaction's consensus branch identifier. Because Zecnero assigns its own branch identifiers (§5.13), a signature valid on Zecnero is not valid on Zcash and the reverse also holds.

4.11 Chain Value Pool Balances

The *chain value pool balance* of a pool at a given block is the total value that has entered the pool minus the total that has left it, over the chain up to and including that block.

Consensus rules:

- The transparent, Sapling, and Orchard chain value pool balances MUST be nonnegative at every block [ZIP-209].
- The total of all chain value pool balances MUST NOT exceed $\mathsf{MAX\_MONEY}$.

Non-normative note: Shielded balances are not otherwise visible. The turnstile rule means that a counterfeiting bug inside a shielded pool is detected when counterfeit value tries to leave the pool, and it bounds the loss to the value that entered.

4.12 In-band Secret Distribution

The sender of a note encrypts its plaintext to the recipient's diversified transmission key. The sender chooses an ephemeral private key esk, derived from rseed, publishes $\mathsf{epk} = [\mathsf{esk}]\,\mathsf{g_d}$, computes the shared secret $\mathsf{KA.Agree}(\mathsf{esk}, \mathsf{pk_d})$, and derives a symmetric key with KDF. The 564-byte note plaintext is encrypted with Sym to give the 580-byte $\mathsf{C^{enc}}$. A second key derived from ovk with $\mathsf{PRF^{ock}}$ encrypts $\mathsf{pk_d} \,\|\, \mathsf{esk}$ to give the 80-byte $\mathsf{C^{out}}$, which lets the holder of ovk recover sent notes.

A recipient trial-decrypts $\mathsf{C^{enc}}$ of each output using ivk, and accepts a note only if the recomputed note commitment matches the one on the chain. The procedures and their checks are specified in [Zcash-Protocol, §4.20].

5 Concrete Protocol

5.1 Integers, Bit Sequences, and Endianness

All integers in Zecnero-specific encodings, including block and transaction fields, are unsigned and little-endian unless stated otherwise. The exception is the nBits compact form, whose internal structure is defined in §5.5.1. Encodings inherited from Zcash follow [Zcash-Protocol, §5.1].

A 32-byte hash output interpreted as a 256-bit integer is read little-endian, that is, by $\mathsf{LEOS2IP}_{256}$. This applies both to SHA-256d block hashes when compared as numbers and to the output of $\mathsf{RandomX}^{\mathsf{Zecnero}}$.

5.2 Constants

Constant	Value
Currency	
COIN	10^8
$\mathsf{MAX\_MONEY}$	$21 \cdot 10^6 \cdot \mathsf{COIN} = 2\,100\,000\,000\,000\,000$
Issuance	
$\mathsf{MaxBlockSubsidy}$	$625\,000\,000$
$\mathsf{HalvingInterval}$	$1\,680\,000$
$\mathsf{FounderStreamNumerator}$	3
$\mathsf{FounderStreamDenominator}$	100

Constant	Value
FounderStreamEndHeight	1 680 000
CoinbaseMaturity	100
Block production	
BlockTargetSpacing	75
PoWLimit	$2^{243} - 1$
GenesisTarget	2^{236}
ASERTHalfLife	7 200
SeedHashEpochBlocks	2 048
SeedHashEpochLag	64
MedianTimePastWindow	11
MaxTimeAboveMedianTimePast	2 700
MaxFutureBlockTime	900
MaxReorgDepth	20
Shielded protocols	
$\mathsf{MerkleDepth}^{\mathsf{Sapling}}$	32
$\mathsf{MerkleDepth}^{\mathsf{Orchard}}$	32
ℓ_{value}	64
ℓ_{d}	88
ℓ_{sk}	256
ℓ_{ovk}	256
ℓ_{dk}	256
ℓ_{memo}	$512 \cdot 8$

Time constants are in seconds. Currency constants are in nero.

Non-normative notes:

- PoWLimit equals the Zcash Mainnet value. GenesisTarget corresponds to an expected 2^{20} evaluations of $\mathsf{RandomX}^{\mathsf{Zecnero}}$ per block, which a desktop processor computing about 14 000 hashes per second completes in about BlockTargetSpacing seconds.
- The shielded protocol constants are identical to those of Zcash [Zcash-Protocol, §5.3].

5.3 Concrete Cryptographic Schemes

Zecnero instantiates every scheme of §4.1 exactly as Zcash does at NU6. Implementations MUST use the definitions, constants, generators, and domain separation strings of the referenced sections without modification. In particular, BLAKE2 personalization strings that begin with "`Zcash`" are used unchanged; renaming them would change every derived key and commitment and make existing proving parameters unusable.

Scheme	Instantiation	Specified in
SHA-256d, SHA-512	FIPS 180-4	[Zcash-Protocol, §5.4.1.1]
BLAKE2b, BLAKE2s	RFC 7693 with personalization	[Zcash-Protocol, §5.4.1.2]
Sapling Merkle hash	Pedersen hash on Jubjub	[Zcash-Protocol, §5.4.1.3]
Orchard Merkle hash	Sinsemilla on Pallas	[Zcash-Protocol, §5.4.1.3]
$\mathsf{CRH}^{\mathsf{ivk}}$	BLAKE2s-256, truncated to 251 bits	[Zcash-Protocol, §5.4.1.5]
$\mathsf{DiversifyHash}^{\mathsf{Sapling}}$, $\mathsf{DiversifyHash}^{\mathsf{Orchard}}$	Group hash into Jubjub; hash-to-curve into Pallas	[Zcash-Protocol, §5.4.1.6]
Pedersen and mixing Pedersen hash	Jubjub	[Zcash-Protocol, §5.4.1.7]

