QRDX Protocol Whitepaper v3.2

QRDX Foundation Research Team

February 16, 2026

Contents

1 QRDX Protocol Whitepaper v3.2

1.1 Quantum-Resistant Layer-0 Protocol: Integrated Exchange, Cross-Chain Oracle & Asset Shielding

Version: 3.2
Last Updated: February 16, 2026

Authors: QRDX Foundation Research Team
Contact: research@mail.qrdx.org

1.2 Abstract

The emergence of quantum computing threatens the cryptographic foundations of blockchain technology. QRDX addresses this existential challenge by introducing the first quantum-resistant **Layer-0 blockchain protocol** with a natively integrated decentralized exchange engine, cross-chain oracle infrastructure, and asset shielding capabilities. Secured by NIST-standardized post-quantum cryptographic algorithms and enforcing post-quantum (PQ) identity at every layer—from bootstrap node discovery to validator consensus to user wallets—QRDX provides an end-to-end quantum-resistant ecosystem.

QRDX operates as a **Layer-0 (L0) substrate**: nodes natively embed light clients and chain adapters for external blockchains (Ethereum, Bitcoin, Solana, etc.), enabling direct cross-chain transaction execution and oracle data provision within a single secure protocol. The integrated exchange engine is built into the blockchain itself—not as an application on top of it—allowing any participant with sufficient staked or subsidized QRDX to spin up liquidity pools permissionlessly, following a model comparable to Hyperliquid's on-chain order book but enhanced with concentrated liquidity AMM mechanics.

The protocol enforces **PQ-native identity** throughout: validators must use Dilithium keypairs exclusively, bootstrap nodes are discovered via a PQ `@`-schema addressing format, and all wallets—including prefunded wallets controlled by a master PQ wallet and PQ multisignature wallets using threshold Dilithium schemes—operate with quantum-resistant cryptography from genesis.

QRDX features comprehensive block height recording for all bridged chains and includes the **Doomsday Protocol** at doomsday.qrdx.org—an automated circuit breaker that immediately halts classical-to-quantum asset bridging if a quantum computer demonstrates the ability to break ECDSA cryptography, while still allowing users to unshield their quantum-protected assets back to classical chains. This whitepaper presents the technical architecture, economic model, and security guarantees of the QRDX ecosystem.

1.3 Table of Contents

1.4 1. Introduction

1.4.1 1.1 Background

Blockchain technology relies on cryptographic primitives that are vulnerable to quantum computing attacks. Shor's algorithm, running on a sufficiently powerful quantum computer, can break elliptic curve cryptography (ECC) and RSA encryption—the backbone of current blockchain security. With major technology companies and governments investing billions in quantum computing research, the timeline for practical quantum attacks is shrinking.

Beyond the quantum threat, the current blockchain landscape suffers from fragmentation: exchanges run as separate applications on top of chains, cross-chain communication requires fragile external bridges, and oracle networks operate as disconnected middleware. Each seam introduces attack surface. QRDX eliminates these seams by integrating exchange, oracle, and cross-chain capabilities directly into the Layer-0 protocol.

1.4.2 1.2 The QRDX Solution

QRDX introduces a comprehensive quantum-resistant ecosystem consisting of:

- **QRDX Layer-0 Protocol**: A substrate blockchain where nodes natively embed light clients for external chains (Ethereum, Bitcoin, Solana, etc.), acting as cross-chain oracles within a single secure protocol
- **Integrated Exchange Engine**: A protocol-native exchange built into the blockchain itself—not deployed on top of it—combining concentrated liquidity AMM mechanics with an on-chain order book, where any participant can spin up pools permissionlessly with sufficient staked or subsidized QRDX
- **PQ-Native Identity**: Mandatory post-quantum identity at every layer: Dilithium-only validators, PQ `@`-schema node addressing for bootstrap, PQ multisignature wallets, and pre-funded wallets controlled by master PQ keypairs
- **Asset Shielding**: Native functionality to convert classical blockchain assets into quantum-resistant equivalents
- **qRC20 Standard**: A quantum-resistant token standard compatible with existing DeFi infrastructure

1.4.3 1.3 Key Innovations

1. **Layer-0 Cross-Chain Substrate**: Nodes embed chain adapters and light clients, enabling native cross-chain transaction execution and oracle data provision without external middleware

2. **Integrated Exchange Engine**: The exchange is the chain—protocol-native trading with concentrated liquidity, on-chain order books, and user-deployable pools backed by stake or subsidy
3. **PQ Multisignatures**: Threshold Dilithium signature scheme enabling m-of-n quantum-resistant multisig wallets at the protocol level
4. **Prefunded Wallet Hierarchies**: Master PQ wallet controls multiple prefunded sub-wallets for operational separation, institutional custody, and automated treasury management
5. **PQ Node Identity & Bootstrap**: `@`-schema addressing (e.g., `dilithium3@<pubkey_hash>@<host>:<port>`) for PQ-authenticated node discovery and bootstrap
6. **Validator PQ Enforcement**: Validators must exclusively use Dilithium keypairs—no classical fallback permitted at the consensus layer
7. **Cross-Chain Shielding**: Trustless mechanism to convert BTC, ETH, and other assets to quantum-resistant versions (qBTC, qETH, etc.)
8. **Block Height Recording**: Comprehensive tracking of all bridged chain states for temporal consistency and fraud detection
9. **Doomsday Protocol**: Automated quantum threat detection at doomsday.qrdx.org that immediately halts classical→quantum bridging if ECDSA is broken
10. **Hooks System**: Extensible plugin architecture for custom pool behaviors

---

1.5 2. The Quantum Threat

1.5.1 2.1 Shor's Algorithm

Shor's algorithm enables quantum computers to factor large integers and solve the discrete logarithm problem in polynomial time, breaking:

- RSA encryption (factorization)
- Elliptic Curve Cryptography (discrete log on elliptic curves)
- DSA and ECDSA signatures

1.5.2 2.2 Grover's Algorithm

Grover's algorithm provides quadratic speedup for brute-force attacks, reducing the effective security level of symmetric cryptography by half. A 256-bit hash function provides only 128-bit security against quantum attacks.

1.5.3 2.3 Timeline Estimates

- **NIST (2023)**: Quantum computers capable of breaking RSA-2048 may exist by 2030-2035
- **IBM Quantum Roadmap**: 4,000+ qubit systems by 2025
- **"Store Now, Decrypt Later"**: Adversaries are already harvesting encrypted data for future quantum decryption

1.5.4 2.4 Impact on Blockchain

Current blockchain vulnerabilities include:

- **Wallet Security**: Private keys derived from public keys (address reuse)
- **Transaction Integrity**: ECDSA signatures can be forged

- **Consensus Security**: Validator keys compromised
- **Smart Contract Security**: Multi-signature wallets, timelock contracts at risk

---

1.6 3. QRDX Layer-0 Architecture

1.6.1 3.1 Overview

QRDX is a purpose-built **Layer-0 (L0) blockchain protocol** designed for post-quantum security, native cross-chain interoperability, and integrated decentralized exchange functionality. Unlike Layer-1 chains that sit in isolation and rely on external bridges and oracles, QRDX nodes natively embed light clients, chain adapters, and oracle components for external blockchains. Each node is a self-contained cross-chain endpoint that can read state from, submit transactions to, and verify proofs against multiple external chains within the QRDX consensus protocol itself.

The exchange engine is not a smart contract application deployed on top of the chain. It is a **protocol-level primitive** embedded in the execution layer, meaning order matching, liquidity pool management, and settlement happen at the same level as token transfers and state transitions.

Key Specifications: - **Protocol Layer**: Layer-0 (cross-chain substrate) - **Consensus**: Quantum-Resistant Proof-of-Stake (QR-PoS) with PQ-only validator identity - **Block Time**: 2 seconds - **Finality**: Sub-second (single slot finality) - **Throughput**: 5,000+ TPS (exchange operations included) - **Smart Contract VM**: QEVM (Quantum-resistant EVM) - **Node Composition**: Modular chain adapters (Ethereum, Bitcoin, Solana, etc.) - **Exchange Engine**: Protocol-native integrated AMM + order book - **Oracle Layer**: Native cross-chain state attestation by validators - **Transaction Model**: OracleTransaction envelope with chain-specific sub-transactions (EthereumTransaction, BitcoinTransaction, SolanaTransaction)

1.6.2 3.2 Layer-0 Design Rationale

Traditional blockchain architectures separate concerns across layers: - **Layer-1**: Base consensus and execution - **Bridges**: Separate protocols with their own trust assumptions - **Oracles**: External networks (Chainlink, Pyth) with additional attack surface - **DEXs**: Smart contract applications with their own liquidity fragmentation

QRDX collapses these into a single L0 protocol:

Traditional Stack	QRDX L0 Equivalent
L1 Chain + External Bridge	L0 nodes with embedded chain adapters
External Oracle Network	Validators attest cross-chain state natively
DEX Smart Contract	Protocol-native exchange engine
Bridge Relayer Network	Node chain adapter components
Separate Multisig Service	Protocol-level PQ threshold signatures

This reduces the total trust surface to a single set of PQ-authenticated validators who perform all roles: consensus, exchange settlement, oracle attestation, and cross-chain relay.

1.6.3 3.3 QEVM: Quantum-Resistant Ethereum Virtual Machine

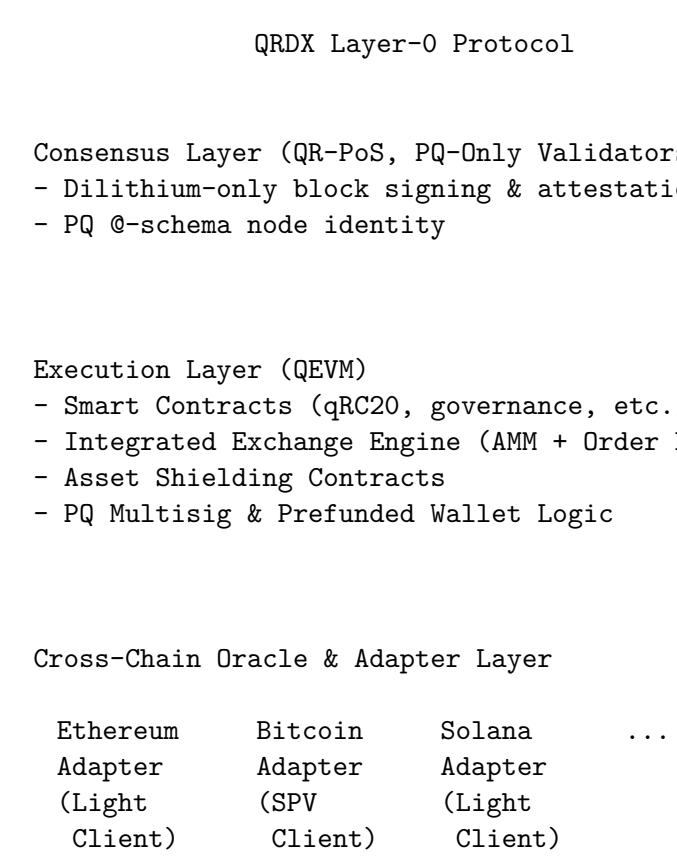
QEVM is a modified EVM that enforces post-quantum cryptographic operations while maintaining backward compatibility with Ethereum tooling.

Modifications: - All signature verification uses CRYSTALS-Dilithium exclusively - Key derivation uses CRYSTALS-Kyber for key encapsulation - Address generation includes quantum-resistant hash functions (SHA3-512, BLAKE3) - Precompiled contracts for efficient post-quantum operations - Native precompiles for exchange engine interaction (pool creation, swaps, order placement) - Oracle precompiles for reading attested cross-chain state

Exchange Engine Precompiles: - `0x0100`: `createPool(tokenA, tokenB, feeTier, initialPrice)` — Deploy a new liquidity pool - `0x0101`: `swap(poolId, direction, amount, minOut)` — Execute a swap - `0x0102`: `addLiquidity(poolId, tickLower, tickUpper, amount)` — Provide concentrated liquidity - `0x0103`: `placeLimitOrder(poolId, price, amount, side)` — Place an on-chain limit order - `0x0104`: `cancelOrder(poolId, orderId)` — Cancel a pending order

Oracle Precompiles: - `0x0200`: `getChainState(chainId)` — Read latest attested block height and state root for an external chain - `0x0201`: `verifyExternalProof(chainId, proof)` — Verify a Merkle/SPV proof against attested state - `0x0202`: `submitCrossChainTx(chainId, txData)` — Queue a transaction for execution on an external chain

1.6.4 3.4 Network Architecture



```
- Native state attestation by validators
- Direct cross-chain transaction submission
- Bridge lock/unlock as protocol operations
```

1.6.5 3.5 Node Composition Model

Every QRDX node is composed of modular components. Validators must run all mandatory components; non-validator full nodes can choose which chain adapters to load.

Mandatory Components (All Validators): - QRDX consensus engine (QR-PoS) - QEVM execution engine - Integrated exchange engine - PQ identity manager (Dilithium keypair store)

Chain Adapter Components (Modular): - **Ethereum Adapter**: Geth light client, tracks ETH block headers, verifies Merkle-Patricia proofs, submits transactions to Ethereum mempool - **Bitcoin Adapter**: SPV client, tracks Bitcoin block headers, verifies SPV proofs, constructs and broadcasts Bitcoin transactions - **Solana Adapter**: Light client tracking slot hashes, verifies transaction inclusion - **Additional Adapters**: BSC, Polygon, Avalanche, Cosmos (IBC), etc.

Validators earn additional oracle rewards for each chain adapter they operate. A validator running Ethereum + Bitcoin + Solana adapters earns oracle fees from all three chains' bridging and attestation operations.

Node Configuration Example:

```yaml
# qrdx-node.yaml
node:
  identity: "dilithium3@qx1a2b3c4d...@validator01.qrdx.org:30303"
  role: validator
  pq_keypair: /keys/dilithium3.key

consensus:
  engine: qr-pos
  min_stake: 100000  # QRDX

exchange_engine:
  enabled: true
  max_pools: 10000
  order_book_depth: 500

chain_adapters:
  ethereum:
    enabled: true
    endpoint: "wss://eth-mainnet.example.com"
    confirmations: 12
    oracle_attestation: true
  bitcoin:
    enabled: true
    endpoint: "tcp://btc-node.example.com:8333"
```

```
    confirmations: 6
    oracle_attestation: true
  solana:
    enabled: false
```

1.6.6 3.6 State Management

QRDX uses a modified Merkle Patricia Trie with quantum-resistant hash functions:

- **State Root Hash**: BLAKE3 (256-bit output extended to 512-bit for quantum resistance)
- **Account State**: Includes quantum-resistant public keys, classical bridge mapping, and exchange positions (open orders, LP positions)
- **Exchange State**: Pool configurations, tick bitmaps, liquidity positions, and order books are stored in protocol-level state, not smart contract storage
- **Oracle State**: Latest attested block heights, state roots, and header hashes for all tracked external chains
- **Storage Optimization**: State pruning and archival nodes for long-term data availability

1.6.7 3.7 Cross-Chain Transaction Execution

Because nodes embed chain adapters, QRDX can execute transactions on external chains directly:

1. **User submits a cross-chain intent** to QRDX (e.g., "send 1 ETH to address X on Ethereum")
2. **Validators with the Ethereum adapter** construct the transaction using locked bridge funds
3. **Threshold Dilithium signature** from 2/3+1 validators authorizes the external transaction
4. **The adapter submits** the signed transaction to the Ethereum mempool
5. **Validators attest** to the external transaction's confirmation once included in an Ethereum block
6. **QRDX state updates** to reflect the completed cross-chain operation

This eliminates the need for separate relayer networks. The validators themselves are the relayers, the oracles, and the bridge operators2014all authenticated with post-quantum cryptography.

1.6.8 3.8 OracleTransaction: Cross-Chain Transaction Type System

QRDX defines a first-class transaction type called **OracleTransaction** that wraps a fully-formed, natively-signed target chain transaction inside a PQ-signed QRDX envelope. This is the fundamental mechanism by which QRDX users execute operations on external blockchains non-custodially, atomically, and with a complete on-chain audit trail.

1.6.8.1 3.8.1 Design Principles The core insight: a user holds both a QRDX PQ keypair and a native keypair for each external chain they interact with. When they want to execute a cross-chain operation, they:

1. **Construct** the target chain transaction locally (e.g., a raw Bitcoin transaction, a signed Ethereum transaction)
2. **Sign it** with the target chain's native key (secp256k1 for Bitcoin/Ethereum, Ed25519 for Solana)
3. **Wrap it** in an OracleTransaction envelope and sign the envelope with their Dilithium key

4. **Submit** the OracleTransaction to QRDX consensus

Validators verify both layers of signatures. The inner transaction is held in QRDX state until execution conditions are met, then validators running the relevant chain adapter broadcast it to the target chain and attest to its confirmation.

This is non-custodial: the user signs the target chain transaction themselves. Validators never hold or derive the user's external chain private keys. They only relay and monitor.

