

ZKPrivacy

Quantum-Secure Privacy Blockchain

Technical Specification v1.0

Draft – January 2026

Phexora AI

<https://quantum.phexora.ai>

This specification is released under CC0 (Public Domain).

Designed for AI-verifiable implementation.

Contents

1	ZKPrivacy: Quantum-Secure Privacy Blockchain	1
1.1	Formal Specification v1.0	1
2	IMMUTABLE REQUIREMENTS	1
2.1	¤ THIS SECTION IS IMMUTABLE ¤	1
2.2	R1. PRIVACY REQUIREMENTS	1
2.2.1	R1.1 Privacy by Default [MANDATORY]	1
2.2.2	R1.2 Sender Privacy [MANDATORY]	1
2.2.3	R1.3 Receiver Privacy [MANDATORY]	2
2.2.4	R1.4 Amount Privacy [MANDATORY]	2
2.2.5	R1.5 Network Privacy [MANDATORY]	2
2.3	R2. SECURITY REQUIREMENTS	2
2.3.1	R2.1 Quantum Security [MANDATORY]	2
2.3.2	R2.2 No Trusted Setup [MANDATORY]	2
2.3.3	R2.3 Cryptographic Binding [MANDATORY]	3
2.3.4	R2.4 Cryptographic Hiding [MANDATORY]	3
2.3.5	R2.5 Proof Soundness [MANDATORY]	3
2.3.6	R2.6 Proof Zero-Knowledge [MANDATORY]	3
2.4	R3. DECENTRALIZATION REQUIREMENTS	3
2.4.1	R3.1 Permissionless Participation [MANDATORY]	3
2.4.2	R3.2 No Privileged Parties [MANDATORY]	3
2.4.3	R3.3 ASIC Resistance [MANDATORY]	4
2.4.4	R3.4 Open Source [MANDATORY]	4
2.5	R4. INTEGRITY REQUIREMENTS	4
2.5.1	R4.1 Fixed Supply [MANDATORY]	4
2.5.2	R4.2 No Inflation Bugs [MANDATORY]	4
2.5.3	R4.3 Double-Spend Prevention [MANDATORY]	4
2.5.4	R4.4 Transaction Finality [MANDATORY]	4
2.6	R5. FUNCTIONAL REQUIREMENTS	5
2.6.1	R5.1 Basic Transaction Support [MANDATORY]	5
2.6.2	R5.2 Wallet Functionality [MANDATORY]	5
2.6.3	R5.3 Light Client Support [MANDATORY]	5
2.7	R6. PERFORMANCE REQUIREMENTS	5
2.7.1	R6.1 Transaction Processing [MANDATORY]	5
2.7.2	R6.2 Storage [MANDATORY]	5
2.7.3	R6.3 Network [MANDATORY]	5
2.8	R7. NON-REQUIREMENTS (Explicitly Excluded)	6
2.8.1	R7.1 NOT Required	6
2.8.2	R7.2 NOT Permitted	6

2.9	Requirement Compliance Matrix	6
2.10	Immutability Declaration	7
3	END OF IMMUTABLE REQUIREMENTS	8
4	Part I: Cryptographic Foundation	8
4.1	1. Notation and Conventions	8
4.1.1	1.1 Mathematical Notation	8
4.1.2	1.2 Security Parameter	8
4.1.3	1.3 Endianness and Encoding	8
4.2	2. Hash Functions	8
4.2.1	2.1 Primary Hash Function: SHAKE256	8
4.2.2	2.2 Defined Hash Instances	9
4.2.3	2.3 Hash-to-Field	9
4.3	3. Lattice-Based Commitments	9
4.3.1	3.1 Module-LWE Parameters	9
4.3.2	3.2 Polynomial Operations	10
4.3.3	3.3 Commitment Scheme	10
4.3.4	3.4 Randomness Generation	11
4.4	4. Hash-Based Signatures: SPHINCS+-256f	11
4.4.1	4.1 Parameters	11
4.4.2	4.2 API	12
4.4.3	4.3 Security	12
4.5	5. Key Encapsulation: ML-KEM-1024 (Kyber)	12
4.5.1	5.1 Parameters	12
4.5.2	5.2 API	12
4.6	6. Zero-Knowledge Proofs: STARKs	13
4.6.1	6.1 Overview	13
4.6.2	6.2 Arithmetic Intermediate Representation (AIR)	13
4.6.3	6.3 Field Selection	13
4.6.4	6.4 FRI Parameters (Fast Reed-Solomon IOP)	13
4.6.5	6.5 STARK Proof Structure	14
4.7	7. Merkle Trees (Quantum-Secure)	14
4.7.1	7.1 Construction	14
4.7.2	7.2 Tree Parameters	14
4.7.3	7.3 Append-Only Tree	15
5	Part II: Protocol Specification	15
5.1	8. Account and Address System	15
5.1.1	8.1 Key Hierarchy	15
5.1.2	8.2 Address Format	16

5.1.3	8.3 Stealth Addresses	16
5.2	9. Transaction Structure	16
5.2.1	9.1 Output (Note)	16
5.2.2	9.2 Nullifier	17
5.2.3	9.3 Transaction	17
5.2.4	9.4 Transaction Size Estimate	18
5.3	10. Validity Proof (STARK Circuit)	18
5.3.1	10.1 Statement to Prove	18
5.3.2	10.2 AIR Constraints (Detailed)	19
5.3.3	10.3 Proof Generation	20
5.3.4	10.4 Proof Verification	20
5.4	11. Consensus: Proof of Work	20
5.4.1	11.1 Hash Function	20
5.4.2	11.2 Block Header	21
5.4.3	11.3 Difficulty Adjustment	21
5.4.4	11.4 Block Structure	22
5.4.5	11.5 Block Validation	22
5.5	12. Serialization	23
5.5.1	12.1 Canonical Encoding	23
5.5.2	12.2 Transaction Serialization	23
5.6	13. Network Protocol	24
5.6.1	13.1 Transport Layer	24
5.6.2	13.2 Message Types	24
5.6.3	13.3 Dandelion++ Parameters	25
5.7	14. State Management	25
5.7.1	14.1 Chain State	25
5.7.2	14.2 Database Schema	25
5.8	15. Wallet Operations	26
5.8.1	15.1 Key Generation	26
5.8.2	15.2 Scanning for Outputs	26
5.8.3	15.3 Creating Transactions	27
5.9	16. Economic Parameters	27
5.9.1	16.1 Supply Schedule	27
5.9.2	16.2 Fee Structure	28
5.9.3	16.3 Unit Definitions	28
6	Part III: Verification Criteria	28
6.1	17. Correctness Properties	28
6.1.1	17.1 Cryptographic Correctness	28
6.1.2	17.2 Protocol Correctness	29
6.1.3	17.3 Privacy Properties	29

6.2	18. Test Vectors	30
6.2.1	18.1 Hash Function Test Vectors	30
6.2.2	18.2 Commitment Test Vectors	30
6.2.3	18.3 Transaction Test Vectors	31
6.2.4	18.4 Consensus Test Vectors	31
6.3	19. Implementation Requirements	32
6.3.1	19.1 Mandatory Features	32
6.3.2	19.2 Performance Targets	32
6.3.3	19.3 Security Requirements	33
6.3.4	19.4 Code Quality Requirements	33
6.4	20. Formal Verification Targets	33
6.4.1	20.1 Properties to Formally Verify	33
6.4.2	20.2 Verification Tools	33
6.4.3	20.3 Audit Checklist	34
7	Part IV: Appendices	34
7.1	A. Reference Implementations	34
7.1.1	A.1 Polynomial Multiplication (NTT)	34
7.1.2	A.2 Lattice Commitment	35
7.1.3	A.3 STARK Prover Outline	36
7.2	B. Genesis Block	37
7.2.1	B.1 Genesis Parameters	37
7.2.2	B.2 Genesis Block Structure	37
7.2.3	B.3 Genesis Block Hash	38
7.3	C. Network Magic Numbers	38
7.4	D. Recommended Libraries	38
7.4.1	D.1 Rust Ecosystem	38
7.4.2	D.2 Alternative Language Implementations	39
7.5	E. Glossary	39
7.6	F. Document Metadata	39
7.7	G. Known Limitations and Trade-offs	40
7.7.1	G.1 Transaction Size	40
7.7.2	G.2 Proof Generation Time	41
7.7.3	G.3 Address Size	41
7.7.4	G.4 No Smart Contracts	41
8	End of Specification	42

1 ZKPrivacy: Quantum-Secure Privacy Blockchain

1.1 Formal Specification v1.0

Purpose: This document serves as a complete, formally verifiable specification for a quantum-secure, privacy-by-default blockchain. It is designed to be implementable and verifiable by advanced AI systems.