Scheme	Instantiation	Specified in
Sinsemilla	Pallas	[Zcash-Protocol, §5.4.1.9]
$\mathsf{PRF}^{\mathsf{expand}}$	BLAKE2b-512	[Zcash-Protocol, §5.4.2]
$\mathsf{PRF}^{\mathsf{nfSapling}}$	BLAKE2s-256	[Zcash-Protocol, §5.4.2]
$\mathsf{PRF}^{\mathsf{nfOrchard}}$	Poseidon	[Zcash-Protocol, §5.4.2]
Sym	AEAD_CHACHA20_POLY1305 with zero nonce	[Zcash-Protocol, §5.4.3]
$\mathsf{PRP}^{\mathsf{d}}$	FF1-AES-256	[Zcash-Protocol, §5.4.4]
$\mathsf{KA}^{\mathsf{Sapling}}$, $\mathsf{KDF}^{\mathsf{Sapling}}$	Diffie–Hellman on Jubjub; BLAKE2b-256	[Zcash-Protocol, §5.4.5.3], [Zcash-Protocol, §5.4.5.4]
$\mathsf{KA}^{\mathsf{Orchard}}$, $\mathsf{KDF}^{\mathsf{Orchard}}$	Diffie–Hellman on Pallas; BLAKE2b-256	[Zcash-Protocol, §5.4.5.5], [Zcash-Protocol, §5.4.5.6]
SpendAuthSig, BindingSig	RedJubjub for Sapling, RedPallas for Orchard	[Zcash-Protocol, §5.4.7]
$\mathsf{NoteCommit}^{\mathsf{Sapling}}$	Windowed Pedersen commitment	[Zcash-Protocol, §5.4.8.2]
$\mathsf{ValueCommit}^{\mathsf{Sapling}}$, $\mathsf{ValueCommit}^{\mathsf{Orchard}}$	Homomorphic Pedersen commitment	[Zcash-Protocol, §5.4.8.3]
$\mathsf{NoteCommit}^{\mathsf{Orchard}}$, $\mathsf{Commit}^{\mathsf{ivk}}$	Sinsemilla commitment	[Zcash-Protocol, §5.4.8.4]
Jubjub, BLS12-381		[Zcash-Protocol, §5.4.9.2], [Zcash-Protocol, §5.4.9.3]
Pallas, Vesta		[Zcash-Protocol, §5.4.9.6]
Groth16	Over BLS12-381	[Zcash-Protocol, §5.4.10.2]
Halo 2	Over Pallas and Vesta	[Zcash-Protocol, §5.4.10.3]
Transparent signatures	ECDSA on secp256k1	[Bitcoin-Protocol]

5.4 RandomX-Zecnero

5.4.1 Parameters

$\mathsf{RandomX}^{\mathsf{Zecnero}}$ is the RandomX algorithm [RandomX] with the configuration below. Every parameter other than RANDOMX_ARGON_SALT equals the reference value of RandomX version 1.2.

Parameter	Value
RANDOMX_ARGON_MEMORY	262 144 (KiB)
RANDOMX_ARGON_ITERATIONS	3
RANDOMX_ARGON_LANES	1
RANDOMX_ARGON_SALT	"ZecneroRX" $\|\|$ [0x01] (10 bytes)
RANDOMX_CACHE_ACCESSES	8
RANDOMX_SUPERSCALAR_LATENCY	170
RANDOMX_DATASET_BASE_SIZE	2 147 483 648
RANDOMX_DATASET_EXTRA_SIZE	33 554 368
RANDOMX_PROGRAM_SIZE	256

Parameter	Value
RANDOMX_PROGRAM_ITERATIONS	2 048
RANDOMX_PROGRAM_COUNT	8
RANDOMX_JUMP_BITS	8
RANDOMX_JUMP_OFFSET	8
RANDOMX_SCRATCHPAD_L3	2 097 152
RANDOMX_SCRATCHPAD_L2	262 144
RANDOMX_SCRATCHPAD_L1	16 384

$\mathsf{RandomX}^{\mathsf{Zecnero}}(k, x) : \mathbb{BY}^{[32]}$ denotes the RandomX hash of input x computed with a virtual machine initialized from the cache for key k.

Security requirement: Implementations MUST NOT change any other parameter. The published analyses of RandomX [RandomX-Audits] cover the reference configuration. Changes to program size, iteration counts, dataset size, or instruction frequencies can invalidate them.

Non-normative note: Monero uses the salt "RandomX" $\|$ [0x03]. A different salt produces an unrelated cache, dataset, and hash function. Hardware and software that implement only Monero's configuration therefore cannot mine Zecnero. Nothing prevents anyone from adding $\mathsf{RandomX}^{\mathsf{Zecnero}}$ to a miner, and Zecnero does not rely on this property for security.

5.4.2 Key Schedule

Define $E = \mathsf{SeedHashEpochBlocks}$ and $L = \mathsf{SeedHashEpochLag}$. For $h : \mathbb{N}^+$,

$$\mathsf{SeedHeight}(h) = \begin{cases} 0, & \text{if } h \le E + L, \\ \mathsf{floor}\left(\dfrac{h - L - 1}{E}\right) \cdot E, & \text{otherwise.} \end{cases}$$

For a block at height h on a chain C:

$$\mathsf{RandomXKey}(h) = \begin{cases} [0\text{x}00]^{32}, & \text{if } h = 0, \\ \mathsf{BlockHash}(C_{\mathsf{SeedHeight}(h)}), & \text{otherwise,} \end{cases}$$

where C_j is the block of C at height j and BlockHash is given in its internal byte order, not in RPC order.

h	$\mathsf{SeedHeight}(h)$	h	$\mathsf{SeedHeight}(h)$
1	0	4 160	2 048
2 112	0	4 161	4 096
2 113	2 048	6 209	6 144

Non-normative notes:

- The key changes every 2 048 blocks, about 42.7 hours, and is fixed 64 blocks before its first use. This rule is Monero's [Monero-RX].
- The key depends on the chain. After a reorganization that replaces the block at $\mathsf{SeedHeight}(h)$, every block keyed by it MUST be verified again under the new key. The limit of §8.1 makes this impossible for keys more than MaxReorgDepth blocks deep.

5.4.3 Proof-of-Work Hash

Let $\mathsf{header}_{140}(B)$ be the first 140 bytes of the serialized header of block B, that is, every field before solutionSize (§5.6). For a block B at height h,

$$\mathsf{PoWHash}(B) = \mathsf{RandomX}^{\mathsf{Zecnero}}\big(\mathsf{RandomXKey}(h),\ \mathsf{header}_{140}(B)\big).$$

Consensus rule: $\mathsf{LEOS2IP}_{256}(\mathsf{PoWHash}(B))$ MUST be less than or equal to $\mathsf{ToTarget}(\mathsf{nBits})$, where nBits is the field of B's header.

Non-normative notes:

- PoWHash is unrelated to BlockHash. The block hash identifies blocks everywhere else in the protocol.
- Mining normally uses RandomX *fast mode*, which needs the full dataset of about 2080 MiB per key. Validation normally uses *light mode*, which needs only the 256 MiB cache and costs on the order of milliseconds per hash. Nodes SHOULD perform every other header check before computing PoWHash, and SHOULD keep caches for no more than the current and next key.
- The nonce occupies bytes 108 to 139 of header_{140}. Miners conventionally vary bytes 108 to 111. Pools assign the remaining bytes as an extra nonce.

