Quantum

AI Implementation & Verification Guide

Version 1.0

Draft – January 2026

Phexora AI

https://quantum.phexora.ai

Companion document to the Quantum Technical Specification.

Designed for AI-assisted implementation and verification.

Contents

1 Quantum: AI Implementation & Verification Guide

1.1 Purpose

This document provides:

1. Structured implementation tasks for AI systems
2. Verification criteria and test oracles
3. Self-check mechanisms for implementation correctness
4. Benchmark metrics for quality assessment

Target: Advanced AI systems capable of autonomous code generation and verification.

---

2 Introduction

2.1 Who This Guide Is For

This guide is designed for **AI systems and developers** implementing the Quantum specification. It assumes familiarity with:

- Cryptographic primitives (hash functions, commitments, signatures)
- Zero-knowledge proof systems (particularly STARKs)
- Blockchain architecture (transactions, blocks, consensus)
- Systems programming (memory safety, constant-time operations)

If you are reading this as a human developer, you should have experience with cryptographic implementations and understand why correctness is paramount in financial systems.

2.2 How To Use This Document

For AI Systems: This document structures the implementation as a directed acyclic graph of tasks. Complete tasks in topological order (respecting dependencies). Each task includes:

- Input/output specifications
- Verification criteria (what "correct" means)
- Test vectors to validate your implementation

For Human Developers: Use this as a verification checklist. After implementing each component, run the specified tests and oracles to confirm correctness. The test vectors are canonical—if your implementation produces different outputs, it is incorrect.

2.3 The Verification Philosophy

Cryptographic implementations have a unique property: **almost correct is completely broken**. A single bit flip in a signature verification, a subtle timing leak in a constant-time operation, or an off-by-one error in a range check can compromise the entire system.

This guide embodies three principles:

1. **Defense in depth**: Multiple independent checks catch different classes of bugs
2. **Fail-safe defaults**: When uncertain, reject rather than accept
3. **Deterministic verification**: All tests produce identical results across implementations

The immutable requirements (R1-R7) are not negotiable. They define the security boundaries that, if violated, invalidate the entire implementation. Before writing any code, internalize these requirements completely.

---

3 CRITICAL: IMMUTABLE REQUIREMENTS CHECK

3.1 ⚠ BEFORE ANY IMPLEMENTATION ⚠

The implementing AI system MUST first verify understanding of the immutable requirements (Section R1-R7 of the main specification). These requirements are **non-negotiable** and **cannot be modified**.

3.1.1 Pre-Implementation Checklist

```
Before writing ANY code, the AI MUST confirm:

[ ] R1: PRIVACY REQUIREMENTS
    [ ] R1.1: No transparent transaction mode will be implemented
    [ ] R1.2: Anonymity set equals entire output set (not rings/subsets)
    [ ] R1.3: Outputs are unlinkable to addresses without view key
    [ ] R1.4: All amounts are hidden via commitments
    [ ] R1.5: Dandelion++ is included in network layer

[ ] R2: SECURITY REQUIREMENTS
    [ ] R2.1: ONLY post-quantum primitives used:
              - Commitments: Lattice-based (Module-LWE)
              - Signatures: SPHINCS+ (hash-based)
              - Key exchange: ML-KEM (Kyber)
              - ZK proofs: STARKs (hash-based)
```

```
                - NO elliptic curves anywhere
    [ ] R2.2: No trusted setup exists
    [ ] R2.3: Commitment binding is proven
    [ ] R2.4: Commitment hiding is proven
    [ ] R2.5: STARK soundness < 2^-100
    [ ] R2.6: STARK is zero-knowledge

[ ] R3: DECENTRALIZATION REQUIREMENTS
    [ ] R3.1: No permissions required to participate
    [ ] R3.2: No privileged parties or special keys
    [ ] R3.3: RandomX (ASIC-resistant) for mining
    [ ] R3.4: All code is open source

[ ] R4: INTEGRITY REQUIREMENTS
    [ ] R4.1: Supply capped at exactly 21,000,000 ZKP
    [ ] R4.2: Balance equation enforced in ZK proof
    [ ] R4.3: Nullifiers prevent double-spending
    [ ] R4.4: Heaviest chain rule for finality

[ ] R5: FUNCTIONAL REQUIREMENTS
    [ ] R5.1: Multi-input/output transactions supported
    [ ] R5.2: Deterministic wallets from seed
    [ ] R5.3: Light client support

[ ] R6: PERFORMANCE REQUIREMENTS
    [ ] R6.1: Proof generation < 120s, verification < 2s
    [ ] R6.2: Runs on 16GB RAM, 500GB storage
    [ ] R6.3: 120s block time,  10 TPS

[ ] R7: NON-REQUIREMENTS ACKNOWLEDGED
    [ ] R7.1: No smart contracts in v1
    [ ] R7.2: No backdoors, no censorship, no blacklisting
```