1.6.8.2 3.8.2 Transaction Type Hierarchy

```
OracleTransaction (PQ Envelope)
|
+-- EthereumTransaction    (secp256k1 / EIP-1559 signed)
+-- BitcoinTransaction     (secp256k1 / P2WPKH or P2TR signed)
+-- SolanaTransaction      (Ed25519 signed)
+-- CosmosTransaction      (secp256k1 or Ed25519 signed)
+-- GenericChainTransaction (extensible for future adapters)
```

1.6.8.3 3.8.3 OracleTransaction Envelope The OracleTransaction is an L0-level consensus primitive. Validators deserialize, validate, and process it directly during block production – it is not a smart contract struct but a core protocol transaction type, encoded in QRDX's binary wire format:

```
OracleTransaction {
    //   L0 Envelope (PQ-signed)
    nonce              : uint64       // QRDX-side nonce (replay protection)
    sender             : bytes32      // QRDX address (BLAKE3 hash of Dilithium pubkey)
    target_chain_id    : uint32       // Target chain identifier (1=Ethereum, 2=Bitcoin, 3=Sola
    tx_type            : uint8        // Sub-transaction type discriminant
                                      //   0x01 = EthereumTransaction
                                      //   0x02 = BitcoinTransaction
                                      //   0x03 = SolanaTransaction
                                      //   0x04 = CosmosTransaction
                                      //   0xFF = GenericChainTransaction

    //   Execution Conditions
    conditions         : ExecutionCondition[]  // Conditions that must be met before broadcast
    deadline           : uint64       // Block timestamp after which tx expires (0 = no expiry)
    max_gas_subsidy    : uint128      // Max QRDX (in base units) user pays for oracle gas rein

    //   Inner Transaction Payload
    inner_transaction  : bytes        // Fully-formed, natively-signed target chain tx (opaque
    inner_signature    : bytes        // Target chain signature (secp256k1, Ed25519, etc.)
    target_chain_pubkey: bytes        // User's public key on the target chain

    //   PQ Signature (outer envelope)
    dilithium_signature: bytes[3293]  // Dilithium3 signature over all preceding fields
    dilithium_pubkey   : bytes[1952]  // Sender's Dilithium3 public key
```

```
    //   Callback (optional)
    callback_tx_hash   : bytes32      // If nonzero, triggers a QRDX state transition on confi
    callback_data      : bytes        // Callback payload (e.g., mint qRC20, update oracle sta
}

ExecutionCondition {
    condition_type     : uint8        // 0x00 = immediate
                                      // 0x01 = after_block_height
                                      // 0x02 = after_oracle_tx_confirms
                                      // 0x03 = price_threshold
                                      // 0x04 = balance_threshold
    chain_id           : uint32       // Which chain the condition references
    value              : uint256      // Block height, price, balance, or OracleTx nonce
    reference          : bytes32      // Reference hash (e.g., txHash of prerequisite OracleTx
}
```

Validators decode the `tx_type` discriminant to determine which chain-specific sub-transaction parser to invoke on the `inner_transaction` payload. The entire envelope (all fields before `dilithium_signature`) is the message that the Dilithium signature covers.

1.6.8.4 3.8.4 Chain-Specific Sub-Transaction Types EthereumTransaction (tx_type = 0x01):

The `inner_transaction` payload for Ethereum targets. This is a fully-formed, RLP-encoded, EIP-1559 (or legacy) Ethereum transaction that the user has already signed with their secp256k1 keypair. QRDX validators parse the following fields for validation, but broadcast the raw RLP bytes directly to Ethereum nodes:

```
EthereumTransaction {
    eth_nonce          : uint64       // Ethereum account nonce
    to                 : bytes20      // Destination address on Ethereum
    value              : uint256      // ETH value in wei
    data               : bytes        // Calldata (for contract interactions)
    max_fee_per_gas    : uint256      // EIP-1559 max fee per gas
    max_priority_fee   : uint256      // EIP-1559 priority fee per gas
    gas_limit          : uint64       // Gas limit
    v                  : uint8        // ECDSA recovery id
    r                  : bytes32      // ECDSA r component
    s                  : bytes32      // ECDSA s component
    rlp_encoded        : bytes        // Full RLP-encoded signed tx (broadcast payload)
}
```

Validators verify: `ecrecover(keccak256(rlp_unsigned), v, r, s) == target_chain_pubkey`. The `rlp_encoded` bytes are what gets submitted to the Ethereum network.

BitcoinTransaction (tx_type = 0x02):

The `inner_transaction` payload for Bitcoin targets. This is a fully-formed, serialized Bitcoin transaction (SegWit v0 P2WPKH or Taproot P2TR) that the user has already signed with their

secp256k1 keypair. QRDX validators parse the structure to verify witness signatures against the referenced UTXOs, then broadcast the raw serialized bytes to the Bitcoin P2P network:

```
BitcoinTransaction {
    version            : uint32       // Transaction version (typically 2)
    inputs             : BTCInput[]   // Transaction inputs (UTXOs being spent)
    outputs            : BTCOutput[]  // Transaction outputs
    locktime           : uint32       // Locktime
    raw_transaction    : bytes        // Full serialized signed tx (broadcast payload)
    txid               : bytes32      // Computed txid for tracking
}

BTCInput {
    prev_tx_hash       : bytes32      // Previous transaction hash (little-endian)
    prev_output_index  : uint32       // Previous output index (vout)
    script_sig         : bytes        // ScriptSig (empty for SegWit inputs)
    sequence           : uint32       // Sequence number
    witness            : bytes[]      // Witness stack (signature + pubkey for P2WPKH,
                                      //   or schnorr sig for P2TR)
}

BTCOutput {
    satoshis           : uint64       // Output value in satoshis
    script_pubkey      : bytes        // Output locking script (P2WPKH, P2TR, etc.)
}
```

Validators verify: for each input, the witness data satisfies the scriptPubKey of the referenced UTXO (checked against the Bitcoin SPV adapter's UTXO set). The `raw_transaction` bytes are what gets submitted to the Bitcoin network.

SolanaTransaction (tx_type = 0x03):

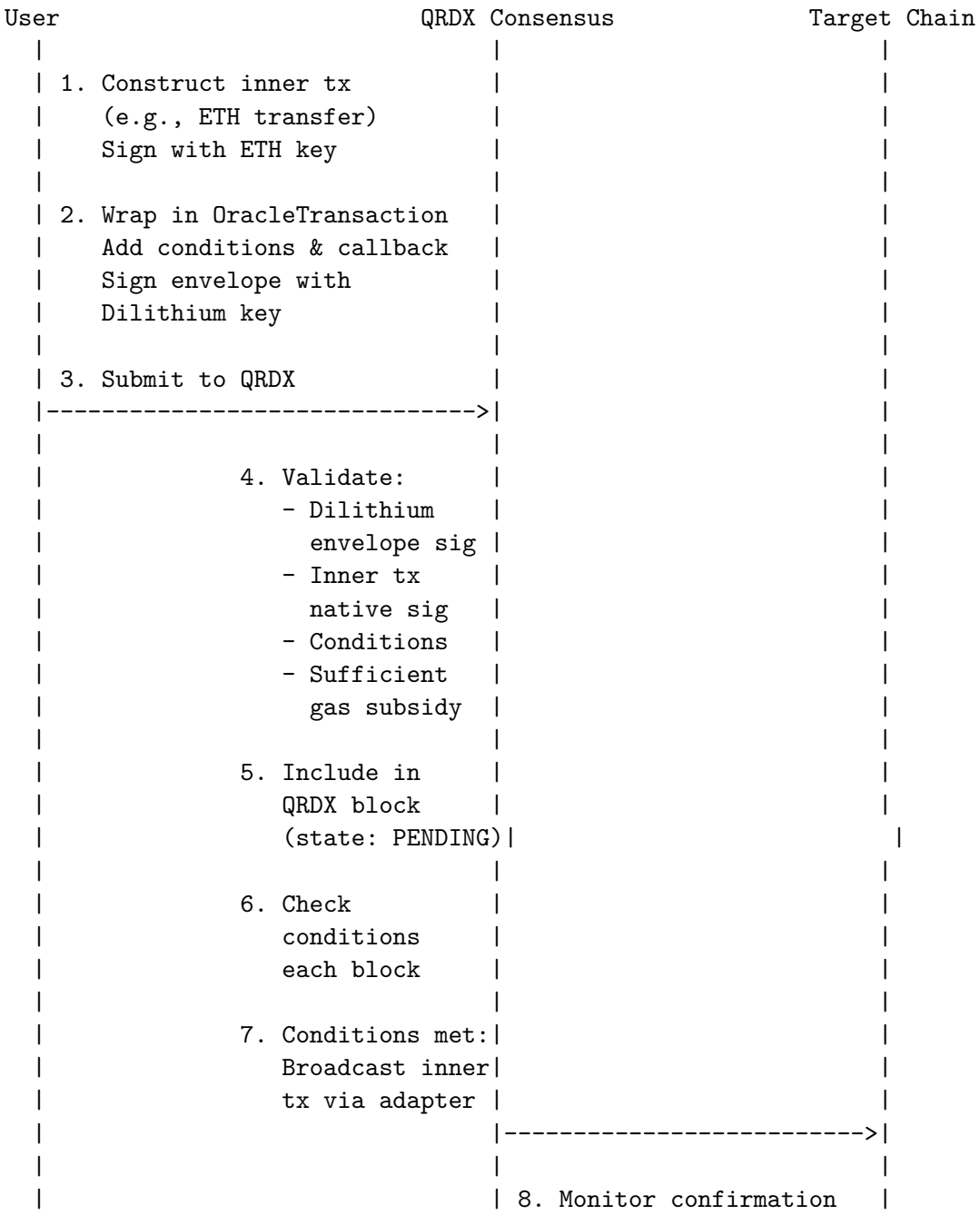
The `inner_transaction` payload for Solana targets. This is a fully-formed Solana transaction that the user has already signed with their Ed25519 keypair. QRDX validators parse the structure to verify the Ed25519 signature, then broadcast the raw serialized bytes to Solana RPC nodes:

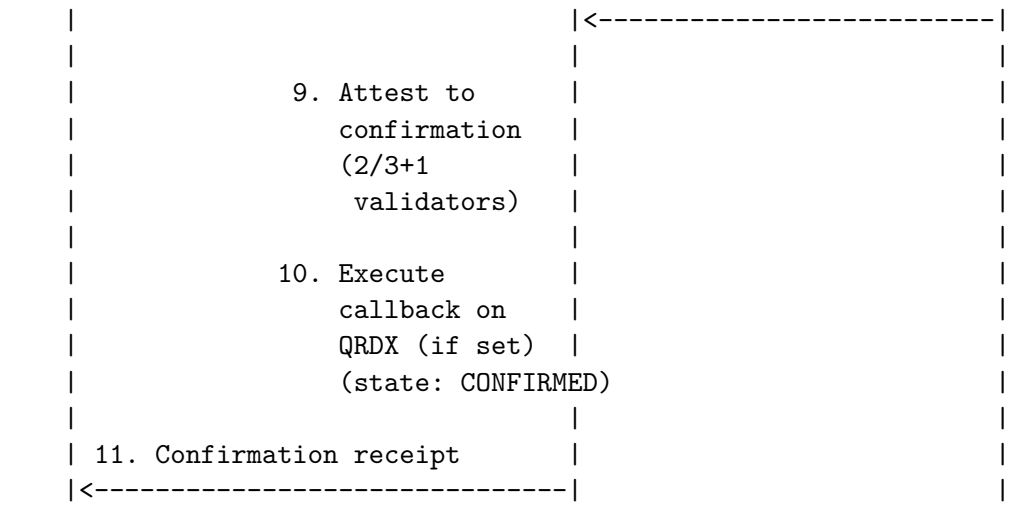
```
SolanaTransaction {
    recent_blockhash   : bytes32           // Solana recent blockhash
    instructions       : SolanaInstruction[] // Program instructions
    signatures         : bytes64[]         // Ed25519 signatures (64 bytes each)
    serialized_message : bytes             // Serialized transaction message
    raw_transaction    : bytes             // Full serialized signed tx (broadcast payload)
}

SolanaInstruction {
    program_id         : bytes32           // Program address
    accounts           : SolanaAccountMeta[] // Account metadata
    data               : bytes             // Instruction data
}
```

```
SolanaAccountMeta {
    pubkey             : bytes32
    is_signer          : bool
    is_writable        : bool
}
```

Validators verify: `ed25519_verify(serialized_message, signatures[0], target_chain_pubkey)`. The `raw_transaction` bytes are what gets submitted to the Solana cluster.

1.6.8.5 3.8.5 Lifecycle of an OracleTransaction





State Transitions:

State	Description
SUBMITTED	OracleTransaction received and validated by QRDX
PENDING	Included in a finalized QRDX block, awaiting conditions
BROADCASTING	Conditions met, inner tx submitted to target chain
CONFIRMING	Inner tx detected on target chain, awaiting confirmations
CONFIRMED	2/3+1 validators attest to confirmation, callback executed
FAILED	Inner tx reverted or expired on target chain
EXPIRED	Deadline passed before conditions were met

1.6.8.6 3.8.6 Execution Conditions & Cross-Chain Composability OracleTransactions support conditional execution, enabling complex cross-chain workflows:

Condition Types:

Type	Description	Example
IMMEDIATE	Broadcast as soon as included in QRDX block	Simple cross-chain transfer
AFTER_BLOCK_HEIGHT	Wait until target chain reaches a specific block	Time-delayed execution
AFTER_ORACLE_TX	Wait until another OracleTransaction confirms	Cross-chain atomic sequences
PRICE_THRESHOLD	Wait until an exchange pair hits a target price	Cross-chain limit orders

Type	Description	Example
`BALANCE_THRESHOLD`	Wait until a target chain address reaches a balance	Conditional funding flows

Cross-Chain Atomic Sequence Example:

A user wants to atomically swap BTC for SOL, routed through QRDX:

```
// Step 1: User constructs a Bitcoin transaction sending BTC to the QRDX bridge
const btcTx = constructBitcoinTx({
    inputs: [userUtxo],
    outputs: [{ address: qrdxBridgeBtcAddress, satoshis: 100_000_000 }], // 1 BTC
});
const signedBtcTx = signWithBitcoinKey(btcTx, userBtcPrivKey);

// Step 2: User constructs a Solana transaction receiving SOL from QRDX bridge
const solTx = constructSolanaInstruction({
    program: QRDX_SOL_BRIDGE_PROGRAM,
    instruction: "release",
    amount: 500_000_000_000, // 500 SOL equivalent
    recipient: userSolAddress,
});
const signedSolTx = signWithSolanaKey(solTx, userSolPrivKey);

// Step 3: Wrap both in OracleTransactions with conditional linking
const oracleTx1 = new OracleTransaction({
    targetChainId: BITCOIN,
    txType: 2, // BitcoinTransaction
    innerTransaction: signedBtcTx.rawTransaction,
    conditions: [{ type: "IMMEDIATE" }],
    callback: { action: "MINT_QBTC", amount: "1.0" },
    dilithiumSignature: signWithDilithium(envelope1, userDilithiumKey),
});

const oracleTx2 = new OracleTransaction({
    targetChainId: SOLANA,
    txType: 3, // SolanaTransaction
    innerTransaction: signedSolTx.rawTransaction,
    conditions: [{
        type: "AFTER_ORACLE_TX",
        reference: oracleTx1.hash(), // Only execute after BTC tx confirms
    }],
    callback: { action: "BURN_QBTC_MINT_QSOL_RELEASE" },
    dilithiumSignature: signWithDilithium(envelope2, userDilithiumKey),
});

// Step 4: Submit both to QRDX as an atomic batch
```

```
await qrdx.submitOracleTransactionBatch([oracleTx1, oracleTx2]);
```

The result: BTC arrives at the bridge, qBTC is minted, an exchange occurs on-chain, qSOL is burned, and the Solana release transaction fires – all orchestrated by QRDX consensus with full PQ audit trail.

1.6.8.7 3.8.7 Advanced Use Cases

1. Cross-Chain DeFi Arbitrage: A user spots a price discrepancy between Uniswap (Ethereum) and Raydium (Solana). They construct an EthereumTransaction buying the underpriced token and a SolanaTransaction selling the overpriced token, linked via `AFTER_ORACLE_TX` conditions. If either leg fails, the sequence halts. QRDX provides the atomicity guarantee across two independent chains.

2. Multi-Chain Treasury Management: A DAO treasury holds assets across Bitcoin, Ethereum, and Solana. Using OracleTransactions signed by a PQ multisig wallet (Section 6), the DAO can rebalance across all three chains in a single QRDX batch submission. The 3-of-5 threshold Dilithium signature authorizes the OracleTransaction envelope, and each inner transaction is signed by the DAO's chain-specific keys.

3. Conditional Cross-Chain Payroll: A company pays contractors in their preferred chain and token. Monthly payroll is prepared as a batch of OracleTransactions: ETH payments for some contractors, BTC for others, SOL for the rest. A `BALANCE_THRESHOLD` condition ensures the master wallet on each chain has sufficient funds before any payments fire. If one chain is underfunded, only that chain's payments are held; the rest proceed.

4. Cross-Chain Liquidation Protection: A user has a leveraged position on an Ethereum lending protocol. They set up a `PRICE_THRESHOLD` OracleTransaction that, if their collateral ratio drops below 150%, automatically submits an Ethereum transaction adding collateral from their wallet. The entire monitoring, triggering, and execution is handled by QRDX validators – no centralized keeper bot required.

5. Atomic Cross-Chain NFT Purchases: A user wants to buy an Ethereum NFT using BTC. They construct a BitcoinTransaction paying the seller's BTC address and an EthereumTransaction calling the NFT contract's `transferFrom` (pre-approved by the seller via a signed listing). Both are linked: the NFT transfer only executes after the BTC payment confirms.

