Shell-Chain: Post-Quantum Security from Genesis

A Clean-Slate Post-Quantum Blockchain with Native NIST-Standardized Cryptography

Shell Team

June 20, 2026

Abstract

We present Shell-Chain, a Layer-1 blockchain architecture and open-source reference implementation effort that integrates NIST-standardized post-quantum (PQ) cryptographic primitives at the protocol level. Unlike retrofits that add quantum resistance through hard forks or Layer-2 overlays, Shell-Chain starts from a *quantum-safe genesis*: the protocol is designed around post-quantum authentication from the first block, not as a later migration. The reference `pqvm` engine contains the PQ-native path—six PQ precompiles, native `PQTx` transactions, 32-byte addresses, and a machine-verified 14-vector conformance suite. Three native opcodes (`PQVERIFY`, `PQHASH`, and `PQADDR`) are specified gas-priced, and wired into the interpreter dispatch table; transport security is likewise staged, with today's nodes using classical libp2p Noise/XX while PQ Noise support matures upstream. Implementation-status notes in the paper mark this boundary explicitly.

Within those boundaries, the design is practical. wPoA combines weighted proposer selection with BFT attestation finality: safety does not depend on network timing, while liveness is a post-GST property that requires timeout/view-change rotation to reach an honest proposer (§7). The active signature set uses ML-DSA-65 as the primary FIPS 204 path, keeps Dilithium3 for legacy-compatible deployments, and retains SLH-DSA-SHA2-256f/SPHINCS+ as a conservative hash-based fallback. Reference measurements put Dilithium3/ML-DSA-65 verification in the $\approx 46\,\mu$s range and SLH-DSA-SHA2-256f at ≈ 2.3 ms on commodity developer hardware. A batch-commitment STARK pipeline lowers per-validator re-verification by about 33× for 100-transaction blocks and 79× for full 500-transaction blocks when `L2StarkMode = Active`; it is an accountability and amortisation layer, not an in-circuit proof of signature validity. Addresses are the full 32-byte value `BLAKE3(algo_id || pubkey)`, rendered as `0x` plus 64 lowercase hex characters, with no bridge to a prior 20-byte address model. The protocol initializes from a compile-time algorithm allowlist, and the on-chain registry in §5.8 governs live transitions.

1 Introduction

Blockchain security rests on a single load-bearing assumption: that computing discrete logarithms over elliptic-curve groups is computationally intractable. This assumption underlies ECDSA, which protects user accounts in Bitcoin, Ethereum, and virtually every production blockchain. Shor's algorithm [Sho97] eliminates that assumption: on a sufficiently powerful quantum computer, it recovers an ECDSA private key from its public key in polynomial time.

What makes the quantum threat urgent *today*—rather than only in some distant future—is *retrospective key forgery*: an adversary can record today's blockchain transactions and, once quantum resources mature, run Shor's algorithm on the public keys already exposed on-chain to recover the corresponding private keys. Because blockchain state is *permanently and publicly recorded*, every public key ever used for signing remains available for later cryptanalysis. The question is not whether to act, but whether the window for preventive action is still open.

This paper makes the following design contributions:

1. **Native PQ-first architecture.** Shell-Chain does not retrofit an ECDSA chain. It begins with post-quantum authentication: ML-DSA-65 is the primary FIPS 204 signing path, Dilithium3 remains available for legacy-compatible deployments, and SLH-DSA-SHA2-256f provides a conservative hash-based fallback. This removes the dual-key transition window that retrofits must carry (Section 2).

2. **STARK-based verification amortisation.** PQ signatures make validator re-verification an $O(V \times n)$ cost. We use a batch-commitment STARK pipeline to reduce common-case re-verification to $O(V)$ short proof checks, while preserving a challenge path for accountability rather than claiming full in-circuit signature verification (Section 11.4).

3. **wPoA consensus with qualified BFT finality.** Permissioned validator authority (wPoA) is the launch-network trust model. A BFT attestation layer gives deterministic finality once a single-block quorum certificate is assembled: safety follows from the weighted Byzantine bound, and liveness requires post-GST synchrony plus successful view-change progression (Section 7).

4. **Post-Quantum Virtual Machine (PQVM).** PQVM is a clean-slate execution environment for PQ cryptography. It makes the necessary changes explicit: 32-byte addresses, native account abstraction, PQ-calibrated opcodes, and a PQ precompile surface from genesis. The native opcodes are specified, gas-priced, and wired into interpreter dispatch (Sections 6 and 14).

5. **Post-Quantum Network Transport (PQ Noise).** Shell-Chain targets the PQ Noise framework [ADH+22] for validator P2P transport, aiming for forward-secret, identity-bound sessions using ML-KEM-768 KEM-based ephemeral key exchange, ML-DSA-65 peer authentication, and ChaCha20-Poly1305 session encryption (Section 8.1), while node transport still uses classical libp2p Noise/XX.

6. **Algorithm Agility via On-Chain Governance.** We specify an on-chain algorithm registry (§5.8) that enables adding new PQ primitives without a protocol hard fork: governance votes ($\lceil 2N/3 \rceil$ quorum) write a new `AlgorithmSpec` on-chain; validators install the verifier ahead of the activation block; no chain split occurs.