5.5 Difficulty

5.5.1 nBits Conversion

A target is encoded in a block header as a 32-bit compact value nBits. Let $s = \mathsf{nBits} \gg 24$ and $w = \mathsf{nBits} \bmod 2^{23}$. Then

$$\mathsf{ToTarget}(\mathsf{nBits}) = \begin{cases} w \gg 8 \cdot (3 - s), & \text{if } s \leq 3, \\ w \ll 8 \cdot (s - 3), & \text{otherwise.} \end{cases}$$

The canonical encoding $\mathsf{ToCompact}(X)$ of a target $X \geq 1$ is computed as follows. Let s be the number of bytes in the minimal big-endian representation of X. If $s \leq 3$, let $w = X \ll 8 \cdot (3 - s)$; otherwise let $w = X \gg 8 \cdot (s - 3)$. If $w \geq 2^{23}$, set $w \leftarrow w \gg 8$ and $s \leftarrow s + 1$. Then $\mathsf{ToCompact}(X) = s \cdot 2^{24} + w$.

Consensus rule: Bit 23 of nBits MUST be zero, and $\mathsf{ToTarget}(\mathsf{nBits})$ MUST be at least 1 and at most PoWLimit.

Non-normative note: $\mathsf{ToCompact}(\mathsf{GenesisTarget}) = \texttt{0x1e100000}$ and $\mathsf{ToCompact}(\mathsf{PoWLimit}) = \texttt{0x1f07ffff}$. $\mathsf{ToTarget}(\texttt{0x1f07ffff})$ is slightly less than PoWLimit because the compact form discards low bits.

5.5.2 ASERT Difficulty Adjustment

The target of a block is determined by the algorithm aserti3-2d [BCH-ASERT], anchored at the genesis block.

Let t_0 be the nTime of the genesis block and $T_0 = \mathsf{GenesisTarget}$. Let the tip of the chain have height $h \geq 0$ and nTime t_h. The target $\mathsf{NextTarget}(h)$ for a block at height $h + 1$ is computed as follows. Intermediate values are signed integers of at least 64 bits, except in steps 6 to 9.

1. $\delta \coloneqq t_h - t_0 - \mathsf{BlockTargetSpacing} \cdot h$.
2. $e \coloneqq \mathsf{trunc}\big((\delta \cdot 65536) / \mathsf{ASERTHalfLife}\big)$.
3. $n \coloneqq e \gg 16$.
4. $f \coloneqq e - 65536 \cdot n$, so that $0 \leq f < 65536$.
5. $F \coloneqq 65536 + \big((195766423245049 \cdot f + 971821376 \cdot f^2 + 5127 \cdot f^3 + 2^{47}) \gg 48\big)$, computed without overflow.
6. $X \coloneqq T_0 \cdot F$, as an unsigned integer.
7. If $n < 0$, $X \coloneqq X \gg (-n)$. Otherwise $X \coloneqq X \ll n$.
8. $X \coloneqq X \gg 16$.
9. If $X = 0$, $X \coloneqq 1$. If $X > \mathsf{PoWLimit}$, $X \coloneqq \mathsf{PoWLimit}$.

$\mathsf{NextTarget}(h) = X$.

Consensus rule: The nBits field of a block at height $h + 1 \geq 1$ MUST equal $\mathsf{ToCompact}(\mathsf{NextTarget}(h))$. The nBits field of the genesis block MUST equal $\mathsf{ToCompact}(\mathsf{GenesisTarget})$.

Non-normative notes:

- Step 5 approximates $65536 \cdot 2^{f/65536}$ with a cubic polynomial whose relative error is below 0.013%. The constants, the rounding in steps 2 and 3, and the order of operations are those of aserti3-2d.
- aserti3-2d measures elapsed time from the timestamp of the anchor's parent and schedules the anchor itself one spacing after it. The genesis block has no parent, so Zecnero measures from t_0 and schedules the block at height h at $t_0 + \mathsf{BlockTargetSpacing} \cdot h$. Implementations checking against aserti3-2d test

vectors MUST account for this difference.
- Each ASERTHalfLife seconds by which the tip is behind schedule doubles the target, halving difficulty. Each ASERTHalfLife seconds ahead halves it.
- In step 7 the shift can make X very large when the chain is far behind schedule. An implementation MAY stop and set $X := \mathsf{PoWLimit}$ as soon as it is clear the result will exceed PoWLimit.
- The target is always computed from T_0, not from the rounded target of the previous block, so compact-form rounding does not accumulate.
- Because the schedule is fixed to t_0, t_0 MUST be the time public mining begins (§5.14). A genesis timestamp earlier than that would leave the chain behind schedule at launch and set difficulty far below the intended level.
- The Zcash Testnet minimum-difficulty rule does not exist on any Zecnero network.

A reference implementation is given in Appendix A.

5.5.3 Definition of Work

The *work* of a block with compact target nBits is

$$\mathsf{Work}(\mathsf{nBits}) = \mathsf{floor}\left(\frac{2^{256}}{\mathsf{ToTarget}(\mathsf{nBits}) + 1}\right).$$

The total work of a chain is the sum of the work of its blocks.

5.6 Block Header Encoding and Consensus

Bytes	Name	Data type	Description
4	nVersion	`int32`	Block version number.
32	hashPrevBlock	`byte[32]`	BlockHash of the parent block.
32	hashMerkleRoot	`byte[32]`	Merkle root of the transaction IDs of the block.
32	hashBlockCommitments	`byte[32]`	Commitment to the chain history tree and transaction authorizing data [ZIP-244].
4	nTime	`uint32`	Unix time in seconds.
4	nBits	`uint32`	Compact target (§5.5.1).
32	nNonce	`byte[32]`	Varied by miners.
1	solutionSize	`compactSize`	Length of solution.
0	solution	`byte[0]`	Empty.

The serialized header is 141 bytes. $\mathsf{BlockHash}(B) = \mathsf{SHA\text{-}256d}(\text{serialized header of } B)$.

Consensus rules:

- nVersion MUST be greater than or equal to 4.
- solutionSize MUST be 0. The Equihash rules of [Zcash-Protocol, §7.7.1] do not apply.
- nBits MUST satisfy §5.5.2, and the block MUST satisfy the proof-of-work rule of §5.4.3.
- nTime MUST satisfy §5.7.
- hashMerkleRoot and hashBlockCommitments MUST be computed as in Zcash at NU6 [Zcash-Protocol, §7.6]. The chain history tree of [ZIP-221] is maintained from height 1 using the Zecnero consensus branch identifier.
- The serialized size of a block MUST NOT exceed 2 000 000 bytes.

Non-normative note: Keeping the solution field as an empty vector preserves the header layout and serialization code of Zcash implementations. Explorers and light clients that read headers need only accept a zero-length solution.