1.6.8.8 3.8.8 Validator Processing of OracleTransactions

When a validator includes an OracleTransaction in a block, the following processing occurs:

1. **Envelope Validation**: Verify Dilithium signature on the OracleTransaction envelope
2. **Inner Signature Validation**: Verify the target chain's native signature on the inner transaction using the appropriate algorithm:
   - EthereumTransaction: ecrecover(hash, v, r, s) must match `targetChainPubKey`
   - BitcoinTransaction: Verify witness/scriptSig against input UTXOs
   - SolanaTransaction: Ed25519 signature verification against `targetChainPubKey`
3. **Condition Evaluation**: Check all `ExecutionCondition` entries against current oracle state
4. **Gas Subsidy Check**: Verify user has sufficient QRDX balance for `maxGasSubsidy`
5. **State Update**: Record OracleTransaction in protocol state with status `PENDING`
6. **Broadcast Scheduling**: If all conditions are met, queue inner transaction for broadcast by the relevant chain adapter

7. **Confirmation Monitoring**: Track inner transaction on target chain via the adapter's light client
8. **Attestation**: Once confirmed, submit attestation; when 2/3+1 validators attest, mark as `CONFIRMED`
9. **Callback Execution**: If `callbackTxHash` is set, execute the callback (e.g., mint qRC20, update exchange state, trigger next OracleTransaction in sequence)

1.6.8.9 3.8.9 Fee Model for OracleTransactions

Component	Fee	Description
Envelope Processing	0.001 QRDX	Fixed fee for envelope validation and state storage
Condition Monitoring	0.0005 QRDX/block	Per-block fee while conditions are being evaluated
Chain Adapter Broadcast	Variable	Reimburses validator for target chain gas/fees
Confirmation Attestation	0.002 QRDX	Paid to attestation validators on confirmation
Callback Execution	Standard QRDX gas	If callback triggers QRDX state changes

Users set `maxGasSubsidy` to cap their total cost. If the target chain's gas exceeds the subsidy, the OracleTransaction is held until gas drops or expires at the deadline.

1.6.8.10 3.8.10 Security Properties

- **Non-custodial**: Users sign inner transactions themselves. Validators never access user private keys for external chains.
- **Double-signature verification**: Both PQ envelope and native chain signatures must be valid. A compromised Dilithium key alone cannot forge a target chain transaction, and a compromised target chain key alone cannot submit an OracleTransaction.
- **Replay protection**: QRDX nonce prevents replay on the L0 side. Target chain nonce/UTXO model prevents replay on the target side.
- **Condition integrity**: Conditions are evaluated against validator-attested oracle state (2/3+1 consensus), not a single source.
- **Atomicity within QRDX**: OracleTransaction batches are atomic on the QRDX side. If batch submission fails, no inner transactions are broadcast.
- **Graceful failure**: If an inner transaction fails or reverts on the target chain, the OracleTransaction enters `FAILED` state and the callback is not executed. The user's QRDX state reflects the failure.

1.7 4. Post-Quantum Cryptography Implementation

1.7.1 4.1 CRYSTALS-Dilithium (Digital Signatures)

Algorithm: Module-Lattice-Based Digital Signature Algorithm
NIST Status: FIPS 204 (Standardized 2024)

Security Level: NIST Level 3 (comparable to AES-192)

Implementation Details: - Public Key Size: 1,952 bytes - Signature Size: 3,293 bytes - Key Generation: ~50 microseconds - Signature Generation: ~100 microseconds - Verification: ~60 microseconds

Usage in QRDX: - Transaction signing - Block signing by validators - Smart contract authentication - Multi-signature wallets

1.7.2 4.2 CRYSTALS-Kyber (Key Encapsulation)

Algorithm: Module-Lattice-Based Key Encapsulation Mechanism
NIST Status: FIPS 203 (Standardized 2024)
Security Level: NIST Level 3

Implementation Details: - Public Key Size: 1,184 bytes - Ciphertext Size: 1,088 bytes - Shared Secret Size: 32 bytes - Encapsulation: ~40 microseconds - Decapsulation: ~50 microseconds

Usage in QRDX: - Secure key exchange for encrypted transactions - Private transaction pools - Validator communication encryption - Cross-chain bridge security

1.7.3 4.3 Hash Functions

Primary: BLAKE3 (512-bit output)
Secondary: SHA3-512
Merkle Tree: BLAKE3-based

Rationale: BLAKE3 provides 256-bit quantum resistance (Grover's algorithm reduces effective security by half) while maintaining high performance.

1.7.4 4.4 Hybrid Security Model

During the transition period, QRDX supports a hybrid mode for **user wallets only** (validators must use PQ-only):

- **Classical + Post-Quantum Signatures**: Dual signing with ECDSA + Dilithium for user transactions during migration
- **Migration Path**: Users can upgrade from classical to quantum-resistant addresses
- **Backward Compatibility**: Legacy transactions supported with quantum-resistant wrapping
- **Validator Exclusion**: Validators, bootstrap nodes, and bridge operators have no classical fallback2014PQ-only from genesis

---

1.8 5. PQ Node Identity & Bootstrap Protocol

1.8.1 5.1 The `@`-Schema Node Addressing Format

Every QRDX node is identified by a post-quantum address using the `@`-schema format. This replaces classical enode URLs (which use secp256k1 public keys) with a PQ-authenticated identity string.

Format:

`<pq_algorithm>@<pubkey_hash>@<host>:<port>`

Examples:

`dilithium3@qx1a2b3c4d5e6f7890abcdef1234567890abcdef12@validator01.qrdx.org:30303`
`dilithium3@qx9f8e7d6c5b4a3210fedcba0987654321fedcba09@boot1.qrdx.org:30303`
`dilithium3@qxdeadbeef00112233445566778899aabbccddeeff@192.168.1.100:30303`

Components: - `dilithium3`: The PQ signature algorithm and security level (NIST Level 3) - `qx1a2b3c...`: BLAKE3 hash of the node's Dilithium public key, prefixed with `qx` - `validator01.qrdx.org:30303`: Host and port for network connectivity

1.8.2 5.2 Bootstrap Node Discovery

Network bootstrap uses a hardcoded set of PQ-authenticated bootstrap nodes. New nodes connecting to the QRDX network for the first time:

1. **Connect to bootstrap nodes** using their `@`-schema addresses from the genesis configuration
2. **Perform PQ handshake**: Both sides exchange Dilithium public keys and sign a nonce challenge
3. **Kyber key exchange**: Establish an encrypted channel using CRYSTALS-Kyber key encapsulation
4. **Peer table population**: Bootstrap node returns a list of known peers, each identified by their `@`-schema address
5. **Recursive discovery**: New node connects to discovered peers, expanding its peer table

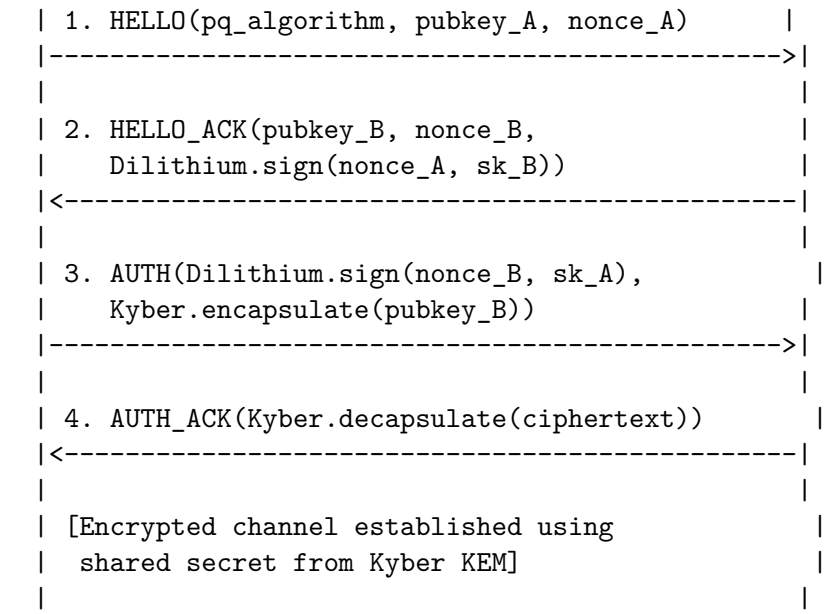
Genesis Bootstrap Configuration:

```
{
  "bootstrap_nodes": [
    "dilithium3@qxboot001...@boot1.qrdx.org:30303",
    "dilithium3@qxboot002...@boot2.qrdx.org:30303",
    "dilithium3@qxboot003...@boot3.qrdx.org:30303",
    "dilithium3@qxboot004...@boot4.qrdx.org:30303",
    "dilithium3@qxboot005...@boot5.qrdx.org:30303"
  ],
  "pq_handshake": {
    "algorithm": "dilithium3",
    "kex": "kyber1024",
    "hash": "blake3-512",
    "challenge_bytes": 64,
    "timeout_ms": 5000
  }
}
```

1.8.3 5.3 PQ Handshake Protocol

The PQ handshake replaces the classical RLPx/devp2p handshake used by Ethereum-derived networks:

```
Node A                                                  Node B
  |                                                        |
```



After the handshake, all communication between nodes is encrypted using the Kyber-derived shared secret with AES-256-GCM. Node identity is permanently bound to the Dilithium public key2014there is no possibility of identity spoofing without breaking lattice-based cryptography.

1.8.4 5.4 Validator Identity Binding

Validators must register their Dilithium public key on-chain before participating in consensus. The on-chain registration permanently binds:

- **Dilithium Public Key** 2192 Validator identity
- **@-Schema Address** 2192 Network reachability
- **Staked QRDX** 2192 Economic commitment
- **Chain Adapter Capabilities** 2192 Which external chains this validator can attest

A validator cannot change their identity key without unstaking, waiting the unbonding period, and re-registering with a new key. This prevents key rotation attacks.

1.9 6. PQ Multisignatures & Wallet Architecture

1.9.1 6.1 Threshold Dilithium Multisignatures

QRDX implements m-of-n post-quantum multisignatures using a threshold Dilithium scheme at the protocol level. Unlike classical ECDSA multisig (which concatenates individual signatures), threshold Dilithium produces a single aggregate signature that is indistinguishable from a standard Dilithium signature, reducing on-chain footprint.

Scheme Properties: - **Threshold**: Any m of n signers can produce a valid signature - **PQ Security**: Based on Module-LWE hardness (same as standard Dilithium) - **Aggregation**: Single signature output (3,293 bytes regardless of m or n) - **Non-interactive**: Signers can produce partial signatures independently - **Verifier efficiency**: Verification cost identical to single-signature Dilithium

Supported Configurations:

Configuration	Use Case
2-of-3	Personal wallets with recovery
3-of-5	Team/DAO treasuries
5-of-9	Protocol governance multisig
10-of-15	Bridge validator threshold
15-of-23	Bitcoin federation multisig
m-of-n (arbitrary)	Custom configurations up to 100-of-150

Multisig Wallet Creation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

/**
 * @title QRDXMultisig
 * @notice Post-quantum m-of-n multisignature wallet using threshold Dilithium
 * @dev All signer keys must be Dilithium3 public keys. Threshold signatures
 *      produce a single aggregated signature verified by the DILITHIUM precompile.
 */
contract QRDXMultisig {
    /// @notice Threshold configuration
    uint8 public immutable threshold;      // m (minimum signers)
    uint8 public immutable totalSigners;   // n (total signers)

    /// @notice Aggregated Dilithium public key for threshold verification
    bytes public aggregatePublicKey;

    /// @notice Individual signer Dilithium public keys
    mapping(uint8 => bytes) public signerKeys;

    /// @notice Transaction nonce
    uint256 public nonce;

    /// @notice Pending transactions awaiting threshold signatures
    mapping(bytes32 => PendingTx) public pendingTxs;

    struct PendingTx {
        address to;
        uint256 value;
        bytes data;
        uint8 signatureCount;
        bool executed;
        mapping(uint8 => bool) hasSigned;
    }
```

```
event TxProposed(bytes32 indexed txId, address to, uint256 value);
event TxSigned(bytes32 indexed txId, uint8 indexed signerIndex);
event TxExecuted(bytes32 indexed txId);

constructor(
    uint8 _threshold,
    uint8 _totalSigners,
    bytes[] memory _signerKeys
) {
    require(_threshold > 0 && _threshold <= _totalSigners, "Invalid threshold");
    require(_signerKeys.length == _totalSigners, "Key count mismatch");

    threshold = _threshold;
    totalSigners = _totalSigners;

    for (uint8 i = 0; i < _totalSigners; i++) {
        signerKeys[i] = _signerKeys[i];
    }

    // Aggregate public key computed off-chain via threshold DKG
    // and verified on deployment
}

/**
 * @notice Submit a partial signature for a pending transaction
 * @dev Once threshold is met, the aggregated signature is verified
 *      via the DILITHIUM precompile and the transaction executes
 */
function submitSignature(
    bytes32 txId,
    uint8 signerIndex,
    bytes calldata partialSignature
) external {
    PendingTx storage ptx = pendingTxs[txId];
    require(!ptx.executed, "Already executed");
    require(!ptx.hasSigned[signerIndex], "Already signed");
    require(signerIndex < totalSigners, "Invalid signer");

    // Verify partial signature via Dilithium precompile
    // DILITHIUM_PRECOMPILE.verifyPartial(txId, partialSignature, signerKeys[signerIndex])

    ptx.hasSigned[signerIndex] = true;
    ptx.signatureCount++;

    emit TxSigned(txId, signerIndex);

    if (ptx.signatureCount >= threshold) {
        _execute(txId);
```

```
        }
    }

    function _execute(bytes32 txId) internal {
        PendingTx storage ptx = pendingTxs[txId];
        ptx.executed = true;

        (bool success, ) = ptx.to.call{value: ptx.value}(ptx.data);
        require(success, "Execution failed");

        nonce++;
        emit TxExecuted(txId);
    }
}
```

1.9.2 6.2 Prefunded Wallet Architecture

QRDX introduces **prefunded wallets**: a hierarchy of sub-wallets that are created, funded, and controlled by a single master PQ wallet. This enables institutional custody patterns, automated treasury management, and operational separation without sacrificing quantum resistance.

Architecture:

```
        Master PQ Wallet
  Key: Dilithium3 Master Keypair
  Controls: All sub-wallets
  Permissions: Create, fund, freeze, reclaim


    Sub-Wallet A     Sub-Wallet B    ...
    (Operations)     (Trading)
    Budget: 1000     Budget: 5000
    Daily: 100       Daily: 500
    Scope: xfer      Scope: swap



    Sub-Wallet C     Sub-Wallet D
    (Staking)        (Bridge)
    Budget: 50k      Budget: 10k
    Daily: N/A       Daily: 2000
    Scope: stake     Scope: bridge
```

Key Properties:

- **Master Control**: Only the master PQ wallet can create, destroy, or reconfigure sub-wallets
- **Budget Limits**: Each sub-wallet has a total budget and optional daily spending limits

- **Scope Restrictions**: Sub-wallets can be restricted to specific operations (transfers, swaps, staking, bridging)
- **Delegated Keys**: Each sub-wallet has its own Dilithium keypair for independent transaction signing, but the master can override or freeze at any time
- **Auto-Refill**: Master wallet can configure automatic refill rules (e.g., "refill to 1000 QRDX when balance drops below 100")
- **Reclaim**: Master wallet can reclaim all funds from any sub-wallet instantly

Prefunded Wallet Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

/**
 * @title PrefundedWalletManager
 * @notice Manages prefunded sub-wallets controlled by a master PQ wallet
 */
contract PrefundedWalletManager {
    struct SubWallet {
        bytes dilithiumPubKey;     // Sub-wallet's own PQ key
        uint256 totalBudget;       // Maximum total funds
        uint256 dailyLimit;        // Daily spending cap (0 = unlimited)
        uint256 spent;             // Total spent
        uint256 dailySpent;        // Spent in current day
        uint256 lastResetDay;      // Day number of last daily reset
        uint8 scope;               // Bitmask: 1=transfer, 2=swap, 4=stake, 8=bridge
        bool active;               // Can be frozen by master
    }

    /// @notice Master wallet Dilithium public key
    bytes public masterPubKey;

    /// @notice Master wallet address (derived from Dilithium key)
    address public immutable masterAddress;

    /// @notice Sub-wallet registry
    mapping(address => SubWallet) public subWallets;
    address[] public subWalletList;

    event SubWalletCreated(address indexed subWallet, uint256 budget, uint8 scope);
    event SubWalletFunded(address indexed subWallet, uint256 amount);
    event SubWalletFrozen(address indexed subWallet);
    event FundsReclaimed(address indexed subWallet, uint256 amount);

    modifier onlyMaster() {
        require(msg.sender == masterAddress, "Not master wallet");
        _;
    }
```

```
constructor(bytes memory _masterPubKey) {
    masterPubKey = _masterPubKey;
    masterAddress = msg.sender;
}

function createSubWallet(
    address subAddr,
    bytes calldata subPubKey,
    uint256 totalBudget,
    uint256 dailyLimit,
    uint8 scope
) external onlyMaster {
    subWallets[subAddr] = SubWallet({
        dilithiumPubKey: subPubKey,
        totalBudget: totalBudget,
        dailyLimit: dailyLimit,
        spent: 0,
        dailySpent: 0,
        lastResetDay: block.timestamp / 1 days,
        scope: scope,
        active: true
    });
    subWalletList.push(subAddr);
    emit SubWalletCreated(subAddr, totalBudget, scope);
}

function freezeSubWallet(address subAddr) external onlyMaster {
    subWallets[subAddr].active = false;
    emit SubWalletFrozen(subAddr);
}

function reclaimFunds(address subAddr) external onlyMaster {
    uint256 balance = address(this).balance; // simplified
    subWallets[subAddr].active = false;
    payable(masterAddress).transfer(balance);
    emit FundsReclaimed(subAddr, balance);
}

function checkSpendingAllowed(
    address subAddr,
    uint256 amount,
    uint8 operationType
) external view returns (bool) {
    SubWallet storage sw = subWallets[subAddr];
    if (!sw.active) return false;
    if (sw.scope & operationType == 0) return false;
    if (sw.spent + amount > sw.totalBudget) return false;
```

```
        if (sw.dailyLimit > 0) {
            uint256 currentDay = block.timestamp / 1 days;
            uint256 todaySpent = (currentDay == sw.lastResetDay) ? sw.dailySpent : 0;
            if (todaySpent + amount > sw.dailyLimit) return false;
        }

        return true;
    }
}
```

1.9.3 6.3 Institutional Custody Model

The combination of PQ multisig and prefunded wallets enables institutional-grade custody:

1. **Treasury**: 5-of-9 PQ multisig holds the organization's master wallet
2. **Operations**: Prefunded sub-wallet with daily limit for routine transactions
3. **Trading**: Prefunded sub-wallet scoped to exchange operations only
4. **Staking**: Prefunded sub-wallet scoped to staking operations, no daily limit
5. **Emergency**: 3-of-5 PQ multisig sub-wallet for emergency recovery

No single compromised key can drain funds. No classical cryptography is involved at any point in the custody chain.