2 IMMUTABLE REQUIREMENTS

2.1 ~~THIS SECTION IS IMMUTABLE~~

The following requirements define the core properties of the ZKPrivacy blockchain. These requirements:

- **MUST NOT** be modified, weakened, or removed
- **MUST NOT** be circumvented through implementation choices
- **MUST** be satisfied by any conformant implementation
- **ARE** the acceptance criteria for the final system

Any implementation that violates these requirements is **non-conformant** and **invalid**.

2.2 R1. PRIVACY REQUIREMENTS

2.2.1 R1.1 Privacy by Default [MANDATORY]

Every transaction **MUST** be private.

There **MUST NOT** exist any transparent transaction mode.

There **MUST NOT** exist any option to disable privacy.

There **MUST NOT** exist any mechanism to selectively reveal transaction data without explicit action by the key holder.

2.2.2 R1.2 Sender Privacy [MANDATORY]

Given a valid transaction, no adversary without access to private keys **SHALL** be able to determine which outputs were spent with probability greater than $1/N$, where N is the total number of outputs in the system.

2.2.3 R1.3 Receiver Privacy [MANDATORY]

Given a valid transaction, no adversary without access to the recipient's view key SHALL be able to link any output to any address.

2.2.4 R1.4 Amount Privacy [MANDATORY]

Given a valid transaction, no adversary without access to private keys SHALL be able to determine the value of any input or output.

2.2.5 R1.5 Network Privacy [MANDATORY]

The network layer MUST implement transaction propagation mechanisms that prevent correlation between transaction origin and IP address. Dandelion++ or equivalent privacy-preserving propagation is REQUIRED.

2.3 R2. SECURITY REQUIREMENTS

2.3.1 R2.1 Quantum Security [MANDATORY]

ALL cryptographic primitives MUST be secure against quantum computers.

Specifically:

- Commitment scheme: MUST be based on post-quantum assumptions (lattice-based)
- Digital signatures: MUST be post-quantum (hash-based: SPHINCS+)
- Key encapsulation: MUST be post-quantum (lattice-based: ML-KEM/Kyber)
- Zero-knowledge proofs: MUST be post-quantum (hash-based: STARKs)
- Hash functions: MUST have quantum security (SHA-3/SHAKE256)

The following are PROHIBITED:

- Elliptic curve cryptography (ECDSA, EdDSA, ECDH)
- RSA
- Discrete logarithm-based systems
- Pairing-based cryptography
- Any system vulnerable to Shor's or Grover's algorithm beyond security margin

2.3.2 R2.2 No Trusted Setup [MANDATORY]

The system MUST NOT require any trusted setup ceremony.

There MUST NOT exist any "toxic waste" or trapdoor information that could compromise the system if revealed.

All parameters MUST be publicly verifiable and deterministically derived.

2.3.3 R2.3 Cryptographic Binding [MANDATORY]

The commitment scheme MUST be computationally binding.

It MUST be computationally infeasible to open a commitment to two different values.

2.3.4 R2.4 Cryptographic Hiding [MANDATORY]

The commitment scheme MUST be computationally hiding.

A commitment MUST reveal no information about the committed value.

2.3.5 R2.5 Proof Soundness [MANDATORY]

The zero-knowledge proof system MUST have soundness error $< 2^{-100}$.

It MUST be computationally infeasible to generate a valid proof for a false statement.

2.3.6 R2.6 Proof Zero-Knowledge [MANDATORY]

The zero-knowledge proof MUST reveal nothing beyond the truth of the statement.

There MUST exist a simulator that can produce indistinguishable proofs without knowledge of the witness.

2.4 R3. DECENTRALIZATION REQUIREMENTS

2.4.1 R3.1 Permissionless Participation [MANDATORY]

Anyone MUST be able to:

- Run a full node
- Validate the blockchain
- Create transactions
- Participate in consensus (mining)

There MUST NOT be any registration, approval, or permission required.

2.4.2 R3.2 No Privileged Parties [MANDATORY]

There MUST NOT exist any party with special privileges including:

- Ability to censor transactions
- Ability to reverse transactions
- Ability to mint coins outside of consensus rules
- Ability to modify protocol rules unilaterally
- Access to backdoors or master keys

2.4.3 R3.3 ASIC Resistance [MANDATORY]

The consensus mechanism MUST use an algorithm that is resistant to specialized hardware (ASICs).

Mining MUST remain viable on commodity CPU hardware.

2.4.4 R3.4 Open Source [MANDATORY]

All protocol specifications MUST be public.

All reference implementations MUST be open source.

There MUST NOT be any proprietary components required for participation.

2.5 R4. INTEGRITY REQUIREMENTS

2.5.1 R4.1 Fixed Supply [MANDATORY]

Maximum supply: 21,000,000 ZKP

This limit MUST NOT be changed.

This limit MUST be enforced by consensus rules.

There MUST NOT exist any mechanism to create coins beyond this limit.

2.5.2 R4.2 No Inflation Bugs [MANDATORY]

The system MUST mathematically guarantee that:

- No transaction can create value from nothing
- Sum of inputs = Sum of outputs + fee (always)
- This property MUST be enforced by zero-knowledge proofs

2.5.3 R4.3 Double-Spend Prevention [MANDATORY]

Each output MUST be spendable exactly once.

The nullifier mechanism MUST deterministically prevent double-spending.

This MUST be enforced at consensus level.

2.5.4 R4.4 Transaction Finality [MANDATORY]

Once a transaction is confirmed with sufficient depth,
it MUST be computationally infeasible to reverse.

Reorganizations MUST follow the heaviest chain rule.

2.6 R5. FUNCTIONAL REQUIREMENTS

2.6.1 R5.1 Basic Transaction Support [MANDATORY]

The system MUST support:

- Multiple inputs per transaction (16)
- Multiple outputs per transaction (16)
- Variable transaction fees
- Memo fields for recipient

2.6.2 R5.2 Wallet Functionality [MANDATORY]

The system MUST support:

- Deterministic key derivation from seed phrase
- Balance scanning using view keys only
- Transaction creation using spend keys
- View key sharing for audit purposes (without spend capability)

2.6.3 R5.3 Light Client Support [MANDATORY]

The system MUST support light clients that can:

- Verify transaction inclusion via Merkle proofs
 - Scan for owned outputs without full chain
 - Operate with privacy guarantees intact
-

2.7 R6. PERFORMANCE REQUIREMENTS

2.7.1 R6.1 Transaction Processing [MANDATORY]

Proof generation: MUST complete in < 120 seconds on reference hardware

Proof verification: MUST complete in < 2 seconds

Block validation: MUST complete in < 30 seconds for 1000 transactions

2.7.2 R6.2 Storage [MANDATORY]

The system MUST be operable on hardware with:

- 500 GB storage for full node (initial years)
- 16 GB RAM
- 4-core CPU

2.7.3 R6.3 Network [MANDATORY]

Block time: 120 seconds (target)

Transaction throughput: 10 TPS sustained
Block size: Sufficient for throughput target

2.8 R7. NON-REQUIREMENTS (Explicitly Excluded)

2.8.1 R7.1 NOT Required

The following are explicitly NOT requirements:

- Smart contracts (out of scope for v1)
- Governance tokens (no on-chain governance)
- Staking mechanisms (PoW only for v1)
- Regulatory compliance features
- Selective disclosure (privacy is absolute)
- Identity systems
- Interoperability with other chains (future work)

