

Quantum

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.

1 Abstract

Quantum is a research specification for a privacy-by-default, high-throughput blockchain designed to remain secure against both classical and quantum adversaries. It addresses three fundamental limitations of existing blockchains: vulnerability to quantum computers, lack of default privacy, and limited scalability.

The core research challenge: No existing blockchain combines DAG-based consensus (like Kaspa’s GhostDAG) with full transaction privacy. Quantum aims to solve this, enabling 1,000+ TPS on L1 while maintaining complete anonymity.

This specification combines five key innovations: **(1)** DAG-based consensus (GhostDAG) enabling parallel block creation and 10-32 blocks per second, **(2)** lattice-based commitment schemes (Module-LWE) that hide transaction amounts while preserving verifiable supply integrity, **(3)** hash-based signatures (SPHINCS+) and key encapsulation (ML-KEM) that resist quantum attacks, **(4)** transparent zero-knowledge proofs (STARKs) that require no trusted setup ceremony, and **(5)** privacy-preserving proofs over DAG structures—the novel cryptographic contribution that enables private transactions in a parallel-block architecture. The result is a system targeting Kaspa-level throughput with Monero-level privacy and post-quantum security.

This document serves as both a formal specification and an implementation guide. It is explicitly designed to be machine-verifiable, enabling AI systems to implement, test, and formally verify conformant implementations. All cryptographic parameters are fully specified, all algorithms are deterministic, and all ambiguities are resolved in favor of security.

Contents

1 Abstract	i
2 Introduction	1
2.1 The Quantum Threat to Blockchain Security	1
2.2 The Privacy Imperative	1
2.3 Our Approach	1
3 Design Philosophy	2
3.1 Why Post-Quantum Now?	2
3.2 Why Lattice-Based Commitments Over Pedersen?	3
3.3 Why STARKs Over SNARKs?	3
3.4 Why Privacy-by-Default With No Opt-Out?	3
3.5 Why DAG-Based Consensus (GhostDAG)?	4

3.6	Mining Algorithm (Open Research Question)	4
3.7	Why These Specific Parameters?	5
3.8	Why Bitcoin's Foundation? UTXO Model, Nakamoto Consensus, Fair Launch	6
4	Project Status and Research Phases	7
4.1	Honest Assessment	7
4.2	Research Phases and Milestones	7
4.2.1	Phase 1: Specification and Formal Analysis (Current)	8
4.2.2	Phase 2: Cryptographic Proof-of-Concept	8
4.2.3	Phase 3: Testnet Implementation	8
4.2.4	Phase 4: Mainnet Launch	9
4.3	What This Means for Stakeholders	9
4.3.1	For Researchers	9
4.3.2	For Investors	9
4.3.3	For the Community	10
4.4	Current Status	10
5	Quantum: Quantum-Secure Privacy Blockchain	11
5.1	Formal Specification v1.0	11
6	IMMUTABLE REQUIREMENTS	11
6.1	¶ THIS SECTION IS IMMUTABLE ¶	11
6.2	R1. PRIVACY REQUIREMENTS	11
6.2.1	R1.1 Privacy by Default [MANDATORY]	11
6.2.2	R1.2 Sender Privacy [MANDATORY]	12
6.2.3	R1.3 Receiver Privacy [MANDATORY]	12
6.2.4	R1.4 Amount Privacy [MANDATORY]	12
6.2.5	R1.5 Network Privacy [MANDATORY]	12
6.3	R2. SECURITY REQUIREMENTS	12
6.3.1	R2.1 Quantum Security [MANDATORY]	12
6.3.2	R2.2 No Trusted Setup [MANDATORY]	13
6.3.3	R2.3 Cryptographic Binding [MANDATORY]	13
6.3.4	R2.4 Cryptographic Hiding [MANDATORY]	13
6.3.5	R2.5 Proof Soundness [MANDATORY]	13
6.3.6	R2.6 Proof Zero-Knowledge [MANDATORY]	13
6.4	R3. DECENTRALIZATION REQUIREMENTS	13
6.4.1	R3.1 Permissionless Participation [MANDATORY]	13
6.4.2	R3.2 No Privileged Parties [MANDATORY]	14
6.4.3	R3.3 ASIC Resistance [MANDATORY]	14
6.4.4	R3.4 Open Source [MANDATORY]	14
6.5	R4. INTEGRITY REQUIREMENTS	14

6.5.1	R4.1 Fixed Supply [MANDATORY]	14
6.5.2	R4.2 No Inflation Bugs [MANDATORY]	14
6.5.3	R4.3 Double-Spend Prevention [MANDATORY]	14
6.5.4	R4.4 Transaction Finality [MANDATORY]	15
6.6	R5. FUNCTIONAL REQUIREMENTS	15
6.6.1	R5.1 Basic Transaction Support [MANDATORY]	15
6.6.2	R5.2 Wallet Functionality [MANDATORY]	15
6.6.3	R5.3 Light Client Support [MANDATORY]	15
6.7	R6. PERFORMANCE REQUIREMENTS	15
6.7.1	R6.1 Transaction Processing [MANDATORY]	15
6.7.2	R6.2 Storage [MANDATORY]	16
6.7.3	R6.3 Network [MANDATORY]	16
6.8	R7. NON-REQUIREMENTS (Explicitly Excluded)	16
6.8.1	R7.1 NOT Required	16
6.8.2	R7.2 NOT Permitted	16
6.9	R8. SCALABILITY REQUIREMENTS	17
6.9.1	R8.1 DAG-Native Architecture [MANDATORY]	17
6.9.2	R8.2 L1 Throughput Targets [MANDATORY]	17
6.9.3	R8.3 Parallel Processing [MANDATORY]	17
6.9.4	R8.4 Privacy-Preserving DAG Consensus [MANDATORY]	17
6.9.5	R8.5 Horizontal Scaling [MANDATORY]	17
6.9.6	R8.6 Layer 2 Compatibility [MANDATORY]	18
6.10	Requirement Compliance Matrix	18
6.11	Immutability Declaration	19
7	END OF IMMUTABLE REQUIREMENTS	19
8	Part I: Cryptographic Foundation	19
8.1	1. Notation and Conventions	20
8.1.1	1.1 Mathematical Notation	20
8.1.2	1.2 Security Parameter	20
8.1.3	1.3 Endianness and Encoding	20
8.2	2. Hash Functions	21
8.2.1	2.1 Primary Hash Function: SHAKE256	21
8.2.2	2.2 Defined Hash Instances	21
8.2.3	2.3 Hash-to-Field	21
8.3	3. Lattice-Based Commitments	22
8.3.1	3.1 Module-LWE Parameters	22
8.3.2	3.2 Polynomial Operations	22
8.3.3	3.3 Commitment Scheme	23
8.3.4	3.4 Randomness Generation	24