5.7 Block Timestamps

$\mathsf{MedianTimePast}(h)$ is the median of the nTime fields of the blocks at heights from $\max(0,\ h - \mathsf{MedianTimePastWindow} + 1)$ to h inclusive.

Consensus rules:

- The nTime of a block at height $h \geq 1$ MUST be greater than MedianTimePast$(h-1)$.
- The nTime of a block at height $h \geq 1$ MUST be less than or equal to MedianTimePast$(h-1)$ + MaxTimeAboveMedianTimePast.

A node MUST NOT accept a block whose nTime is more than MaxFutureBlockTime seconds ahead of the node's clock. It MAY accept the block later when its clock has advanced.

Non-normative note: The second rule and the future limit are tighter than in Zcash, where the corresponding bounds are 90 minutes and 2 hours. With an exponential difficulty algorithm, a miner who can advance timestamps can lower difficulty for its own blocks, so the bounds are kept as tight as clock accuracy allows.

5.8 Transaction Encoding and Consensus

Zecnero accepts only version 5 transactions, encoded as specified in [ZIP-225] and [Zcash-Protocol, §7.1]. The fields are listed below for reference. In any discrepancy, [ZIP-225] governs.

Bytes	Name	Description
Common transaction fields		
4	header	fOverwintered = 1 and version 5: 0x80000005.
4	nVersionGroupId	0x26A7270A.
4	nConsensusBranchId	The Zecnero consensus branch identifier.
4	lock_time	As in Bitcoin.
4	nExpiryHeight	Height after which the transaction is invalid, or 0.
Transparent fields		
var	tx_in_count, tx_in	Transparent inputs.
var	tx_out_count, tx_out	Transparent outputs.
Sapling fields		
var	nSpendsSapling	Number of Spend descriptions.
$96 \cdot n$	vSpendsSapling	cv, nf, rk of each Spend.
var	nOutputsSapling	Number of Output descriptions.
$756 \cdot n$	vOutputsSapling	cv, cm_u, epk, $\mathsf{C}^{\mathsf{enc}}$ (580), $\mathsf{C}^{\mathsf{out}}$ (80).
8	valueBalanceSapling	Present if there are Spends or Outputs. Signed.
32	anchorSapling	Present if there are Spends.
$192 \cdot n$	vSpendProofsSapling	Groth16 proofs for Spends.
$64 \cdot n$	vSpendAuthSigsSapling	Spend authorization signatures.
$192 \cdot n$	vOutputProofsSapling	Groth16 proofs for Outputs.
64	bindingSigSapling	Present if there are Spends or Outputs.
Orchard fields		
var	nActionsOrchard	Number of Action descriptions.
$820 \cdot n$	vActionsOrchard	$\mathsf{cv}^{\mathsf{net}}$, nf, rk, cm_x, epk, $\mathsf{C}^{\mathsf{enc}}$, $\mathsf{C}^{\mathsf{out}}$.
1	flagsOrchard	Bit 0 enableSpends, bit 1 enableOutputs, other bits zero.
8	valueBalanceOrchard	Signed.
32	anchorOrchard	
var	sizeProofsOrchard, proofsOrchard	Aggregated Halo 2 proof.
$64 \cdot n$	vSpendAuthSigsOrchard	Spend authorization signatures.
64	bindingSigOrchard	

The Orchard fields after nActionsOrchard are present only if nActionsOrchard > 0.

Consensus rules:

- The transaction version MUST be 5, fOverwintered MUST be 1, and nVersionGroupId MUST be 0x26A7270A.
- nConsensusBranchId MUST equal the consensus branch identifier in effect at the height of the block containing the transaction (§6).
- A transaction MUST have at least one input (transparent, Spend, or Action with enableSpends = 1) and at least one output (transparent, Output, or Action with enableOutputs = 1).

- If nExpiryHeight is nonzero, the transaction MUST NOT be included in a block of greater height.
- All other rules for version 5 transactions in [Zcash-Protocol, §7.1.2] apply, with the Zecnero branch identifier and without the rules conditioned on network upgrades before NU5.

Non-normative note: Zcash also accepts version 4 transactions. Zecnero has no history before NU5, so it accepts only version 5, which removes one encoding and its rules from every implementation.

5.9 Block Subsidy, Founder Stream, and Coinbase

5.9.1 Block Subsidy

For $h : \mathbb{N}^+$, let $\mathsf{Halving}(h) = \mathsf{floor}(h/\mathsf{HalvingInterval})$. The block subsidy in nero is

$$\mathsf{BlockSubsidy}(h) = \begin{cases} \mathsf{MaxBlockSubsidy} \gg \mathsf{Halving}(h), & \text{if } \mathsf{Halving}(h) < 64, \\ 0, & \text{otherwise.} \end{cases}$$

The genesis block has no spendable subsidy (§5.14). There is no slow start.

$\mathsf{BlockSubsidy}(h)$ reaches zero when $\mathsf{Halving}(h) = 30$, at height 50 400 000, about 119.8 years after genesis. From then on, block producers are paid by transaction fees alone. Total issuance is 2 099 999 356 520 000 nero, that is, 20 999 993.5652 ZMR. The complete schedule is given in Appendix B.

Halving period	Subsidy (ZMR)	Issued in period (ZMR)	Cumulative (ZMR)
1	6.25	10 499 993.75	10 499 993.75
2	3.125	5 250 000	15 749 993.75
3	1.5625	2 625 000	18 374 993.75
4	0.78125	1 312 500	19 687 493.75
5	0.390625	656 250	20 343 743.75
6	0.1953125	328 125	20 671 868.75

Non-normative note: The first halving period contains heights 1 to $\mathsf{HalvingInterval} - 1$, one block fewer than later periods, because height 0 is the genesis block.

5.9.2 Founder Stream

FounderScriptPubKey is a transparent P2SH script, fixed in the reference implementation and published before launch. For $h : \mathbb{N}^+$,

$$\mathsf{FounderStreamValue}(h) = \begin{cases} \mathsf{floor}(\mathsf{BlockSubsidy}(h) \cdot a / b), & \text{if } h < \mathsf{FounderStreamEndHeight}, \\ 0, & \text{otherwise,} \end{cases}$$

where $a = \mathsf{FounderStreamNumerator}$ and $b = \mathsf{FounderStreamDenominator}$, and $\mathsf{MinerSubsidy}(h) = \mathsf{BlockSubsidy}(h) - \mathsf{FounderStreamValue}(h)$.