1.10 7. QRDX Integrated Exchange Engine

1.10.1 7.1 Design Philosophy

The QRDX exchange is not a smart contract deployed on top of the blockchain. It is a **protocol-native execution engine** embedded directly into the QEVM execution layer. Every QRDX node processes exchange operations2014pool creation, swaps, liquidity provision, and order matching2014as first-class transaction types, just like token transfers.

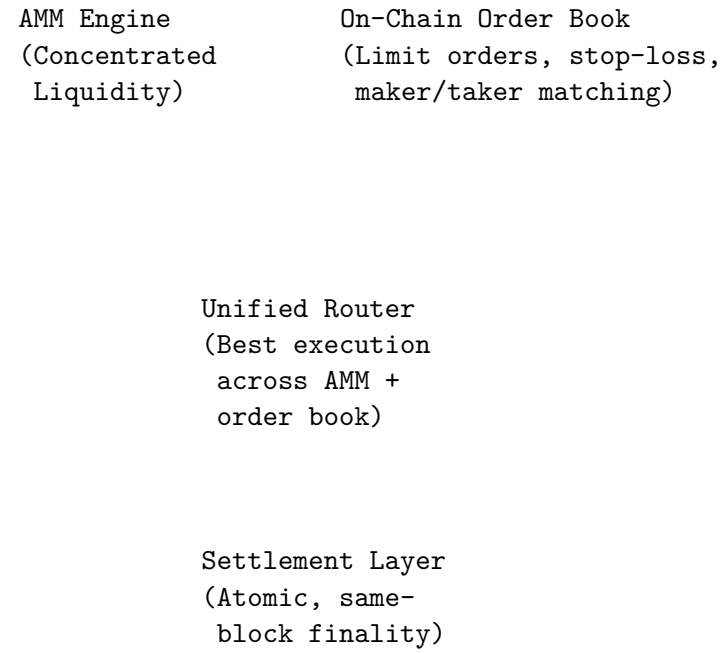
This design, comparable to Hyperliquid's approach of building the exchange into the chain itself, provides:

- **Zero deployment cost**: Pools and order books exist as protocol state, not contract storage
- **Deterministic execution**: Exchange logic is part of the state transition function, not subject to smart contract gas estimation variance
- **Native settlement**: Trades settle atomically within the same block as the state transition
- **Validator-enforced fairness**: Order matching follows a deterministic rule set enforced by consensus, preventing MEV extraction at the protocol level
- **Permissionless pool creation**: Any participant can spin up a new liquidity pool by staking or subsidizing sufficient QRDX

1.10.2 7.2 Exchange Architecture

The exchange engine consists of three integrated components:

```
QRDX Integrated Exchange Engine
```



Component 1: AMM Engine (Concentrated Liquidity)

Liquidity providers (LPs) can concentrate liquidity within specific price ranges, following proven Uniswap v3/v4 concentrated liquidity principles:

```
Price Range: [P_a, P_b]
Liquidity Density: L(P) = k / (P - P_a) for P in [P_a, P_b]
```

Benefits: - Up to 4000x capital efficiency vs. constant product AMMs - Higher fee APY for active LPs - Reduced slippage for traders

Component 2: On-Chain Order Book

For pairs with sufficient liquidity, the exchange engine maintains a fully on-chain order book:

- **Limit orders**: Placed at specific prices, filled when the market reaches them
- **Stop-loss orders**: Triggered when price crosses a threshold
- **Maker/taker model**: Makers (limit orders) pay lower fees than takers (market orders)
- **Time priority**: Orders at the same price are filled in the order they were submitted (enforced by consensus)
- **Order book depth**: Configurable per pool (default: 500 price levels per side)

Component 3: Unified Router

The router determines the best execution path for each trade: - Pure AMM fill (small trades, low-liquidity pairs) - Pure order book fill (large trades with matching limit orders) - Hybrid fill (split across AMM and order book for optimal price)

1.10.3 7.3 Permissionless Pool Creation

Any QRDX participant can create a new liquidity pool by meeting the pool creation requirements. This is a protocol-level operation, not a smart contract deployment.

Pool Creation Requirements:

Pool Type	Stake/Subsidy Required	Description
Standard Pool	10,000 QRDX staked	For qRC20/qRC20 pairs with existing liquidity
Bootstrap Pool	25,000 QRDX staked	For new token pairs, includes 30-day incentive period
Subsidized Pool	5,000 QRDX burned (subsidy)	Reduced ongoing costs, permanent pool
Institutional Pool	100,000 QRDX staked	Higher order book depth, lower fees, priority matching

Staked vs. Subsidized: - **Staked**: QRDX is locked for the pool's lifetime. Pool creator earns a share of pool fees. If the pool is closed, staked QRDX is returned after a 7-day unbonding period. - **Subsidized**: QRDX is burned permanently. The pool has lower ongoing fees and cannot be closed by the creator. Community-owned infrastructure.

Pool Creation Transaction:

```
// Protocol-level pool creation (processed as a native transaction type)
struct CreatePoolTx {
    // Token pair
    address tokenA;
    address tokenB;

    // Pool configuration
    uint24 feeTier;           // 100 (0.01%), 500 (0.05%), 3000 (0.30%), 10000 (1.00%)
    int24 tickSpacing;        // Price granularity
    uint160 initialSqrtPrice; // Starting price

    // Creator configuration
    uint8 poolType;           // 0=standard, 1=bootstrap, 2=subsidized, 3=institutional
    uint256 stakeAmount;      // QRDX staked/burned for pool creation
    bool enableOrderBook;     // Whether to activate the on-chain order book
    uint16 orderBookDepth;    // Max price levels per side (if order book enabled)

    // Hooks (optional plugin logic)
    address hooksContract;    // Address of hooks contract (address(0) for none)
    uint256 hookFlags;        // Bitmask of enabled hook points

    // Creator signature
    bytes dilithiumSignature;
}
```

1.10.4 7.4 Hooks System

Hooks allow developers to customize pool behavior at key lifecycle points, similar to Uniswap v4's hook architecture but operating at the protocol level:

Hook Points: - `beforeInitialize` / `afterInitialize` - `beforeSwap` / `afterSwap` - `beforeModifyLiquidity` / `afterModifyLiquidity` - `beforeOrderPlace` / `afterOrderFill` - `beforeDonate` / `afterDonate`

Use Cases: - Time-weighted average price (TWAP) oracles - Dynamic fees based on volatility - Limit orders and stop-loss (augmenting native order book) - KYC/AML compliance checks - Liquidity mining rewards distribution - Cross-chain arbitrage automation

1.10.5 7.5 Protocol-Level Pool Manager

All QRDX pools exist within the protocol's state trie, managed by the PoolManager2014a protocol-level singleton that is part of the QEVM execution engine, not a deployed contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {IPoolManager} from "./interfaces/IPoolManager.sol";
import {IHooks} from "./interfaces/IHooks.sol";
import {PoolKey} from "./types/PoolKey.sol";
import {PoolId, PoolIdLibrary} from "./types/PoolId.sol";
import {Currency} from "./types/Currency.sol";
import {BalanceDelta} from "./types/BalanceDelta.sol";

/**
 * @title PoolManager
 * @notice Protocol-native pool manager embedded in QEVM execution layer.
 * @dev This is not a deployed contract. It is a precompile that operates
 *      on protocol-level state. Shown in Solidity for interface clarity.
 */
contract PoolManager is IPoolManager {
    using PoolIdLibrary for PoolKey;

    /// @notice Mapping of pool IDs to pool state (protocol-level storage)
    mapping(PoolId id => Pool.State) public pools;

    /// @notice Pool creator stakes
    mapping(PoolId id => PoolStake) public poolStakes;

    struct PoolStake {
        address creator;
        uint256 stakedAmount;
        uint8 poolType;
        bool subsidized;
    }
```

```
/// @notice The current locker (flash accounting)
address public locker;

/// @notice Reentrancy guard
uint256 private unlocked = 1;

modifier lock() {
    require(locker == address(0), "PoolManager: LOCKED");
    locker = msg.sender;
    _;
    locker = address(0);
}

modifier onlyLocker() {
    require(msg.sender == locker, "PoolManager: NOT_LOCKER");
    _;
}

/// @inheritdoc IPoolManager
function swap(
    PoolKey memory key,
    IPoolManager.SwapParams memory params,
    bytes calldata hookData
) external override onlyLocker returns (BalanceDelta delta) {
    PoolId id = key.toId();
    Pool.State storage pool = pools[id];

    // Execute hooks
    if (key.hooks.shouldCallBeforeSwap()) {
        key.hooks.beforeSwap(msg.sender, key, params, hookData);
    }

    // Unified router: check order book first, then AMM, or split
    delta = pool.swap(params);

    // Execute after hooks
    if (key.hooks.shouldCallAfterSwap()) {
        key.hooks.afterSwap(msg.sender, key, params, delta, hookData);
    }

    emit Swap(id, msg.sender, delta.amount0(), delta.amount1(), pool.slot0.sqrtPriceX96, po
}

/// @inheritdoc IPoolManager
function modifyLiquidity(
    PoolKey memory key,
    IPoolManager.ModifyLiquidityParams memory params,
    bytes calldata hookData
```

```
) external override onlyLocker returns (BalanceDelta delta) {
    PoolId id = key.toId();
    Pool.State storage pool = pools[id];

    if (key.hooks.shouldCallBeforeModifyLiquidity()) {
        key.hooks.beforeModifyLiquidity(msg.sender, key, params, hookData);
    }

    delta = pool.modifyLiquidity(
        Pool.ModifyLiquidityParams({
            owner: msg.sender,
            tickLower: params.tickLower,
            tickUpper: params.tickUpper,
            liquidityDelta: params.liquidityDelta
        })
    );

    if (key.hooks.shouldCallAfterModifyLiquidity()) {
        key.hooks.afterModifyLiquidity(msg.sender, key, params, delta, hookData);
    }

    emit ModifyLiquidity(id, msg.sender, params.tickLower, params.tickUpper, params.liquidi
}

/// @inheritdoc IPoolManager
function initialize(
    PoolKey memory key,
    uint160 sqrtPriceX96,
    bytes calldata hookData
) external override returns (int24 tick) {
    PoolId id = key.toId();
    require(pools[id].slot0.sqrtPriceX96 == 0, "PoolManager: ALREADY_INITIALIZED");

    // Verify pool creator has sufficient stake/subsidy
    // (enforced by protocol, not contract logic)

    if (key.hooks.shouldCallBeforeInitialize()) {
        key.hooks.beforeInitialize(msg.sender, key, sqrtPriceX96, hookData);
    }

    tick = pools[id].initialize(sqrtPriceX96, key.fee);

    if (key.hooks.shouldCallAfterInitialize()) {
        key.hooks.afterInitialize(msg.sender, key, sqrtPriceX96, tick, hookData);
    }

    emit Initialize(id, key.currency0, key.currency1, key.fee, key.tickSpacing, key.hooks)
}
```

```
}
```

Gas Savings: ~60% reduction compared to application-layer DEXs through protocol-native execution, flash accounting, and elimination of contract call overhead.

1.10.6 7.6 Fee Structure

QRDX exchange implements a flexible fee tier system:

Fee Tier	Typical Use Case	Expected Volume
0.01%	Stablecoin pairs (qUSDC/qUSDT)	High
0.05%	Major pairs (qETH/qUSDC)	High
0.30%	Standard pairs	Medium
1.00%	Exotic / low-liquidity pairs	Low

Fee Distribution: - 70% to liquidity providers - 15% to pool creator (stake reward) - 10% to QRDX protocol treasury (governed by QRDX token holders) - 5% to validator oracle/settlement rewards

Order Book Fee Model: - Maker fee: 0.02% (incentivizes limit orders and liquidity depth) - Taker fee: 0.05% (market orders pay for immediacy)

1.10.7 7.7 Price Oracle

QRDX maintains geometric mean time-weighted average price (TWAP) oracles as protocol-level state:

```
TWAP(t_0, t_1) = exp((log(P(t_1)) * t_1 - log(P(t_0)) * t_0) / (t_1 - t_0))
```

Oracles are quantum-resistant by design, with Dilithium signatures securing price updates. Cross-chain oracle data (external asset prices) is provided by validators running the relevant chain adapters.

1.11 8. Asset Shielding Mechanism

1.11.1 8.1 Overview

Asset shielding, also known as **Quantum Shielding**, is the core mechanism that protects traditional blockchain assets from future quantum attacks by converting them into quantum-resistant equivalents on QRDX Chain. This process involves locking classical assets on their native chains and minting corresponding qRC20 tokens secured by post-quantum cryptography.

Quantum Shielding converts: - BTC → qBTC (Bitcoin to Quantum Bitcoin) - ETH → qETH (Ethereum to Quantum Ethereum) - WBTC → qBTC (Wrapped Bitcoin to Quantum Bitcoin) - USDC → qUSDC (USD Coin to Quantum USD Coin) - USDT → qUSDT (Tether to Quantum Tether) - Any ERC-20 → qRC20 equivalent

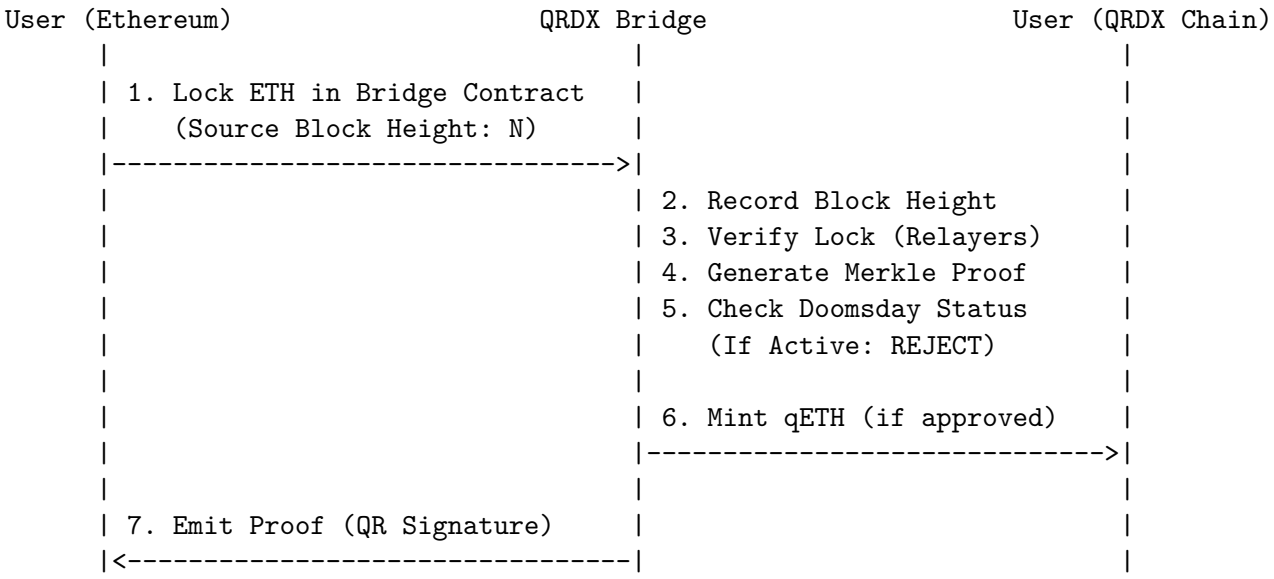
Why Quantum Shielding?

Classical blockchain assets use ECDSA signatures and elliptic curve cryptography, both vulnerable to Shor's algorithm running on sufficiently powerful quantum computers. Quantum shielding provides a migration path to post-quantum security while maintaining 1:1 backing and liquidity between classical and quantum-resistant assets.

Key Features: - **Trustless Bridge**: No centralized custodian required - **1:1 Backing**: Every qETH/qBTC is backed by locked ETH/BTC - **Bi-directional**: Shield and unshield assets freely - **Block Height Recording**: All bridged chain states are tracked - **Doomsday Protocol**: Automatic circuit breaker if quantum threat detected

1.11.2 8.2 Shielding Process (Classical → Quantum)

The shielding process converts classical assets like BTC or ETH into their quantum-resistant equivalents:



Step-by-Step Process:

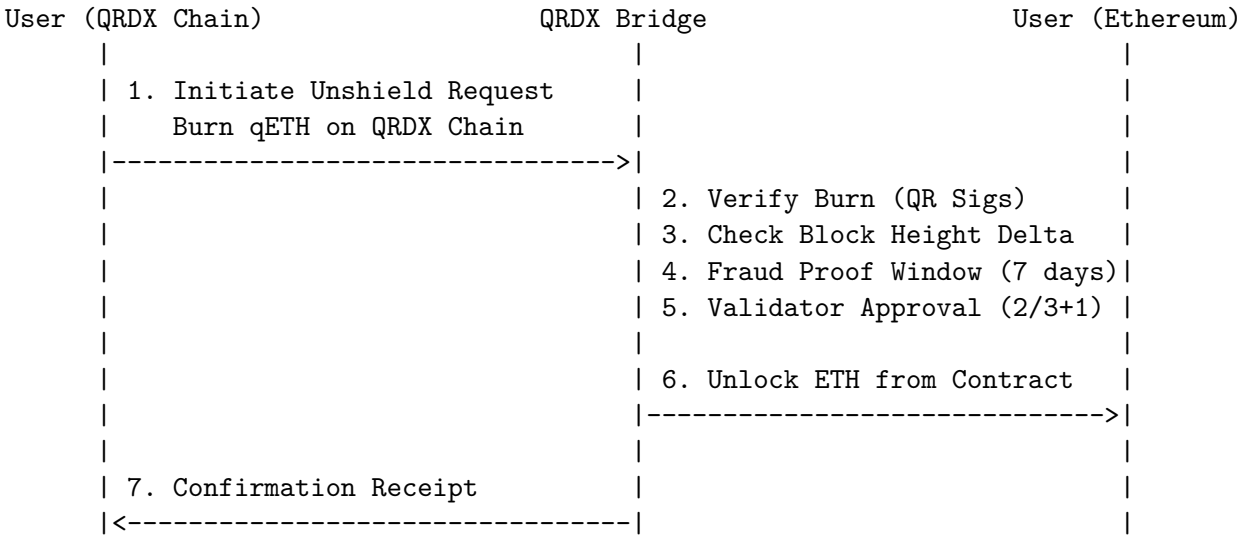
1. **User Initiates Shielding**: User sends ETH (or BTC via wrapped BTC) to the QRDX Bridge lock contract on the source chain
2. **Block Height Recording**: The bridge records the exact block height of the source chain at the time of locking
3. **Relayer Monitoring**: Decentralized relayers monitor the source chain for lock events
4. **Confirmation Period**: Bridge waits for sufficient confirmations (12 blocks for Ethereum, 6 for Bitcoin)
5. **Doomsday Check**: Bridge verifies the doomsday protocol has not been triggered (see Section 8.5)
6. **Proof Generation**: Relayers generate cryptographic Merkle proofs of the lock transaction
7. **Validator Consensus**: QRDX Chain validators verify proofs using quantum-resistant signatures
8. **qRC20 Minting**: If approved, corresponding qETH/qBTC is minted to user's QRDX Chain address

Security Properties: - Trustless operation via cryptographic proofs - Quantum-resistant

Dilithium signatures on all bridge operations - Time-lock mechanisms for fraud prevention - Multi-validator consensus for large transfers (>$100K equivalent) - Block height anchoring ensures temporal consistency - Doomsday protocol prevents shielding if quantum threat detected

1.11.3 8.3 Unshielding (Quantum → Classical)

Users can convert qRC20 tokens back to classical assets. **Important**: Unshielding remains available even if the doomsday protocol is active, but shielding (classical → quantum) is blocked.



Step-by-Step Process:

1. **User Initiates Unshielding**: User burns qETH/qBTC on QRDX Chain
2. **Burn Verification**: Validators verify burn transaction with quantum-resistant signatures
3. **Block Height Verification**: Bridge checks that source chain block height matches recorded state
4. **Fraud Proof Window**: 7-day challenge period for disputed transactions
5. **Validator Approval**: 2/3 + 1 validators must sign the unlock transaction
6. **Asset Release**: Locked ETH/BTC released from bridge contract on source chain
7. **Confirmation**: User receives classical assets

Security Measures: - Minimum unshielding delay: 7 days (fraud proof window for amounts >$100K) - Multi-validator approval threshold: 2/3 + 1 (10/15 validators minimum) - Emergency pause mechanism (governance-controlled) - **Doomsday Mode Behavior**: Unshielding continues to function even when doomsday is active

Unshielding During Doomsday: If the doomsday protocol detects a quantum threat, users can still convert qETH → ETH and qBTC → BTC, allowing them to exit to classical chains. However, new shielding (ETH → qETH, BTC → qBTC) is permanently blocked to prevent locking assets in vulnerable classical chains.

1.11.4 8.4 Block Height Recording and State Anchoring

QRDX Bridge maintains a comprehensive record of bridged chain states to ensure consistency and enable fraud detection.

Recorded Information:

For each bridge operation, QRDX records: - **Source Chain ID**: Ethereum (1), Bitcoin (via wrapped), BSC (56), etc. - **Block Height**: Exact block number when assets were locked - **Block Hash**: Cryptographic hash of the source block - **Timestamp**: Unix timestamp of the lock event - **Transaction Hash**: Source chain transaction identifier - **Amount**: Quantity of assets locked/unlocked - **User Address**: Both source and destination addresses

Storage Schema:

```
struct BridgeRecord {
    uint256 sourceChainId;        // Chain identifier
    uint256 blockHeight;          // Block height on source chain
    bytes32 blockHash;            // Block hash for verification
    uint256 timestamp;            // Event timestamp
    bytes32 txHash;               // Source transaction hash
    uint256 amount;               // Amount bridged
    address sourceAddress;        // Address on source chain
    bytes32 qrdxAddress;          // Address on QRDX Chain
    bool isShielding;             // true = classical→quantum, false = quantum→classical
}

mapping(bytes32 => BridgeRecord) public bridgeRecords;
mapping(uint256 => uint256) public latestBlockHeight; // Per-chain tracking
```

Use Cases:

1. **Fraud Detection**: Verify source chain state hasn't been reorganized
2. **Temporal Consistency**: Ensure operations occur in correct sequence
3. **Audit Trail**: Complete history of all bridge operations
4. **Doomsday Verification**: Check if quantum attack occurred before shielding
5. **State Recovery**: Reconstruct bridge state from block heights

Automatic Updates:

Block heights are updated automatically: - Every bridge operation updates the latest block height for that chain - Relayers submit periodic heartbeat updates (every 100 blocks) - Validators can challenge outdated block heights - Emergency updates triggered by significant chain reorganizations

1.11.5 8.5 Doomsday Protocol: Quantum Threat Detection

The **QRDX Doomsday Protocol** is an automated circuit breaker designed to protect users if a quantum computer successfully breaks classical cryptography.

Overview:

QRDX maintains a canary wallet at **doomsday.qrdx.org** with a publicly known public key. If any entity can derive the private key (proving they have a quantum computer capable of breaking ECDSA), they can call a special contract function that immediately halts all classical→quantum bridging.

Technical Implementation:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";
import {ReentrancyGuard} from "@openzeppelin/contracts/security/ReentrancyGuard.sol";

/**
 * @title DoomsdayProtocol
 * @notice Quantum threat detection system using canary ECDSA address
 * @dev If anyone can derive the private key from the public canary key,
 *      they can trigger doomsday, proving quantum computers can break ECDSA.
 *      This immediately halts classical→quantum bridging while allowing
 *      quantum→classical unshielding to continue.
 */
contract DoomsdayProtocol is AccessControl, ReentrancyGuard {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    /// @notice Public canary address published at doomsday.qrdx.org
    /// @dev Private key is unknown - if someone derives it, proves quantum threat
    address public constant CANARY_ADDRESS = 0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb1;

    /// @notice Bounty pool for whoever triggers doomsday
    uint256 public constant BOUNTY_AMOUNT = 1_000_000 ether; // 1M QRDX tokens

    /// @notice Whether doomsday has been triggered
    bool public doomsdayTriggered;

    /// @notice Block number when doomsday was triggered
    uint256 public doomsdayBlockHeight;

    /// @notice Address that triggered doomsday (derived canary key)
    address public triggerAddress;

    /// @notice Timestamp when doomsday was activated
    uint256 public doomsdayTimestamp;

    /// @notice Additional verification data
    bytes32 public verificationHash;

    /// @notice Registered bridge contracts that need notification
    address[] public registeredBridges;
    mapping(address => bool) public isBridgeRegistered;

    event DoomsdayActivated(
        address indexed triggeredBy,
        uint256 blockHeight,
        uint256 timestamp,
        bytes32 verificationHash
```

```
);

event BridgeRegistered(address indexed bridge);
event BridgeUnregistered(address indexed bridge);
event BountyPaid(address indexed recipient, uint256 amount);

error AlreadyTriggered();
error InvalidCaller();
error BountyTransferFailed();
error BridgeAlreadyRegistered();

constructor() {
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(ADMIN_ROLE, msg.sender);
}

/**
 * @notice Trigger doomsday protocol
 * @dev Only callable by signing with CANARY_ADDRESS private key
 *      If you can call this, you've broken ECDSA with a quantum computer!
 * @param proof Additional proof data for verification
 */
function triggerDoomsday(bytes32 proof) external nonReentrant {
    // Verify caller is the canary address (proves private key derived)
    if (msg.sender != CANARY_ADDRESS) revert InvalidCaller();
    if (doomsdayTriggered) revert AlreadyTriggered();

    // Record doomsday activation
    doomsdayTriggered = true;
    doomsdayBlockHeight = block.number;
    doomsdayTimestamp = block.timestamp;
    triggerAddress = msg.sender;
    verificationHash = proof;

    // Emit event for monitoring systems
    emit DoomsdayActivated(
        msg.sender,
        block.number,
        block.timestamp,
        proof
    );

    // Notify all registered bridges to disable shielding
    _notifyBridges();

    // Pay bounty to whoever triggered it (proves quantum capability)
    _payBounty(msg.sender);
}
```

```
/**
 * @notice Check if doomsday is active
 * @return active Whether doomsday protocol has been triggered
 */
function isDoomsdayActive() external view returns (bool active) {
    return doomsdayTriggered;
}

/**
 * @notice Get full doomsday status
 * @return triggered Whether doomsday is active
 * @return blockHeight Block when triggered
 * @return timestamp Time when triggered
 * @return triggeredBy Address that triggered it
 */
function getDoomsdayStatus() external view returns (
    bool triggered,
    uint256 blockHeight,
    uint256 timestamp,
    address triggeredBy
) {
    return (
        doomsdayTriggered,
        doomsdayBlockHeight,
        doomsdayTimestamp,
        triggerAddress
    );
}

/**
 * @notice Register a bridge contract for doomsday notifications
 * @param bridge Address of bridge contract
 */
function registerBridge(address bridge) external onlyRole(ADMIN_ROLE) {
    if (isBridgeRegistered[bridge]) revert BridgeAlreadyRegistered();

    registeredBridges.push(bridge);
    isBridgeRegistered[bridge] = true;

    emit BridgeRegistered(bridge);
}

/**
 * @notice Unregister a bridge contract
 */
function unregisterBridge(address bridge) external onlyRole(ADMIN_ROLE) {
    isBridgeRegistered[bridge] = false;
```

```
    emit BridgeUnregistered(bridge);
}

/**
 * @notice Notify all registered bridges of doomsday
 * @dev Called internally when doomsday is triggered
 */
function _notifyBridges() internal {
    // In production, this would call each bridge's onDoomsday() function
    // to immediately disable classical→quantum shielding
    for (uint256 i = 0; i < registeredBridges.length; i++) {
        if (isBridgeRegistered[registeredBridges[i]]) {
            // Bridge interface: IDoomsdayAware(bridge).onDoomsday();
            // Omitted for brevity - actual implementation would call interface
        }
    }
}

/**
 * @notice Pay bounty to doomsday trigger
 * @dev Rewards whoever proves quantum computers can break ECDSA
 */
function _payBounty(address recipient) internal {
    // In production, this transfers QRDX tokens from treasury
    // (bool success, ) = recipient.call{value: BOUNTY_AMOUNT}("");
    // if (!success) revert BountyTransferFailed();

    emit BountyPaid(recipient, BOUNTY_AMOUNT);
}

/**
 * @notice Get list of registered bridges
 */
function getRegisteredBridges() external view returns (address[] memory) {
    return registeredBridges;
}

/**
 * @notice Get canary public key info for verification
 * @return canaryAddress The address to monitor
 * @return bounty The bounty amount for triggering
 */
function getCanaryInfo() external pure returns (
    address canaryAddress,
    uint256 bounty
) {
    return (CANARY_ADDRESS, BOUNTY_AMOUNT);
}
```

```
}
```

Doomsday Website: doomsday.qrdx.org

The doomsday website displays: - **Canary Public Key**: The ECDSA public key (openly published) - **Canary Address**: Ethereum address derived from the public key - **Challenge**: "Derive the private key and call triggerDoomsday() to win $1M" - **Status**: Real-time status of doomsday protocol (ACTIVE/INACTIVE) - **Block Height**: Last recorded block heights of all bridged chains - **Bounty Pool**: Reward for anyone who triggers doomsday (proving quantum threat)

Behavior After Doomsday Activation:

Operation	Before Doomsday	After Doomsday
ETH → qETH	Allowed	**BLOCKED**
BTC → qBTC	Allowed	**BLOCKED**
qETH → ETH	Allowed	**STILL ALLOWED**
qBTC → BTC	Allowed	**STILL ALLOWED**
QRDX Chain Trading	Normal	**NORMAL**
qETH/qBTC Trading	Normal	**NORMAL**

Rationale:

Once doomsday is triggered, it means quantum computers can break classical ECDSA. At this point:

1. **Shielding Blocked**: Prevents users from locking assets on vulnerable classical chains
2. **Unshielding Continues**: Allows users to exit to classical chains if desired
3. **QRDX Trading Unaffected**: qETH and qBTC remain secure and tradeable on QRDX Chain
4. **Permanent Protection**: Assets already shielded remain protected by post-quantum cryptography

Bounty Incentive:

A $1,000,000 QRDX bounty is allocated to whoever successfully triggers the doomsday protocol. This incentivizes research institutions, quantum computing labs, and security researchers to immediately alert the ecosystem when quantum computers reach sufficient capability.

False Positive Protection:

The doomsday trigger requires: - Valid signature from the canary address (proves key was derived) - On-chain transaction (public and verifiable) - Irreversible (cannot be undone)

This ensures only genuine quantum capability can trigger the protocol, not false alarms or social engineering.

1.11.6 8.6 Bridge Security Model

Components: 1. **Relayer Network**: Decentralized operators monitoring both chains 2. **Validator Set**: QRDX Chain validators with bonded stake 3. **Merkle Proof System**: Efficient proof verification 4. **Fraud Proof Mechanism**: Challenge period for disputed transfers 5. **Block**

Height Anchoring: Temporal consistency verification 6. **Doomsday Protocol**: Quantum threat detection and response

Collateral Requirements: - Validators must bond QRDX tokens (minimum 100,000 QRDX) - Slashing conditions: Invalid proofs, downtime, malicious behavior, ignoring doomsday - Insurance fund: 5% of protocol revenue allocated to bridge insurance - Doomsday bounty pool: $1M QRDX reserved

---

1.12 9. qRC20 Token Standard

1.12.1 9.1 Specification

qRC20 is the quantum-resistant token standard for QRDX Chain, designed for compatibility with ERC-20 tooling while enforcing post-quantum security.

Interface:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {IERC20Metadata} from "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol

interface IqRC20 is IERC20, IERC20Metadata {
    /// @notice Emitted when tokens are transferred with quantum-resistant proof
    event TransferWithProof(address indexed from, address indexed to, uint256 value, bytes32 pr

    /// @notice Emitted when tokens are minted from bridge
    event BridgeMint(address indexed to, uint256 amount, uint256 sourceChainId, bytes32 sourceT

    /// @notice Emitted when tokens are burned for bridge
    event BridgeBurn(address indexed from, uint256 amount, address destinationAddress);

    /// @notice Emitted when doomsday trading preference is updated
    event DoomsdayTradingPreferenceUpdated(bool allowTrading);

    /// @notice Standard ERC-20 functions inherited from IERC20
    // function totalSupply() external view returns (uint256);
    // function balanceOf(address account) external view returns (uint256);
    // function transfer(address recipient, uint256 amount) external returns (bool);
    // function allowance(address owner, address spender) external view returns (uint256);
    // function approve(address spender, uint256 amount) external returns (bool);
    // function transferFrom(address sender, address recipient, uint256 amount) external return

    /// @notice Quantum-resistant transfer with Dilithium signature verification
    /// @param recipient The address receiving the tokens
    /// @param amount The amount of tokens to transfer
    /// @param dilithiumSignature The post-quantum signature proving authorization
```

```
    /// @return success Whether the transfer succeeded
    function transferWithProof(
        address recipient,
        uint256 amount,
        bytes calldata dilithiumSignature
    ) external returns (bool success);

    /// @notice Get bridge metadata for this token
    /// @return sourceChainId The chain ID where the original token exists
    /// @return sourceToken The address of the original token
    /// @return totalShielded The total amount of tokens currently shielded
    function bridgeInfo() external view returns (
        uint256 sourceChainId,
        address sourceToken,
        uint256 totalShielded
    );

    /// @notice Check if this token should be traded after doomsday
    /// @dev This is an advisory flag that trading clients CAN respect but are not forced to
    /// @dev Returns true if token is safe to trade post-doomsday (e.g., qETH, qBTC)
    /// @dev Returns false if token represents classical assets that may be compromised
    /// @return shouldTrade Whether trading is recommended after doomsday activation
    function shouldTradeAfterDoomsday() external view returns (bool shouldTrade);

    /// @notice Mint tokens (only callable by bridge contract)
    /// @param to The address receiving minted tokens
    /// @param amount The amount to mint
    function mint(address to, uint256 amount) external;

    /// @notice Burn tokens for unshielding
    /// @param from The address burning tokens
    /// @param amount The amount to burn
    function burn(address from, uint256 amount) external;
}
```

1.12.2 9.2 Key Features

Quantum-Resistant Transfers: - All transfers require Dilithium signatures - Address derivation uses post-quantum key derivation functions - Support for quantum-resistant multi-sig

Bridge Integration: - Native bridging metadata for cross-chain tracking - Automatic mint/burn on shield/unshield operations - Source chain provenance tracking

Doomsday Trading Advisory: - Each qRC20 token declares whether it should be traded after doomsday via `shouldTradeAfterDoomsday()` - This is an **advisory flag** that trading clients and DEX interfaces can respect but are not forced to honor - Tokens backed by quantum-resistant assets (qETH, qBTC) return `true` - safe to trade post-doomsday - Tokens backed by classical chain assets that may be compromised return `false` - trading not recommended - Ultimate decision rests with users and trading platforms - the protocol does not enforce restrictions

Gas Optimization: - Batch transfers for reduced costs - Optimized storage layout - EIP-2612 permit functionality (with Dilithium)

1.12.3 9.3 Post-Doomsday Trading Behavior

When the doomsday protocol is triggered, the `shouldTradeAfterDoomsday()` flag provides guidance to ecosystem participants:

Recommended Behavior (Client Implementation):