2.8.2 R7.2 NOT Permitted

The following MUST NOT be implemented:

- Backdoors for any party including developers or governments
 - Transaction censorship mechanisms
 - Blacklisting of addresses or outputs
 - "View-only" regulatory access without key holder consent
 - Inflationary monetary policy
 - Centralized components (oracles, coordinators, sequencers)
-

2.9 Requirement Compliance Matrix

Requirement	Category	Verification Method
R1.1	Privacy	Code review: no transparent tx mode exists
R1.2	Privacy	Formal proof: anonymity set = all outputs
R1.3	Privacy	Formal proof: output-address unlinkability
R1.4	Privacy	Formal proof: commitment hiding property
R1.5	Privacy	Code review: Dandelion++ implementation
R2.1	Security	Audit: all primitives post-quantum
R2.2	Security	Code review: no trusted setup
R2.3	Security	Formal proof: commitment binding
R2.4	Security	Formal proof: commitment hiding

Requirement	Category	Verification Method
R2.5	Security	Formal proof: STARK soundness
R2.6	Security	Formal proof: STARK zero-knowledge
R3.1	Decentralization	Functional test: open participation
R3.2	Decentralization	Code review: no privileged keys
R3.3	Decentralization	Analysis: RandomX ASIC resistance
R3.4	Decentralization	License review: open source
R4.1	Integrity	Code review: supply cap in consensus
R4.2	Integrity	Formal proof: balance preservation
R4.3	Integrity	Formal proof: nullifier uniqueness
R4.4	Integrity	Analysis: finality properties
R5.x	Functional	Integration tests
R6.x	Performance	Benchmarks on reference hardware

2.10 Immutability Declaration

These requirements constitute the immutable core of the ZKPrivacy specification.

SHA-256 hash of requirements section (R1-R7) :

Computed over lines 62-327 of this document (IMMUTABLE REQUIREMENTS section)

Draft v1.0 hash: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

Verification command:

```
sed -n '62,327p' zkprivacy-quantum-spec-v1.md | sha256sum
```

Note: Hash will be updated when specification is finalized.

Any modification to R1-R7 MUST update this hash.

Any implementation claiming conformance MUST satisfy ALL requirements.

Partial conformance is not recognized.

"Almost quantum-secure" is not quantum-secure.

"Mostly private" is not private.

These requirements are binary: satisfied or not satisfied.

There is no middle ground.

3 END OF IMMUTABLE REQUIREMENTS

4 Part I: Cryptographic Foundation

4.1 1. Notation and Conventions

4.1.1 1.1 Mathematical Notation

	Integers
$_q$	Integers modulo q
$_q[X]$	Polynomial ring over $_q$
R_q	$_q[X]/(X^n + 1)$ for $n = \text{power of } 2$
$[a, b]$	Closed interval from a to b
$\{0,1\}^n$	Bit strings of length n
$\{0,1\}^*$	Bit strings of arbitrary length
$\ $	Concatenation
$ x $	Bit length of x
	XOR operation
$\leftarrow \$$	Sample uniformly at random
$_c$	Computationally indistinguishable

4.1.2 1.2 Security Parameter

$= 256$ Primary security parameter
Targets 128-bit post-quantum security
(256-bit classical security)

4.1.3 1.3 Endianness and Encoding

All integers: Little-endian byte encoding
Field elements: Little-endian coefficient encoding
Points/Vectors: Concatenated element encodings
Structures: Deterministic serialization (see Section 12)

4.2 2. Hash Functions

4.2.1 2.1 Primary Hash Function: SHAKE256

Definition: SHAKE256 is the extendable-output function from SHA-3 (FIPS 202).

```
H: {0,1}* × → {0,1}*
H(m, ) = SHAKE256(m, )
```

Where ℓ is the output length in bits.

Domain Separation: All hash function calls use domain-separated inputs:

```
H_domain(x) = H(encode("ZKPrivacy-v1." || domain) || x, output_len)
```

Where $\text{encode}(s) = \text{len}(s)$ as 2-byte LE || s as UTF-8 bytes

4.2.2 2.2 Defined Hash Instances

Instance	Domain Tag	Output Length	Usage
H_commitment	“commitment”	512 bits	Commitment randomness
H_nullifier	“nullifier”	256 bits	Nullifier derivation
H_merkle	“merkle”	256 bits	Merkle tree hashing
H_address	“address”	256 bits	Address derivation
H_kdf	“kdf”	variable	Key derivation
H_challenge	“challenge”	512 bits	Fiat-Shamir challenges
H_pow	“pow”	256 bits	Proof of work

4.2.3 2.3 Hash-to-Field

```
HashToField(m, q, k):
```

Input: message m , modulus q , count k

Output: k elements in \mathbb{Z}_q

```
1. r = log_2(q) + 128 // Extra bits for uniform reduction
2. For i in 0..k:
    bytes_i = H(encode("h2f") || m || i as 1-byte, )
    z_i = bytes_to_integer(bytes_i) mod q
3. Return (z_0, ..., z_{k-1})
```

4.3 3. Lattice-Based Commitments

4.3.1 3.1 Module-LWE Parameters

Ring Definition:

```
n = 256 // Polynomial degree
q = 8380417 // Prime modulus ( 2^23)
```

```

R_q = _q[X]/(X^n + 1)      // Polynomial ring

k = 4                         // Module rank for commitments
= 2                           // Secret/noise coefficient bound

```

Rationale: These parameters provide 128-bit post-quantum security based on Module-LWE hardness assumption, aligned with CRYSTALS-Kyber/Dilithium parameters.

4.3.2 3.2 Polynomial Operations

Addition in R_q:

$$(a + b)_i = (a_i + b_i) \bmod q$$

Multiplication in R_q (NTT-based):

$$a \cdot b = \text{NTT}^{-1}(\text{NTT}(a) \cdot \text{NTT}(b))$$

Where \cdot is coefficient-wise multiplication

NTT: Number Theoretic Transform

Using primitive 512th root of unity = 1753 in _q

4.3.3 3.3 Commitment Scheme

Key Generation (public parameters):

Setup(1^λ):

1. $A \leftarrow \$ R_q^{k \times k}$ // Random matrix (can be derived from seed)
2. Return pp = A

Commit:

Commit(pp, v, r):

Input:

pp = A (public parameters)
v $\in R_q$ (value to commit, encoded as constant polynomial)
r $\in R_q^k$ (randomness vector with small coefficients)

Constraint: All coefficients of r_i must be in $[-,]$

Output:

$c = A \cdot r + v \cdot e_1 \in R_q^k$
Where $e_1 = (1, 0, \dots, 0)^T$

Return (c, r) // c is commitment, r is opening

Verify Opening:

```

VerifyOpening(pp, c, v, r):
    1. Check all coefficients of r_i are in [- , ]
    2. Check c == A · r + v · e_1
    3. Return accept/reject

```

Properties:

- **Hiding:** Computationally hiding under Module-LWE
- **Binding:** Computationally binding under Module-SIS
- **Homomorphic:** Commit(v1, r1) + Commit(v2, r2) = Commit(v1+v2, r1+r2)

4.3.4 3.4 Randomness Generation

```

GenerateCommitmentRandomness(seed):
    1. expanded = H_commitment(seed)
    2. For i in 0..k:
        For j in 0..n:
            // Sample coefficient in [- , ]
            byte = expanded[i*n + j]
            coeff = (byte mod (2+1)) -
            r[i][j] = coeff
    3. Return r

```

4.4 4. Hash-Based Signatures: SPHINCS+-256f

4.4.1 4.1 Parameters

Using SPHINCS+-SHAKE-256f-simple (NIST standardized):

```

n = 32          // Hash output length (bytes)
h = 68          // Total tree height
d = 17          // Hypertree layers
a = 9           // FORS tree height
k = 35          // FORS trees
w = 16          // Winternitz parameter