Consensus rules:

- For $1 \le h < \mathsf{FounderStreamEndHeight}$, the coinbase transaction of the block at height h MUST have at least one transparent output whose scriptPubKey equals FounderScriptPubKey and whose value is at least $\mathsf{FounderStreamValue}(h)$.
- No other coinbase output is required. The Founders' Reward, funding streams, deferred lockbox, and lockbox disbursement rules of Zcash [Zcash-Protocol, §7.8]–[Zcash-Protocol, §7.10] do not apply.

At MaxBlockSubsidy, $\mathsf{FounderStreamValue}(h) = 18\,750\,000$ nero, which is 0.1875 ZMR. Over the first halving period the founder stream totals 314 999.8125 ZMR: 3% of issuance in that period and 1.5% of total issuance.

Non-normative notes:

- The founder stream is paid by every block below FounderStreamEndHeight, whoever mines it. No height is reserved for, or restricted to, any miner.

- Coins the founder obtains by mining are not part of the founder stream. The founder's mining addresses are published before launch (§10).
- A P2SH script allows the founder stream to be paid to a multisignature script, so that it is not lost with a single key.

5.9.3 Coinbase Transactions

Consensus rules:

- The first transaction of every block MUST be a coinbase transaction, and no other transaction in the block may be one.
- The coinbase input script MUST begin with the block height, encoded as in [BIP-34].
- nExpiryHeight of a coinbase transaction MUST equal the block height.
- A coinbase transaction MUST NOT have Spend descriptions, and its enableSpends flag MUST be 0.
- The total value of the outputs of a coinbase transaction MUST equal $\mathsf{BlockSubsidy}(h)$ plus the fees of the other transactions in the block.
- Sapling and Orchard outputs of a coinbase transaction MUST be decryptable with the all-zero outgoing viewing key [ZIP-213].
- Transparent outputs of a coinbase transaction MUST NOT be spent by a transaction in a block less than CoinbaseMaturity blocks after the block containing the coinbase. Other rules on spending coinbase outputs are as in [Zcash-Protocol, §7.1.2].

5.10 Encodings of Note Plaintexts and Memo Fields

A Sapling or Orchard note plaintext is encoded as 564 bytes: leadByte (1), d (11), v as $\mathsf{I2LEOSP}_{64}$ (8), rseed (32), memo (512). The encoding is that of [Zcash-Protocol, §5.5]. Encryption adds a 16-byte authentication tag, giving the 580-byte $\mathsf{C}^{\mathsf{enc}}$.

5.11 Encodings of Addresses and Keys

5.11.1 Transparent Addresses

Transparent addresses are encoded with Base58Check [Bitcoin-Base58]. The payload is a 2-byte prefix followed by the 20-byte hash of a public key (P2PKH) or of a script (P2SH). The prefixes are chosen so that every address of a given kind begins with the same two characters.

Kind	Mainnet prefix	Mainnet leading	Testnet prefix	Testnet leading
P2PKH	[0x0B, 0xD3]	t1	[0x19, 0xC4]	tm
P2SH	[0x0B, 0xD8]	t3	[0x19, 0x59]	t2

Every encoded transparent address is 35 characters long. Transparent private keys are encoded as in Zcash [Zcash-Protocol, §5.6.1.2].

5.11.2 Sapling Encodings

Sapling payment addresses, viewing keys, and spending keys are encoded with Bech32 [BIP-173] using the raw encodings of [Zcash-Protocol, §5.6.3] and [ZIP-32], with the following human-readable parts.

Encoding	Mainnet HRP	Testnet HRP
Payment address	zs	ztestsapling
Incoming viewing key	zivks	zivktestsapling
Full viewing key	zviews	zviewtestsapling
Extended full viewing key	zxviews	zxviewtestsapling
Spending key	zmr-secret-spending-key-main	zmr-secret-spending-key-test
Extended spending key	zmr-secret-extended-key-main	zmr-secret-extended-key-test

5.11.3 Unified and Orchard Encodings

Unified addresses and viewing keys are encoded as specified in [ZIP-316], with F4Jumble and Bech32m, and with the following human-readable parts. Receiver typecodes are those of [ZIP-316]. Orchard raw payment addresses are 43 bytes: d (11) followed by $\mathsf{repr}_{\mathbb{P}}(\mathsf{pk_d})$ (32).

Encoding	Mainnet HRP	Testnet HRP
Unified address	`nu`	`nutest`
Unified full viewing key	`nuview`	`nuviewtest`
Unified incoming viewing key	`nuivk`	`nuivktest`

Consensus rule: None; these encodings are not consensus-critical. Implementations MUST nevertheless reject Zcash encodings when a Zecnero encoding is expected, so that users cannot send ZMR to an address typed for Zcash or the reverse.

5.12 Groth16 zk-SNARK Parameters

Sapling proofs are created and verified with the Spend and Output parameters generated by the Zcash Sapling multi-party computation [BGM2017]. Zecnero uses those parameters without change. Implementations MUST check the SHA-256 digests of the parameter files against those published by Zcash before use.

Halo 2 proofs for Orchard require no parameter generation.

5.13 Network Parameters

Parameter	Mainnet	Testnet
Network magic	`[0x7A, 0x6D, 0x72, 0xD1]`	`[0x7A, 0x6D, 0x72, 0xE3]`
Default peer-to-peer port	8733	18733
Default RPC port	8732	18732
Initial consensus branch identifier	`0x01524D5A`	`0x01524D5A`
Protocol version	180000	180000
SLIP-44 coin type	*[unassigned]*	1
DNS seeds	*[unassigned]*	*[unassigned]*

Non-normative notes:

- The branch identifier `0x01524D5A` serializes little-endian as the bytes of "`ZMR`" $\|\|$ `[0x01]`. It differs from every Zcash branch identifier.
- Mainnet and Testnet share a branch identifier, as in Zcash. Transactions are not protected against replay between the two networks, whose currencies are distinguished only by genesis block.
- A SLIP-44 coin type will be registered before Mainnet launch.

5.14 Genesis Block

The genesis block has height 0, a zero hashPrevBlock, and one transaction. As in Zcash, the genesis transaction is a version 1 transaction; the rules of §5.8 and §5.9.3 apply from height 1.

Consensus rules:

- The genesis coinbase input script MUST contain the ASCII string "`Zecnero`", the block hash of a Bitcoin block, and the height of that Bitcoin block. The Bitcoin block MUST have been mined no more than 24 hours before t_0.
- The genesis coinbase transaction MUST have exactly one output, with value 0 and scriptPubKey consisting of the single opcode `OP_RETURN`.
- The genesis nTime is t_0, the announced time at which the software is released and public mining begins.

- The genesis nBits is ToCompact(GenesisTarget) and the block satisfies it under the all-zero key of §5.4.2.

Non-normative note: The Bitcoin block commitment proves that the genesis block did not exist before that Bitcoin block was mined. Together with the ASERT schedule anchored at t_0, it bounds the time before launch in which any block could have been produced.