8.4	4. Hash-Based Signatures: SPHINCS+-256f	24
8.4.1	4.1 Parameters	24
8.4.2	4.2 API	24
8.4.3	4.3 Security	25
8.5	5. Key Encapsulation: ML-KEM-1024 (Kyber)	25
8.5.1	5.1 Parameters	25
8.5.2	5.2 API	25
8.6	6. Zero-Knowledge Proofs: STARKs	26
8.6.1	6.1 Overview	26
8.6.2	6.2 Arithmetic Intermediate Representation (AIR)	26
8.6.3	6.3 Field Selection	26
8.6.4	6.4 FRI Parameters (Fast Reed-Solomon IOP)	27
8.6.5	6.5 STARK Proof Structure	27
8.7	7. Merkle Trees (Quantum-Secure)	27
8.7.1	7.1 Construction	27
8.7.2	7.2 Tree Parameters	28
8.7.3	7.3 Append-Only Tree	28
9	Part II: Protocol Specification	28
9.1	8. Account and Address System	29
9.1.1	8.1 Key Hierarchy	29
9.1.2	8.2 Address Format	29
9.1.3	8.3 Stealth Addresses	30
9.2	9. Transaction Structure	30
9.2.1	9.1 Output (Note)	30
9.2.2	9.2 Nullifier	31
9.2.3	9.3 Transaction	31
9.2.4	9.4 Transaction Size Estimate	32
9.3	10. Validity Proof (STARK Circuit)	32
9.3.1	10.1 Statement to Prove	32
9.3.2	10.2 AIR Constraints (Detailed)	33
9.3.3	10.3 Proof Generation	33
9.3.4	10.4 Proof Verification	34
9.4	11. Consensus: Proof of Work	34
9.4.1	11.1 Hash Function	34
9.4.2	11.2 Block Header (DAG)	35
9.4.3	11.3 Difficulty Adjustment (DAG)	35
9.4.4	11.4 Block Structure	36
9.4.5	11.5 Block Validation	36
9.5	12. Serialization	37
9.5.1	12.1 Canonical Encoding	37

9.5.2	12.2 Transaction Serialization	38
9.6	13. Network Protocol	38
9.6.1	13.1 Transport Layer	38
9.6.2	13.2 Message Types	38
9.6.3	13.3 Dandelion++ Parameters	39
9.7	14. State Management	39
9.7.1	14.1 Chain State	39
9.7.2	14.2 Database Schema	40
9.8	15. Wallet Operations	40
9.8.1	15.1 Key Generation	40
9.8.2	15.2 Scanning for Outputs	41
9.8.3	15.3 Creating Transactions	41
9.9	16. Economic Parameters	42
9.9.1	16.1 Supply Schedule	42
9.9.2	16.2 Fee Structure	42
9.9.3	16.3 Unit Definitions	42
10	Part III: Verification Criteria	43
10.1	17. Correctness Properties	43
10.1.1	17.1 Cryptographic Correctness	43
10.1.2	17.2 Protocol Correctness	43
10.1.3	17.3 Privacy Properties	44
10.2	18. Test Vectors	44
10.2.1	18.1 Hash Function Test Vectors	44
10.2.2	18.2 Commitment Test Vectors	45
10.2.3	18.3 Transaction Test Vectors	45
10.2.4	18.4 Consensus Test Vectors	46
10.3	19. Implementation Requirements	47
10.3.1	19.1 Mandatory Features	47
10.3.2	19.2 Performance Targets	47
10.3.3	19.3 Security Requirements	47
10.3.4	19.4 Code Quality Requirements	47
10.4	20. Formal Verification Targets	47
10.4.1	20.1 Properties to Formally Verify	47
10.4.2	20.2 Verification Tools	48
10.4.3	20.3 Audit Checklist	48
11	Part IV: Appendices	48
11.1	A. Reference Implementations	48
11.1.1	A.1 Polynomial Multiplication (NTT)	48
11.1.2	A.2 Lattice Commitment	49

11.1.3 A.3 STARK Prover Outline	50
11.2 B. Genesis Block	51
11.2.1 B.1 Genesis Parameters	51
11.2.2 B.2 Genesis Block Structure	52
11.2.3 B.3 Genesis Block Hash	52
11.3 C. Network Magic Numbers	53
11.4 D. Recommended Libraries	53
11.4.1 D.1 Rust Ecosystem	53
11.4.2 D.2 Alternative Language Implementations	53
11.5 E. Glossary	54
11.6 F. Document Metadata	54
11.7 G. Known Limitations and Trade-offs	54
11.7.1 G.1 Transaction Size	55
11.7.2 G.2 Proof Generation Time	55
11.7.3 G.3 Address Size	55
11.7.4 G.4 No Smart Contracts	56
11.8 H. Research Challenges and Future Directions	56
11.8.1 H.1 Core Research Challenge: Privacy-Preserving DAG Consensus	56
11.8.2 H.2 Alternative Scaling Approaches Considered	58
11.8.3 H.3 Mining Algorithm Analysis (Open Research)	60
11.8.4 H.4 Proof System Improvements	61
11.8.5 H.5 Layer 2 Scaling	62
11.8.6 H.6 Post-Quantum Signature Alternatives	62
12 End of Specification	63

2 Introduction

2.1 The Quantum Threat to Blockchain Security

The security of virtually all deployed blockchain systems rests on a single assumption: the computational hardness of the discrete logarithm problem in elliptic curve groups. Bitcoin, Ethereum, and nearly every major cryptocurrency use ECDSA or EdDSA signatures and ECDH key exchange—all of which would be broken by a sufficiently powerful quantum computer running Shor’s algorithm.

This is not a distant hypothetical. Quantum computers capable of breaking 256-bit elliptic curve cryptography could emerge within the next 10-20 years. More concerning is the “harvest now, decrypt later” threat: adversaries can record encrypted blockchain data today, store it, and decrypt it once quantum capabilities mature. For a blockchain designed to be permanent and immutable, this means that cryptographic choices made today have consequences that extend decades into the future.

2.2 The Privacy Imperative

Beyond quantum security, current blockchain systems suffer from a fundamental privacy deficiency: transaction transparency. While often marketed as a feature (auditability, trustlessness), transparent transactions create severe problems:

- **Fungibility:** When transaction history is visible, individual coins can be “tainted” by their history, breaking the interchangeability essential to functioning money
- **Commercial confidentiality:** Businesses cannot use transparent blockchains without exposing their operations to competitors
- **Personal safety:** Visible balances and transaction patterns create physical security risks for users
- **Surveillance:** Transaction graphs enable mass surveillance without user consent

Privacy is not a luxury feature—it is a prerequisite for a system that claims to be censorship-resistant. Without privacy, any sufficiently motivated adversary can identify and target users.

2.3 Our Approach

Quantum addresses both challenges simultaneously through careful selection of cryptographic primitives:

Component	Primitive	Quantum Security	Why This Choice
Commitments	Module-LWE Lattice	Yes	Binding + hiding + homomorphic under quantum-hard assumptions
Signatures	SPHINCS+	Yes	Stateless hash-based, NIST standardized
Key Exchange	ML-KEM (Kyber)	Yes	Lattice-based KEM, NIST standardized
ZK Proofs	STARKs	Yes	Hash-based, no trusted setup
Hashing	SHAKE256	Yes	128-bit post-quantum security with domain separation

The system enforces privacy-by-default with no opt-out mechanism. This is a deliberate design choice: optional privacy creates a smaller anonymity set and marks private transactions as “suspicious.” When all transactions are private, privacy is the norm rather than the exception.

3 Design Philosophy

This section explains the rationale behind key design decisions. Understanding *why* the specification makes certain choices is essential for implementers and auditors.

3.1 Why Post-Quantum Now?

The Risk: Cryptographically relevant quantum computers (CRQC) may be 10-20 years away, but blockchain data is permanent. An address created today may hold value for decades. If the underlying cryptography is broken, all historical transactions become vulnerable.