3.1.2 Automatic Requirement Violation Detection

The implementation MUST include automated checks that detect requirement violations:

```rust
/// Run at compile time and runtime
pub fn verify_requirements_compliance() -> Result<(), RequirementViolation> {
    // R1.1: No transparent mode
    #[cfg(feature = "transparent_transactions")]
```

```
    compile_error!("REQUIREMENT VIOLATION R1.1: Transparent transactions are forbidde

    // R2.1: No elliptic curves
    #[cfg(feature = "elliptic_curves")]
    compile_error!("REQUIREMENT VIOLATION R2.1: Elliptic curves are forbidden");

    // R4.1: Supply cap
    const MAX_SUPPLY: u64 = 21_000_000 * 100_000_000; // in satoshi
    assert!(TOTAL_SUPPLY_CAP == MAX_SUPPLY, "REQUIREMENT VIOLATION R4.1");

    // R3.2: No privileged keys
    assert!(PRIVILEGED_KEYS.is_empty(), "REQUIREMENT VIOLATION R3.2");

    Ok(())
}
```

3.1.3 Requirement Violation = Implementation Failure

```
IF any requirement R1-R7 is violated:
    THEN implementation is INVALID
    THEN implementation MUST be rejected
    THEN implementation MUST be restarted from scratch

There is NO partial compliance.
There are NO exceptions.
There are NO workarounds.
```

---

4 Part I: Implementation Task Decomposition

This section breaks the Quantum implementation into discrete, verifiable tasks. The tasks are organized as a directed acyclic graph (DAG) where edges represent dependencies: task B depends on task A means you cannot correctly implement B without a working A.

Why This Structure? Cryptographic systems have strict layering. Field arithmetic must be correct before polynomial operations can work. Polynomial operations must be correct before commitments can work. Commitments must be correct before transactions can work. An error in a lower layer propagates upward, causing cascading failures that may be difficult to debug.

Verification At Each Level: Each task includes verification criteria that must pass before pro-

ceeding. Do not skip verification "to save time"—debugging a high-level failure caused by a low-level bug is far more expensive than verifying each layer in isolation.

4.1 1. Dependency Graph

```
Implementation order (topological sort):

Level 0 (No dependencies):
       T001: Field arithmetic (Goldilocks)
       T002: Polynomial ring arithmetic
       T003: Hash functions (SHAKE256 wrappers)
       T004: Serialization primitives

Level 1 (Depends on Level 0):
       T101: NTT implementation
       T102: Merkle tree
       T103: Domain-separated hash instances
       T104: Random number generation

Level 2 (Depends on Level 1):
       T201: Lattice commitment scheme
       T202: SPHINCS+ integration
       T203: ML-KEM integration
       T204: Nullifier derivation

Level 3 (Depends on Level 2):
       T301: Output structure
       T302: Stealth addresses
       T303: Key hierarchy
       T304: Address encoding

Level 4 (Depends on Level 3):
       T401: Transaction structure
       T402: STARK AIR definition
       T403: Wallet scanning
       T404: Transaction creation

Level 5 (Depends on Level 4):
       T501: STARK prover
       T502: STARK verifier
       T503: Block structure
```

```
        T504: Consensus rules

Level 6 (Depends on Level 5):
        T601: Full node
        T602: P2P networking
        T603: Chain state management
        T604: Complete wallet

Level 7 (Integration):
        T701: System integration and testing
```

4.2 2. Task Specifications

Each task below specifies:

- **What** to implement (required operations)
- **How** to verify correctness (test vectors and properties)
- **Why** this component matters (security context)

Understanding the "why" helps catch subtle bugs that might technically pass tests but violate security assumptions.