6 Network Upgrades

Zecnero changes its consensus rules by *network upgrades*, following the mechanism of [ZIP-200]. Each upgrade has an activation height and a new consensus branch identifier. Blocks at or above the activation height are validated under the new rules, and transactions in them MUST carry the new branch identifier.

The rules of Zecnero at launch are defined as the network upgrade Genesis. It incorporates the Zcash upgrades Overwinter, Sapling, Blossom, Heartwood, Canopy, NU5, and NU6, together with the security fixes of §6.1 and the changes of §7.

Upgrade	Branch identifier	Mainnet height	Testnet height
Genesis	`0x01524D5A`	1	1

Changes made to Zcash after NU6 do not apply to Zecnero unless adopted in a later Zecnero network upgrade specified in a revision of this document, with the exception of the security fixes listed in §6.1, which Genesis incorporates from height 1. Those fixes correct unsound or malleable behaviour in rules Zecnero inherits; they add no feature, change no issuance, and are not conditioned on any Zcash activation height.

A network upgrade is *settled* when there is social consensus that it activated at a given block hash. A full validator that risks Mainnet funds or displays Mainnet transaction information to a user MUST do so only on a chain that includes the activation block of the most recent settled upgrade.

6.1 Adopted Post-NU6 Security Fixes

Zecnero has no block chain history before its own genesis block, so it does not need to verify blocks produced under superseded Zcash rules. Where Zcash retains a rule only for that purpose, Zecnero uses the corrected rule from height 1.

Consensus rules:

- The Action statement of §4.7 is the Orchard Action circuit introduced in Zcash NU6.2, in which the base of the variable-base scalar multiplication is anchored. The circuit used by Zcash from NU5 until NU6.2 is not sound; it MUST NOT be used for proving or verification on any Zecnero network, and Action proofs MUST verify under the verifying key of the anchored-base circuit.
- sizeProofsOrchard MUST equal the size determined by nActionsOrchard for the aggregated Halo 2 proof. A transaction whose Orchard proof field is longer than that size MUST be rejected [GHSA-jfw5].

Non-normative note: Zcash reached these rules at NU6.2, after a hotfix that temporarily disabled Orchard Action transfers. Zecnero applies them from height 1 and never enables the temporary disabling rule, whose Zcash activation heights have no meaning on a Zecnero chain.

7 Consensus Changes from Zcash

This section lists every consensus difference between Zecnero at Genesis and Zcash at NU6. Anything not listed here, or specified differently elsewhere in this document, is as in Zcash.

7.1 Removed

- The Sprout shielded protocol: JoinSplit descriptions, the Sprout note commitment tree and nullifier set, Ed25519 JoinSplit signatures, and BCTV14 proofs.
- Version 1 to 4 transactions, for blocks at height 1 or above.
- The Founders' Reward, funding streams, deferred development fund lockbox, and lockbox disbursement.
- The slow start interval and the Blossom subsidy adjustment.
- Equihash.
- The Zcash difficulty adjustment algorithm and the Testnet minimum-difficulty rule.
- Rules that exist only to manage the transition into an upgrade that Zecnero activates at height 1, such as the grace period for note plaintext lead byte `0x01`.

7.2 Replaced

- Proof of work: $\mathsf{RandomX}^{\mathsf{Zecnero}}$ over the 140-byte header prefix, with an empty solution field (§5.4, §5.6).
- Difficulty adjustment: aserti3-2d anchored at genesis (§5.5.2).
- Block subsidy: 6.25 ZMR halving every 1 680 000 blocks (§5.9).
- Timestamp bounds: MaxTimeAboveMedianTimePast and MaxFutureBlockTime (§5.7).
- Network identity: magic, ports, branch identifier, address encodings, and genesis block (§5.11, §5.13, §5.14).
- The Orchard Action circuit and the encoding rule for the aggregated Orchard proof: the corrected NU6.2 versions apply from height 1 (§6.1).

7.3 Added

- The founder stream (§5.9.2).
- The requirement that the genesis block commit to a recent Bitcoin block (§5.14).
- The maximum reorganization depth and checkpoint rules (§8).

8 Chain Selection

8.1 Maximum Reorganization Depth

A node MUST NOT switch its best valid block chain to a chain that would require disconnecting more than MaxReorgDepth blocks from its current best valid block chain, even if the other chain has greater total work.

8.2 Checkpoints

Each release of a full validator contains a list of (height, block hash) checkpoints on the Mainnet chain. A node MUST NOT accept a chain that conflicts with any checkpoint in its release. A node MAY skip verification of proofs of work and zk-SNARK proofs for blocks at or below the highest checkpoint that it has verified to be an ancestor of its chain.

Non-normative notes:

- These are rules of node behaviour, not block validity. Two correctly operating nodes can disagree about the best chain if they have observed blocks in different orders. The rules are intended to protect nodes that are already synchronized against an attacker who mines a longer chain in private and then publishes it.
- Zcash implementations limit rollback to 100 blocks. At BlockTargetSpacing seconds, MaxReorgDepth blocks is 25 minutes.
- If the network partitions for longer than MaxReorgDepth blocks, the parts will not reconverge without

operators choosing a chain.
- A node that synchronizes from genesis is protected by checkpoints only up to the most recent one in its release. Beyond it, the node relies on connecting to honest peers.

9 Rationale

This section is non-normative.

9.1 Choice of Proof of Work

Zecnero is intended to let ordinary computers compete with specialized operations on equal terms per unit of energy. The most efficient hardware for a proof of work should therefore resemble the hardware people already own. RandomX achieves this by executing randomly generated programs whose instruction mix, memory hierarchy use, and branch behaviour are modelled on general-purpose processors. A device designed to run RandomX efficiently converges toward a general-purpose processor, and processor manufacturers already produce those at the largest scale in the industry.

RandomX has secured Monero since November 2019 and was reviewed in four independent audits before deployment. Dedicated RandomX hardware sold to date has been reported to offer little efficiency advantage over high-end desktop processors.

Other options were considered:

- *Equihash with new parameters.* Equihash performs well on GPUs, and ASICs supporting several parameter sets have been sold. New parameters would delay specialized hardware without preventing it.
- *ProgPoW, Autolykos, Cuckaroo.* These are designed for GPUs. A large supply of idle GPU mining equipment exists and would dominate the early chain.
- *A new algorithm.* An unreviewed proof of work may contain shortcuts. Whoever first writes an optimized implementation gains a private advantage during launch, which is the outcome Zecnero is designed to avoid.