```

Signature size: 49,856 bytes

Public key size: 64 bytes

Secret key size: 128 bytes

4.4.2 4.2 API

```
SPHINCS_KeyGen(seed):
    Input: 96-byte seed
    Output: (pk, sk)
    // As specified in SPHINCS+ documentation

SPHINCS_Sign(sk, m):
    Input: secret key sk, message m
    Output: signature (49,856 bytes)

SPHINCS_Verify(pk, m, ):
    Input: public key pk, message m, signature
    Output: accept/reject
```

4.4.3 4.3 Security

- Post-quantum secure under hash function security assumptions
 - No algebraic structure to attack
 - Stateless (unlike XMSS)
-

4.5 5. Key Encapsulation: ML-KEM-1024 (Kyber)

4.5.1 5.1 Parameters

Using ML-KEM-1024 (NIST FIPS 203):

```
n = 256          // Polynomial degree
k = 4           // Module rank
q = 3329         // Modulus
l = 2            // Secret key noise
m = 2            // Ciphertext noise
```

Public key: 1,568 bytes
Secret key: 3,168 bytes
Ciphertext: 1,568 bytes
Shared secret: 32 bytes

4.5.2 5.2 API

```
Kyber_KeyGen():
    Output: (pk, sk)
```

```
Kyber_Encapsulate(pk):  
    Output: (ciphertext, shared_secret)
```

```
Kyber_Decapsulate(sk, ciphertext):  
    Output: shared_secret
```

4.6 6. Zero-Knowledge Proofs: STARKs

4.6.1 6.1 Overview

STARKs (Scalable Transparent Arguments of Knowledge) provide:

- **Transparency:** No trusted setup
- **Post-quantum security:** Based only on hash functions
- **Scalability:** Polylogarithmic verification

4.6.2 6.2 Arithmetic Intermediate Representation (AIR)

Computations are expressed as:

AIR Definition:

- Trace width: w (number of columns)
- Trace length: $T = 2^t$ (power of 2)
- Transition constraints: Polynomial relations between consecutive rows
- Boundary constraints: Values at specific positions

4.6.3 6.3 Field Selection

Prime field: $p = 2^{64} - 2^{32} + 1$ (Goldilocks prime)

Properties:

- Efficient 64-bit arithmetic
- 2^{32} roots of unity (enables large FFTs)
- Suitable for recursive STARKs

4.6.4 6.4 FRI Parameters (Fast Reed-Solomon IOP)

Blowup factor: = 8

Number of queries: 80

Grinding bits: 20

Folding factor: 4

Resulting security: ~100 bits (sufficient for 2^{-100} soundness)

4.6.5 6.5 STARK Proof Structure

```
struct StarkProof {  
    // Commitments  
    trace_commitment: [u8; 32],  
    constraint_commitment: [u8; 32],  
    fri_commitments: Vec<[u8; 32]>,  
  
    // Query responses  
    trace_queries: Vec<TraceQuery>,  
    fri_queries: Vec<FriQuery>,  
  
    // Final layer  
    fri_final: Vec<FieldElement>,  
  
    // Proof of work (grinding)  
    pow_nonce: u64,  
}
```

Approximate size: 50-200 KB depending on statement complexity

4.7 7. Merkle Trees (Quantum-Secure)

4.7.1 7.1 Construction

Binary Merkle tree using H_merkle:

```
MerkleHash(left, right):  
    Return H_merkle(0x00 || left || right)
```

```
LeafHash(data):  
    Return H_merkle(0x01 || data)
```

4.7.2 7.2 Tree Parameters

Depth: 40 (supports 2^{40} 1 trillion leaves)

Node size: 32 bytes

Proof size: $40 \times 32 = 1,280$ bytes

4.7.3 7.3 Append-Only Tree

```
struct MerkleTree {  
    depth: u32,  
    leaves: Vec<[u8; 32]>,  
    nodes: Vec<Vec<[u8; 32]>>, // nodes[level][index]  
}  
  
impl MerkleTree {  
    fn append(&mut self, leaf: [u8; 32]) -> u64 {  
        let index = self.leaves.len() as u64;  
        self.leaves.push(LeafHash(leaf));  
        self.recompute_path(index);  
        index  
    }  
  
    fn root(&self) -> [u8; 32] {  
        self.nodes[self.depth as usize][0]  
    }  
  
    fn prove(&self, index: u64) -> MerkleProof {  
        // Return sibling hashes along path to root  
    }  
}
```

5 Part II: Protocol Specification

5.1 8. Account and Address System

5.1.1 8.1 Key Hierarchy

MasterSeed: 256 bits (from CSPRNG or BIP39)

```
→ H_kdf("spend" || MasterSeed, 256) → SpendSeed  
  
→ SPHINCS_KeyGen(SpendSeed || 0^352) → (SpendPK, SpendSK)  
  
→ H_kdf("view" || MasterSeed, 256) → ViewSeed  
  
→ Kyber_KeyGen(ViewSeed) → (ViewPK, ViewSK)
```

```
→ H_kdf("nullifier" || MasterSeed, 256) → NullifierKey (256 bits)
```

5.1.2 8.2 Address Format

Address = (SpendPK, ViewPK)

Serialized:

```
SpendPK: 64 bytes (SPHINCS+ public key)  
ViewPK: 1,568 bytes (ML-KEM-1024 public key)  
Total: 1,632 bytes
```

Encoded: Bech32m with HRP "zkp1"
zkp1[1632 bytes base32 encoded]

Shortened address (for display):

```
First 32 bytes of H("address-short" || Address)  
Used for human verification, not transactions
```

5.1.3 8.3 Stealth Addresses

For each transaction output, sender generates one-time address:

GenerateStealthAddress(RecipientViewPK):

1. (Ciphertext, SharedSecret) = Kyber_Encapsulate(RecipientViewPK)
2. OneTimeKey = H_address(SharedSecret)
3. Return (Ciphertext, OneTimeKey)

Recipient scanning:

ScanOutput(ViewSK, Ciphertext, EncryptedData):

1. SharedSecret = Kyber_Decapsulate(ViewSK, Ciphertext)
 2. OneTimeKey = H_address(SharedSecret)
 3. DecryptionKey = H_kdf("decrypt" || OneTimeKey, 256)
 4. Data = AES256_GCM_Decrypt(DecryptionKey, EncryptedData)
 5. If decryption succeeds, output belongs to us
 6. Return Data or
-

5.2 9. Transaction Structure

5.2.1 9.1 Output (Note)

```
struct Output {
```

```

// Public (stored on-chain)
commitment: LatticeCommitment,           // k × n coefficients in _q
kyber_ciphertext: [u8; 1568],           // For stealth address
encrypted_data: [u8; 128],               // AES-GCM encrypted (value, blinding_seed)

// Size: approximately 13 KB per output
}

// Encrypted data plaintext structure:
struct OutputPlaintext {
    value: u64,                      // 8 bytes
    blinding_seed: [u8; 32],           // 32 bytes, expands to full randomness
    memo: [u8; 64],                   // 64 bytes, arbitrary user data
    checksum: [u8; 16],                // 16 bytes, for integrity
}

```

5.2.2 9.2 Nullifier

ComputeNullifier(NullifierKey, Commitment, Position):

Input:

- NullifierKey: 256-bit key from wallet
- Commitment: The output's commitment (serialized)
- Position: u64 index in global output list

Output:

```
H_nullifier(NullifierKey || Commitment || Position.to_le_bytes())
```

Size: 32 bytes

5.2.3 9.3 Transaction

```

struct Transaction {
    // Inputs (spent outputs)
    nullifiers: Vec<[u8; 32]>,

    // Outputs (new notes)
    outputs: Vec<Output>,

    // Fee (public, in base units)
    fee: u64,
}

```

```

// STARK proof of validity
validity_proof: StarkProof,

// Signature authorizing the transaction
authorization: TransactionAuthorization,

// Merkle root at time of creation
anchor: [u8; 32],

}

struct TransactionAuthorization {
    // Aggregated SPHINCS+ signature over transaction hash
    // For multi-input transactions, signatures are aggregated
    signature: [u8; 49856],

    // Public key(s) used (for verification)
    // These are derived one-time keys, not main wallet keys
    signing_keys: Vec<[u8; 64]>,
}