4.2.1 T001: Field Arithmetic (Goldilocks)

Why This Matters: Every cryptographic operation in the STARK prover and verifier reduces to field arithmetic. If addition wraps incorrectly, if multiplication overflows, if inversion fails for some inputs—the resulting proofs may be unsound (accepting invalid transactions) or incomplete (rejecting valid ones). Both are catastrophic.

Input specification:

```
Field: F_p where p = 2^64 - 2^32 + 1
Elements: 64-bit unsigned integers
```

Required operations:

```
trait GoldilocksField {
    fn add(a: u64, b: u64) -> u64;
    fn sub(a: u64, b: u64) -> u64;
    fn mul(a: u64, b: u64) -> u64;
    fn inv(a: u64) -> u64;  // Multiplicative inverse
    fn pow(base: u64, exp: u64) -> u64;
    fn neg(a: u64) -> u64;
```

```
    // Batch operations for efficiency
    fn batch_inv(elements: &[u64]) -> Vec<u64>;
}
```

Verification criteria:

```
V001.1: add(a, b) = (a + b) mod p
V001.2: mul(a, b) = (a × b) mod p
V001.3: inv(a) × a = 1 mod p for a   0
V001.4: All operations complete in constant time
V001.5: No overflow in intermediate computations
```

Test vectors:

```
add(p-1, 1) = 0
mul(p-1, p-1) = 1
inv(2) = (p+1)/2 = 9223372034707292161
pow(7, p-1) = 1 (Fermat's little theorem)
```

4.2.2 T002: Polynomial Ring Arithmetic

Input specification:

```
Ring: R_q = Z_q[X]/(X^256 + 1)
q = 8380417
Coefficients: Signed 32-bit integers (reduced mod q)
```

Required operations:

```
trait PolynomialRing {
    fn add(a: &Poly, b: &Poly) -> Poly;
    fn sub(a: &Poly, b: &Poly) -> Poly;
    fn mul(a: &Poly, b: &Poly) -> Poly;  // Via NTT
    fn scalar_mul(a: &Poly, s: i32) -> Poly;
    fn reduce(a: &Poly) -> Poly;  // Reduce coefficients mod q
}

type Poly = [i32; 256];
```

Verification criteria:

```
V002.1: Coefficients always in [-(q-1)/2, (q-1)/2] after reduce
V002.2: mul satisfies (X^256 + 1) reduction
V002.3: Ring axioms hold (associativity, distributivity)
```

4.2.3 T003: Hash Functions

Input specification:

```
Base: SHAKE256 (FIPS 202)
Domain separation: Prefix with tagged length-encoded domain string
```

Required instances:

```rust
trait HashInstances {
    fn h_commitment(input: &[u8]) -> [u8; 64];
    fn h_nullifier(input: &[u8]) -> [u8; 32];
    fn h_merkle(input: &[u8]) -> [u8; 32];
    fn h_address(input: &[u8]) -> [u8; 32];
    fn h_kdf(input: &[u8], output_len: usize) -> Vec<u8>;
    fn h_challenge(input: &[u8]) -> [u8; 64];
    fn h_pow(input: &[u8]) -> [u8; 32];
}
```

Verification criteria:

```
V003.1: h_X(m)   h_Y(m) for X   Y (domain separation)
V003.2: Output matches SHAKE256 reference implementation
V003.3: Streaming API for large inputs
```

Test vectors (must match specification Section 18.1):

```
h_nullifier(0x00^64):
    Domain: "Quantum-v1.nullifier"
    Output: 0x3a7f2c9e8b4d1a6f5c0e7b3d9a2f8c4e1b6d0a5f3e9c7b2d8a4e6f1c0b5d9a3e

h_merkle(0x00 || 0x00^32 || 0x00^32):
    Domain: "Quantum-v1.merkle"
    Output: 0x5c9a3e7f1b4d8c2e6a0f5b9d3c7e1a4f8b2d6e0a4c9f3b7e1d5a8c2f6e0b4d9a

Domain separation test:
    h_nullifier(0x00^32)   h_commitment(0x00^32)   h_merkle(0x00^32)
```