The reference `pqvm` execution engine implements the PQ-native path: six PQ precompiles, 32-byte addresses, native `PQTx` transactions, and a machine-verified conformance suite. Remaining partial or target-state work, including PQ Noise transport, is called out explicitly.

Implementation-status legend. The paper uses four status labels. **Implemented** means the reference implementation contains the consensus-critical code. **Partial** means the interface or subsystem exists, but a consensus or execution path is not finished. **Target** marks a specified design that still depends on upstream or future work. **Planned** marks work outside the immediate implementation scope.

Paper organization. Section 2 explains why migration-after-the-fact leaves residual security and coordination risks. Section 3 states the adversary model and design principles. Section 4 presents the system architecture. Section 5 addresses post-quantum authentication (algorithm selection, signature security, address derivation, key storage, and algorithm agility). Section 6 specifies the Post-Quantum Virtual Machine as the target execution environment. Section 7 covers the wPoA consensus mechanism and Section 8 the network layer, including the PQ Noise post-quantum transport (§8.1). Section 9 covers node roles, STARK deployment modes, and storage profiles (Archive / Full / Pruned (Rolling)). Section 10 provides a unified security

Table 1: Implementation status of major subsystems

Subsystem	Status	Note
PQ precompile suite	Implemented	Six 32-byte-address precompiles
Native `PQTx` / 32-byte addresses	Implemented	Canonical Shell address model
Algorithm registry governance	Implemented	Live registry transitions
wPoA view-change	Implemented	Timeout quorum rotates proposer
PQ-native opcodes	Implemented	Defined/gas-priced; interpreter dispatch wired
STARK batch commitment	Partial	Accumulator/accountability proof, not in-circuit signature validity
PQ Noise transport	Target	Current nodes use classical libp2p Noise/XX
Permissionless validator admission	Planned	Future PQ staking upgrade

analysis. Section 11 evaluates performance viability against design targets. Section 12 positions Shell-Chain in prior literature, Section 13 presents a comparative evaluation, Section 14 discusses the current implementation gap and other limitations, and Section 15 concludes.

2 The Retrofitting Trap

The natural response to the quantum threat is to upgrade existing chains. We examine the four principal strategies and show that each leaves a residual vulnerability or imposes unsolvable coordination costs.

Layer-2 overlay schemes. Adding PQ signature validation at a layer atop an existing chain [FCFL20] protects application-layer messages but does not protect the base-layer consensus. An ECDSA break compromises validator signing keys, finality proofs, and the bridge contracts through which L2 value settles. The PQ wrapper rests on a quantum-vulnerable foundation.

Hard-fork migration. Ethereum's proposed quantum emergency fork [But22] would replace ECDSA accounts via a protocol upgrade. The coordination problem is formidable: consensus must be reached across heterogeneous client implementations, wallet vendors, exchange operators, and deployed DeFi contracts, under time pressure, while an adversary with quantum capability may already be active. Even a successful fork leaves a long tail of unupgraded accounts indefinitely vulnerable, and it does nothing to protect historical transactions already archived under classical keys.

Hybrid classical/post-quantum schemes. Transitional designs that accept both ECDSA and PQ signatures during a migration window [BBF+19] double implementation complexity and create a subtle security regression: the system's security degrades to the weaker component unless the hybrid composition is formally proven. More practically, "legacy mode" is difficult to terminate once deployed, because removing it breaks existing applications. Hybrid periods tend to become permanent.

Native PQ chains without EVM-familiar tooling. Starting fresh with PQ foundations eliminates retrofit problems but prior proposals either require custom virtual machines or custom toolchains, severing the Ethereum developer ecosystem. This confines such chains to greenfield deployments and imposes large adoption costs.

EVM extension is insufficient. The most natural retrofitting approach for an Ethereum-derived chain is to add PQ signature support as an EVM extension: keep the EVM, add a `pqrecover` precompile, and preserve ECDSA for compatibility. We argue this fails for the same structural reason as all retrofitting: the `address` type in Solidity, the ABI encoding standard, and the EVM opcode semantics all encode 20-byte (160-bit) addresses. Changing this requires modifying the Solidity type system, the ABI specification, the EVM opcode semantics for `ADDRESS`, `CALLER`, `ORIGIN`, and all call opcodes, and recompiling every contract. At that point, compatibility with existing Ethereum bytecode is broken regardless. A clean-slate VM is no harder to migrate to than a heavily patched EVM, and produces a simpler, more principled design.

Strongest available guarantee. Each retrofit strategy either fails to fully close the classical attack surface, creates coordination problems that realistic networks cannot reliably solve, or sacrifices developer compatibility. A quantum-safe genesis—integrating PQ cryptography at every layer from the first block, while retaining Cancun-equivalent opcode semantics and developer-familiar tooling where possible—is the strongest available guarantee under current cryptographic assumptions. Shell-Chain is built on this conclusion.