```
// Example DEX client logic
async function shouldShowTradingWarning(tokenAddress: string): Promise<boolean> {
    const qrc20 = new Contract(tokenAddress, IqRC20_ABI);
    const doomsday = new Contract(DOOMSDAY_ADDRESS, IDoomsdayProtocol_ABI);

    const isDoomsdayActive = await doomsday.isDoomsdayActive();

    if (!isDoomsdayActive) {
        return false; // No warning needed before doomsday
    }

    // Check token's advisory flag
    const shouldTrade = await qrc20.shouldTradeAfterDoomsday();

    if (!shouldTrade) {
        // Show warning to user, but allow trade if they choose
        return true;
    }

    return false;
}
```

Trading Client Options:

1. **Strict Mode**: Disable trading UI for tokens that return `false`
2. **Warning Mode**: Show warning but allow users to proceed (recommended)
3. **Permissionless Mode**: Ignore flag entirely (possible but not recommended)

1.12.4 9.4 Example Tokens

Token	Symbol	Source	Backing	Supply	Post-Doomsday Trading
Quantum Ether	qETH	Ethereum	1:1 ETH locked	Dynamic	Recommended (`true`)
Quantum Bitcoin	qBTC	Bitcoin (via WBTC)	1:1 WBTC locked	Dynamic	Recommended (`true`)
Quantum USD Coin	qUSDC	Ethereum	1:1 USDC locked	Dynamic	Recommended (`true`)

Token	Symbol	Source	Backing	Supply	Post-Doomsday Trading
Quantum Tether	qUSDT	Ethereum	1:1 USDT locked	Dynamic	Recommended (`true`)
QRDX Token	QRDX	Native	N/A	Fixed (100M)	Recommended (`true`)

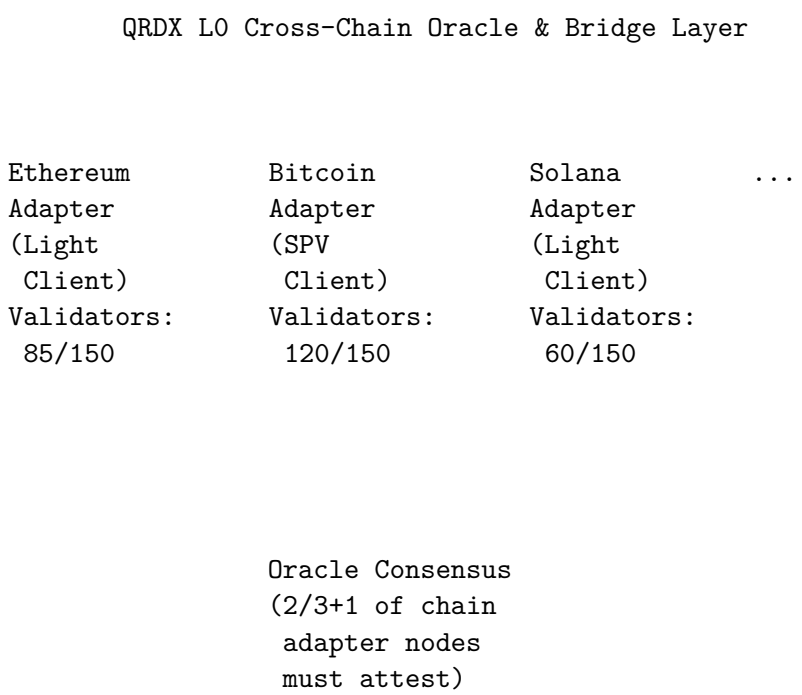
Rationale for Flags:

- **qETH, qBTC, qUSDC, qUSDT**: Return `true` because these are quantum-resistant tokens backed by locked classical assets. Even if classical chains are compromised, the qRC20 versions remain secure with post-quantum cryptography.
- **Theoretical Classical Wrappers**: If someone created a token representing an unsecured classical chain asset without proper locking, it should return `false` to warn users.
- **Native QRDX**: Returns `true` as it's a native quantum-resistant token with no classical backing dependency.

1.13 10. Cross-Chain Oracle & Bridge Infrastructure

1.13.1 10.1 Architecture

In QRDX's L0 model, cross-chain bridging and oracle data provision are unified. Validators running chain adapter components attest to external chain state, and bridge operations are processed as native protocol transactions rather than through a separate relayer network.



```
Bridge Core
(Lock/Unlock
 qRC20 Mint/
 Burn Logic)
```

1.13.2 10.2 Oracle Attestation Model

Validators running a chain adapter continuously track the external chain's state. At each QRDX block, validators with the relevant adapter submit attestations:

Attestation Contents: - External chain ID - Latest confirmed block height - Block hash - State root (for EVM chains) - Timestamp - Dilithium signature of the attestation

Consensus Rule: An external chain state is considered finalized on QRDX when 2/3+1 of validators running that chain's adapter agree on the same block height and hash. This is the oracle consensus, and it runs in parallel with QRDX block consensus.

Oracle Rewards: Validators earn oracle fees proportional to the number of chain adapters they run and the accuracy of their attestations. A validator running 3 adapters earns roughly 3x the oracle rewards of one running a single adapter.

1.13.3 10.3 Ethereum Bridge

Lock Contract (Ethereum):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import {ReentrancyGuard} from "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import {Pausable} from "@openzeppelin/contracts/security/Pausable.sol";
import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";

contract QRDXBridge is ReentrancyGuard, Pausable, AccessControl {
    using SafeERC20 for IERC20;

    bytes32 public constant RELAYER_ROLE = keccak256("RELAYER_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    /// @notice Minimum lock amount to prevent dust attacks
    uint256 public constant MIN_LOCK_AMOUNT = 0.001 ether;

    /// @notice Nonce for preventing replay attacks
    uint256 public unlockNonce;

    /// @notice Mapping of user balances locked in the bridge
```

```
mapping(address => mapping(address => uint256)) public lockedBalances;

/// @notice Mapping to prevent double-processing of mints
mapping(bytes32 => bool) public processedMints;

/// @notice Block height records for temporal consistency
mapping(uint256 => BlockHeightRecord) public blockHeightRecords;

struct BlockHeightRecord {
    uint256 blockHeight;
    bytes32 blockHash;
    uint256 timestamp;
    bool exists;
}

struct ValidatorProof {
    bytes32 messageHash;
    bytes[] dilithiumSignatures;
    address[] signers;
    uint256 nonce;
}

event AssetLocked(
    address indexed user,
    address indexed token,
    uint256 amount,
    bytes32 indexed qrdxAddress,
    uint256 blockHeight,
    bytes32 txId
);

event AssetUnlocked(
    address indexed recipient,
    address indexed token,
    uint256 amount,
    uint256 sourceBlockHeight,
    uint256 nonce
);

error InsufficientAmount();
error InvalidProof();
error InsufficientBalance();
error TransferFailed();
error AlreadyProcessed();

constructor() {
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(ADMIN_ROLE, msg.sender);
```

```
}

/**
 * @notice Lock ETH for quantum shielding (ETH → qETH)
 * @param qrdxAddress The recipient address on QRDX Chain (bytes32 for quantum addresses)
 * @dev Records block height for temporal consistency and fraud detection
 */
function lockETH(bytes32 qrdxAddress)
    external
    payable
    nonReentrant
    whenNotPaused
{
    if (msg.value < MIN_LOCK_AMOUNT) revert InsufficientAmount();

    // Generate unique transaction ID
    bytes32 txId = keccak256(
        abi.encodePacked(
            msg.sender,
            address(0), // ETH
            msg.value,
            qrdxAddress,
            block.number,
            block.timestamp
        )
    );

    // Record block height for this operation
    blockHeightRecords[block.number] = BlockHeightRecord({
        blockHeight: block.number,
        blockHash: blockhash(block.number - 1),
        timestamp: block.timestamp,
        exists: true
    });

    // Update user's locked balance
    lockedBalances[msg.sender][address(0)] += msg.value;

    emit AssetLocked(
        msg.sender,
        address(0), // ETH represented as address(0)
        msg.value,
        qrdxAddress,
        block.number,
        txId
    );
}
```

```
/**
 * @notice Lock ERC20 tokens for quantum shielding
 * @param token The ERC20 token address
 * @param amount The amount to lock
 * @param qrdxAddress The recipient address on QRDX Chain
 */
function lockERC20(
    address token,
    uint256 amount,
    bytes32 qrdxAddress
) external nonReentrant whenNotPaused {
    if (amount == 0) revert InsufficientAmount();

    bytes32 txId = keccak256(
        abi.encodePacked(
            msg.sender,
            token,
            amount,
            qrdxAddress,
            block.number,
            block.timestamp
        )
    );

    // Record block height
    if (!blockHeightRecords[block.number].exists) {
        blockHeightRecords[block.number] = BlockHeightRecord({
            blockHeight: block.number,
            blockHash: blockhash(block.number - 1),
            timestamp: block.timestamp,
            exists: true
        });
    }

    // Transfer tokens from user to bridge
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

    // Update user's locked balance
    lockedBalances[msg.sender][token] += amount;

    emit AssetLocked(
        msg.sender,
        token,
        amount,
        qrdxAddress,
        block.number,
        txId
    );
```

```
}

/**
 * @notice Unlock ETH for unshielding (qETH → ETH)
 * @param recipient The recipient address on Ethereum
 * @param amount The amount to unlock
 * @param proof Quantum-resistant proof from QRDX validators
 * @dev This function works even after doomsday is triggered
 */
function unlockETH(
    address recipient,
    uint256 amount,
    ValidatorProof calldata proof
) external nonReentrant onlyRole(RELAYER_ROLE) {
    // Verify proof and nonce
    if (proof.nonce != unlockNonce) revert InvalidProof();
    if (!_verifyValidatorProof(recipient, address(0), amount, proof)) {
        revert InvalidProof();
    }

    bytes32 proofHash = keccak256(abi.encode(proof));
    if (processedMints[proofHash]) revert AlreadyProcessed();

    // Mark as processed
    processedMints[proofHash] = true;
    unlockNonce++;

    // Transfer ETH
    (bool success, ) = payable(recipient).call{value: amount}("");
    if (!success) revert TransferFailed();

    emit AssetUnlocked(
        recipient,
        address(0),
        amount,
        block.number,
        proof.nonce
    );
}

/**
 * @notice Verify quantum-resistant validator proof
 * @dev Implements Dilithium signature verification with 2/3+1 threshold
 */
function _verifyValidatorProof(
    address recipient,
    address token,
    uint256 amount,
```

```
    ValidatorProof calldata proof
) internal view returns (bool) {
    // Reconstruct message hash
    bytes32 expectedHash = keccak256(
        abi.encodePacked(
            recipient,
            token,
            amount,
            proof.nonce,
            block.chainid
        )
    );

    if (proof.messageHash != expectedHash) return false;

    // Verify we have enough signatures (2/3 + 1 threshold)
    uint256 requiredSigs = (proof.signers.length * 2) / 3 + 1;
    if (proof.dilithiumSignatures.length < requiredSigs) return false;

    // Verify each Dilithium signature (simplified - actual implementation uses precompile)
    for (uint256 i = 0; i < proof.dilithiumSignatures.length; i++) {
        if (!_verifyDilithiumSignature(
            proof.messageHash,
            proof.dilithiumSignatures[i],
            proof.signers[i]
        )) {
            return false;
        }
    }

    return true;
}

/**
 * @notice Verify a single Dilithium signature
 * @dev In production, this calls a precompile for efficient PQC verification
 */
function _verifyDilithiumSignature(
    bytes32 messageHash,
    bytes memory signature,
    address signer
) internal view returns (bool) {
    // This would call a precompile in production:
    // return DILITHIUM_PRECOMPILE.verify(messageHash, signature, signer);
    // For now, placeholder:
    return signature.length > 0 && signer != address(0);
}
```

```
    /**
     * @notice Emergency pause function
     */
    function pause() external onlyRole(ADMIN_ROLE) {
        _pause();
    }

    /**
     * @notice Unpause the contract
     */
    function unpause() external onlyRole(ADMIN_ROLE) {
        _unpause();
    }

    /**
     * @notice Get the latest recorded block height
     */
    function getLatestBlockHeight() external view returns (uint256) {
        return block.number;
    }

    receive() external payable {}
}
```

Mint Contract (QRDX Chain):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {IqRC20} from "./interfaces/IqRC20.sol";
import {IDoomsdayProtocol} from "./interfaces/IDoomsdayProtocol.sol";
import {ReentrancyGuard} from "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import {Pausable} from "@openzeppelin/contracts/security/Pausable.sol";
import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";
import {MerkleProof} from "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract QRDXBridgeMinter is ReentrancyGuard, Pausable, AccessControl {
    bytes32 public constant RELAYER_ROLE = keccak256("RELAYER_ROLE");
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    /// @notice Minimum confirmations required on source chain
    uint256 public constant MIN_CONFIRMATIONS = 12;

    /// @notice Doomsday protocol contract
    IDoomsdayProtocol public immutable doomsdayProtocol;

    /// @notice Mapping of chain IDs to their latest recorded block heights
    mapping(uint256 => uint256) public chainBlockHeights;
```

```
/// @notice Mapping of qRC20 tokens to their configurations
mapping(address => TokenConfig) public tokenConfigs;

/// @notice Processed mint records to prevent double-minting
mapping(bytes32 => MintRecord) public mintRecords;

/// @notice Merkle roots for each source chain block
mapping(uint256 => mapping(uint256 => bytes32)) public blockMerkleRoots;

struct TokenConfig {
    uint256 sourceChainId;
    address sourceToken;
    bool active;
    uint256 minAmount;
    uint256 maxAmount;
}

struct MintRecord {
    uint256 sourceChainId;
    uint256 sourceBlockHeight;
    bytes32 sourceBlockHash;
    bytes32 sourceTxHash;
    uint256 amount;
    uint256 timestamp;
    bool processed;
}

struct MintProof {
    bytes32[] merkleProof;
    bytes[] dilithiumSignatures;
    address[] validators;
    bytes32 merkleRoot;
}

event qTokenMinted(
    address indexed recipient,
    address indexed token,
    uint256 amount,
    uint256 sourceChainId,
    uint256 sourceBlockHeight,
    bytes32 indexed sourceTxHash
);

event UnshieldRequested(
    address indexed from,
    address indexed token,
    address destinationAddress,
    uint256 amount,
```

```
    uint256 blockNumber,
    bytes32 requestId
);

event BlockHeightUpdated(
    uint256 indexed chainId,
    uint256 oldHeight,
    uint256 newHeight
);

error DoomsdayActive();
error InvalidProof();
error StaleBlockHeight();
error AlreadyProcessed();
error InvalidAmount();
error TokenNotConfigured();
error InsufficientValidatorSignatures();

constructor(address _doomsdayProtocol) {
    doomsdayProtocol = IDoomsdayProtocol(_doomsdayProtocol);
    _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
    _grantRole(ADMIN_ROLE, msg.sender);
}

/**
 * @notice Mint qETH from locked Ethereum ETH
 * @dev BLOCKED if doomsday protocol is active
 * @param recipient The address receiving minted qETH
 * @param token The qRC20 token to mint
 * @param amount The amount to mint
 * @param sourceBlockHeight Block height on source chain where lock occurred
 * @param sourceBlockHash Block hash for verification
 * @param sourceTxHash Transaction hash on source chain
 * @param proof Merkle proof and validator signatures
 */
function mintFromSource(
    address recipient,
    address token,
    uint256 amount,
    uint256 sourceBlockHeight,
    bytes32 sourceBlockHash,
    bytes32 sourceTxHash,
    MintProof calldata proof
) external nonReentrant whenNotPaused onlyRole(RELAYER_ROLE) {
    TokenConfig memory config = tokenConfigs[token];

    // CHECK 1: Token is configured
    if (!config.active) revert TokenNotConfigured();
```

```
// CHECK 2: Doomsday not triggered (shielding disabled if active)
if (doomsdayProtocol.isDoomsdayActive()) revert DoomsdayActive();

// CHECK 3: Amount within bounds
if (amount < config.minAmount || amount > config.maxAmount) {
    revert InvalidAmount();
}

// CHECK 4: Block height consistency (prevent old/reorg attacks)
if (sourceBlockHeight < chainBlockHeights[config.sourceChainId]) {
    revert StaleBlockHeight();
}

// CHECK 5: Not already processed
bytes32 mintId = keccak256(
    abi.encodePacked(sourceTxHash, recipient, amount)
);
if (mintRecords[mintId].processed) revert AlreadyProcessed();

// CHECK 6: Verify Merkle proof
bytes32 leaf = keccak256(
    abi.encodePacked(
        recipient,
        token,
        amount,
        sourceTxHash,
        sourceBlockHeight
    )
);

if (!MerkleProof.verify(proof.merkleProof, proof.merkleRoot, leaf)) {
    revert InvalidProof();
}

// CHECK 7: Verify validator signatures (2/3 + 1 threshold)
if (!_verifyValidatorSignatures(
    leaf,
    proof.dilithiumSignatures,
    proof.validators
)) {
    revert InsufficientValidatorSignatures();
}

// Record this mint operation
mintRecords[mintId] = MintRecord({
    sourceChainId: config.sourceChainId,
    sourceBlockHeight: sourceBlockHeight,
```

```
        sourceBlockHash: sourceBlockHash,
        sourceTxHash: sourceTxHash,
        amount: amount,
        timestamp: block.timestamp,
        processed: true
    });

    // Update tracked block height if newer
    if (sourceBlockHeight > chainBlockHeights[config.sourceChainId]) {
        uint256 oldHeight = chainBlockHeights[config.sourceChainId];
        chainBlockHeights[config.sourceChainId] = sourceBlockHeight;
        emit BlockHeightUpdated(config.sourceChainId, oldHeight, sourceBlockHeight);
    }