4.2.4 T101: NTT Implementation

Input specification:

```
Transform size: 256
```

```
Field: Z_q where q = 8380417
Primitive 512th root of unity:   = 1753
```

Required operations:

```rust
trait NTT {
    fn forward(a: &Poly) -> Poly;
    fn inverse(a: &Poly) -> Poly;
    fn pointwise_mul(a: &Poly, b: &Poly) -> Poly;  // In NTT domain
}
```

Verification criteria:

```
V101.1: inverse(forward(a)) = a
V101.2: forward(a * b) = pointwise_mul(forward(a), forward(b))
V101.3:  ^512 = 1 mod q
V101.4:  ^256 = -1 mod q
```

4.2.5 T102: Merkle Tree

Input specification:

```
Hash: h_merkle
Depth: 40
Leaf prefix: 0x01
Node prefix: 0x00
```

Required operations:

```rust
trait MerkleTree {
    fn new(depth: u32) -> Self;
    fn append(&mut self, leaf: &[u8; 32]) -> u64;  // Returns position
    fn root(&self) -> [u8; 32];
    fn prove(&self, position: u64) -> MerkleProof;
    fn verify(root: &[u8; 32], leaf: &[u8; 32], position: u64, proof: &MerkleProof) -
}

struct MerkleProof {
    siblings: Vec<[u8; 32]>,  // Length = depth
    path_bits: u64,           // Left/right indicators
}
```

Verification criteria:

```
V102.1: verify(tree.root(), leaf, pos, tree.prove(pos)) = true
```

```
V102.2: Proof size exactly depth × 32 bytes
V102.3: Empty tree root is well-defined
V102.4: Append-only (no modification of existing leaves)
```

4.2.6 T201: Lattice Commitment Scheme

Input specification:

```
Parameters from spec: n=256, q=8380417, k=4,  =2
Public parameters: Matrix A   R_q^{k×k}
```

Required operations:

```
trait LatticeCommitment {
    fn setup(seed: &[u8; 32]) -> PublicParams;
    fn commit(pp: &PublicParams, value: u64, randomness: &PolyVec) -> Commitment;
    fn verify_opening(pp: &PublicParams, c: &Commitment, v: u64, r: &PolyVec) -> bool
    fn add(c1: &Commitment, c2: &Commitment) -> Commitment;

    fn generate_randomness(seed: &[u8; 32]) -> PolyVec;
}

type PolyVec = [Poly; 4];
struct Commitment { data: PolyVec }
```

Verification criteria:

```
V201.1: Randomness coefficients in [- ,  ]
V201.2: commit(v1, r1) + commit(v2, r2) = commit(v1+v2, r1+r2)
V201.3: Cannot find collision (binding)
V201.4: Commitment reveals nothing about value (hiding)
```

4.2.7 T301: Output Structure

Required structure:

```
struct Output {
    commitment: Commitment,           // ~3KB
    kyber_ciphertext: [u8; 1568],     // ML-KEM-1024
    encrypted_data: [u8; 128],        // AES-GCM
}

struct OutputPlaintext {
```

```
    value: u64,
    blinding_seed: [u8; 32],
    memo: [u8; 64],
    checksum: [u8; 16],
}

impl Output {
    fn create(
        pp: &PublicParams,
        recipient_view_pk: &KyberPublicKey,
        value: u64,
        memo: &[u8; 64],
    ) -> (Self, OutputSecrets);

    fn try_decrypt(
        &self,
        view_sk: &KyberSecretKey,
    ) -> Option<OutputPlaintext>;

    fn serialize(&self) -> Vec<u8>;
    fn deserialize(data: &[u8]) -> Result<Self, Error>;
}
```

Verification criteria:

```
V301.1: Serialization is canonical and deterministic
V301.2: try_decrypt succeeds only with correct key
V301.3: Checksum validates integrity
V301.4: Output size matches specification (~13KB)
```