3 Threat Model and Design Principles

3.1 Adversary Model

Definition 1 (CRQC)**.** *A* Cryptographically Relevant Quantum Computer *(CRQC) is a quantum computing device with sufficient qubit count, gate fidelity, and coherence time to execute Shor's algorithm on cryptographic-scale inputs (e.g., 256-bit elliptic curves or 2048-bit RSA moduli) within a practical time bound. We make no assumption about when a CRQC will exist; our threat model treats retrospective key forgery as an active threat vector today.*

We consider a computationally bounded adversary capable of:

1. Running Shor's algorithm to invert discrete-logarithm and integer-factorization problems in polynomial time.
2. Running Grover's algorithm, yielding a quadratic speedup for unstructured search (e.g., preimage attacks on 256-bit hashes).
3. **Retrospective key forgery**: recording on-chain public keys today and running Shor's algorithm after quantum capability is acquired to recover private keys, enabling forged future transactions or authorizations from historically exposed public keys.
4. Controlling up to $f \leq \lfloor (N-1)/3 \rfloor$ validators (Byzantine fault model).
5. Mounting classical network attacks: Eclipse, Sybil, DDoS, and transaction flooding.

This adversary can compromise any component whose security relies on hardness of classical number-theoretic problems. It *cannot* break Module-LWE, Module-SIS, or SHA-256 one-wayness—the assumptions underlying Shell-Chain's signature schemes. We model the adversary as a *quantum polynomial-time* (QPT) algorithm: one that runs in polynomial time on a quantum computer and may query hash functions in quantum superposition (Quantum Random Oracle Model, QROM).

3.2 Three Engineering Challenges

Designing a practical PQ blockchain against this adversary reduces to three independent but interrelated challenges:

C1 — Authentication: Replace ECDSA transaction signing with a PQ alternative, define a 32-byte PQ-native address space with sufficient security margin, preserve EVM-familiar opcode semantics where they remain valid, and support arbitrary signing policies (multisig, account upgrades).

C2 — Consensus integrity: Ensure that validator identities, block seals, and finality proofs are all quantum-resistant, so that a CRQC cannot impersonate a validator, rewrite finalized blocks, or violate BFT safety.

C3 — Practical performance: Absorb the $\approx$52$\times$ larger PQ signatures (3,309 bytes for ML-DSA-65 vs. 64 bytes for ECDSA) and higher per-verification latency without degrading transaction throughput to impractical levels.

Synchrony model. The system operates under the partial synchrony model of Dwork, Lynch, and Stockmeyer [DLS88]: after an unknown *Global Stabilization Time* (GST), message delays are bounded by Δ. By the FLP impossibility result [FLP85], deterministic consensus is impossible in a fully asynchronous network; Shell-Chain therefore states safety without synchrony assumptions and states liveness only after GST, bounded message delay, and a view-change path that eventually rotates to an honest proposer (see Section 7).

These challenges are ordered by dependency: C2 relies on C1 (consensus requires quantum-resistant validator identities), and C3 is a consequence of solving C1 and C2 honestly—we cannot pretend PQ signatures are small.

3.3 Design Principles

Three principles govern Shell-Chain's response to these challenges:

P1 — Quantum-safe genesis (answers C1 + C2): No classical signature algorithm is accepted anywhere in the protocol. PQ primitives are present from block zero. There is no migration event and no hybrid period.

P2 — Post-Quantum Virtual Machine (answers C1): Shell-Chain retains EVM-familiar arithmetic, memory, storage, and control-flow semantics, but adopts a clean-slate PQVM with 32-byte PQ-native addresses, native account abstraction, and a dedicated PQ opcode/precompile layer.

P3 — Defense in depth at every boundary (answers C2 + C3): Classical attack surfaces (ECDSA recovery, oversized signature deserialization, classical KDF) are removed or hard-bounded at protocol boundaries, not merely discouraged by documentation.

Migration urgency: the Mosca inequality. The Mosca inequality [Mos18] formalizes why early action is critical: if $x + y > z$, where x is the estimated years until a CRQC exists, y is the years required for full chain migration, and z is the required data secrecy lifetime, then migration must begin *now*. For a public blockchain, several national security agency assessments place x within roughly a decade or more, migration requires $y \sim$ 2–5 years, and on-chain data (public keys, transaction history) must be protected indefinitely ($z = \infty$). Shell-Chain's quantum-safe genesis satisfies this inequality by construction.

4 System Architecture

The architecture is implemented as a layered Rust workspace in which the dependency graph is strictly acyclic: foundational crates (`shell-primitives`, `shell-crypto`) carry no dependencies on higher-layer services (`shell-rpc`, `shell-node`). This layering enforces the invariant that the PQ cryptographic substrate is self-contained and independently auditable.

Table 2: Shell-Chain layered architecture

Layer	Crate(s)	Responsibility
1	`shell-primitives`	Hashes (Keccak-256, BLAKE3), `Address`, `U256`
2	`shell-crypto`, `shell-core`,	PQ signing/verification; block and transaction
	`shell-storage`, `shell-genesis`	data structures; RocksDB state; genesis
3	`shell-consensus`, `shell-evm`,	PoA / optional wPoA + BFT finality; PQVM/revm execution path;
	`shell-mempool`, `shell-network`	fee-priority mempool; libp2p P2P
4	`shell-stark-prover`	Async batch-commitment STARK pipeline
5	`shell-rpc`, `shell-node`	Ethereum JSON-RPC; node orchestration
6	`shell-cli`, `shell-keystore`	CLI; argon2id + XChaCha20-Poly1305 key store

The challenge-to-crate mapping is direct: C1 (authentication) is addressed by `shell-crypto` and `shell-evm`; C2 (consensus) by `shell-consensus`; C3 (performance) by `shell-stark-prover` and `shell-mempool`. The following three sections treat each challenge in turn.