Harvest-Now-Decrypt-Later: Nation-state adversaries are already collecting encrypted traffic for future decryption. Blockchain transactions are public by design, making them trivially available for future cryptanalysis.

Migration is Hard: Upgrading cryptographic primitives in a decentralized system requires coordination across millions of users and thousands of implementations. It is far easier to build quantum-secure from the beginning than to migrate later under adversarial conditions.

NIST Standardization: The post-quantum primitives used in Quantum (ML-KEM, SPHINCS+) have completed NIST standardization. They are no longer experimental but rather ready for

production use.

3.2 Why Lattice-Based Commitments Over Pedersen?

Traditional privacy coins (Monero, Zcash) use Pedersen commitments based on elliptic curve discrete logarithm. These are elegant and efficient but broken by Shor's algorithm.

Lattice commitments based on Module-LWE provide:

- **Quantum resistance:** Security reduces to the hardness of finding short vectors in lattices, which resists known quantum attacks
- **Homomorphic property:** Like Pedersen commitments, lattice commitments can be added together, enabling balance verification without revealing values
- **Binding and hiding:** Computationally binding (cannot open to two values) and computationally hiding (reveals nothing about the committed value)

The trade-off is size: lattice commitments are larger (~3KB vs ~32 bytes). This is acceptable given the security requirements.

3.3 Why STARKs Over SNARKs?

SNARKs (used by Zcash) offer smaller proofs but require a *trusted setup*—a ceremony where participants generate parameters and must destroy their secret inputs. If any participant is compromised, they could create undetectable counterfeit coins.

STARKs require no trusted setup:

- **Transparency:** All parameters are publicly derived from hash functions
- **No toxic waste:** No secret information that could compromise the system
- **Post-quantum security:** Security relies only on collision-resistant hash functions
- **Scalability:** Prover time scales quasi-linearly with computation size

The trade-off is proof size: STARK proofs are larger (~100KB vs ~1KB for SNARKs). This specification accepts this trade-off because:

1. Trusted setup risk is unacceptable for a system designed to outlast its creators
2. Proof size can be reduced through future optimizations (recursive STARKs, proof aggregation)
3. Storage and bandwidth costs decrease over time

3.4 Why Privacy-by-Default With No Opt-Out?

Many privacy coins offer “selective transparency” or optional privacy. This is a mistake:

Anonymity sets shrink: If 10% of users opt into privacy, only that 10% provides cover for each other. If 100% use privacy, everyone benefits from the full user base as their anonymity set.

Privacy becomes suspicious: When privacy is optional, choosing it signals that you have “something to hide.” This invites enhanced scrutiny on private transactions.

Network effects: Privacy is a collective good. Individual opt-out degrades privacy for everyone else.

Quantum makes privacy mandatory and identical for all transactions. There is no “transparent mode” to implement, no configuration to accidentally disable privacy, and no way for users to harm the privacy of others.

3.5 Why DAG-Based Consensus (GhostDAG)?

Traditional blockchains process blocks sequentially, limiting throughput to ~10 TPS regardless of network capacity. DAG-based consensus (Directed Acyclic Graph) allows parallel block creation, fundamentally changing the scalability equation.

GhostDAG advantages:

- **Parallel blocks:** Multiple miners create valid blocks simultaneously; none are orphaned
- **No wasted work:** All valid proof-of-work contributes to consensus weight
- **Sub-second finality:** Practical for payment applications
- **Linear scaling:** Throughput increases with block rate, limited only by propagation delay

The research challenge: No existing DAG blockchain provides privacy. Kaspa achieves 10+ blocks/second but transactions are transparent. Quantum’s core research contribution is solving **privacy-preserving consensus over parallel block structures**:

1. **Nullifier ordering:** Preventing double-spend when blocks are created in parallel
2. **Anonymity set coherence:** Maintaining full output set as anonymity set across DAG branches
3. **STARK proofs over DAG:** Proving transaction validity with membership in a non-linear history
4. **Taint resistance:** Ensuring DAG structure doesn’t enable transaction graph analysis

This is novel cryptographic research. The specification defines the requirements; the implementation must solve these open problems.

3.6 Mining Algorithm (Open Research Question)

ASIC resistance is essential for decentralization. When mining requires specialized hardware:

- Manufacturing concentration creates geographic centralization

- Capital requirements exclude casual participants
- Supply chain dependencies create potential single points of failure

The Challenge: DAG consensus with 10-32 blocks/second requires fast hash verification. Traditional ASIC-resistant algorithms have trade-offs:

Algorithm	Hardware	Verification Speed	ASIC Resistance	DAG Suitability
RandomX	CPU	~2-3ms/hash	Excellent	Uncertain (designed for 2-min blocks)
kHeavyHash	GPU	<0.1ms/hash	Good (ASICs emerging)	Proven (Kaspa)
Blake3	Any	Very fast	Poor	Fast but centralizes

Current Position: This is an open research question. The algorithm must balance:

1. **Decentralization:** Accessible to commodity hardware
2. **Speed:** Fast enough for high block rate validation
3. **Security:** Resistant to ASIC/FPGA dominance
4. **Efficiency:** Reasonable power consumption

See Appendix H.3 for detailed analysis of alternatives.

3.7 Why These Specific Parameters?

Parameter	Value	Rationale
Security level	128-bit post-quantum	NIST Level 3, balanced security/efficiency
Field (STARKs)	Goldilocks ($2^{64} - 2^{32} + 1$)	Efficient 64-bit arithmetic, 2^{32} roots of unity
Ring modulus	$q = 8380417$	Matches CRYSTALS-Dilithium, enables NTT
Polynomial degree	$n = 256$	Standard lattice parameter, efficient NTT
Block rate	10-32 blocks/second	DAG consensus enables high parallelism
Confirmation time	< 10 seconds	Sub-second block time + DAG finality
Target throughput	1,000+ TPS	Competitive with Kaspa, plus privacy

Parameter	Value	Rationale
Supply cap	21,000,000	Familiar, deflationary, no tail emission debate

3.8 Why Bitcoin's Foundation? UTXO Model, Nakamoto Consensus, Fair Launch

Quantum builds on Bitcoin's battle-tested foundations rather than reinventing proven concepts.

UTXO Model: Bitcoin's Unspent Transaction Output model is superior for privacy:

- Transactions consume and create discrete outputs rather than modifying account balances
- No address reuse is natural (one-time stealth addresses extend this)
- Parallel transaction validation (outputs are independent)
- Simpler SPV proofs and state verification
- Proven secure for 15+ years with trillions of dollars at stake

Account-based models (Ethereum) leak information through balance changes and require complex state management. UTXO is the natural foundation for privacy.

Nakamoto Consensus: The longest-chain (most cumulative proof-of-work) rule provides:

- Objective fork choice without trusted coordinators
- Security proportional to honest hashrate (51% threshold)
- Permissionless participation—anyone can mine
- No stake-based plutocracy or validator cartels
- Proven game-theoretic security since 2009

GhostDAG extends Nakamoto consensus to parallel blocks while preserving these properties. All valid proof-of-work contributes to consensus weight; the PHANTOM ordering provides deterministic transaction ordering.