4.2.8 T401: Transaction Structure

Required structure:

```
struct Transaction {
    nullifiers: Vec<[u8; 32]>,
    outputs: Vec<Output>,
    fee: u64,
    validity_proof: StarkProof,
    authorization: Authorization,
    anchor: [u8; 32],
```

```
}

impl Transaction {
    fn create(
        wallet: &Wallet,
        inputs: &[OwnedOutput],
        recipients: &[(Address, u64)],
        fee: u64,
        anchor: [u8; 32],
    ) -> Result<Self, Error>;

    fn verify(&self, utxo_tree_root: &[u8; 32]) -> bool;

    fn nullifier_count(&self) -> usize;
    fn output_count(&self) -> usize;
    fn serialized_size(&self) -> usize;
}
```

Verification criteria:

```
V401.1: Valid transaction passes verify()
V401.2: Invalid balance fails verify()
V401.3: Reused nullifier fails verify()
V401.4: Wrong anchor fails verify()
```

4.2.9 T501: STARK Prover

Input specification:

```
Statement: Transaction validity (balance, range, membership, nullifiers)
Security: 100-bit soundness
Proof size: < 200 KB
```

Required interface:

```
trait StarkProver {
    fn prove(
        public_inputs: &PublicInputs,
        witness: &Witness,
    ) -> StarkProof;
}

struct PublicInputs {
```

```
    nullifiers: Vec<[u8; 32]>,
    output_commitments: Vec<Commitment>,
    fee: u64,
    anchor: [u8; 32],
}

struct Witness {
    input_values: Vec<u64>,
    input_blindings: Vec<PolyVec>,
    input_positions: Vec<u64>,
    input_merkle_paths: Vec<MerkleProof>,
    input_nullifier_keys: Vec<[u8; 32]>,
    output_values: Vec<u64>,
    output_blindings: Vec<PolyVec>,
}
```

Verification criteria:

```
V501.1: Proof verifies for valid witness
V501.2: Cannot generate valid proof for invalid statement
V501.3: Proof size < 200 KB
V501.4: Proving time < 60 seconds (benchmark hardware)
```

4.2.10 T502: STARK Verifier

Required interface:

```
trait StarkVerifier {
    fn verify(
        public_inputs: &PublicInputs,
        proof: &StarkProof,
    ) -> bool;
}
```

Verification criteria:

```
V502.1: Accepts valid proofs
V502.2: Rejects invalid proofs with overwhelming probability
V502.3: Verification time < 1 second
V502.4: Constant-time execution (no timing leaks)
```

4.3 3. Integration Tests

4.3.1 IT001: End-to-End Transaction

```
Setup:
    1. Generate two wallets (Alice, Bob)
    2. Initialize chain with genesis block
    3. Mine blocks to give Alice coins

Test:
    4. Alice creates transaction sending to Bob
    5. Transaction is validated and included in block
    6. Bob's wallet scans and finds output
    7. Bob can spend the received output

Verify:
    - Alice's balance decreased correctly
    - Bob's balance increased correctly
    - Nullifiers are recorded
    - Chain state is consistent
```

4.3.2 IT002: Double-Spend Prevention

```
Setup:
    1. Wallet with single UTXO

Test:
    2. Create transaction spending the UTXO
    3. Create second transaction spending same UTXO
    4. Submit both to node

Verify:
    - First transaction accepted
    - Second transaction rejected
    - Only one nullifier recorded
```

4.3.3 IT003: DAG Reordering

```
Setup:
    1. DAG with blue score 1000 at tips
    2. Receive delayed blocks that change canonical ordering
```

```
Test:
    3. Receive late-arriving blocks with valid PoW
    4. Process DAG reordering

Verify:
    - All valid blocks remain in DAG (none orphaned)
    - Canonical ordering updates based on GhostDAG blue score
    - Transactions in reordered blocks maintain validity
    - Nullifier set remains consistent after reorder
```

---

5 Part II: Verification Oracles

Verification oracles are automated tests that check critical security properties. Unlike unit tests (which verify that code does what you wrote), oracles verify that code does what *cryptography requires*. A passing unit test with a failing oracle indicates a specification bug or fundamental misunderstanding.

5.1 4. Cryptographic Oracles

Cryptographic properties are the foundation of system security. If any of these oracles fail, the entire system is compromised—not degraded, not weakened, but broken.