4.1 Cryptographic Substrate

Two cross-cutting properties of Shell-Chain's cryptographic foundation apply throughout:

- **ML-DSA-65 (FIPS 204)**: the primary governed signature algorithm in the reference implementation, implemented in `shell-crypto` via `MlDsaSigner`. It uses a 1,952-byte public key and 3,309-byte signature and targets NIST Level 3 security.
- **Dilithium3**: a legacy-compatible active path implemented via `pqcrypto-dilithium`. It shares the same key and signature size profile as ML-DSA-65 and remains valid for deployments and tooling that predate the FIPS 204 naming.
- **SLH-DSA-SHA2-256f (FIPS 205)**: the hash-only fallback (NIST Level 5 ($\geq$128-bit quantum / $\geq$256-bit classical security)), for high-assurance operations where larger signatures are acceptable.
- **BLAKE3**: used for address derivation via `BLAKE3(algo_id || pubkey)`, yielding a full 32-byte address encoded as `0x`-prefixed 64-character lowercase hex.

Both signature schemes are proven EUF-CMA in the Quantum Random Oracle Model (QROM) [KLS18]—security holds even against an adversary who can query hash functions in quantum superposition. (Under the QROM, BLAKE3 and SHA3-256 are modeled as quantum-accessible random oracles; this is a standard heuristic assumption.)

5 Post-Quantum Authentication

The first challenge is replacing ECDSA at the account level while maintaining a deployable address model. Shell-Chain uses 32-byte `0x`-prefixed hex addresses from genesis. This requires three sub-solutions: a PQ signature scheme selection with formal security reductions, a quantum-resistant address derivation, and a protocol-level account abstraction that works without a classical base layer.

5.1 Algorithm Selection Rationale

Three post-quantum signature families are NIST-standardised: lattice-based (ML-DSA / FIPS 204), hash-based (SLH-DSA / FIPS 205), and code-based schemes (not yet standardised for signatures). Table 3 summarises the design trade-offs.

Table 3: Post-quantum signature algorithm trade-offs

Algorithm	Sig. size	Verify	Security basis	Role
ML-DSA-65	3,309 B	$\approx$46 μs class	MLWE/MLSIS	Primary FIPS 204 path
Dilithium3	3,309 B	$\approx$46 μs class	MLWE/MLSIS	Legacy-compatible active path
SLH-DSA-SHA2-256f	49,856 B	$\approx$2.3 ms	SHA-256 hash	High-assurance fallback
ECDSA (secp256k1)	65 B[1]	$\approx$40 μs	ECDLP	**Eliminated; quantum-broken**

Why ML-DSA-65 is primary. ML-DSA-65 is the FIPS 204-standardized lattice-signature path implemented in `shell-crypto`. Repository benchmarks place it in the same $\approx$46 μs verification class as Dilithium3 on the reference hardware. That keeps it comfortably inside the 2-second block target while aligning governance with the finalized NIST standard.

Why retain Dilithium3. Dilithium3 has the same 3.3 KB signature profile and Level-3 security target. It remains active for legacy-compatible deployments, test vectors, and tooling that predate the FIPS 204 ML-DSA naming. It is not a classical fallback and does not reintroduce ECDSA exposure.

Why retain SLH-DSA-SHA2-256f. SLH-DSA provides an *independent* security foundation: its security relies primarily on hash-function assumptions (DSPR and PRF security of the underlying hash, with no lattice assumption). If a future lattice attack were discovered, SLH-DSA transactions remain unaffected. The cost is large signatures (49 KB) and slow signing (84 ms); Shell-Chain assigns SLH-DSA to high-assurance use cases (governance, large-value transfers) where these costs are acceptable.

Why eliminate ECDSA entirely. Retaining ECDSA in any configuration—even as optional—preserves the attack surface: any account secured by ECDSA can be compromised by a CRQC regardless of whether other accounts use PQ signatures. The design choice to *eliminate* ECDSA removes this residual vulnerability class at the cost of requiring migration tooling.

Rejected alternatives. Alternative schemes considered include FN-DSA (Falcon) [Nat24d] (rejected: variable-time Gaussian sampling complicates constant-time implementation), FrodoKEM (rejected: larger key sizes and lack of NIST standardization), and NTRU-family

schemes (rejected: a more complex patent landscape and exclusion from the final NIST selection). Shell-Chain adopts only NIST-standardized schemes to maximize ecosystem interoperability and avoid assuming parameter-selection responsibility.

5.2 Signature Security Foundations

Definition 2 (EUF-CMA)**.** *A signature scheme* $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is* existentially unforgeable under adaptive chosen-message attack *(EUF-CMA) if no QPT (quantum polynomial-time) adversary, given access to a signing oracle, can produce a valid forgery* (m^*, σ^*) *for a message* m^* *not previously queried.*

ML-DSA-65 achieves EUF-CMA under Module-SIS and Module-LWE [DKL+18]—lattice assumptions believed hard for both classical and quantum adversaries. SLH-DSA (FIPS 205) achieves EUF-CMA under the combined assumptions of (1) PRF security of the keyed hash function, (2) collision resistance, (3) second-preimage resistance, and (4) undetectability of the PRF output [BHK+19]—a *hash-only* assumption set immune to algebraic attack. Supporting both schemes provides defence in depth: if unforeseen progress against Module-LWE weakens ML-DSA, SLH-DSA remains unaffected.