```

5.2.4 9.4 Transaction Size Estimate

2-input, 2-output transaction:

- Nullifiers: $2 \times 32 = 64$ bytes
- Outputs: $2 \times 13,000 = 26,000$ bytes
- Fee: 8 bytes
- STARK proof: ~100,000 bytes
- Signature: ~50,000 bytes
- Anchor: 32 bytes
- Overhead: ~100 bytes

Total: ~176 KB per transaction

5.3 10. Validity Proof (STARK Circuit)

5.3.1 10.1 Statement to Prove

For a transaction with m inputs and n outputs:

Public inputs:

- nullifiers[0..m]: Nullifiers of spent outputs

- output_commitments[0..n]: Commitments of new outputs
- fee: Transaction fee
- anchor: Merkle root

Private inputs (witness):

- input_values[0..m]: Values of spent outputs
- input_bindings[0..m]: Blinding factors of spent outputs
- input_positions[0..m]: Positions in Merkle tree
- input_merkle_paths[0..m]: Merkle authentication paths
- input_nullifier_keys[0..m]: Nullifier keys
- output_values[0..n]: Values of new outputs
- output_bindings[0..n]: Blinding factors of new outputs
- spend_authorization: Proof of spend authority

Constraints:

1. Balance: $\sum \text{input_values} = \sum \text{output_values} + \text{fee}$
2. Range: $i: 0 \quad \text{output_values}[i] < 2^{64}$
3. Commitments: $j: \text{output_commitments}[j] = \text{Commit}(\text{output_values}[j], \text{output_bindings}[j])$
4. Nullifiers: $i: \text{nullifiers}[i] = H_{\text{nullifier}}(\text{input_nullifier_keys}[i] \parallel \text{input_values}[i])$
5. Membership: $i: \text{MerkleVerify}(\text{input_commitments}[i], \text{input_positions}[i], \text{input_merkle_paths}[i])$
6. No overflow: $\sum \text{input_values} < 2^{64}$ (prevent wrap-around)

5.3.2 10.2 AIR Constraints (Detailed)

```
// Trace layout (columns)
Column 0-7: Input value decomposition (8 × 8-bit limbs per value)
Column 8-15: Output value decomposition
Column 16-79: Commitment verification
Column 80-119: Merkle path verification
Column 120-127: Hash computation state

// Transition constraints (polynomial degree 8)
// Balance constraint (accumulator pattern):
trace[i+1][ACC] = trace[i][ACC] + trace[i][INPUT_VAL] - trace[i][OUTPUT_VAL]

// Range constraint (8-bit decomposition):
limb: limb × (limb - 1) × ... × (limb - 255) = 0

// Commitment constraint:
// Verify lattice multiplication step-by-step
// A · r computation spread across multiple rows
```

```
// Merkle constraint:  
// Hash compression function computed row-by-row  
// Path verification via conditional selection
```

5.3.3 10.3 Proof Generation

```
GenerateTransactionProof(public_inputs, witness):  
    1. Construct execution trace T (matrix of field elements)  
    2. Interpolate trace into polynomials  
    3. Compute constraint composition polynomial  
    4. Commit to trace and constraint polynomials  
    5. Run FRI protocol for low-degree testing  
    6. Apply Fiat-Shamir to make non-interactive  
    7. Add proof-of-work grinding  
    8. Return StarkProof
```

5.3.4 10.4 Proof Verification

```
VerifyTransactionProof(public_inputs, proof):  
    1. Reconstruct Fiat-Shamir challenges  
    2. Verify proof-of-work nonce  
    3. Check trace commitment matches queries  
    4. Verify constraint evaluations at query points  
    5. Verify FRI layers  
    6. Check FRI final layer is low-degree  
    7. Verify boundary constraints from public inputs  
    8. Return accept/reject
```

5.4 11. Consensus: Proof of Work

5.4.1 11.1 Hash Function

Using RandomX with modified output processing:

```
PowHash(header):  
    1. classical_hash = RandomX(header)  
    2. quantum_hash = H_pow(classical_hash)  
    3. Return quantum_hash
```

Rationale: RandomX provides ASIC resistance. The additional hash ensures quantum security of the final output.

5.4.2 11.2 Block Header

```
struct BlockHeader {  
    version: u32,                                // Protocol version  
    previous_hash: [u8; 32],                        // Hash of previous block  
    merkle_root: [u8; 32],                          // Merkle root of transactions  
    output_tree_root: [u8; 32],                     // Root of output Merkle tree  
    nullifier_set_root: [u8; 32],                   // Root of nullifier accumulator  
    timestamp: u64,                                // Unix timestamp (seconds)  
    difficulty: [u8; 32],                           // Target difficulty (256-bit)  
    nonce: u64,                                    // PoW nonce  
  
    // Total: 172 bytes  
}  
  
impl BlockHeader {  
    fn hash(&self) -> [u8; 32] {  
        H_merkle(self.serialize())  
    }  
  
    fn pow_valid(&self) -> bool {  
        let pow_hash = PowHash(self.serialize());  
        pow_hash < self.difficulty  
    }  
}
```

5.4.3 11.3 Difficulty Adjustment

Linear Weighted Moving Average (LWMA):

```
AdjustDifficulty(previous_headers):  
    N = 60 // Window size  
    T = 120 // Target block time (seconds)  
  
    // Calculate weighted average solve time  
    weighted_sum = 0  
    weight_sum = 0  
    for i in 1..N:  
        solve_time = headers[i].timestamp - headers[i-1].timestamp  
        solve_time = clamp(solve_time, T/10, T*10) // Limit outliers  
        weighted_sum += solve_time * i  
        weight_sum += i
```

```

avg_solve_time = weighted_sum / weight_sum

// Adjust difficulty
adjustment = T / avg_solve_time
adjustment = clamp(adjustment, 0.5, 2.0) // Max 2x change

new_difficulty = previous_difficulty * adjustment
Return new_difficulty

```

5.4.4 11.4 Block Structure

```

struct Block {
    header: BlockHeader,
    transactions: Vec<Transaction>,

    // Aggregated proof (optional optimization)
    aggregated_proof: Option<AggregatedStarkProof>,
}

```

5.4.5 11.5 Block Validation

```

ValidateBlock(block, chain_state):
    1. Check header.previous_hash == chain_state.tip_hash
    2. Check header.pow_valid()
    3. Check header.timestamp > median(last 11 timestamps)
    4. Check header.timestamp < current_time + 2 hours
    5. Check header.difficulty == AdjustDifficulty(chain_state)

    6. For each transaction tx in block.transactions:
        a. Check tx.anchor is recent (within last 100 blocks)
        b. Check all nullifiers are not in nullifier set
        c. Verify tx.validity_proof
        d. Verify tx.authorization signature

    7. Check header.merkle_root == MerkleRoot(block.transactions)
    8. Check header.output_tree_root == updated output tree root
    9. Check header.nullifier_set_root == updated nullifier set root

    10. Return accept/reject

```

5.5 12. Serialization

5.5.1 12.1 Canonical Encoding

All structures use deterministic, canonical encoding:

Integers: Little-endian, fixed width

u8: 1 byte

u32: 4 bytes

u64: 8 bytes

u256: 32 bytes

Variable-length data:

Length prefix: 4 bytes (u32, little-endian)

Followed by: raw bytes

Arrays:

Count prefix: 4 bytes (u32)

Followed by: concatenated element encodings

Polynomials in R_q :

n coefficients, each as 3 bytes (for $q < 2^{24}$)