5.1.1 Why Commitment Binding Matters

If commitments aren't binding, an attacker could commit to "100 coins" but later open it as "1,000,000 coins"—creating money from nothing. This would silently inflate the supply while all proofs verify correctly. The binding property ensures that once you commit to a value, you're locked to that value forever.

5.1.2 Why Commitment Hiding Matters

If commitments leak information about values, transaction privacy is broken. Even partial leakage (e.g., "this commitment is probably a large amount") enables statistical attacks that deanonymize users over time. The hiding property ensures that commitments are indistinguishable regardless of the committed value.

5.1.3 Why STARK Soundness Matters

If invalid proofs can pass verification, attackers can steal funds, double-spend, or violate any protocol rule. Soundness with error < 2^-100 means that even an attacker who generates 2^99

proof attempts has negligible chance of forging a valid proof for a false statement.

5.1.4 O001: Commitment Binding Oracle

```
def test_binding(implementation):
    """Test that commitments are binding."""
    pp = implementation.setup(random_seed())

    # Try to find collision
    for _ in range(1000000):
        v1, r1 = random_value(), random_blinding()
        v2, r2 = random_value(), random_blinding()

        c1 = implementation.commit(pp, v1, r1)
        c2 = implementation.commit(pp, v2, r2)

        if c1 == c2 and (v1, r1) != (v2, r2):
            return FAIL("Found collision")

    return PASS("No collision found in 10^6 attempts")
```

5.1.5 O002: STARK Soundness Oracle

```
def test_soundness(implementation):
    """Test STARK soundness with invalid witnesses."""

    test_cases = [
        # Unbalanced transaction
        {"inputs": [100], "outputs": [50, 60], "fee": 0},  # 110 > 100

        # Negative value (overflow attempt)
        {"inputs": [100], "outputs": [2**64-1, 101], "fee": 0},

        # Invalid Merkle proof
        {"valid_merkle": False},

        # Wrong nullifier
        {"correct_nullifier": False},
    ]

    for case in test_cases:
```

```
        witness = generate_invalid_witness(case)
        proof = implementation.prove(witness)

        if implementation.verify(proof):
            return FAIL(f"Accepted invalid case: {case}")

    return PASS("Rejected all invalid cases")
```

5.1.6 O003: Privacy Oracle

```
def test_transaction_privacy(implementation):
    """Test that transactions reveal no private information."""

    # Create many transactions with different parameters
    transactions = []
    for _ in range(1000):
        tx = implementation.create_transaction(
            random_inputs(),
            random_outputs(),
            random_fee()
        )
        transactions.append(tx)

    # Statistical tests on serialized transactions
    serialized = [tx.serialize() for tx in transactions]

    # Test 1: No correlation between tx size and value
    if correlation(sizes, values) > 0.1:
        return FAIL("Size leaks value information")

    # Test 2: Byte distribution is uniform
    if not chi_squared_uniform(concatenate(serialized)):
        return FAIL("Non-uniform byte distribution")

    # Test 3: No timing correlation
    prove_times = measure_prove_times(transactions)
    if correlation(prove_times, values) > 0.1:
        return FAIL("Timing leaks value information")

    return PASS("No detectable information leakage")
```

5.2 5. Performance Oracles

5.2.1 O004: Proof Generation Benchmark

```python
def benchmark_proving(implementation):
    """Benchmark proof generation time."""

    configurations = [
        {"inputs": 1, "outputs": 2},   # Minimal
        {"inputs": 2, "outputs": 2},   # Typical
        {"inputs": 4, "outputs": 2},   # Multi-input
        {"inputs": 2, "outputs": 8},   # Multi-output
        {"inputs": 16, "outputs": 2},  # Large
    ]

    results = {}
    for config in configurations:
        times = []
        for _ in range(10):
            witness = generate_valid_witness(config)
            start = time.monotonic()
            implementation.prove(witness)
            elapsed = time.monotonic() - start
            times.append(elapsed)

        results[str(config)] = {
            "mean": statistics.mean(times),
            "std": statistics.stdev(times),
            "max": max(times),
        }

    # Check against requirements
    for config, result in results.items():
        if result["mean"] > 60.0:
            return FAIL(f"Proving too slow for {config}: {result['mean']:.1f}s")

    return PASS(f"All configurations within limits: {results}")
```