Definition 3 (QROM security)**.** *A scheme is* secure in the Quantum Random Oracle Model *(QROM) if security holds when the adversary may query the hash oracle in quantum superposition. ML-DSA has been proven secure in the QROM [KLS18]; SLH-DSA security in the QROM is addressed in [H+22].*

5.3 Signature Parameters

Table 4: ML-DSA-65 Parameters

Property	Value
Standard	NIST FIPS 204 (ML-DSA-65)
Security Level	NIST Category 3: ≥128-bit quantum security / ≥192-bit classical security (AES-192 equivalent for classical-security comparisons)
Security Assumption	Module-LWE + Module-SIS
Public Key Size	1,952 bytes
Secret Key Size	4,032 bytes
Signature Size	3,309 bytes

Table 5: ML-KEM-768 Parameters

Property	Value
Standard	NIST FIPS 203 (ML-KEM-768)
Security Level	NIST Level 3 (conjectured ≥128-bit quantum security)
Security Assumption	Module-LWE
Public Key (ek) Size	1,184 bytes
Ciphertext (ct) Size	1,088 bytes
Shared Secret Size	32 bytes

‡ `MAX_SIGNATURE_BYTES = 51,200` in `shell-crypto`: a safe deserialization bound; actual SLH-DSA-SHA2-256f signatures are 49,856 bytes.

Table 6: SLH-DSA-SHA2-256f Parameters

Property	Value
Standard	NIST FIPS 205 (SLH-DSA-SHA2-256f)
Security Level	NIST Category 5: ≥128-bit quantum security / ≥256-bit classical security
Security Assumption	SHA-256 one-wayness and collision resistance
Public Key Size	64 bytes
Signature Size	49,856 bytes‡

5.4 Address Derivation

Shell-Chain derives addresses as

$$addr = \text{BLAKE3}(algo\_id \| pk) \tag{1}$$

where *algo_id* is a one-byte algorithm identifier and *pk* is the serialised public key. The full 32-byte BLAKE3 output is used with no truncation. User-facing addresses are encoded as `0x`-prefixed 64-character lowercase hex strings. The `algo_id` byte provides domain separation across supported signature schemes, and on a clean-slate chain no separate version byte is required for legacy compatibility. Addresses derived from Dilithium3, ML-DSA-65, and SLH-DSA-SHA2-256f/SPHINCS+ therefore occupy domain-separated sub-spaces; cross-algorithm collisions remain possible only with negligible BLAKE3 collision probability. The `algo_id` values and their future governance are discussed in §5.8.

5.5 Protocol-Level Account Abstraction

Why not ERC-4337? ERC-4337 [B+23] implements account abstraction via a smart contract bundler that submits user operations at the base layer. The bundler's own transactions must be authenticated—and on a classical base layer, that authentication is ECDSA. A PQ wallet built on an ERC-4337 overlay chain is only as secure as the classical base-layer ECDSA it ultimately depends on. Protocol-level AA removes the bundler entirely, placing PQ validation directly in the block import pipeline.

Transaction validation follows three paths:

1. **First-use (implicit key publication)**: The transaction must carry the sender's full PQ public key in the `pubkey` field. The node derives the sender address via §5.4, verifies the PQ signature, and stores the public key hash on-chain.

2. **Returning account**: The stored public key hash is looked up and matched against the transaction's public key before signature verification.

3. **Custom AA account (`AaBundle`)**: An `AaBundle` carries an ordered list of `inner_calls`, an optional `paymaster` address (gas sponsorship), and an optional `session_auth` field (session-key delegation for limited-permission sub-keys). The bundle is verified as an atomic unit: all inner calls succeed or the entire bundle reverts. Multisig policies, time-locks, and future algorithm upgrades—all without address changes—follow from this design, enforced at the protocol level without a smart-contract bundler. `AaBundle` execution is part of the account model and uses the same 32-byte address semantics as standard transactions.

Standard transactions and `AaBundle` execution are fully supported within the protocol-defined account model.

5.6 Key Storage

Shell-Chain adapts the Ethereum Web3 Secret Storage format for PQ keys, replacing PBKDF2/scrypt with Argon2id [BDK16] (memory-hard, providing ASIC/GPU resistance; quantum speedup is limited by the memory-hard structure) and AES-CTR with XChaCha20-Poly1305 [Ber08] (authenticated encryption). All PQ key pairs are generated using the OS CSPRNG (`getrandom` syscall on Linux/macOS), ensuring cryptographically secure randomness independent of user-space state. Raw deterministic key-generation APIs are not intended for ad-hoc production key management; production HD wallets may use deterministic derivation *only* through the Shell PQ-HD v1 scheme (see §5.7), with encrypted seed storage and hardened-only derivation. The legacy single-key keystore format remains:

```
{
  "version": 1,
  "address": "0x<64 lowercase hex characters>",
  "key_type": "dilithium3",
  "kdf": "argon2id",
  "kdf_params": {
    "m_cost": 65536, "t_cost": 3, "p_cost": 4,
    "salt": "<32-byte CSPRNG salt, 64 hex chars>"
  },
  "cipher": "xchacha20-poly1305",
  "cipher_params": { "nonce": "<24-byte nonce, 48 hex chars>" },
  "ciphertext": "<encrypted secret key, hex>",
  "public_key": "<PQ public key, hex>"
}
```

Key derivation uses 64 MiB memory, 3 iterations, and 4-thread parallelism, targeting resistance against attackers with $\leq$1 GiB-DRAM ASICs. Shell-Chain uses the `fips204`-based ML-DSA-65 implementation as the primary FIPS path and keeps the `pqcrypto-dilithium` path active for legacy-compatible deployments; side-channel analysis of the full node implementation is deferred to a dedicated security audit. A 32-byte CSPRNG-generated salt is stored alongside the ciphertext and bound via ChaCha20-Poly1305 AAD to prevent precomputed dictionary attacks. The 192-bit XChaCha20-Poly1305 nonce space eliminates nonce-reuse risk for random nonce selection.