Fair Launch / No Premine: Quantum follows Bitcoin's fair distribution model:

- **No premine:** Zero coins allocated before public mining begins
- **No ICO/IEO:** No token sale to insiders or investors
- **No founder's reward:** No perpetual tax on mining rewards
- **No venture capital allocation:** No preferential access for investors
- **Day-one mining:** Anyone can participate from genesis block

This ensures that Quantum's value, if any emerges, accrues to those who secure the network through proof-of-work rather than to early insiders. Fair launch is essential for credibility as a

decentralized, censorship-resistant system.

Why This Matters: Projects that launch with premines, founder rewards, or investor allocations create conflicts of interest that undermine decentralization claims. Bitcoin's fair launch is a key reason it remains the most credibly neutral cryptocurrency. Quantum inherits this principle.

4 Project Status and Research Phases

4.1 Honest Assessment

This specification describes a system that **does not yet exist**. The core innovation—privacy-preserving proofs over DAG-based consensus—is an open research problem. No existing blockchain has solved this.

What is proven (low risk):

- Post-quantum cryptography: SPHINCS+, ML-KEM, lattice commitments are NIST-standardized
- STARKs: Production-ready (StarkNet, StarkEx), no trusted setup
- GhostDAG: Production-ready (Kaspa), achieves 10+ blocks/second
- Privacy transactions: Production-ready (Monero, Zcash) on sequential chains

What is unproven (research required):

- Privacy-preserving nullifier schemes for parallel blocks
- STARK proofs over DAG membership (not Merkle trees)
- Anonymity set coherence across DAG branches
- Transaction graph privacy in DAG structure
- Mining algorithm for high-frequency DAG (RandomX vs kHeavyHash vs hybrid)

What could fail:

- Proof sizes may be impractical (>1MB per transaction)
- Privacy analysis may reveal fundamental DAG information leakage
- Performance may not meet targets despite theoretical feasibility
- Unknown unknowns in the interaction of components

4.2 Research Phases and Milestones

This project follows a phased approach with explicit go/no-go decision points.

4.2.1 Phase 1: Specification and Formal Analysis (Current)

Deliverables:

- Complete specification (this document)
- Formal security model for privacy in DAG consensus
- Academic paper: "Privacy-Preserving Proofs Over Directed Acyclic Graphs"
- Peer review by independent cryptographers

Success Criteria:

- No fundamental flaws identified in security model
- Peer reviewers agree the approach is theoretically sound
- At least one viable solution identified for each open problem (H.1)

Failure Mode:

- Fundamental incompatibility discovered between DAG and privacy
- Action: Publish findings, pivot to sequential chain, or sunset project

4.2.2 Phase 2: Cryptographic Proof-of-Concept

Deliverables:

- Reference implementation of DAG-aware nullifier scheme
- STARK circuits for DAG membership proofs
- Benchmark suite for proof generation/verification
- Privacy analysis of DAG transaction graph

Success Criteria:

- Nullifier scheme prevents double-spend in parallel blocks (proven)
- STARK proof size < 500 KB per transaction
- Proof generation < 60 seconds on reference hardware
- No privacy leaks identified in DAG structure analysis

Failure Mode:

- Proof sizes exceed 1 MB (impractical for users)
- Proof generation exceeds 5 minutes (unusable UX)
- Privacy analysis reveals unavoidable information leakage
- Action: Publish findings, evaluate if trade-offs acceptable

4.2.3 Phase 3: Testnet Implementation

Deliverables:

- Full node implementation (Rust reference)
- Wallet implementation with proof generation

- Block explorer (privacy-preserving)
- Public testnet with community participation

Success Criteria:

- Sustained 1,000+ TPS on testnet
- < 10 second confirmation time
- Independent security audit passed
- No critical vulnerabilities in 6-month testnet operation

Failure Mode:

- Performance targets not met at scale
- Security audit reveals critical flaws
- Action: Fix issues and re-audit, or downgrade targets

4.2.4 Phase 4: Mainnet Launch

Prerequisites:

- All Phase 3 success criteria met
- Two independent security audits passed
- Formal verification of critical components
- Community governance established

Launch Criteria:

- Audit firms sign off on production readiness
- Bug bounty program running for 3+ months
- Economic model validated

4.3 What This Means for Stakeholders

4.3.1 For Researchers

This is a **genuine research project** with hard, unsolved problems:

- Novel contributions possible in each open problem area (H.1)
- Publication opportunities in top venues
- Clear problem statements with measurable success criteria
- Honest acknowledgment of what we don't know

We welcome collaboration. The specification is public domain (CC0).

4.3.2 For Investors

This is a **high-risk, high-reward research bet**:

Risk factors:

- Core research may not succeed
- Timeline is research-dependent, not calendar-driven
- No guarantee of mainnet launch

Mitigating factors:

- Clear milestones with go/no-go decisions
- Honest communication about progress
- Fallback options at each phase
- Novel positioning if successful

Upside:

- First mover in privacy + high-throughput L1
- Patent-free, open technology (CC0 license)
- Attracts top cryptographic talent

4.3.3 For the Community

We promise:

1. **Honesty:** We will communicate clearly about what works and what doesn't
2. **No vaporware:** We will not hype features that don't exist
3. **Open research:** All findings published, including failures
4. **Milestones:** Clear checkpoints where we evaluate progress
5. **Integrity:** If the research fails, we will say so rather than ship broken privacy

4.4 Current Status

Phase 1: Specification and Formal Analysis

Status: IN PROGRESS

Completed:

- Core specification (this document)
- Requirements definition (R1-R8)
- Design philosophy documented
- Open problems identified (H.1)

In Progress:

- Formal security model
- Academic paper draft
- Peer review outreach

Next Milestone:

- Complete formal analysis
 - Submit paper for peer review
-

5 Quantum: Quantum-Secure Privacy Blockchain

5.1 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.

6 IMMUTABLE REQUIREMENTS

6.1 ~~THIS SECTION IS IMMUTABLE~~

The following requirements define the core properties of the Quantum 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**.

6.2 R1. PRIVACY REQUIREMENTS

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.3 R2. SECURITY REQUIREMENTS

6.3.1 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

6.3.2 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.

6.3.3 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.

6.3.4 R2.4 Cryptographic Hiding [MANDATORY]

The commitment scheme MUST be computationally hiding.

A commitment MUST reveal no information about the committed value.

6.3.5 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.

6.3.6 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.

6.4 R3. DECENTRALIZATION REQUIREMENTS

6.4.1 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.

6.4.2 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

6.4.3 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.

6.4.4 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.

6.5 R4. INTEGRITY REQUIREMENTS

6.5.1 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.

6.5.2 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

6.5.3 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.

6.5.4 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.

6.6 R5. FUNCTIONAL REQUIREMENTS

6.6.1 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

6.6.2 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)

6.6.3 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
-

6.7 R6. PERFORMANCE REQUIREMENTS

6.7.1 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

6.7.2 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

6.7.3 R6.3 Network [MANDATORY]

Block rate: 10-32 blocks per second (DAG consensus)

Transaction throughput: 1,000 TPS sustained (see R8.2)

Confirmation time: < 10 seconds for high-confidence finality

Block propagation: < 1 second to 90% of nodes

6.8 R7. NON-REQUIREMENTS (Explicitly Excluded)

6.8.1 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)

6.8.2 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)
-

6.9 R8. SCALABILITY REQUIREMENTS

6.9.1 R8.1 DAG-Native Architecture [MANDATORY]

The protocol MUST use DAG-based consensus (GhostDAG or equivalent).

The protocol MUST NOT use sequential single-chain block ordering.