5.2.2 O005: Verification Benchmark

```
def benchmark_verification(implementation):
    """Benchmark proof verification time."""

    proofs = [generate_valid_proof() for _ in range(100)]

    times = []
    for proof in proofs:
        start = time.monotonic()
        result = implementation.verify(proof)
        elapsed = time.monotonic() - start
        times.append(elapsed)
        assert result == True

    mean_time = statistics.mean(times)
    max_time = max(times)

    if max_time > 1.0:
        return FAIL(f"Verification too slow: max {max_time:.3f}s")

    return PASS(f"Verification time: mean={mean_time:.3f}s, max={max_time:.3f}s")
```

5.3 6. Fuzz Testing Specifications

5.3.1 F001: Serialization Fuzzing

```
def fuzz_serialization(implementation):
    """Fuzz test serialization/deserialization."""

    # Property: deserialize(serialize(x)) == x
    for _ in range(1000000):
        tx = generate_random_transaction()
        serialized = implementation.serialize(tx)
        deserialized = implementation.deserialize(serialized)
        assert tx == deserialized

    # Property: Invalid bytes should not crash
    for _ in range(1000000):
        random_bytes = os.urandom(random.randint(0, 1000000))
```

```
        try:
            implementation.deserialize(random_bytes)
        except DeserializationError:
            pass  # Expected
        except Exception as e:
            return FAIL(f"Unexpected exception: {e}")

    return PASS("Serialization fuzzing passed")
```

5.3.2 F002: Arithmetic Fuzzing

```
def fuzz_field_arithmetic(implementation):
    """Fuzz test field operations."""

    p = 2**64 - 2**32 + 1

    for _ in range(10000000):
        a = random.randint(0, p-1)
        b = random.randint(0, p-1)

        # Addition
        assert implementation.add(a, b) == (a + b) % p

        # Multiplication
        assert implementation.mul(a, b) == (a * b) % p

        # Inverse
        if a != 0:
            inv_a = implementation.inv(a)
            assert implementation.mul(a, inv_a) == 1

    return PASS("Field arithmetic fuzzing passed")
```

---

6 Part III: Quality Metrics

6.1 7. Code Quality Checklist

```
[ ] CQ001: No compiler warnings (strict mode)
[ ] CQ002: All public APIs documented
```

```
[ ] CQ003: Test coverage > 80%
[ ] CQ004: No unsafe blocks (or justified and audited)
[ ] CQ005: No panics in library code
[ ] CQ006: All errors are typed and recoverable
[ ] CQ007: No hardcoded magic numbers (use named constants)
[ ] CQ008: Consistent naming conventions
[ ] CQ009: No dead code
[ ] CQ010: Passes clippy/lint with no warnings
```

6.2 8. Security Checklist

```
[ ] SC001: All secret operations are constant-time
[ ] SC002: Sensitive memory is zeroized after use
[ ] SC003: No secret-dependent branches
[ ] SC004: No secret-dependent memory access patterns
[ ] SC005: CSPRNG used for all randomness
[ ] SC006: Input validation on all external data
[ ] SC007: No integer overflow vulnerabilities
[ ] SC008: No buffer overflow vulnerabilities
[ ] SC009: Rate limiting on network inputs
[ ] SC010: Timeout handling on all operations
```

6.3 9. Performance Metrics

Target metrics (consumer hardware: 8-core CPU, 16GB RAM):

Operation	Target	Measured
Field multiplication	< 10 ns	
Polynomial multiplication	< 100 s	
Commitment creation	< 10 ms	
Merkle proof generation	< 1 ms	
Merkle proof verification	< 1 ms	
Output encryption	< 5 ms	
Output decryption	< 5 ms	
Wallet scan (per output)	< 5 ms	
STARK proof generation (2-2)	< 60 s	
STARK proof verification	< 1 s	
Block validation (1000 tx)	< 10 s	
Transaction serialization	< 1 ms	
Transaction deserialization	< 1 ms	