Note: Argon2id's post-quantum security is not formally proven; its primary benefit is memory-hard resistance to GPU/ASIC brute-force, and any quantum speedup is expected to be limited rather than eliminated by the memory-bound structure.

5.7 Hierarchical-Deterministic Key Derivation (Shell PQ-HD v1)

Classical HD wallets (BIP-32 [Wui12]) rely on the additive homomorphism of secp256k1 (child_pub = parent_pub + $H \cdot G$). ML-DSA-65 and SLH-DSA-SHA2-256f have no such algebraic structure, making BIP-32-style non-hardened derivation impossible. Shell PQ-HD v1 defines a *hardened-only* symmetric-KDF tree that is quantum-safe end to end and produces byte-identical keys and addresses in both the TypeScript SDK and the Rust node.

Mnemonic to seed. A 24-word BIP-39 [P+13] mnemonic (256-bit entropy) is NFKD-normalised, lower-cased, and joined with single spaces. The 512-bit seed is:

$$\text{seed}_{512} = \text{PBKDF2-HMAC-SHA512}(\text{mnemonic},\ \text{“mnemonic”} \parallel \text{passphrase},\ 2048)$$

PBKDF2-HMAC-SHA512 is a symmetric-only primitive; its post-quantum security reduces to a quadratic Grover advantage, preserving a $\geq$ 128-bit quantum-security floor at 256-bit entropy.

Master node.

```
KEY_MASTER        = BLAKE3("Shell-Chain PQ-HD master key v1")
I                 = BLAKE3_keyed(KEY_MASTER, seed_512, dkLen=64)
master_secret     = I[0:32]
master_chain_code = I[32:64]
```

Child derivation (hardened-only). For a raw index n ($0 \le n < 2^{31}$):

```
encoded_index = 0x80000000 | n                    // BIP-32 hardened bit
data = "Shell-Chain PQ-HD child v1" || 0x00
     || parent_secret[32] || ser32BE(encoded_index)
I             = BLAKE3_keyed(parent_chain_code, data, dkLen=64)
child_secret  = I[0:32];  child_chain_code = I[32:64]
```

`child_secret` and chain codes are HD-internal and never stored or exported as account private keys.

Derivation path.

$$m \,/\, 9000' \,/\, 8888' \,/\, \text{algo}' \,/\, \text{account}' \,/\, \text{change}' \,/\, \text{address\_index}'$$

All levels are hardened ($'$). `algo'=1` selects ML-DSA-65; `algo'=2` selects SLH-DSA-SHA2-256f. Paths are *network-independent*: the same mnemonic produces the same addresses on testnet and mainnet because Shell addresses are derived from algo_id ‖ pubkey, not from a network byte.

Leaf keypair.

```
KEY_MLDSA_LEAF = BLAKE3("Shell-Chain PQ-HD ML-DSA-65 leaf seed v1")
KEY_SLH_LEAF   = BLAKE3("Shell-Chain PQ-HD SLH-DSA-SHA2-256f leaf seed v1")

ML-DSA-65:
  ml_seed32 = BLAKE3_keyed(KEY_MLDSA_LEAF, child_secret, dkLen=32)
  (pk, sk)  = ml_dsa65.keygen(ml_seed32)
  address   = 0x || BLAKE3(0x01 || pk)[0:32]

SLH-DSA-SHA2-256f:
  slh_seed96 = BLAKE3_keyed(KEY_SLH_LEAF, child_secret, dkLen=96)
               // layout: SK.seed(32) || SK.prf(32) || PK.seed(32)
  (pk, sk)   = slh_dsa_sha2_256f.keygen(slh_seed96)
  address    = 0x || BLAKE3(0x02 || pk)[0:32]
```

The BLAKE3-keyed KDF is Shell-specific and intentionally non-FIPS at the KDF layer; the signature algorithms remain FIPS 204 (ML-DSA-65) and FIPS 205 (SLH-DSA-SHA2-256f). Canonical test vectors that lock the byte-exact pubkey formats, hardened-index encoding, and SLH seed layout are committed at `test-vectors/pq-hd-v1.json` (ADR-011).

Wallet storage. The 512-bit seed is stored under a new keystore type `"hd-seed"`, encrypted with the same Argon2id + XChaCha20-Poly1305 key-wrap used for single-key keystores. The mnemonic is displayed once at wallet creation and never persisted. Per-account data (`path`, `publicKey`, `address`) is derived on demand from the decrypted seed. Legacy single-key keystores remain valid as type `"imported"`.