Parallel block creation MUST be supported natively.

All valid blocks MUST contribute to consensus (no orphans discarded).

Research Challenge: Privacy-preserving proofs over DAG structures.

This is a core research goal, not a deferrable optimization.

6.9.2 R8.2 L1 Throughput Targets [MANDATORY]

Block rate: 10 blocks per second (target: 32 blocks/second)

Transaction throughput: 1,000 TPS sustained on L1

Peak capacity: 5,000 TPS burst without failure

Confirmation time: < 10 seconds for high-confidence finality

Comparative baseline: Match or exceed Kaspa's throughput
while adding privacy and quantum security.

6.9.3 R8.3 Parallel Processing [MANDATORY]

Transaction validation MUST support parallel execution.

STARK proof verification MUST be parallelizable across blocks.

Nullifier checking MUST scale with DAG width.

Merkle tree updates MUST handle concurrent block insertions.

6.9.4 R8.4 Privacy-Preserving DAG Consensus [MANDATORY]

The DAG structure MUST NOT weaken privacy guarantees:

- Anonymity set: MUST remain the full output set across all DAG branches
- Transaction graph: DAG parent references MUST NOT enable taint analysis
- Nullifier ordering: MUST prevent double-spend across parallel blocks
- Proof validity: STARKs MUST prove membership across DAG structure

This is the core research contribution of Quantum:

Solving privacy-preserving consensus over parallel block structures.

6.9.5 R8.5 Horizontal Scaling [MANDATORY]

Node operations MUST scale horizontally:

- Validation load MUST be distributable across CPU cores
- Block propagation MUST use DAG-aware protocols
- State synchronization MUST handle DAG branch merging
- Mempool MUST support high-throughput transaction ingestion (10K+ TPS)

6.9.6 R8.6 Layer 2 Compatibility [MANDATORY]

The protocol MUST support L2 scaling in addition to L1:

- Payment channels: Off-chain transactions with on-chain settlement
- Validity rollups: Batched proofs for higher throughput
- State channels: Generalized off-chain state transitions

L2 solutions provide: 100,000+ TPS for specific use cases.

L1 provides: Base layer security, finality, and censorship resistance.

6.10 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
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: Mining algorithm 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

Requirement	Category	Verification Method
R8.1	Scalability	Architecture review: DAG consensus implemented
R8.2	Scalability	Load testing: 1000+ TPS sustained
R8.3	Scalability	Benchmark: parallel validation scaling
R8.4	Scalability	Formal proof: privacy preserved over DAG
R8.5	Scalability	Integration test: horizontal node scaling
R8.6	Scalability	Design review: L2 compatibility

6.11 Immutability Declaration

These requirements constitute the immutable core of the Quantum specification.

SHA-256 hash of requirements section (R1-R8) :

Computed over the IMMUTABLE REQUIREMENTS section of this document.

Draft v1.0 hash: [to be computed when specification is finalized]

Note: Hash will be updated when specification is finalized.

Any modification to R1-R8 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.

7 END OF IMMUTABLE REQUIREMENTS

8 Part I: Cryptographic Foundation

Part I defines the cryptographic building blocks that underpin Quantum. Each component is selected for its quantum resistance and well-understood security properties. Together, these

primitives enable private transactions with amounts hidden, senders unlinkable, and receivers unidentifiable—all without trusted setup or quantum-vulnerable assumptions.

The components build on each other: hash functions provide the foundation for domain-separated operations; polynomial rings enable efficient lattice arithmetic; commitments hide values while preserving verifiability; signatures authorize spending; key encapsulation enables secure one-time addressing; and STARKs prove transaction validity without revealing private data.

8.1 1. Notation and Conventions

8.1.1 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

8.1.2 1.2 Security Parameter

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

8.1.3 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)

8.2 2. Hash Functions

Hash functions are the most fundamental cryptographic primitive in Quantum. Unlike elliptic curve operations, hash functions remain secure against quantum computers—Grover’s algorithm provides only a quadratic speedup, which is addressed by using 256-bit security parameters that yield 128-bit post-quantum security.

Why SHAKE256? We use SHAKE256 (SHA-3 family) as the universal hash function because:

- **Extendable output:** Can produce arbitrary-length output, simplifying API design
- **Domain separation:** Different hash instances are created by prefixing distinct domain tags
- **No length extension attacks:** Unlike SHA-2, SHA-3’s sponge construction prevents length extension
- **NIST standardized:** FIPS 202 provides implementation confidence

8.2.1 2.1 Primary Hash Function: SHAKE256

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

$$\begin{aligned} H &: \{0,1\}^* \times \mathbb{N} \rightarrow \{0,1\}^* \\ H(m, l) &= \text{SHAKE256}(m, l) \end{aligned}$$

Where l is the output length in bits.

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

$$H_{\text{domain}}(x) = H(\text{encode}("Quantum-v1.") \parallel \text{domain}) \parallel x, \text{output_len})$$

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

8.2.2 2.2 Defined Hash Instances

Instance	Domain Tag	Output Length	Usage
$H_{\text{commitment}}$	“commitment”	512 bits	Commitment randomness
$H_{\text{nullifier}}$	“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
$H_{\text{challenge}}$	“challenge”	512 bits	Fiat-Shamir challenges
H_{pow}	“pow”	256 bits	Proof of work

8.2.3 2.3 Hash-to-Field

$\text{HashToField}(m, q, k)$:

```
Input: message m, modulus q, count k
Output: k elements in _q
```

```
1.   = 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})
```

8.3 3. Lattice-Based Commitments

Commitments are how Quantum hides transaction amounts while proving they balance. A commitment scheme allows you to “commit” to a value (like a transaction amount) in a way that hides the value but binds you to it—you cannot later claim you committed to a different value.

Why lattice-based? Traditional Pedersen commitments use elliptic curve points, which are broken by quantum computers. Lattice-based commitments achieve the same functionality under quantum-hard assumptions. The security reduces to the Module Learning With Errors (Module-LWE) problem: given noisy linear equations over polynomial rings, recover the secret. This problem resists all known classical and quantum algorithms.

The key property: Commitments are *additively homomorphic*. If $C(a)$ commits to value a and $C(b)$ commits to value b , then $C(a) + C(b) = C(a+b)$. This allows us to verify that inputs equal outputs plus fee without decrypting any amounts.

8.3.1 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.