    // Store merkle root for this block
    blockMerkleRoots[config.sourceChainId][sourceBlockHeight] = proof.merkleRoot;

    // Mint qRC20 tokens
    IqRC20(token).mint(recipient, amount);

    emit qTokenMinted(
        recipient,
        token,
        amount,
        config.sourceChainId,
        sourceBlockHeight,
        sourceTxHash
    );
}

/**
 * @notice Burn qETH to initiate unshielding (qETH → ETH)
 * @dev This works even after doomsday is triggered
 * @param token The qRC20 token to burn
 * @param amount The amount to burn
 * @param destinationAddress The recipient address on the source chain
 */
function burnForUnshielding(
    address token,
    uint256 amount,
    address destinationAddress
) external nonReentrant whenNotPaused {
    TokenConfig memory config = tokenConfigs[token];
    if (!config.active) revert TokenNotConfigured();
    if (amount == 0) revert InvalidAmount();

    // Generate unique request ID
    bytes32 requestId = keccak256(
```

```
        abi.encodePacked(
            msg.sender,
            token,
            amount,
            destinationAddress,
            block.number,
            block.timestamp
        )
    );

    // Burn qRC20 tokens (works regardless of doomsday status)
    IqRC20(token).burn(msg.sender, amount);

    // Emit event for validators to process unlock on source chain
    emit UnshieldRequested(
        msg.sender,
        token,
        destinationAddress,
        amount,
        block.number,
        requestId
    );
}

/**
 * @notice Verify quantum-resistant validator signatures
 * @dev Requires 2/3 + 1 threshold of valid Dilithium signatures
 */
function _verifyValidatorSignatures(
    bytes32 messageHash,
    bytes[] calldata signatures,
    address[] calldata validators
) internal view returns (bool) {
    if (signatures.length != validators.length) return false;

    uint256 requiredSigs = (validators.length * 2) / 3 + 1;
    if (signatures.length < requiredSigs) return false;

    // In production, this would call DILITHIUM precompile
    // For each signature, verify: DILITHIUM.verify(messageHash, sig, validator)

    return true; // Placeholder
}

/**
 * @notice Configure a qRC20 token
 */
function configureToken(
```

```
        address token,
        uint256 sourceChainId,
        address sourceToken,
        uint256 minAmount,
        uint256 maxAmount
    ) external onlyRole(ADMIN_ROLE) {
        tokenConfigs[token] = TokenConfig({
            sourceChainId: sourceChainId,
            sourceToken: sourceToken,
            active: true,
            minAmount: minAmount,
            maxAmount: maxAmount
        });
    }

    /**
     * @notice Check if shielding is currently allowed
     */
    function isShieldingEnabled() external view returns (bool) {
        return !doomsdayProtocol.isDoomsdayActive();
    }

    /**
     * @notice Get recorded block height for a specific chain
     */
    function getChainBlockHeight(uint256 chainId) external view returns (uint256) {
        return chainBlockHeights[chainId];
    }

    /**
     * @notice Emergency pause
     */
    function pause() external onlyRole(ADMIN_ROLE) {
        _pause();
    }

    function unpause() external onlyRole(ADMIN_ROLE) {
        _unpause();
    }
}
        bytes calldata dilithiumSignature
    ) external {
        require(verifyMerkleProof(merkleProof), "Invalid Merkle proof");
        require(verifyDilithiumSignature(dilithiumSignature), "Invalid signature");
        qETH.mint(recipient, amount);
    }
}
```

1.13.4 10.4 Bitcoin Bridge

Bitcoin integration uses a threshold Dilithium signature scheme. Validators running the Bitcoin adapter collectively manage a PQ multisig address on QRDX that controls locked BTC (via the Bitcoin adapter's ability to construct and broadcast Bitcoin transactions):

1. **PQ Federation Multisig**: 15-of-23 threshold Dilithium multi-signature (protocol-level, not a separate federation)
2. **SPV Proofs**: Bitcoin block headers verified on QRDX Chain
3. **HTLC Adapters**: Hash Time-Locked Contracts for atomic swaps
4. **Block Height Tracking**: Bitcoin block heights recorded for all BTC → qBTC operations
5. **Doomsday Compliance**: BTC → qBTC shielding blocked if doomsday triggered

Bitcoin Shielding Process:

1. User sends BTC to federation multisig address
2. Federation waits for 6 confirmations
3. Bitcoin block height recorded on QRDX Chain
4. Doomsday protocol checked (must be inactive)
5. SPV proof generated and verified
6. qBTC minted to user's QRDX address

Bitcoin Unshielding Process:

1. User burns qBTC on QRDX Chain
2. Unshield request submitted with BTC address
3. 15-of-23 validators approve (quantum-resistant signatures)
4. BTC released from multisig (works even during doomsday)
5. User receives BTC on Bitcoin network

1.13.5 10.5 Security Measures

Multi-Layer Verification (L0 Oracle Model): 1. Source chain confirmation (12+ blocks for Ethereum, 6+ for Bitcoin) 2. Oracle consensus (2/3+1 of chain adapter validators must attest) 3. Validator Dilithium signature threshold (2/3 + 1) 4. Fraud proof challenge period (7 days for large amounts)

Economic Security: - Total validator bonded stake: $100M+ equivalent in QRDX - Slashing penalty: 50% of bonded stake for provable fraud - Insurance fund: $10M+ reserved for bridge exploits

---

1.14 11. Consensus Mechanism & Validator Requirements

1.14.1 11.1 Quantum-Resistant Proof-of-Stake (QR-PoS)

QRDX uses a modified Proof-of-Stake consensus with **mandatory post-quantum cryptographic identity**. No validator may participate using classical cryptography. Every block proposal, attestation, oracle attestation, and cross-chain bridge signature uses CRYSTALS-Dilithium exclusively.

Key Components: 1. **Validator Set**: 150 active validators (expandable via governance) 2. **Staking Requirement**: Minimum 100,000 QRDX per validator 3. **Identity Requirement**: Dilithium3 keypair registered on-chain via `@`-schema address (Section 5) 4. **Block Proposal**: Pseudo-random selection weighted by stake 5. **Finality**: Single-slot finality via BFT consensus 6. **Oracle Duty**: Validators attest to external chain states using their chain adapter components 7. **Exchange Settlement**: Validators execute exchange engine state transitions as part of block processing

1.14.2 11.2 Validator Roles (Unified)

In the L0 model, validators perform all critical protocol roles simultaneously:

Role	Description	Requirement
Block Producer	Proposes blocks containing transactions, exchange operations, and oracle attestations	Dilithium keypair, minimum stake
Attestor	Attests to block validity using Dilithium signatures	Dilithium keypair, online
Oracle Attestor	Attests to external chain states (block heights, state roots)	Chain adapter(s) running
Bridge Operator	Signs cross-chain transactions using threshold Dilithium	Chain adapter + bridge signer role
Exchange Settler	Processes exchange engine state transitions (swaps, fills, LP changes)	Part of block processing
OracleTx Relay	Validates, broadcasts, and attests to OracleTransaction inner transactions on target chains	Chain adapter + OracleTx processing

There is no separate relayer network, no separate oracle network, and no separate bridge operator set. The validator set is the single source of truth for all operations, all authenticated with PQ cryptography.

1.14.3 11.3 Validator Selection

```
Selection Probability = (Validator Stake / Total Staked) * Uptime Factor * Adapter Bonus
Uptime Factor = min(1.0, Blocks Signed / Expected Blocks)
Adapter Bonus = 1.0 + (0.05 * number_of_chain_adapters)
```

Validators running more chain adapters get a slight selection probability bonus, incentivizing broader cross-chain coverage.

1.14.4 11.4 Block Production

Timeline: - Slot Duration: 2 seconds - Block Proposal: Validator signature (Dilithium) - Attestation Period: 1 second - Finality: 1 second (after 2/3+ attestations)

Block Structure:

```
Block {
    header: {
        number: uint64,
        parentHash: bytes32,
        stateRoot: bytes32,
        transactionsRoot: bytes32,
        exchangeStateRoot: bytes32,     // NEW: root of exchange engine state
        oracleAttestationsRoot: bytes32, // NEW: root of oracle attestations
        timestamp: uint64,
        validatorPubKey: bytes (Dilithium),
        validatorSignature: bytes (Dilithium),
        validatorNodeId: string (@-schema)
    },
    transactions: Transaction[],
    oracleTransactions: OracleTransaction[], // Cross-chain sub-tx envelopes (Section 3.8)
    exchangeOps: ExchangeOperation[],    // Swaps, orders, LP changes
    oracleAttestations: OracleAttestation[], // Cross-chain state attestations
    attestations: Attestation[]
}
```

1.14.5 11.5 Finality Gadget

QRDX implements a BFT-style finality mechanism:

1. Validator proposes block (including exchange ops and oracle attestations)
2. Other validators verify block validity, exchange state transitions, and oracle attestations
3. Block becomes final when 2/3+ of stake has attested
4. Finalized blocks cannot be reverted

Safety Guarantee: As long as >2/3 of validators are honest, no conflicting blocks can be finalized.

1.14.6 11.6 Slashing Conditions

Validators are slashed for: - **Double-signing**: Proposing two blocks at same height (50% stake) - **Invalid attestation**: Attesting to provably invalid block (30% stake) - **Downtime**: Missing >10% of attestations in epoch (5% stake) - **Bridge fraud**: Submitting false bridge proofs (100% stake) - **Oracle fraud**: Submitting false oracle attestations for external chain states (50% stake) - **Exchange manipulation**: Submitting invalid exchange state transitions or front-running through block manipulation (75% stake) - **Classical key usage**: Attempting to participate in consensus with non-Dilithium keys (immediate ejection, 100% stake) - **OracleTx relay fraud**: Falsely attesting to OracleTransaction confirmation or selectively censoring OracleTransactions (75% stake) - **Doomsday violation**: Processing shielding transactions after doomsday activation (100% stake)

1.15 12. Tokenomics

1.15.1 12.1 QRDX Token

Token Name: QRDX
Total Supply: 100,000,000 QRDX (fixed)
Token Standard: qRC20 (native to QRDX Chain)
Decimals: 18

1.15.2 12.2 Token Distribution

Allocation	Amount	Percentage	Vesting
Public Sale	20,000,000	20%	Immediate
Team & Advisors	15,000,000	15%	4-year linear, 1-year cliff
Treasury	20,000,000	20%	Governance-controlled
Ecosystem Fund	15,000,000	15%	5-year release schedule
Liquidity Mining	20,000,000	20%	4-year emission curve
Early Backers	10,000,000	10%	2-year linear, 6-month cliff

1.15.3 12.3 Token Utility

Staking: - Validator staking (minimum 100,000 QRDX) - Delegated staking (minimum 100 QRDX) - Pool creation staking (10,000-100,000 QRDX depending on pool type) - Staking rewards: 5-12% APY (dynamic based on total staked)

Pool Creation: - Standard Pool: 10,000 QRDX staked (returned on pool closure) - Bootstrap Pool: 25,000 QRDX staked (includes 30-day incentive) - Subsidized Pool: 5,000 QRDX burned permanently (community-owned) - Institutional Pool: 100,000 QRDX staked (premium features)

Oracle & Chain Adapter Rewards: - Validators earn oracle fees per chain adapter operated - Attestation accuracy bonuses for consistent cross-chain state reporting - Additional rewards for running less-common chain adapters (incentivizes coverage)

Governance: - Protocol parameter adjustments - Treasury fund allocation - Validator set changes - Bridge and oracle security parameters - Exchange engine configuration

Fee Discounts: - Trading fee discounts (up to 50% with sufficient stake) - Bridge fee discounts (up to 30%) - Priority transaction inclusion

Liquidity Mining: - LP rewards for QRDX pairs - Incentivized pools for new qRC20 assets - Bootstrap liquidity programs

1.15.4 12.4 Fee Economics

Transaction Fees: - Base fee: Burned (deflationary mechanism) - Priority fee: Paid to validators

Exchange Trading Fees (Protocol-Native): - 70% to liquidity providers - 15% to pool creator (stake reward) - 10% to protocol treasury - 5% to validator oracle/settlement rewards

Order Book Fees: - Maker: 0.02% (limit orders) - Taker: 0.05% (market orders)

Bridge Fees: - Shielding: 0.1% of amount (minimum $1) - Unshielding: 0.1% of amount (minimum $1) - Fees distributed: 50% burned, 30% to validators, 20% to insurance fund

1.15.5 12.5 Emission Schedule

Liquidity mining rewards follow a decreasing emission curve:

```
Year 1: 8,000,000 QRDX
Year 2: 6,000,000 QRDX
Year 3: 4,000,000 QRDX
Year 4: 2,000,000 QRDX
Total:  20,000,000 QRDX
```

1.15.6 12.6 Deflationary Mechanics

Token Burns: - Base transaction fees (100% burned) - 50% of bridge fees - Subsidized pool creation burns (5,000 QRDX per subsidized pool) - Protocol revenue buybacks (quarterly)

Projected Burn Rate: 1-3% of total supply annually, depending on network activity.

1.16 13. Governance Model

1.16.1 13.1 Overview

QRDX implements on-chain governance allowing token holders to propose and vote on protocol changes.

Governance Scope: - Protocol parameter adjustments (fees, limits, etc.) - Treasury fund allocation and spending - Validator set management - Smart contract upgrades - Ecosystem grants and partnerships - Exchange engine parameters (fee tiers, pool creation requirements) - Oracle configuration (chain adapter requirements, attestation thresholds) - Node identity and bootstrap configuration

1.16.2 13.2 Proposal Process

Stages: 1. **Discussion**: Forum discussion (minimum 3 days) 2. **Temperature Check**: Informal vote (minimum 1M QRDX support) 3. **Formal Proposal**: On-chain proposal submission (requires 10M QRDX or delegation) 4. **Voting Period**: 7-day voting window 5. **Timelock**: 2-day execution delay 6. **Execution**: Automatic on-chain execution

1.16.3 13.3 Voting Mechanics

Voting Power: - 1 QRDX = 1 vote - Delegated voting supported - Vote locking for increased weight (optional)

Quorum Requirements: - Minimum participation: 10% of circulating supply - Approval threshold: 60% of votes cast (for parameter changes) - Supermajority: 75% of votes cast (for protocol upgrades)

Vote Types: - For - Against - Abstain (counts toward quorum)

1.16.4 13.4 Timelock Contract

All governance actions pass through a timelock contract with quantum-resistant signatures:

- Minimum delay: 2 days
- Maximum delay: 14 days
- Guardian role: Can veto critical vulnerabilities (3-of-5 PQ multisig using threshold Dilithium)

1.16.5 13.5 Governance Parameters (Initial)

Parameter	Value	Governance Required
Trading Fee Tiers	0.01%, 0.05%, 0.30%, 1.00%	Yes
Order Book Maker/Taker Fees	0.02% / 0.05%	Yes
Bridge Fee	0.1%	Yes
Minimum Validator Stake	100,000 QRDX	Yes
Pool Creation Stake (Standard)	10,000 QRDX	Yes
Pool Creation Subsidy (Burned)	5,000 QRDX	Yes
Validator Set Size	150	Yes
Block Time	2 seconds	Yes (requires upgrade)
Proposal Threshold	10,000,000 QRDX	Yes
Voting Period	7 days	Yes
Oracle Attestation Threshold	2/3+1 of adapter validators	Yes
Min Chain Adapters per Validator	1 (Ethereum mandatory)	Yes

1.17 14. Security Analysis

1.17.1 14.1 Threat Model

Adversary Capabilities: - Classical computational resources (unlimited) - Quantum computers (up to 10,000 logical qubits) - Network-level attacks (DDoS, eclipse attacks) - Economic attacks (stake manipulation, MEV) - Cross-chain oracle manipulation attempts - Exchange engine front-running attempts

Security Assumptions: - >2/3 of validators are honest - Post-quantum cryptographic assumptions hold - Dilithium-only identity prevents classical key compromise from affecting protocol security - Chain adapter implementations correctly track external chain state - Classical blockchains (Ethereum, Bitcoin) remain secure during bridge operations

1.17.2 14.2 Cryptographic Security

Post-Quantum Security Levels: - CRYSTALS-Dilithium: NIST Level 3 (equivalent to AES-192) - CRYSTALS-Kyber: NIST Level 3 - BLAKE3 (512-bit): 256-bit quantum security

Attack Resistance: - Shor's Algorithm: Resistant (lattice-based crypto) - Grover's Algorithm: Mitigated (doubled hash output) - Collision Attacks: Resistant (512-bit hashes) - Side-Channel Attacks: Constant-time implementations

1.17.3 14.3 Consensus Security

Byzantine Fault Tolerance: - Safety: Guaranteed with <1/3 Byzantine validators - Liveness: Guaranteed with >2/3 online validators - Finality: Single-slot finality with >2/3 attestations

Economic Security: - Cost to attack: >$300M (33% of staked value) - Slashing penalties: Up to 100% of stake - Social consensus: Community can fork to remove attackers

1.17.4 14.4 Bridge & Oracle Security

Attack Vectors & Mitigations:

Attack	Mitigation
Double-spend on source chain	12+ block confirmations
Fake mint proof	Merkle proof + validator Dilithium signatures
Validator collusion	High stake requirements + slashing
Oracle data manipulation	2/3+1 attestation consensus from chain adapter validators
Exchange front-running	Deterministic matching enforced by consensus
Quantum attack on classical chains	Doomsday protocol automatically blocks shielding
Classical key compromise	No classical keys in protocol (PQ-only)

Insurance Mechanism: - $10M insurance fund - Coverage: Up to $1M per incident - Claims: Governance-approved

Doomsday Protocol Integration: - Canary wallet monitored at doomsday.qrdx.org - Automatic circuit breaker if quantum threat detected - Shielding blocked, unshielding continues - $1M bounty incentivizes early detection

1.17.5 14.5 Smart Contract & Exchange Engine Security

Audit Partners: - Trail of Bits (Q3 2025) - OpenZeppelin (Q4 2025) - Quantstamp (Q1 2026)

Security Practices: - Formal verification for core contracts - Bug bounty program ($1M max reward) - Continuous monitoring and incident response - Timelocked upgrades with community review

1.17.6 14.6 Security Roadmap

Phase 1 (2025): - Third-party security audits - Public bug bounty launch - Formal verification of core contracts

Phase 2 (2026): - Quantum computer testing (collaboration with IBM/Google) - Real-world attack simulations - Insurance protocol integration

Phase 3 (2027+): - Continuous security monitoring - Annual audits and penetration testing - Upgrade to NIST Level 5 when available

---

1.18 15. Performance Benchmarks

1.18.1 15.1 Transaction Throughput

Testnet Results (Q2 2025): - Peak TPS: 5,247 transactions per second - Average TPS: 3,850 TPS (under normal load) - Block gas limit: 50,000,000 gas - Average transaction: ~21,000 gas

(simple transfer)

Comparison: | Blockchain | TPS | Finality | |————|—|———-| | Ethereum | 15-30 | 13-15 minutes | | Binance Smart Chain | 100-160 | 3 seconds | | Solana | 2,000-3,000 | 400ms | | **QRDX Chain** | **5,000+** | **1 second** |

1.18.2 15.2 Latency

Block Propagation: - Average: 350ms - P95: 680ms - P99: 1,200ms

Transaction Finality: - Single-slot finality: 2 seconds - Economic finality: 2 seconds (irreversible)

1.18.3 15.3 Signature Performance

CRYSTALS-Dilithium Benchmarks (Hardware: AMD EPYC 7763): - Key Generation: 48 s - Signing: 105 s - Verification: 62 s

CRYSTALS-Kyber Benchmarks: - Key Generation: 52 s - Encapsulation: 42 s - Decapsulation: 48 s

Comparison to ECDSA (secp256k1): - Dilithium signing: ~3x slower - Dilithium verification: ~2.5x slower - Key sizes: ~60x larger - **Trade-off**: Quantum resistance worth the overhead

1.18.4 15.4 Storage Requirements

Validator Node: - Initial sync: ~50 GB (genesis + 1 month) - Growth rate: ~2 GB/day (at 3,000 TPS) - Annual growth: ~730 GB - Pruned mode: ~100 GB (3-month history)

Archive Node: - Full history: 50 GB + all historical data - No pruning

1.18.5 15.5 Network Requirements

Minimum Validator Requirements: - CPU: 8 cores @ 3.0 GHz - RAM: 32 GB - Storage: 1 TB SSD - Network: 100 Mbps symmetric

Recommended Validator Requirements: - CPU: 16 cores @ 3.5 GHz - RAM: 64 GB - Storage: 2 TB NVMe SSD - Network: 1 Gbps symmetric

1.18.6 15.6 Gas Costs

QRDX Gas Costs (vs. Ethereum Application-Layer DEX):

Operation	QRDX Gas	ETH Gas	Savings
Simple Transfer	21,000	21,000	0%
qRC20 Transfer	45,000	65,000	31%
Swap (Protocol-Native)	65,000	150,000	57%
Add Liquidity	90,000	200,000	55%
Place Limit Order	40,000	N/A	N/A
Cancel Order	25,000	N/A	N/A
Create Pool (Standard)	150,000	4,500,000	97%
Bridge Deposit	75,000	N/A	N/A

Operation	QRDX Gas	ETH Gas	Savings
Multisig Submit Signature	55,000	65,000	15%

Gas Price: - Average: 0.1 Gwei (QRDX) - Transaction cost: ~$0.002-0.01 (at $2 QRDX price)

1.19 16. Roadmap

1.19.1 16.1 Phase 1: Foundation (Q3 2025 - Q4 2025)

Q3 2025: - Whitepaper release - Testnet launch (v1.0) - Core protocol implementation - Basic bridge functionality (Ethereum) - PQ node identity (@-schema) implementation

Q4 2025: - Security audits (Trail of Bits, OpenZeppelin) - Public testnet stress testing - Bug bounty program launch ($1M pool) - Community testing incentives - Integrated exchange engine testnet deployment - Mainnet launch (December 2025)

1.19.2 16.2 Phase 2: Ecosystem Growth (Q1 2026 - Q2 2026)

Q1 2026: - Bitcoin chain adapter launch (native L0 integration) - Multi-chain adapter expansion (BSC, Polygon, Avalanche, Solana) - Permissionless pool creation goes live - PQ multisig wallet tooling release - Prefunded wallet manager deployment - Liquidity mining program start - Mobile wallet launch (PQ-native)

Q2 2026: - Governance activation - On-chain order book activation for high-liquidity pairs - Oracle attestation rewards live - Third-party protocol integrations - Fiat on-ramp partnerships - Institutional custody SDK (multisig + prefunded wallets)

1.19.3 16.3 Phase 3: Advanced Features (Q3 2026 - Q4 2026)

Q3 2026: - Private transaction pools (Kyber-based encryption) - Cross-chain messaging protocol (native L0 inter-chain calls) - NFT bridge (quantum-resistant NFT standard) - Lending/borrowing protocol (protocol-native) - Options and derivatives on integrated exchange

Q4 2026: - Institutional custody solutions (enterprise PQ multisig) - Compliance tools (optional KYC hooks for pools) - Enterprise partnerships - Quantum computer stress testing - Real-world quantum attack simulations

1.19.4 16.4 Phase 4: Maturity & Expansion (2027+)

2027: - Layer-2 scaling solutions (quantum-resistant rollups on L0 substrate) - Cross-protocol interoperability (Cosmos IBC adapter, Polkadot XCMP adapter) - Advanced privacy features (quantum-resistant zero-knowledge proofs) - AI-powered trading tools integrated with exchange engine - Decentralized sequencer network for L2s built on QRDX

2028+: - NIST Level 5 cryptography upgrade (when standardized) - Quantum Internet integration - Post-quantum smart contract languages - Full DeFi ecosystem parity with Ethereum (all on protocol-native exchange) - Global institutional adoption - 50+ chain adapters with universal cross-chain composability

1.19.5 16.5 Research & Development

Ongoing Initiatives: - Post-quantum zero-knowledge proofs (zkSNARKs/zkSTARKs) - Quantum-resistant threshold signatures (advanced threshold Dilithium schemes) - Advanced cross-chain communication protocols (native L0 inter-chain calls) - Scalability improvements (sharding, Layer-2 on L0 substrate) - Formal verification of all core contracts and exchange engine - Threshold Dilithium optimization for larger signer sets - Oracle attestation aggregation for reduced block overhead

---

1.20 17. Conclusion

QRDX represents a paradigm shift in blockchain architecture, addressing the existential threat posed by quantum computing while eliminating the fragmentation that plagues current cross-chain infrastructure. By operating as a Layer-0 protocol with natively integrated exchange, oracle, and bridge capabilities—all secured by NIST-standardized post-quantum cryptography—QRDX delivers a unified, quantum-resistant ecosystem where the chain, the exchange, and the cross-chain infrastructure are one and the same.

The enforcement of PQ-only identity at every layer—from bootstrap node discovery using `Q`-schema addressing, to Dilithium-only validator consensus, to threshold Dilithium multisig wallets and master-controlled prefunded wallet hierarchies—eliminates any classical cryptographic attack surface. The integrated exchange engine, where users can permissionlessly spin up liquidity pools backed by staked or subsidized QRDX, provides capital-efficient trading with protocol-level fairness guarantees that application-layer DEXs cannot match.

The asset shielding mechanism enables users to protect their holdings against future quantum attacks, creating a bridge between the classical and quantum-resistant blockchain eras. As quantum computers continue to advance, QRDX provides a safe haven for digital assets, ensuring that trillions of dollars in cryptocurrency value remain secure.

Key Achievements: - First quantum-resistant L0 protocol with integrated exchange engine - Protocol-native cross-chain oracle and bridge (no external middleware) - PQ-only identity at every layer (nodes, validators, wallets, multisig) - Permissionless pool creation with stake/subsidy model - Threshold Dilithium multisignatures and prefunded wallet hierarchies - Native asset shielding (ETH → qETH, etc.) - 5,000+ TPS with sub-second finality - Full EVM compatibility (QEVM) - Community-driven governance

Call to Action: The QRDX ecosystem invites developers, liquidity providers, traders, and institutions to join us in building the future of quantum-resistant DeFi. Whether you're looking to shield your assets, provide liquidity, build applications, or participate in governance, QRDX offers a comprehensive platform for the post-quantum era.

Join the Revolution: - Website: https://qrdx.org - Documentation: https://docs.qrdx.org - GitHub: https://github.com/qrdx-org - Telegram: https://t.me/qrdx_org - Twitter: https://twitter.com/qrdx_org

---

1.21 18. References

1.21.1 Academic Papers

1. Shor, P. W. (1997). "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." SIAM Journal on Computing.
2. Grover, L. K. (1996). "A Fast Quantum Mechanical Algorithm for Database Search." Proceedings of the 28th Annual ACM Symposium on Theory of Computing.
3. Ducas, L., et al. (2018). "CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme." IACR Transactions on Cryptographic Hardware and Embedded Systems.
4. Bos, J., et al. (2018). "CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM." IEEE European Symposium on Security and Privacy.
5. Aumasson, J. P., et al. (2021). "BLAKE3: One Function, Fast Everywhere." Official BLAKE3 Specification.

1.21.2 Industry Standards

6. NIST (2024). "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard." National Institute of Standards and Technology.
7. NIST (2024). "FIPS 204: Module-Lattice-Based Digital Signature Standard." National Institute of Standards and Technology.
8. NIST (2016). "Post-Quantum Cryptography Standardization." https://csrc.nist.gov/projects/post-quantum-cryptography

1.21.3 Blockchain & DeFi

9. Adams, H., et al. (2021). "Uniswap v3 Core." Uniswap Whitepaper.
10. Adams, H., et al. (2023). "Uniswap v4 Core." Uniswap Technical Documentation.
11. Buterin, V., et al. (2022). "Ethereum 2.0 Specification." Ethereum Foundation.
12. Nakamoto, S. (2008). "Bitcoin: A Peer-to-Peer Electronic Cash System." Bitcoin Whitepaper.

1.21.4 Quantum Computing

13. IBM (2024). "IBM Quantum Roadmap." https://www.ibm.com/quantum/roadmap
14. Google (2023). "Quantum AI Research." https://quantumai.google/
15. Mosca, M. (2018). "Cybersecurity in an Era with Quantum Computers: Will We Be Ready?" IEEE Security & Privacy.

1.21.5 Cryptography

16. Bernstein, D. J., et al. (2017). "Post-Quantum Cryptography." Springer.
17. Chen, L., et al. (2016). "Report on Post-Quantum Cryptography." NIST Internal Report 8105.

18. Peikert, C. (2016). "A Decade of Lattice Cryptography." Foundations and Trends in Theoretical Computer Science.

1.22 Appendix A: Glossary

@-Schema Addressing: Post-quantum node identity format (`<pq_algorithm>@<pubkey_hash>@<host>:<port>`) used for bootstrap node discovery and peer identification.

AMM (Automated Market Maker): A decentralized exchange mechanism using liquidity pools and mathematical formulas to determine asset prices.

Block Height: The sequential number of a block in a blockchain, used for temporal ordering and state verification.

Chain Adapter: A modular component embedded in QRDX nodes that runs a light client or SPV client for an external blockchain, enabling native cross-chain state reading and transaction submission.

CRYSTALS-Dilithium: A lattice-based digital signature algorithm standardized by NIST for post-quantum cryptography.

CRYSTALS-Kyber: A lattice-based key encapsulation mechanism standardized by NIST for post-quantum cryptography.

Concentrated Liquidity: A feature allowing liquidity providers to allocate capital within specific price ranges for higher capital efficiency.

Doomsday Protocol: Automated circuit breaker at doomsday.qrdx.org that halts classical-to-quantum asset bridging if a quantum computer breaks ECDSA, while allowing quantum-to-classical unshielding to continue.

Finality: The point at which a transaction or block becomes irreversible.

Hooks: Extensible plugin architecture allowing custom logic at key points in pool lifecycle.

Integrated Exchange Engine: The protocol-native exchange built into the QRDX blockchain itself, combining concentrated liquidity AMM with on-chain order book, where pools are protocol-level state rather than smart contract deployments.

Layer-0 (L0): A blockchain protocol that operates as a cross-chain substrate, natively embedding capabilities for interacting with multiple external blockchains without external middleware.

Lattice-Based Cryptography: Cryptographic schemes based on the hardness of lattice problems, resistant to quantum attacks.

NIST: National Institute of Standards and Technology, responsible for cryptographic standards in the United States.

Oracle Attestation: The process by which validators running chain adapters attest to the state of external blockchains, providing oracle data as part of QRDX consensus.

Post-Quantum Cryptography: Cryptographic algorithms designed to be secure against attacks by quantum computers.

PQ Multisig: Post-quantum multisignature wallet using threshold Dilithium signatures, producing a single aggregated signature regardless of threshold configuration.

Prefunded Wallet: A sub-wallet created, funded, and controlled by a master PQ wallet, with configurable budget limits, daily spending caps, and operation scope restrictions.

qBTC: Quantum-resistant Bitcoin2014BTC locked on Bitcoin network and represented as a quantum-resistant token on QRDX Chain.

qETH: Quantum-resistant Ethereum2014ETH locked on Ethereum and represented as a quantum-resistant token on QRDX Chain.

qRC20: Quantum-resistant token standard for QRDX Chain, based on ERC-20 with post-quantum extensions.

Quantum Resistance: Property of cryptographic systems that remain secure even against quantum computer attacks.

Quantum Shielding: Process of converting classical blockchain assets (BTC, ETH, etc.) into quantum-resistant equivalents (qBTC, qETH, etc.) by locking them on source chains and minting backed qRC20 tokens.

Shielding: Process of converting classical blockchain assets into quantum-resistant equivalents (classical-to-quantum).

Shor's Algorithm: Quantum algorithm that can efficiently factor large numbers and solve discrete logarithm problems.

Slashing: Penalty mechanism where validators lose staked tokens for malicious or negligent behavior.

Subsidized Pool: A liquidity pool created by burning QRDX permanently, resulting in community-owned exchange infrastructure with lower ongoing fees.

Threshold Dilithium: A post-quantum threshold signature scheme where m-of-n signers produce a single aggregated Dilithium signature.

TWAP (Time-Weighted Average Price): Price averaging method that weights prices by the time they were active.

OracleTransaction: An L0 cross-chain transaction envelope that wraps a fully-formed, natively-signed target chain transaction inside a PQ-signed Dilithium wrapper. Enables non-custodial cross-chain execution with conditional logic and callbacks.

EthereumTransaction (sub-type): An OracleTransaction inner payload containing a fully-formed EIP-1559 Ethereum transaction signed with secp256k1.

BitcoinTransaction (sub-type): An OracleTransaction inner payload containing a fully-formed Bitcoin transaction with witness data, signed with secp256k1.

SolanaTransaction (sub-type): An OracleTransaction inner payload containing a fully-formed Solana transaction signed with Ed25519.

Execution Condition: A rule attached to an OracleTransaction that must be satisfied before the inner transaction is broadcast to the target chain (e.g., price threshold, block height, prerequisite OracleTransaction confirmation).

Unshielding: Process of converting quantum-resistant assets back to classical blockchain assets (quantum-to-classical).

1.23 Appendix B: Mathematical Formulas

1.23.1 Constant Product Formula (Base AMM)

```
x × y = k
```

Where: - x = reserves of token A - y = reserves of token B - k = constant product

1.23.2 Concentrated Liquidity Formula

```
L = √(x × y)
P = y / x
ΔL = Δx × √P + Δy / √P
```

Where: - L = liquidity - P = price - Δ = delta (change)

1.23.3 Price Impact

```
Price Impact = |P_final - P_initial| / P_initial
```

1.23.4 Impermanent Loss

```
IL = (2 × √(P_ratio) / (1 + P_ratio)) - 1
```

Where P_ratio = P_final / P_initial

1.23.5 Validator Selection Probability

```
P_selection = (stake_i / total_stake) × uptime_factor_i
```

1.23.6 TWAP Calculation

```
TWAP = exp((Σ(log(P_i) × Δt_i)) / Σ(Δt_i))
```

1.24 Appendix C: Contract Addresses (Mainnet - Post Launch)

QRDX Chain (Chain ID: TBD) - PoolManager (Exchange Engine): `0x0000000000000000000000000000000000000000` - qETH Token: `0x0000000000000000000000000000000000000002` - qBTC Token: `0x0000000000000000000000000000000000000003` - qUSDC Token: `0x0000000000000000000000000000000000000004` - Bridge Core Contract: `0x0000000000000000000000000000000000000005` - Governance: `0x0000000000000000000000000000000000000006` - Timelock (3-of-5 PQ Multisig Guardian): `0x0000000000000000000000000000000000000007` - Doomsday Protocol: `0x0000000000000000000000000000000000000008` - PQ Multisig Factory: `0x0000000000000000000000000000000000000009` - Prefunded Wallet Manager: `0x000000000000000000000000000000000000000A` - Oracle Attestation Registry: `0x000000000000000000000000000000000000000B`

Ethereum Mainnet - QRDX Bridge Lock: TBD (Post-launch) - QRDX Token (ERC-20): TBD (Post-launch) - Doomsday Canary Address: `0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb1` (Public key published at doomsday.qrdx.org)

Addresses will be updated after mainnet deployment in Q4 2025.

Document Version: 3.2
Last Updated: February 16, 2026
Authors: QRDX Foundation Research Team

Disclaimer: This whitepaper is for informational purposes only and does not constitute investment advice, financial advice, trading advice, or any other sort of advice. QRDX does not guarantee the accuracy or completeness of the information provided. The protocol is under active development and specifications may change.

For the latest updates, visit https://qrdx.org