A private salt was preferred to changing other RandomX parameters because it separates Zecnero from Monero's hardware and hashrate market while leaving the audited design untouched.

9.2 Choice of Difficulty Algorithm

A new chain experiences large and sudden changes in hashrate. The Zcash algorithm averages 17 block intervals and limits each adjustment, so after a large miner leaves, blocks can remain slow for a long time. ASERT depends only on the anchor and the current tip, has no window to manipulate, and recovers from a change of any size within a few half-lives. It has been analysed at length and has run on Bitcoin Cash since November 2020.

A half-life of two hours was chosen over Bitcoin Cash's two days because Zecnero blocks are eight times more frequent and because a small chain needs to react within hours rather than days.

9.3 Issuance

The schedule follows Bitcoin and Zcash: a fixed maximum supply of about 21 million, with a halving about every four years. After issuance ends, block producers are paid only by fees. Monero instead pays a permanent tail subsidy. A tail was considered and not adopted; the question can be revisited by network upgrade long before issuance becomes small.

9.4 Founder Allocation

The founder stream is fixed in consensus and published before launch so that the full allocation is known to anyone deciding whether to participate. It is paid by every block regardless of who mines it and ends

at the first halving. There is no period in which only the founder may mine, and no allocation other than the stream.

9.5 Removal of Sprout

Sprout was the original Zcash shielded protocol. Its proof system, BCTV14, was found in 2018 to contain a flaw that would have allowed undetectable counterfeiting; the flaw was repaired by moving Sprout to Groth16 with the Sapling upgrade. Creating Sprout proofs was slow and memory-intensive, and Zcash has prohibited new value from entering the Sprout pool since Canopy. A new chain has no Sprout funds to support, current wallets do not create Sprout transactions, and carrying the code would add risk without benefit.

10 Launch Procedure

This section is non-normative.

1. A public Testnet operates for several weeks with independent miners before Mainnet parameters are fixed.
2. At least one week before launch, the following are published: t_0; the source code of the reference implementation; FounderScriptPubKey and its address; and the addresses from which the founder will mine.
3. Shortly before t_0, a recent Bitcoin block is selected and the genesis block is constructed and mined.
4. At t_0, the node and miner software, reproducible build hashes, and the genesis block hash are published together.
5. From t_0, anyone may mine. The founder mines with the published software from the published addresses under the same rules as every other participant, and does not mine before the software is public. This is a commitment, not a protocol guarantee; the Bitcoin block committed to in the genesis block bounds how early any block could have been produced.

Participation in the first hours depends on how widely the launch is known, and the founder may produce a large share of early blocks. Publishing the founder's mining addresses in advance lets anyone measure that share.

11 Security Considerations

This section is non-normative.

11.0.0.1 Majority attacks. A new chain has little hashrate, and processor time can be rented in bulk from cloud providers. An attacker with most of the hashrate can censor transactions and reverse recent ones. The rule of §8.1 bounds reversals seen by running nodes to MaxReorgDepth blocks. Recipients of significant payments SHOULD wait for at least MaxReorgDepth confirmations. A finality mechanism that does not depend on hashrate is intended to be specified before ZMR is listed on any exchange.

11.0.0.2 Compromised machines. A proof of work efficient on general-purpose processors is also efficient on compromised computers and misused cloud accounts. The protocol cannot distinguish them from other miners.

11.0.0.3 Specialized hardware. If hardware with a large efficiency advantage for RandomX$^{\text{Zecnero}}$ appears, the proof of work can be changed by network upgrade.

11.0.0.4 Timestamp manipulation. ASERT responds to timestamps directly. The bounds of §5.7 limit how far a miner can move time, and so how much it can lower difficulty for its own blocks.

11.0.0.5 Trusted setup. The soundness of Sapling depends on at least one participant in the Zcash Sapling ceremony having destroyed their secret. Orchard has no such assumption. Wallets SHOULD prefer Orchard.

11.0.0.6 Cross-network linkability. Keys and addresses derived from one seed are identical on Zcash and Zecnero unless wallets separate them. See §3.1.

11.0.0.7 Implementation risk. Zecnero modifies consensus-critical code. A difference between implementations, or between an implementation and this document, can split the network. Security fixes to upstream Zcash implementations must be reviewed and adopted promptly.

11.0.0.8 Economic expectations. A miner's expected share of issuance equals its share of total hashrate. Zecnero aims to keep ordinary processors competitive per unit of energy. It does not make any miner profitable.

11.0.0.9 Regulation. Assets with strong privacy face delisting and legal restrictions in several jurisdictions.

12 Acknowledgements

This specification depends throughout on the Zcash Protocol Specification by Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox, and on the Zcash Improvement Proposals and their authors. The reference implementation is derived from Zebra, developed by the Zcash Foundation. Zcash is based on Zerocash, by Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. RandomX was designed by tevador with contributions from the Monero community. The aserti3-2d algorithm was developed for Bitcoin Cash by Mark Lundeberg, Jonathan Toomim, and others. Any errors in this document are the responsibility of the Zecnero developers.

13 Change History

13.0.0.1 2026.0.3-draft 2026-09-19

- Adopted the NU6.2 Orchard security fixes from height 1: the anchored-base Action circuit and the canonical Orchard proof size (§6.1).
- Stated that the temporary Orchard-disabling rule of the Zcash hotfix does not apply to Zecnero.

13.0.0.2 2026.0.2-draft 2026-09-19

- Rewritten as a self-contained specification in the structure of the Zcash Protocol Specification.
- Removed the tail subsidy. Issuance ends after 30 halvings.
- Assigned transparent address prefixes, network magic, ports, and the initial branch identifier.
- Restricted transactions to version 5.
- Added a reference implementation of the difficulty algorithm and the full issuance schedule.

13.0.0.3 2026.0.1-draft 2026-09-19

- Initial draft, specified as a set of differences from Zcash.

14 References

BCGGMTV2014 Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin*. In Proceedings of the IEEE Symposium on Security and Privacy, 2014.

BCH-ASERT Mark Lundeberg, Jonathan Toomim, et al. *ASERT Difficulty Adjustment Algorithm (aserti3-2d)*. Bitcoin Cash upgrade specification, November 2020.

BGM2017 Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050.

BIP-34 Gavin Andresen. *Block v2, Height in Coinbase*. Bitcoin Improvement Proposal 34.

BIP-173 Pieter Wuille and Greg Maxwell. *Base32 address format for native v0-16 witness outputs*. Bitcoin Improvement Proposal 173.

Bitcoin-Base58 *Base58Check encoding*. Bitcoin Wiki.

Bitcoin-Protocol *Protocol documentation*. Bitcoin Wiki.