8.3.2 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 $_q$

8.3.3 3.3 Commitment Scheme

Key Generation (public parameters):

Setup(1^λ):

1. $A \leftarrow R_{q^k}$ // 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 $c == A \cdot r + v \cdot e_1$
3. Return accept/reject

Properties:

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

8.3.4 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
```

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

8.4.1 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

8.4.2 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

8.4.3 4.3 Security

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

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

8.5.1 5.1 Parameters

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

```
n = 256          // Polynomial degree
k = 4           // Module rank
q = 3329         // Modulus
1 = 2           // Secret key noise
2 = 2           // Ciphertext noise
```

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

8.5.2 5.2 API

Kyber_KeyGen():
 Output: (pk , sk)

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

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

8.6 6. Zero-Knowledge Proofs: STARKs

Zero-knowledge proofs are the cryptographic core of Quantum's privacy guarantees. They allow a prover to convince a verifier that a statement is true (e.g., "this transaction is valid and balanced") without revealing *anything* beyond that fact—not the amounts, not which outputs were spent, not the sender or receiver.

Why STARKs specifically? The choice of STARK over other ZK systems (SNARKs, Bulletproofs) is driven by two requirements:

1. **No trusted setup:** SNARKs require a ceremony where participants generate parameters and destroy secrets. If anyone cheats, they can forge proofs forever. STARKs derive all parameters from public randomness, eliminating this catastrophic risk.
2. **Quantum security:** SNARKs typically rely on elliptic curve pairings or discrete log assumptions, both broken by quantum computers. STARKs rely only on collision-resistant hash functions, which remain secure.

The trade-off is proof size: STARK proofs are larger ($\sim 100\text{KB}$ vs $\sim 1\text{KB}$). For a system designed to remain secure for decades, this is acceptable.

8.6.1 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

8.6.2 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

8.6.3 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

8.6.4 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)

8.6.5 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

8.7 7. Merkle Trees (Quantum-Secure)

8.7.1 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)

```

8.7.2 7.2 Tree Parameters

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

Node size: 32 bytes

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

8.7.3 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
    }
}

```

9 Part II: Protocol Specification

Part II specifies how the cryptographic primitives from Part I combine into a working blockchain protocol. This includes the account model, transaction lifecycle, consensus mechanism, and network layer.

How a Transaction Works (High-Level):

1. **Creating outputs:** The sender generates commitments to amounts, encrypts output data to recipients using their view keys, and creates new “notes” (outputs)
2. **Spending inputs:** To spend previous outputs, the sender reveals nullifiers (deterministic tags that mark outputs as spent) without revealing *which* outputs they correspond to
3. **Proving validity:** The sender generates a STARK proof that the transaction is balanced (inputs = outputs + fee), all spent outputs existed, and all nullifiers are correctly formed
4. **Authorization:** The sender signs the transaction with their spend key
5. **Propagation:** The transaction propagates through the network using Dandelion++ to hide the sender’s IP address
6. **Inclusion:** Miners validate the proof and signature, check nullifiers aren’t already spent, and include the transaction in a block

This design provides complete privacy: observers see only nullifiers (unlinkable to outputs) and new commitments (hiding amounts and recipients).

9.1 8. Account and Address System

9.1.1 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)
```

9.1.2 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
```

9.1.3 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
-

9.2 9. Transaction Structure

9.2.1 9.1 Output (Note)

```
struct Output {  
    // Public (stored on-chain)  
    commitment: LatticeCommitment,      // k x 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
}

```

9.2.2 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

9.2.3 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]>,
}

```

9.2.4 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

9.3 10. Validity Proof (STARK Circuit)

9.3.1 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: $i: \text{MerkleVerify}(\text{input_commitments}[i], \text{input_positions}[i], \text{input_merkle_root})$
6. No overflow: $\sum \text{input_values} < 2^{64}$ (prevent wrap-around)

9.3.2 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
```

9.3.3 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

9.3.4 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
```

9.4 11. Consensus: Proof of Work

Why Proof of Work? In a privacy-focused system, proof of stake creates problematic dynamics: stake is visible on-chain (compromising privacy) or requires trusted infrastructure to verify (compromising decentralization). Proof of work provides permissionless participation without revealing participant identity or holdings.

Why ASIC-resistant? When mining centralizes around specialized hardware manufacturers, the network becomes vulnerable to supply chain attacks, geographic concentration, and regulatory capture. The mining algorithm must keep mining accessible to commodity hardware (CPUs or GPUs).

9.4.1 11.1 Hash Function

The mining algorithm is an **open research question** (see Design Philosophy and Appendix H.3). The general structure:

PowHash(header):

1. classical_hash = ASIC_RESISTANT_HASH(header) // RandomX, kHeavyHash, or TBD
2. quantum_hash = H_pow(classical_hash) // SHAKE256 for quantum security
3. Return quantum_hash

Candidates:

- **RandomX**: Maximum ASIC resistance (CPU), but verification speed may limit DAG block rate
- **kHeavyHash**: Proven for DAG (Kaspa), GPU-friendly, ASICs exist but GPUs remain viable
- **Hybrid**: Custom algorithm optimized for privacy-DAG requirements

Rationale: The classical hash provides ASIC resistance. The additional SHAKE256 hash ensures quantum security of the final output. Final algorithm choice requires empirical testing at target block rates.

9.4.2 11.2 Block Header (DAG)

```
struct BlockHeader {
    version: u32,                                // Protocol version
    parent_hashes: Vec<[u8; 32]>,                // Hashes of parent blocks (DAG structure)
    num_parents: u8,                               // Number of parents (typically 1-64)
    blue_score: u64,                               // GhostDAG blue score
    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 (milliseconds for DAG)
    difficulty: [u8; 32],                          // Target difficulty (256-bit)
    nonce: u64,                                   // PoW nonce

    // Variable size due to parent_hashes
    // Minimum: 140 bytes (1 parent)
    // Typical: 300-500 bytes (5-10 parents)
}

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
    }
}
```

9.4.3 11.3 Difficulty Adjustment (DAG)

DAG-aware difficulty adjustment based on block rate:

```

AdjustDifficulty(dag_state):
    EPOCH_DURATION = 100      // Epoch duration in seconds
    TARGET_RATE = 10          // Target blocks per second
    EXPECTED_BLOCKS = EPOCH_DURATION * TARGET_RATE // 1000 blocks

    // Count blocks in epoch window
    epoch_start = current_time - EPOCH_DURATION
    blocks_in_epoch = count_blocks_since(epoch_start)

    // Calculate adjustment ratio
    ratio = blocks_in_epoch / EXPECTED_BLOCKS

    // Smooth adjustment (DAG requires more stability)
    adjustment = clamp(ratio, 0.75, 1.25) // Max 25% change per epoch

    new_difficulty = previous_difficulty * adjustment
    Return new_difficulty

```

Note: DAG requires smoother difficulty adjustments than sequential chains because high block rates amplify oscillations.

9.4.4 11.4 Block Structure

```

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

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

```

9.4.5 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
-

9.5 12. Serialization

9.5.1 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 $k=4$

9.5.2 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)
```

9.6 13. Network Protocol

9.6.1 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

9.6.2 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,
}

```

9.6.3 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

9.7 14. State Management

9.7.1 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  
}
```

9.7.2 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
```

9.8 15. Wallet Operations

9.8.1 15.1 Key Generation

```
GenerateWallet():  
1. entropy = CSPRNG(256 bits)
```

```

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

```

9.8.2 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)

```

9.8.3 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 }
```

9.9 16. Economic Parameters

9.9.1 16.1 Supply Schedule

Distribution: Fair launch (no premine, no ICO, no founder's reward)

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)

9.9.2 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 $176,000 \times 1 \text{ sat} = 0.00176 \text{ ZKP}$

9.9.3 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

10 Part III: Verification Criteria

10.1 17. Correctness Properties

10.1.1 17.1 Cryptographic Correctness

Property 1: Commitment Binding

For all PPT adversaries A:

$$\Pr[\text{VerifyOpening}(\text{pp}, \text{c}, \text{v}_1, \text{r}_1) \wedge \text{VerifyOpening}(\text{pp}, \text{c}, \text{v}_2, \text{r}_2) \mid \text{v}_1 = \text{v}_2] < \text{negl}()$$

Property 2: Commitment Hiding

For all PPT adversaries A, all v_0, v_1 :

$$|\Pr[A(\text{Commit}(v_0)) = 1] - \Pr[A(\text{Commit}(v_1)) = 1]| < \text{negl}()$$