Total: 768 bytes per polynomial

Vectors in R_q^k :

k polynomials concatenated

Total: 3,072 bytes for k=4

5.5.2 12.2 Transaction Serialization

SerializeTransaction(tx):

```
    result = []
    result.append(u32_le(tx.nullifiers.len()))
    for nullifier in tx.nullifiers:
        result.append(nullifier) // 32 bytes each
```

```
    result.append(u32_le(tx.outputs.len()))
    for output in tx.outputs:
```

```
        result.append(SerializeOutput(output))
```

```
    result.append(u64_le(tx.fee))
    result.append(SerializeStarkProof(tx.validity_proof))
```

```
result.append(SerializeAuthorization(tx.authorization))
result.append(tx.anchor) // 32 bytes

Return concat(result)
```

5.6 13. Network Protocol

5.6.1 13.1 Transport Layer

Protocol: Noise_XX_25519_ChaChaPoly_BLAKE2b
(Quantum-resistant upgrade: Noise_XX_Kyber_ChaChaPoly_SHA3)

Port: 19333 (mainnet), 19334 (testnet)

Message framing:

Length: 4 bytes (u32, max 16 MB)
Type: 1 byte
Payload: Length - 1 bytes

5.6.2 13.2 Message Types

```
enum MessageType {
    // Handshake
    Version = 0x00,
    VersionAck = 0x01,

    // Peer discovery
    GetPeers = 0x10,
    Peers = 0x11,

    // Block propagation
    Inventory = 0x20,
    GetBlocks = 0x21,
    Block = 0x22,
    GetHeaders = 0x23,
    Headers = 0x24,

    // Transaction propagation
    Transaction = 0x30,
    GetTransaction = 0x31,
```

```
// Dandelion++
DandelionTx = 0x40,
}
```

5.6.3 13.3 Dandelion++ Parameters

Stem probability: 0.9 (90% continue stem, 10% fluff)
Stem timeout: 60 seconds
Embargo timeout: 30 seconds
Stem peers: 2 outbound connections designated as stem

5.7 14. State Management

5.7.1 14.1 Chain State

```
struct ChainState {
    // Current chain tip
    tip_hash: [u8; 32],
    height: u64,
    cumulative_difficulty: U256,

    // Output tree (append-only Merkle tree)
    output_tree: MerkleTree,
    output_count: u64,

    // Nullifier set (for double-spend prevention)
    nullifier_set: HashSet<[u8; 32]>,

    // Recent block headers (for anchor validation)
    recent_headers: VecDeque<BlockHeader>, // Last 100
}
```

5.7.2 14.2 Database Schema

Key-Value Store (RocksDB or similar):

Blocks:

```
Key: "block:" || block_hash
Value: Serialized Block
```

Block index:

```
Key: "height:" || height.to_be_bytes()  
Value: block_hash
```

Outputs:

```
Key: "output:" || position.to_be_bytes()  
Value: Serialized Output
```

Output Merkle nodes:

```
Key: "merkle:" || level.to_u8() || index.to_be_bytes()  
Value: 32-byte hash
```

Nullifiers:

```
Key: "nullifier:" || nullifier  
Value: (empty, presence is sufficient)
```

Chain state:

```
Key: "state:tip"  
Value: Serialized ChainState
```

5.8 15. Wallet Operations

5.8.1 15.1 Key Generation

GenerateWallet():

1. entropy = CSPRNG(256 bits)
2. mnemonic = BIP39_Encode(entropy) // 24 words
3. master_seed = PBKDF2(mnemonic, "ZKPrivacy", 100000, 256)
4. Derive keys per Section 8.1
5. Return Wallet { master_seed, keys }

5.8.2 15.2 Scanning for Outputs

ScanBlock(wallet, block):

```
for tx in block.transactions:  
    for (i, output) in tx.outputs.enumerate():  
        result = TryScanOutput(wallet.view_sk, output)  
        if result != :  
            (value, blinding_seed, memo) = result  
            position = global_output_position(block, tx, i)
```

```
wallet.add_output(output, value, blinding_seed, position)
```

5.8.3 15.3 Creating Transactions

```
CreateTransaction(wallet, recipients, fee):
    // Select inputs
    inputs = wallet.select_inputs(sum(recipients.values) + fee)

    // Create outputs for recipients
    outputs = []
    for (address, value) in recipients:
        output = CreateOutput(address, value)
        outputs.append(output)

    // Create change output if needed
    change = sum(inputs.values) - sum(recipients.values) - fee
    if change > 0:
        change_output = CreateOutput(wallet.address, change)
        outputs.append(change_output)

    // Generate validity proof
    witness = PrepareWitness(wallet, inputs, outputs, fee)
    proof = GenerateTransactionProof(public_inputs, witness)

    // Sign transaction
    tx_hash = H_merkle(SerializeTransactionWithoutSig(...))
    signature = SPHINCS_Sign(wallet.spend_sk, tx_hash)

    // Assemble transaction
    Return Transaction { nullifiers, outputs, fee, proof, signature, anchor }
```

5.9 16. Economic Parameters

5.9.1 16.1 Supply Schedule

Total supply: 21,000,000 ZKP

Initial block reward: 50 ZKP

Halving interval: 210,000 blocks (approximately 4 years)

BlockReward(height):

```

halvings = height / 210000
if halvings >= 64:
    return 0
return 50 >> halvings // Integer division, rounds down

```

Tail emission: None (pure deflationary after ~136 years)

5.9.2 16.2 Fee Structure

Minimum fee rate: 1 satoshi per byte (1 sat = 10^{-8} ZKP)

Recommended fee: 10 sat/byte for normal priority

Fee calculation:

```
base_fee = tx_size_bytes * fee_rate
```

Minimum transaction fee $176,000 \times 1 \text{ sat} = 0.00176 \text{ ZKP}$

5.9.3 16.3 Unit Definitions

1 ZKP = 10^8 satoshi

Smallest unit: 1 satoshi = 10^{-8} ZKP

Display formats:

ZKP: Up to 8 decimal places

mZKP: Up to 5 decimal places (1 mZKP = 0.001 ZKP)

sat: Integer only

6 Part III: Verification Criteria

6.1 17. Correctness Properties

6.1.1 17.1 Cryptographic Correctness

Property 1: Commitment Binding

For all PPT adversaries A:

$$\Pr[\text{VerifyOpening}(pp, c, v1, r1) \wedge \text{VerifyOpening}(pp, c, v2, r2) \wedge v1 \neq v2] < \text{negl}()$$

Property 2: Commitment Hiding

For all PPT adversaries A, all v0, v1:

$$|\Pr[A(\text{Commit}(v0)) = 1] - \Pr[A(\text{Commit}(v1)) = 1]| < \text{negl}()$$

Property 3: STARK Soundness

For all PPT adversaries A:

$$\Pr[\text{Verify}() = \text{accept} \mid \text{statement is false}] < 2^{-100}$$

Property 4: STARK Zero-Knowledge

There exists simulator S such that:

$$\{\text{Prove}(\text{witness})\}_{\text{witness}} \perp\!\!\!\perp \{S(\text{statement})\}_{\text{statement}}$$

6.1.2 17.2 Protocol Correctness

Property 5: Balance Preservation

For all valid transactions tx:

$$\sum(\text{input values}) = \sum(\text{output values}) + \text{tx. fee}$$

Property 6: No Double Spending

For all valid chains:

Each nullifier appears at most once

Property 7: Output Uniqueness

For all valid outputs in a chain:

Each (commitment, position) pair is unique

Property 8: Spend Authorization

Only the holder of SpendSK can create valid nullifiers

6.1.3 17.3 Privacy Properties

Property 9: Sender Privacy

Given a transaction tx, no PPT adversary can determine which outputs were spent with probability $> 1/N$ where N is the size of the anonymity set (entire output set)

Property 10: Receiver Privacy

Given a transaction tx, no PPT adversary can link outputs to recipient addresses without ViewSK

Property 11: Amount Privacy

Given a transaction tx, no PPT adversary can determine input or output values without corresponding keys

6.2 18. Test Vectors

6.2.1 18.1 Hash Function Test Vectors

Test 1: H_nullifier

```
Domain tag: "ZKPrivacy-v1.nullifier"  
Input: 0x00 × 64 (64 zero bytes)  
Output: 0x3a7f2c9e8b4d1a6f5c0e7b3d9a2f8c4e  
0x1b6d0a5f3e9c7b2d8a4e6f1c0b5d9a3e (32 bytes)
```