7 Part IV: Acceptance Criteria

7.1 10. Minimum Viable Implementation

An implementation is considered **complete** when:

```
Cryptographic Layer:
    [x] All hash functions implemented with test vectors passing
    [x] Lattice commitment scheme with binding/hiding tests
    [x] ML-KEM integration with KAT vectors
    [x] SPHINCS+ integration with KAT vectors
    [x] Merkle tree with correctness tests

Transaction Layer:
    [x] Output creation and encryption
    [x] Output scanning and decryption
    [x] Nullifier derivation
    [x] Transaction structure serialization

Proof System:
    [x] AIR constraints correctly specified
    [x] STARK prover generates valid proofs
    [x] STARK verifier rejects invalid proofs
    [x] Soundness tests pass

Consensus:
    [x] Block header validation
    [x] Difficulty adjustment
    [x] Chain selection rules
    [x] Block validation with transaction verification

Networking:
    [x] Peer discovery
    [x] Block propagation
    [x] Transaction propagation
    [x] Dandelion++ for transaction privacy

Wallet:
    [x] Key generation from seed
```

```
    [x] Address derivation
    [x] Balance tracking
    [x] Transaction creation
    [x] Output scanning

Integration:
    [x] End-to-end transaction test passes
    [x] Chain synchronization works
    [x] No memory leaks in long-running tests
    [x] All performance targets met
```

7.2 11. Formal Verification Requirements

Mandatory formal verification targets:

```
1. Field arithmetic correctness
   Tool: Bounded model checking (Kani)
   Property: All operations produce correct results mod p

2. Memory safety
   Tool: MIRI
   Property: No undefined behavior

3. Merkle tree correctness
   Tool: Property-based testing (QuickCheck/Proptest)
   Property: Verify-proof always succeeds for valid proofs

4. STARK verifier soundness
   Tool: Cryptographic review + testing
   Property: Rejects invalid proofs

5. No secret-dependent timing
   Tool: dudect or similar
   Property: Timing is independent of secret values
```

Optional formal verification targets:

```
6. Full STARK soundness proof (EasyCrypt/Coq)
7. Protocol-level security proof (TLA+)
8. Economic incentive analysis (game theory)
```

7.3 12. Benchmarking Protocol

For reproducible benchmarks:

```
Hardware specification:
    CPU: AMD Ryzen 9 5900X or equivalent
    RAM: 64 GB DDR4-3200
    Storage: NVMe SSD
    OS: Ubuntu 22.04 LTS

Software configuration:
    Rust: Latest stable
    Compiler flags: --release with LTO
    No other significant processes running

Benchmark procedure:
    1. Warm-up: Run operation 100 times, discard results
    2. Measurement: Run operation 1000 times
    3. Report: Mean, median, std dev, min, max
    4. Verify: Results reproducible across runs
```

7.4 13. Self-Verification Commands

An AI implementation should expose these verification commands:

```
# Run all unit tests
zkprivacy test --all

# Run cryptographic test vectors
zkprivacy test --vectors

# Run fuzz tests (1 hour)
zkprivacy fuzz --duration 3600

# Run performance benchmarks
zkprivacy bench --full

# Run security checks
zkprivacy audit --security

# Verify against specification
```

```
zkprivacy verify --spec ./zkprivacy-quantum-spec-v1.md

# Generate implementation report
zkprivacy report --output implementation-report.json
```

Expected report format:

```
{
    "version": "1.0.0",
    "spec_version": "1.0",
    "timestamp": "2026-01-14T12:00:00Z",
    "tests": {
        "unit": {"passed": 1234, "failed": 0, "skipped": 0},
        "integration": {"passed": 56, "failed": 0, "skipped": 0},
        "fuzz": {"iterations": 10000000, "failures": 0}
    },
    "benchmarks": {
        "proof_generation_2_2": {"mean_ms": 45000, "target_ms": 60000},
        "proof_verification": {"mean_ms": 800, "target_ms": 1000}
    },
    "security": {
        "constant_time_check": "PASS",
        "memory_safety": "PASS",
        "input_validation": "PASS"
    },
    "coverage": {
        "line": 0.87,
        "branch": 0.82
    },
    "verification_hash": "sha256:abc123..."
}
```

---

8 End of Verification Guide

This document provides complete criteria for implementing and verifying the Quantum specification. An implementation that satisfies all requirements in this document is considered conformant.