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:

$$\{\text{Prove}(\text{witness})\}_{\text{witness}} \xrightarrow{c} \{S(\text{statement})\}_{\text{statement}}$$

10.1.2 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

10.1.3 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

10.2 18. Test Vectors

10.2.1 18.1 Hash Function Test Vectors

Test 1: H_nullifier

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

Test 2: H_merkle leaf hash

Domain tag: "Quantum-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)
```

10.2.2 18.2 Commitment Test Vectors

Test 5: Zero commitment

Input: $v = 0$, r = zero polynomial vector
Output: $c = A \cdot 0 + 0 \cdot e_1 = 0$ (zero vector in R_q^k)
Serialized: $0x00 \times 3072$ (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 r_1, r_2 be random polynomial vectors with coefficients in $[-2, 2]$
 $\text{Commit}(100, r_1) + \text{Commit}(50, r_2) = \text{Commit}(150, r_1+r_2)$
Verification: Extract value component, verify $100 + 50 = 150 \bmod q$

Test 8: Binding test (negative)

Property: Cannot find $(v_1, r_1), (v_2, r_2)$ such that $\text{Commit}(v_1, r_1) = \text{Commit}(v_2, r_2)$
Test: Generate 10^6 random commitments, verify no collisions

10.2.3 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: $0x1a2b3c4d\dots$ (32 bytes)

Output:

- Value: 999999 satoshi
- Recipient: test address

Fee: 1 satoshi

Expected nullifier: $H_{\text{nullifier}}(nk \parallel \text{commitment} \parallel 0) = 0x4f8a2c\dots$
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)

10.2.4 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 (DAG)

Given: 1000 blocks in epoch window

Target block rate: 10 blocks per second

Epoch duration: 100 seconds (expected 1000 blocks)

If actual blocks in epoch = 1200 (too fast):

New difficulty = old difficulty \times (1200/1000) = old difficulty \times 1.2

If actual blocks in epoch = 800 (too slow):

New difficulty = old difficulty × (800/1000) = old difficulty × 0.8

Clamped to range [0.75, 1.25] per adjustment (DAG requires smoother adjustments)

Test 14: DAG block ordering (GhostDAG blue score)

Block A: blue score = 1000, parents = [genesis]

Block B: blue score = 1050, parents = [A, C]

Block C: blue score = 1020, parents = [genesis]

Canonical ordering: Blocks ordered by blue score (GhostDAG algorithm)

All valid blocks included in DAG (none orphaned)

10.3 19. Implementation Requirements

10.3.1 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

10.3.2 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

10.3.3 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

10.3.4 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
-

10.4 20. Formal Verification Targets

10.4.1 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

10.4.2 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

10.4.3 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
-

11 Part IV: Appendices

11.1 A. Reference Implementations

11.1.1 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
}

```

11.1.2 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;
    }

    ntt_inverse(&a_ntt)
}

```

11.1.3 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,
}
}

}

```

11.2 B. Genesis Block

11.2.1 B.1 Genesis Parameters

Version: 1

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

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

Previous hash: 0x00000000000000000000000000000000
Nonce: 0 (genesis block has special validation rules)

Transactions: None (empty block)

Genesis message (encoded in first 80 bytes):

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

11.2.2 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
```

}

```
// 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
```

11.2.3 B.3 Genesis Block Hash

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

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

11.3 C. Network Magic Numbers