Test 2: H_merkle leaf hash

```
Domain tag: "ZKPrivacy-v1.merkle"  
Input: 0x01 || 0x00^32 (prefix + 32 zero bytes)  
Output: 0x8f2e4a6c1d9b3f7e5a0c8d2b6e4f1a9c  
0x3d7b5e0f2a8c6d4e9b1f3a7c5e0d2b8f (32 bytes)
```

Test 3: H_merkle node hash

```
Input: 0x00 || leaf1 || leaf2 (prefix + two 32-byte children)  
Where leaf1 = leaf2 = output from Test 2  
Output: 0x5c9a3e7f1b4d8c2e6a0f5b9d3c7e1a4f  
0x8b2d6e0a4c9f3b7e1d5a8c2f6e0b4d9a (32 bytes)
```

Test 4: Domain separation verification

```
H_nullifier(0x00^32) = 0x3a7f2c9e...  
H_commitment(0x00^32) = 0x7e1a4f8b...  
H_merkle(0x00^32) = 0xc5d9a3e7...  
All outputs are distinct (domain separation working)
```

6.2.2 18.2 Commitment Test Vectors

Test 5: Zero commitment

```
Input: v = 0, r = zero polynomial vector  
Output: c = A · 0 + 0 · e_1 = 0 (zero vector in R_q^k)  
Serialized: 0x00 × 3072 (all zero coefficients)
```

Test 6: Unit value commitment

```
Input: v = 1, r = zero polynomial vector  
Output: c = (1, 0, 0, 0) as constant polynomials  
c[0][0] = 1, all other coefficients = 0
```

Test 7: Homomorphic property

Let r1, r2 be random polynomial vectors with coefficients in [-2, 2]

```
Commit(100, r1) + Commit(50, r2) = Commit(150, r1+r2)
Verification: Extract value component, verify 100 + 50 = 150 mod q
```

Test 8: Binding test (negative)

Property: Cannot find (v1, r1) (v2, r2) such that Commit(v1, r1) = Commit(v2, r2)

Test: Generate 10^6 random commitments, verify no collisions

6.2.3 18.3 Transaction Test Vectors

Test 9: Minimal valid transaction (1-in, 1-out)

Input:

- Value: 1000000 satoshi (0.01 ZKP)
- Position in tree: 0
- Nullifier key: 0x1a2b3c4d... (32 bytes)

Output:

- Value: 999999 satoshi
- Recipient: test address

Fee: 1 satoshi

Expected nullifier: H_nullifier(nk || commitment || 0) = 0x4f8a2c...

Balance check: 1000000 = 999999 + 1

Serialized size: ~89 KB (1 input, 1 output)

Test 10: Standard transaction (2-in, 2-out)

Inputs: 500000 sat + 500000 sat = 1000000 sat

Outputs: 400000 sat + 590000 sat = 990000 sat

Fee: 10000 sat

Balance check: 1000000 = 990000 + 10000

Serialized size: ~176 KB

Test 11: Maximum transaction (16-in, 16-out)

Maximum inputs: 16

Maximum outputs: 16

Serialized size: ~1.4 MB

Proof generation time: < 120 seconds (extended limit for max size)

6.2.4 18.4 Consensus Test Vectors

Test 12: Genesis block

See Appendix B for complete genesis block structure

First block (height 1) hash: [computed at launch]

Test 13: Difficulty adjustment example

Given: 60 blocks with timestamps $T[0] \dots T[59]$

Target block time: 120 seconds

If average solve time = 100 seconds:

New difficulty = old_difficulty × (120/100) = old_difficulty × 1.2

If average solve time = 150 seconds:

New difficulty = old_difficulty × (120/150) = old_difficulty × 0.8

Clamped to range [0.5, 2.0] per adjustment

Test 14: Chain selection

Chain A: cumulative_difficulty = 1000, height = 100

Chain B: cumulative_difficulty = 1050, height = 99

Selected: Chain B (higher cumulative difficulty wins, not height)

6.3 19. Implementation Requirements

6.3.1 19.1 Mandatory Features

- [MUST] Implement all cryptographic primitives from Part I
 - [MUST] Implement full transaction validation
 - [MUST] Implement STARK prover and verifier
 - [MUST] Implement full node with P2P networking
 - [MUST] Implement wallet with key management
 - [MUST] Pass all test vectors
 - [MUST] Achieve specified performance targets

6.3.2 19.2 Performance Targets

Proof generation: < 60 seconds per transaction (consumer CPU)

Proof verification: < 1 second per transaction

Block validation: < 10 seconds per block (1000 transactions)

Wallet scanning: < 1 second per block

Merkle proof: < 10 ms

Nullifier lookup: < 1 ms

6.3.3 19.3 Security Requirements

- [MUST] Use constant-time implementations for all secret operations
- [MUST] Zeroize sensitive memory after use
- [MUST] Validate all inputs before processing
- [MUST] Implement rate limiting against DoS
- [MUST] Use cryptographically secure random number generation

6.3.4 19.4 Code Quality Requirements

- [MUST] Compile without warnings on strict settings
 - [MUST] Pass static analysis (clippy for Rust, etc.)
 - [MUST] Have >80% test coverage
 - [MUST] Document all public APIs
 - [MUST] Include fuzzing targets for parsers
-

6.4 20. Formal Verification Targets

6.4.1 20.1 Properties to Formally Verify

1. Type safety of all data structures
2. Memory safety (no buffer overflows, use-after-free)
3. Correctness of finite field arithmetic
4. Correctness of polynomial operations
5. Soundness of STARK verifier
6. Balance preservation in transaction validation
7. Nullifier uniqueness enforcement
8. Merkle tree correctness

6.4.2 20.2 Verification Tools

Recommended:

- Rust: MIRI for undefined behavior detection
- Rust: Kani for bounded model checking
- General: TLA+ for protocol logic
- Cryptographic: EasyCrypt for proof verification

Optional:

- Coq/Lean for full formal proofs
- F* for verified implementation extraction

6.4.3 20.3 Audit Checklist

- [] Cryptographic review by domain expert
 - [] Implementation review by security firm
 - [] Formal verification of critical paths
 - [] Fuzzing campaign (>1 billion iterations)
 - [] Incentivized testnet with bug bounty
 - [] Economic audit of incentive mechanisms
-

7 Part IV: Appendices

7.1 A. Reference Implementations

7.1.1 A.1 Polynomial Multiplication (NTT)

```
// Goldilocks field element
type Felt = u64;
const P: u64 = 0xFFFFFFFF00000001; // 2^64 - 2^32 + 1

fn ntt(a: &mut [Felt; 256], omega: Felt) {
    let n = 256;
    let mut m = 1;
    while m < n {
        let w_m = pow_mod(omega, (n / (2 * m)) as u64);
        let mut k = 0;
        while k < n {
            let mut w = 1u64;
            for j in 0..m {
                let t = mul_mod(w, a[k + j + m]);
                let u = a[k + j];
                a[k + j] = add_mod(u, t);
                a[k + j + m] = sub_mod(u, t);
                w = mul_mod(w, w_m);
            }
            k += 2 * m;
        }
        m *= 2;
    }
}
```

```

fn mul_mod(a: u64, b: u64) -> u64 {
    // Montgomery multiplication or Barrett reduction
    ((a as u128 * b as u128) % P as u128) as u64
}