GHSA-jfw5 Zcash Foundation. *Zebra security advisory GHSA-jfw5-j458-pfv6: Orchard Action circuit soundness and non-canonical Orchard proof size*. 2026. https://github.com/ZcashFoundation/zebra/security/advisories/GHSA-jfw5-j458-pfv6

Monero-RX The Monero Project. *RandomX seed height computation*, `rx_seedheight` in the Monero source code.

Nakamoto2008 Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008.

NIST2015 NIST. *Secure Hash Standard (SHS)*. FIPS PUB 180-4, August 2015.

RandomX tevador et al. *RandomX: specification, design notes, and reference implementation*, version 1.2.

RandomX-Audits Trail of Bits, X41 D-Sec, Kudelski Security, and QuarksLab. *Security audits of RandomX*. 2019.

RFC-2119 Scott Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. BCP 14, RFC 2119, March 1997.

Zcash-Protocol Daira-Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. *Zcash Protocol Specification*, version 2026.7.0 [NU6.2] or any later version that does not change the NU6 rules referenced here. https://zips.z.cash/protocol/protocol.pdf

ZIP-32 Jack Grigg and Daira-Emma Hopwood. *Shielded Hierarchical Deterministic Wallets*.

ZIP-200 Jack Grigg. *Network Upgrade Mechanism*.

ZIP-209 Sean Bowe. *Prohibit Negative Shielded Chain Value Pool Balances*.

ZIP-212 Sean Bowe. *Allow Recipient to Derive Ephemeral Secret from Note Plaintext*.

ZIP-213 Jack Grigg. *Shielded Coinbase*.

ZIP-221 Jack Grigg. *FlyClient: Consensus-Layer Changes*.

ZIP-225 Daira-Emma Hopwood, Jack Grigg, Sean Bowe, Kris Nuttycombe, and Ying Tong Lai. *Version 5 Transaction Format*.

ZIP-239 Daira-Emma Hopwood and Jack Grigg. *Relay of Version 5 Transactions*.

ZIP-244 Kris Nuttycombe, Daira-Emma Hopwood, and Jack Grigg. *Transaction Identifier Non-Malleability*.

ZIP-302 Jay Graber. *Standardized Memo Field Format*.

ZIP-316 Daira-Emma Hopwood, Nathan Wilcox, Taylor Hornby, Jack Grigg, Sean Bowe, Kris Nuttycombe, and Ying Tong Lai. *Unified Addresses and Unified Viewing Keys*.

A Reference Implementation of the Difficulty Algorithm

The following Rust function computes NextTarget(h) of §5.5.2. It uses a 512-bit unsigned integer type `U512` for steps 6 to 9. It is normative only to the extent that it agrees with §5.5.2.

```
const TARGET_SPACING: i64 = 75;
const HALF_LIFE: i64 = 7_200;

/// Returns the target for the block after `tip_height`, per spec section 5.5.2.
pub fn next_target(genesis_time: i64, tip_height: i64, tip_time: i64) -> U512 {
    let genesis_target = U512::one() << 236;
    let pow_limit = (U512::one() << 243) - 1;

    let delta = tip_time - genesis_time - TARGET_SPACING * tip_height;
    // Rust integer division truncates toward zero, as step 2 requires.
    let exponent = (delta as i128 * 65_536 / HALF_LIFE as i128) as i64;
    // Arithmetic shift rounds toward negative infinity, as step 3 requires.
    let shifts = exponent >> 16;
```

```
    let frac = (exponent - shifts * 65_536) as u128;

    let factor = 65_536
        + ((195_766_423_245_049u128 * frac
            + 971_821_376u128 * frac * frac
            + 5_127u128 * frac * frac * frac
            + (1u128 << 47))
            >> 48);

    let mut target = genesis_target * U512::from(factor);
    if shifts < 0 {
        target >>= (-shifts) as u32;
    } else if shifts >= 256 {
        return pow_limit;
    } else {
        target <<= shifts as u32;
    }
    target >>= 16;

    if target.is_zero() {
        U512::one()
    } else if target > pow_limit {
        pow_limit
    } else {
        target
    }
}
```

Worked examples

Each row gives the tip height h, the elapsed time $t_h - t_0$ in seconds, and the resulting compact target of the next block.

h	$t_h - t_0$	Position of tip	$\mathsf{NextTarget}(h)/T_0$	ToCompact
0	0	genesis	1	`0x1e100000`
100	7 500	on schedule	1	`0x1e100000`
100	11 100	1 hour behind	1.41409 …	`0x1e16a020`
100	14 700	2 hours behind	2	`0x1e200000`
100	300	2 hours ahead	0.5	`0x1e080000`
1 000	161 400	24 hours behind	clamped	`0x1f07ffff`

B Issuance Schedule

Subsidy per block and cumulative issuance at the end of each halving period. Period k covers heights $(k-1) \cdot \mathsf{HalvingInterval}$ to $k \cdot \mathsf{HalvingInterval} - 1$, excluding height 0.

Period	Subsidy (nero)	Issued in period (ZMR)	Cumulative (ZMR)
1	625 000 000	10 499 993.75	10 499 993.75
2	312 500 000	5 250 000	15 749 993.75
3	156 250 000	2 625 000	18 374 993.75
4	78 125 000	1 312 500	19 687 493.75
5	39 062 500	656 250	20 343 743.75
6	19 531 250	328 125	20 671 868.75
7	9 765 625	164 062.5	20 835 931.25
8	4 882 812	82 031.2416	20 917 962.4916
9	2 441 406	41 015.6208	20 958 978.1124
10	1 220 703	20 507.8104	20 979 485.9228
11	610 351	10 253.8968	20 989 739.8196

Period	Subsidy (nero)	Issued in period (ZMR)	Cumulative (ZMR)
12	305 175	5 126.94	20 994 866.7596
13	152 587	2 563.4616	20 997 430.2212
14	76 293	1 281.7224	20 998 711.9436
15	38 146	640.8528	20 999 352.7964
16	19 073	320.4264	20 999 673.2228
17	9 536	160.2048	20 999 833.4276
18	4 768	80.1024	20 999 913.53
19	2 384	40.0512	20 999 953.5812
20	1 192	20.0256	20 999 973.6068
21	596	10.0128	20 999 983.6196
22	298	5.0064	20 999 988.626
23	149	2.5032	20 999 991.1292
24	74	1.2432	20 999 992.3724
25	37	0.6216	20 999 992.994
26	18	0.3024	20 999 993.2964
27	9	0.1512	20 999 993.4476
28	4	0.0672	20 999 993.5148
29	2	0.0336	20 999 993.5484
30	1	0.0168	20 999 993.5652
31 onward	0	0	20 999 993.5652