```
Mainnet magic: 0x5A4B5031 ("ZKP1" in ASCII)  
Testnet magic: 0x5A4B5430 ("ZKT0" in ASCII)  
Regtest magic: 0x5A4B5230 ("ZKRO" in ASCII)
```

Protocol version: 1

Minimum supported version: 1

11.4 D. Recommended Libraries

11.4.1 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

11.4.2 D.2 Alternative Language Implementations

Go:

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

C/C++:

- liboqs: Post-quantum algorithms

- libstark: STARK implementation
-

11.5 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

11.6 F. Document Metadata

Title: Quantum 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

11.7 G. Known Limitations and Trade-offs

This section documents known limitations and design trade-offs:

11.7.1 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

11.7.2 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)

11.7.3 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

11.7.4 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
-

11.8 H. Research Challenges and Future Directions

This section documents the core research challenges that define Quantum, as well as potential future improvements.

11.8.1 H.1 Core Research Challenge: Privacy-Preserving DAG Consensus

This is the defining research contribution of Quantum. No existing blockchain combines DAG-based consensus with full transaction privacy. Solving this enables a system that is simultaneously:

- Fast (1,000+ TPS via parallel blocks)
- Private (full anonymity set, hidden amounts)
- Quantum-secure (post-quantum cryptography throughout)

The problem statement:

Given: GhostDAG consensus (parallel block creation, no orphans)

Given: STARK-based transaction proofs (quantum-secure, no trusted setup)

Given: Privacy requirements (hidden amounts, unlinkable outputs)

Design: A protocol where privacy guarantees hold despite parallel block ordering

Open Research Problems (must be solved for conformant implementation):

1. Parallel Nullifier Commitment

Problem: Nullifiers prevent double-spend, but two parallel blocks might both try to spend the same output.

Approaches under investigation:

- Nullifier epochs: Commit nullifiers in batches at DAG "checkpoints"
- Optimistic execution: Accept parallel spends, resolve in ordering phase
- DAG-aware nullifier sets: Nullifier validity depends on DAG ancestry

Required property: No double-spend possible, even with adversarial miners

2. Anonymity Set Coherence

Problem: "All outputs" as anonymity set assumes linear history.
DAG has multiple valid topological orderings.

Approaches under investigation:

- Canonical ordering: Define deterministic topological sort of DAG
- Epoch-based sets: Anonymity set = outputs confirmed before epoch boundary
- Branch-aware proofs: Prove membership in "any valid ordering"

Required property: Anonymity set size never smaller than sequential chain

3. STARK Proofs Over DAG Structure

Problem: Proving transaction validity requires proving output membership.
DAG membership is more complex than Merkle tree membership.

Approaches under investigation:

- DAG Merkle structures: Generalized authenticated data structures for DAGs
- Epoch snapshots: Prove against periodic linear snapshots
- Recursive proofs: Prove validity relative to parent blocks' proofs

Required property: Proof size and verification time remain practical

4. Transaction Graph Privacy

Problem: DAG's explicit parent-child structure might enable taint analysis even when amounts and addresses are hidden.

Approaches under investigation:

- Decoy references: Transactions reference random additional parents

- Delayed inclusion: Random delay before transaction enters DAG
- Reference obfuscation: Parents selected to minimize information leakage

Required property: DAG structure reveals no more than sequential chain

Research Milestones:

- M1: Formal security model for privacy in DAG consensus
- M2: Nullifier scheme with proof of double-spend prevention
- M3: STARK circuit for DAG membership proofs
- M4: Privacy proof under DAG adversary model
- M5: Reference implementation passing all security tests
- M6: Independent security audit

11.8.2 H.2 Alternative Scaling Approaches Considered

Before selecting GhostDAG, several alternative scaling approaches were evaluated:

11.8.2.1 Ouroboros Leios (Cardano) [Ouroboros Leios](#) achieves ~1,000 TPS through pipelining with three block types:

Architecture:

- | | |
|-----------------------|-------------------------------------------|
| Input Blocks (IBs) | → Collect user transactions |
| Endorser Blocks (EBs) | → Committee validates transactions |
| Ranking Blocks (RBs) | → Finalize ordering (20-second intervals) |

Results: 30–50x throughput improvement over base Praos

Why not adopted:

1. Proof of Stake conflicts with privacy
 - Stake is visible on-chain, compromising holder privacy
 - Stake-weighted selection reveals economic information
 - Committee membership leaks participation data
2. Committee-based endorsement adds trust assumptions
 - Requires honest committee majority
 - Committee selection must be unpredictable
 - Conflicts with Quantum's trustless design goal
3. Complexity
 - Three block types vs. GhostDAG's uniform blocks
 - More attack surface for privacy analysis

11.8.2.2 Sharding (Ethereum-style) Sharding splits the network into parallel chains processing different transactions.

Why not adopted:

1. Cross-shard privacy is unsolved
 - Transactions crossing shards leak information
 - Anonymity set fragments across shards
 - Atomic cross-shard private transactions are an open problem
2. Shard assignment reveals information
 - Which shard holds your outputs?
 - Shard-specific scanning leaks user locations
3. Complexity
 - Shard coordination protocols
 - Data availability across shards
 - Reorganization handling across shards

11.8.2.3 Parallel Execution (Solana-style) Solana achieves high throughput through parallel transaction execution with declared state access.

Why not adopted:

1. State access declarations leak privacy
 - Transactions must declare which accounts they touch
 - This reveals transaction graph information
 - Conflicts with unlinkability requirements
2. Hardware requirements harm decentralization
 - Solana requires high-end hardware
 - Conflicts with commodity hardware goal (R3.3)
3. Not ASIC-resistant
 - Validator hardware becomes specialized
 - Centralization pressure from hardware requirements

11.8.2.4 GhostDAG (Selected) Why GhostDAG fits Quantum:

1. Pure Proof of Work
 - No stake visibility (privacy preserved)
 - No committees (trustless)
 - ASIC-resistant mining (decentralized)

2. Uniform block structure
 - All blocks are equal (no special roles)
 - Simpler privacy analysis
 - Cleaner anonymity set definition

3. Proven in production
 - Kaspa demonstrates 10+ blocks/second
 - Well-understood security properties
 - Active research community

4. Privacy-compatible (with research)
 - Parallel blocks don't inherently leak more than sequential
 - Research challenges are tractable (see H.1)
 - No fundamental conflicts with privacy requirements

Summary:

Approach	Throughput	Privacy Compatible	Trust Model	Selected
Leios (PoS)	~1,000 TPS	No (stake visible)	Committee	No
Sharding	~10,000 TPS	No (cross-shard leaks)	Per-shard	No
Solana-style	~65,000 TPS	No (state declarations)	Validators	No
GhostDAG (PoW)	~1,000+ TPS	Yes (with research)	Pure PoW	Yes

11.8.3 H.3 Mining Algorithm Analysis (Open Research)

The mining algorithm choice is an **open research question** for DAG-based privacy blockchains. The requirements are more demanding than traditional sequential chains.

DAG-Specific Requirements:

1. **Fast verification:** 10-32 blocks/second means verifying potentially 320+ block headers per 10-second window
2. **Low latency:** Block propagation must be fast to prevent DAG fragmentation
3. **ASIC resistance:** Decentralization is critical for censorship resistance
4. **Post-quantum consideration:** Hash-based PoW is inherently quantum-resistant

Algorithm Comparison:

	RandomX	kHeavyHash	Blake3-256
Hardware	CPU	GPU	Any

Verification	~2-3ms/hash	<0.1ms/hash	<0.01ms/hash
ASIC resistance	Excellent	Good*	Poor
Decentralization	Maximum	Good	Low (ASIC farms)
Power efficiency	Lower	Higher	Highest
Production use	Monero (5+ yr)	Kaspa (2+ yr)	Various
DAG suitability	Uncertain	Proven	Unproven

* kHeavyHash ASICs emerged in 2023, but GPU mining remains viable

Analysis:

1. **RandomX**: Designed for Monero's 2-minute blocks. Verification at ~2-3ms/hash may create bottlenecks at high block rates. CPU mining maximizes decentralization but may not scale to DAG requirements. Needs benchmarking.
2. **kHeavyHash**: Proven for DAG consensus (Kaspa). GPU-friendly provides good decentralization (GPUs are widely owned). ASICs exist but haven't eliminated GPU mining. Most practical choice based on Kaspa's success.
3. **Blake3**: Very fast but provides no ASIC resistance. Would lead to mining centralization.
4. **Hybrid approach**: Possible to combine algorithms (e.g., memory-hard component + fast verification). Requires custom research.

Research Required:

- Benchmark RandomX verification at 10-32 blocks/second
- Analyze kHeavyHash ASIC impact on decentralization
- Evaluate hybrid approaches for privacy-DAG specific requirements
- Consider optical PoW or other novel approaches

Current Position: No final decision. The specification requires ASIC resistance and DAG compatibility. Implementation phase must resolve this with empirical testing.

11.8.4 H.4 Proof System Improvements

Current: STARKs with ~100KB proofs, 30-120 second generation

Research Directions:

1. Recursive STARKs
 - Prove verification of other proofs
 - Enables proof aggregation: one proof for entire block
 - Could reduce per-transaction overhead by 10-100x
2. Hardware Acceleration

- GPU-based STARK provers
 - FPGA implementations for mobile/embedded
 - Could reduce proof generation to <10 seconds
3. Folding Schemes (Nova, SuperNova)
- Incrementally verifiable computation
 - Could enable real-time proof updates
 - Active research area, not yet production-ready

4. Hybrid STARK-SNARK
- Use STARKs for quantum security
 - Wrap in SNARK for smaller on-chain footprint
 - Maintains no-trusted-setup property

11.8.5 H.5 Layer 2 Scaling

Rather than modifying L1 consensus, scaling could be achieved via Layer 2:

Payment Channels:

- Off-chain transactions between parties
- Only settlement on L1
- Challenge: Privacy-preserving channel design

Rollups:

- Batch transactions off-chain
- Single validity proof on L1
- Challenge: Maintaining privacy across rollup boundary

State Channels:

- Generalized off-chain state updates
- Requires ZK-compatible state transition proofs

Note: L2 solutions are complementary to, not replacements for, L1 improvements. Both can be pursued independently.

11.8.6 H.6 Post-Quantum Signature Alternatives

Current: SPHINCS+-256f (49KB signatures)

Potential Alternatives:

FALCON (NIST standardized):

- Signature size: ~1.3 KB (40x smaller)
- Trade-off: Requires careful implementation (floating point)

- Risk: More complex, potential side-channel vulnerabilities

SLH-DSA (SPHINCS+ successor):

- NIST's final standard (2024)
- Similar to SPHINCS+ with minor improvements
- Migration path: Parameter update, not architectural change

Dilithium (ML-DSA):

- Signature size: ~2.4 KB
- Lattice-based (same assumption family as commitments)
- Trade-off: Larger than FALCON, simpler than SPHINCS+

v1.0 Rationale: SPHINCS+ is the most conservative choice—pure hash-based security with no algebraic structure to attack. Size is acceptable for a chain prioritizing security over throughput. Future versions may offer signature algorithm flexibility if smaller alternatives prove equally secure in production.

12 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).