5.8 Algorithm Agility via On-Chain Governance

Cryptographic standards evolve. New cryptanalytic advances may weaken existing algorithms, or future NIST standardisation rounds may produce superior post-quantum primitives. ShellChain addresses this through a governance-controlled *algorithm registry*, following standard algorithm-agility guidance [Hou15]: the set of `ALLOWED_ALGORITHMS` is stored in genesis state and may be updated by on-chain governance without changing address formats or forcing account-by-account migration.

Algorithm registry. Implementation status: the algorithm registry is a live on-chain component initialised from the compile-time allowlist `ALLOWED_ALGORITHMS = {Dilithium3, MlDsa65, SphincsSha2256f}`. ML-DSA-65 is the primary algorithm; Dilithium3 remains active for legacy compatibility; SLH-DSA-SHA2-256f is the conservative hash-based fallback. The initial active entries map to Dilithium3, ML-DSA-65 (NIST FIPS 204 [Nat24b]), and SLH-DSA-SHA2-256f (NIST FIPS 205 [Nat24c]); identifiers beyond those active entries remain reserved for governance-approved additions. Each registry entry holds:

```
AlgorithmSpec {
  algo_id:           u8,
  name:              &str,   // e.g. "ML-DSA-87"
  pk_size:           u32,    // public-key bytes
  sig_size:          u32,    // signature bytes
  verify_gas:        u64,    // gas cost for PQVERIFY
  batch_gas:         u64,    // gas per-sig in batch verify
  verifier_hash:     [u8; 32],  // BLAKE3 hash of reference verifier bytecode
  status:            Active | Deprecated,
  activation_height: u64,
}
```

Governance process. Any active validator may submit an `AddAlgorithmProposal` governance transaction carrying a candidate `AlgorithmSpec` and a target `activation_height`. Acceptance requires a BFT super-majority: $\lceil 2N/3 \rceil$ ML-DSA-65 signed votes collected within a 7-day voting window (measured in blocks at the prevailing block interval). The `activation_height` must be at least current_height + $\Delta_{\min}$, where $\Delta_{\min} = 30$ days in blocks (approximately 1,296,000 blocks at a 2-second slot time). Proposals with `activation_height` in the past or within $\Delta_{\min}$ blocks are invalid and rejected by consensus. On quorum, the new entry is committed to the registry state. At `activation_height`, the entry transitions to ACTIVE; new accounts may then be created using the approved algorithm.

The quorum threshold is deliberately set equal to the BFT finality threshold (§7): adding a cryptographically weak or backdoored algorithm would require corrupting more than $N/3$ validators—the same adversarial capability required to break finality. This alignment prevents an algorithm-agility channel from becoming a weaker attack surface than the consensus itself.

Bootstrap Safety. A circular dependency exists if the only approved algorithm is ML-DSA-65 and it is compromised: governance votes would require ML-DSA-65 signatures to add a replacement. We resolve this with *dual-algorithm governance*: algorithm registry changes require either (a) ML-DSA-65 $\lceil 2N/3 \rceil$ quorum, *or* (b) SLH-DSA-SHA2-256f $\lceil 2N/3 \rceil$ quorum as an emergency fallback. Genesis validators register both an ML-DSA-65 key and an SLH-DSA-SHA2-256f key. In the event ML-DSA-65 is deprecated, the SLH-DSA-SHA2-256f governance path remains operative at the same BFT quorum threshold. Non-registry governance (parameter changes, upgrades) continues to require only ML-DSA-65 quorum.

Proposal identity and replay prevention. Each `AddAlgorithmProposal` carries:

$$proposal\_id = \text{BLAKE3}(algo\_id \| spec \| activation\_height \| proposer\_pk)$$

Each vote binds `proposal_id`; duplicate `proposal_id` submissions are rejected. This prevents replay attacks where an approved proposal for one algorithm is reused for another. Nodes verify that their locally installed verifier matches the `verifier_hash` field before accepting signatures of the new algorithm type.

Deprecation and migration. A separate `DeprecateAlgorithmProposal` (same $\lceil 2N/3 \rceil$ quorum) sets an entry's status to DEPRECATED. Deprecation prevents *new* address derivation under the affected `algo_id` while all existing accounts remain fully operational. Wallets can then migrate at their own pace by creating a new account under the replacement algorithm and transferring funds—no forced migration event is required.

Network upgrade, not hard fork. Adding an algorithm already implemented and audited in released clients requires no protocol-level fork: governance writes the `AlgorithmSpec` on-chain and the activation height gates when the network may accept it. Introducing a brand-new primitive still requires a coordinated client release that ships the verifier before activation. We therefore distinguish a *network upgrade* (coordinated software update, no chain split) from a *hard fork* (incompatible protocol change causing chain divergence); algorithm addition is intended to be the former only when the verifier is already present on participating nodes.

Address-space isolation. The $algo\_id$ prefix in $\text{BLAKE3}(algo\_id \| pk)$ (full 32-byte output) provides domain separation between accounts under distinct algorithms. Adding a new algorithm does not create a structured collision channel with any pre-existing address; residual collision risk is bounded by the underlying 256-bit BLAKE3 address derivation.

Operational constraint. An `AddAlgorithmProposal` is a standard governance transaction processed by the existing block import pipeline as a state update to the algorithm-registry trie. However, governance can activate only algorithms that have already been implemented, audited, and shipped in a prior client release. Adding a fundamentally new verifier therefore still requires a coordinated software rollout even if activation itself is state-driven rather than fork-triggered.