```

7.1.2 A.2 Lattice Commitment

```

const N: usize = 256; // Polynomial degree
const K: usize = 4; // Module rank
const Q: u32 = 8380417; // Modulus
const ETA: i32 = 2; // Noise bound

type Poly = [i32; N];
type PolyVec = [Poly; K];

fn commit(a: &[PolyVec; K], v: u64, r: &PolyVec) -> PolyVec {
    let mut c = [[0i32; N]; K];

    // c = A + r
    for i in 0..K {
        for j in 0..K {
            let product = poly_mul(&a[i][j], &r[j]);
            for k in 0..N {
                c[i][k] = (c[i][k] + product[k]) % Q as i32;
            }
        }
    }

    // c[0] += v (as constant term)
    c[0][0] = (c[0][0] + (v % Q as u64) as i32) % Q as i32;

    c
}

fn poly_mul(a: &Poly, b: &Poly) -> Poly {
    // NTT-based multiplication in R_q
    let mut a_ntt = ntt_forward(a);
    let b_ntt = ntt_forward(b);

    for i in 0..N {
        a_ntt[i] = ((a_ntt[i] as i64 * b_ntt[i] as i64) % Q as i64) as i32;
    }

    let mut result = Poly::new();
    for i in 0..N {
        for j in 0..N {
            result[i] += a_ntt[i] * b_ntt[j];
        }
    }

    result
}

```

```

    }

    ntt_inverse(&a_ntt)
}

```

7.1.3 A.3 STARK Prover Outline

```

struct StarkProver {
    air: ArithmeticIntermediateRepresentation,
    fri_params: FriParameters,
}

impl StarkProver {
    fn prove(&self, witness: &Witness) -> StarkProof {
        // 1. Generate execution trace
        let trace = self.generate_trace(witness);

        // 2. Commit to trace polynomials
        let trace_polys = self.interpolate_trace(&trace);
        let trace_commitment = self.commit_polynomials(&trace_polys);

        // 3. Get challenge for constraint composition
        let alpha = self.fiat_shamir_challenge(&trace_commitment);

        // 4. Compute constraint composition polynomial
        let composition = self.compute_composition(&trace_polys, alpha);
        let composition_commitment = self.commit_polynomials(&[composition]);

        // 5. Get challenge for DEEP composition
        let z = self.fiat_shamir_challenge(&composition_commitment);

        // 6. Compute DEEP quotient
        let deep_quotient = self.compute_deep_quotient(&trace_polys, &composition, z);

        // 7. Run FRI protocol
        let fri_proof = self.fri_prove(&deep_quotient);

        // 8. Generate query responses
        let queries = self.generate_queries(&trace_commitment, &fri_proof);

        // 9. Proof of work grinding
    }
}

```

```
let pow_nonce = self.grind_pow(&queries);
```



```
StarkProof {
    trace_commitment,
    composition_commitment,
    fri_proof,
    queries,
    pow_nonce,
}
```

```
}
```

7.2 B. Genesis Block

7.2.1 B.1 Genesis Parameters

Version: 1

Timestamp: 2026-01-01T00:00:00Z (Unix: 1767225600)

Difficulty target: 0x00000ffffffffffffffffff \dots ffff (approximately 2^{236} , allows ~16 hashes to find valid block)

Nonce: 0 (genesis block has special validation rules)

Transactions: None (empty block)

Genesis message (encoded in first 80 bytes):

"ZKPrivacy Genesis - Quantum-Secure Privacy for All - 2026-01-01"

7.2.2 B.2 Genesis Block Structure

```
struct GenesisBlock {  
    header: BlockHeader {  
        version: 1,  
        previous_hash: [0u8; 32],  
        merkle_root: [0u8; 32],  
        output_tree_root: [0u8; 32],  
        nullifier set root: [0u8; 32],
```

```

        timestamp: 1767225600,
        difficulty: GENESIS_DIFFICULTY,
        nonce: 0,
    },
    transactions: [] ,
}

// Genesis block is validated specially:
// - No PoW check (nonce = 0 is accepted)
// - No previous block check
// - Empty transaction list is valid
// - First mined block (height 1) follows normal rules

```

7.2.3 B.3 Genesis Block Hash

Genesis block hash (SHA3-256 of serialized header):

0x00000000000000000000000000000000[to be computed at mainnet launch]

Note: Testnet will use a different genesis block with timestamp of testnet launch.

7.3 C. Network Magic Numbers

Mainnet magic: 0x5A4B5031 ("ZKP1" in ASCII)

Testnet magic: 0x5A4B5430 ("ZKT0" in ASCII)

Regtest magic: 0x5A4B5230 ("ZKR0" in ASCII)

Protocol version: 1

Minimum supported version: 1

7.4 D. Recommended Libraries

7.4.1 D.1 Rust Ecosystem

Cryptography:

- sha3: SHAKE256 implementation
- blake3: Fast hashing
- curve25519-dalek: For any EC operations needed
- pqcrypto-kyber: ML-KEM implementation
- pqcrypto-sphincsplus: SPHINCS+ implementation

Proof systems:

- winterfell: STARK prover/verifier
- plonky2: Alternative STARK implementation

Networking:

- tokio: Async runtime
- snow: Noise protocol
- libp2p: P2P networking

Storage:

- rocksdb: Key-value store
- sled: Pure Rust alternative

7.4.2 D.2 Alternative Language Implementations

Go:

- gnark: ZK proof systems
- circl: Post-quantum crypto

C/C++:

- liboqs: Post-quantum algorithms
 - libstark: STARK implementation
-

7.5 E. Glossary

AIR: Arithmetic Intermediate Representation

FRI: Fast Reed-Solomon Interactive Oracle Proof of Proximity

ML-KEM: Module Lattice Key Encapsulation Mechanism (Kyber)

NTT: Number Theoretic Transform

STARK: Scalable Transparent Argument of Knowledge

UTXO: Unspent Transaction Output

ZK: Zero-Knowledge

7.6 F. Document Metadata

Title: ZKPrivacy Quantum-Secure Blockchain Specification

Version: 1.0

Status: Draft

Date: 2026-01-14

License: CC0 (Public Domain)

Authors: Phexora AI Research Team

Repository: <https://github.com/phexora/quantum>

Website: <https://quantum.phexora.ai>

Review status:

Cryptographic review: Pending (seeking external reviewers)

Implementation audit: Pending (no implementation yet)

Community feedback: Open for comments via GitHub issues

Document hash (SHA-256):

To be computed when document is finalized.

Use: sha256sum zkprivacy-quantum-spec-v1.md

7.7 G. Known Limitations and Trade-offs

This section documents known limitations and design trade-offs:

7.7.1 G.1 Transaction Size

Issue: Transactions are large (~176 KB for 2-in, 2-out)

Breakdown:

- STARK proof: ~100 KB (dominant factor)
- SPHINCS+ signature: ~50 KB
- Lattice commitments: ~13 KB per output
- Kyber ciphertext: ~1.5 KB per output

Impact:

- Higher bandwidth requirements
- Larger blockchain storage
- ~10 TPS limit with 2 MB blocks

Mitigation:

- Proof aggregation for blocks (future optimization)
- Recursive STARKs for smaller proofs (research area)
- Accept trade-off for quantum security

7.7.2 G.2 Proof Generation Time

Issue: Proof generation takes 30-120 seconds

Cause: STARK provers are computationally intensive

Impact:

- User experience: waiting time for transaction confirmation
- Mobile devices may struggle

Mitigation:

- Hardware acceleration (GPU/FPGA)
- Incremental proving during wallet sync
- Pre-computation of partial proofs
- Accept trade-off for transparency (no trusted setup)

7.7.3 G.3 Address Size

Issue: Addresses are large (1,632 bytes)

Cause:

- SPHINCS+ public key: 64 bytes
- ML-KEM-1024 public key: 1,568 bytes

Impact:

- Cannot use short addresses for display
- QR codes are large

Mitigation:

- Use shortened address (32-byte hash) for display/verification
- Full address only needed in transaction data

7.7.4 G.4 No Smart Contracts

Issue: Version 1.0 does not support programmable transactions

Reason: Complexity and attack surface reduction

Future work:

- Version 2.0 may add ZK-compatible smart contracts
- Research into STARKs for general computation

8 End of Specification

This document contains all information necessary to implement a complete, quantum-secure, privacy-by-default blockchain. Implementations MUST conform to all requirements marked [MUST] and SHOULD implement all performance optimizations.

Any ambiguity in this specification should be resolved by reference to the stated security properties and the principle of conservative security (when in doubt, choose the more secure option).