6 Post-Quantum Virtual Machine (PQVM)

Shell-Chain specifies PQVM as its execution environment: a clean-slate design in which post-quantum security is structural rather than applied after the fact. The execution environment determines how contracts read and write state, how accounts are identified, and how cryptographic operations are priced—each of which the shift to PQ-native primitives reshapes from the ground up.

6.1 Why a New VM?

The most natural path for an Ethereum-derived chain is to extend the EVM: add a `pqrecover` precompile, keep the account model, and preserve bytecode compatibility. We argue this approach fails for the same structural reason as all retrofitting: the classical assumptions are too deeply embedded to patch without cascading incompatibilities.

The 20-byte address is pervasive. The `address` type in Solidity, the ABI encoding standard, and the EVM opcode semantics all encode 20-byte (160-bit) addresses. Changing this requires modifying the Solidity type system, the ABI specification, and the EVM opcode semantics for `ADDRESS`, `CALLER`, `ORIGIN`, and all call opcodes—and recompiling every contract. Once this work is done, compatibility with existing Ethereum bytecode is broken regardless. A clean-slate VM is no harder to migrate to than a heavily patched EVM, and produces a simpler, more principled design.

The ECDSA-based account model cannot be isolated. Ethereum's EOA (Externally Owned Account) concept is baked into the protocol: an EOA is an account whose validity is determined by a fixed ECDSA `ecrecover` rule, not by code. Replacing `ecrecover` with a PQ equivalent still leaves the 20-byte address space and the implicit assumption that account control flows from a single classical key. Keeping the EOA concept alongside a PQ-verified wallet creates two classes of accounts with different security properties—precisely the kind of residual vulnerability that a clean-slate design avoids.

Classical precompiles create quantum-vulnerable contracts. Ethereum precompiles 0x01–0x09 include `ecrecover` (ECDSA), `bn256Add`, `bn256ScalarMul`, and `bn256Pairing` (pairing-based cryptography). A PQ-native chain that retains these precompiles allows contracts to depend on quantum-vulnerable cryptography, undermining the PQ-native guarantee. Any contract calling `ecrecover` or BN256 pairings would have a classical attack surface regardless of the underlying transaction authentication.

Design decision. PQVM retains EVM-familiar arithmetic, memory, storage, and control-flow opcodes while replacing the wider cryptographic layer: address model, account type, precompile set, transaction format, and ABI encoding. The resulting system makes post-quantum security a structural property—not a layer added on top.

Patched EVM vs. PQVM. Table 7 compares a patched EVM approach with the clean-slate PQVM design.

Table 7: Patched EVM vs. clean-slate PQVM

Approach	Address model	ABI impact	AA support	Crypto surface	Tooling reuse
Patched EVM	20-byte legacy type retained or shimmed	Breaks at every `address`-typed interface	Overlay-style only	Classical precompiles linger unless removed	Highest short-term bytecode reuse
PQVM	Native 32-byte `PQAddress`	Single explicit ABI transition	Native protocol-level AA	Clean PQ-native opcode and precompile surface	Reuse of EVM-familiar semantics and tooling patterns

6.2 PQVM Architecture

Shell-Chain uses a stack-based, 256-bit word-width PQVM architecture that follows EVM execution semantics closely enough to support Solidity contracts, while replacing the cryptographic layer for a clean-slate PQ design.

Execution model. The PQVM execution model follows the EVM in structure:

- **Stack-based**: all operands are pushed and popped from a 1024-element stack of 256-bit words.
- **256-bit word width**: integer arithmetic operates on 256-bit big-endian unsigned integers, matching EVM semantics for all arithmetic opcodes.
- **Linear memory**: byte-addressable, expandable in 32-byte chunks; gas-charged per word expansion.
- **Persistent storage**: key-value store mapping 256-bit keys to 256-bit values, accessed via `SLOAD`/`SSTORE`.
- **Call frames**: each call creates an isolated context with its own stack, memory, and gas counter; return data flows through `RETURNDATASIZE`/`RETURNDATACOPY`.

PQ-native address type: `PQAddress`. In Shell-Chain, `PQAddress` is a first-class 32-byte (256-bit) address type:

$$addr = \text{BLAKE3}(algo\_id \| pk) \tag{2}$$

where $algo\_id$ is a one-byte algorithm identifier providing domain separation, and pk is the serialized public key. The full 256-bit BLAKE3 output is used—no truncation. User-facing addresses are encoded as `0x`-prefixed 64-character lowercase hex. Shell-Chain starts from its own genesis; all addresses are 32-byte `0x`-encoded from day one. Address-returning opcodes (`ADDRESS`, `CALLER`, `ORIGIN`, `COINBASE`) return 32-byte values and the Solidity `address` type occupies 32 bytes in all ABI-encoded contexts (see §6.8).

6.3 Native Account Abstraction

Shell-Chain eliminates the EOA/contract distinction entirely. Native account abstraction uses a three-path validator model: first-use key publication, returning-account verification against the stored PQ public-key hash, and an optional account-specific `validation_code_hash` path.

Implementation status. Reference implementation. The protocol includes native `AccountManager` and `ValidatorRegistry` system contracts at deterministic 32-byte addresses, supports key rotation without changing the account address, and exposes custom validation through account-managed validator code. `AaBundle` execution is implemented as part of the same native account model and uses 32-byte addresses throughout.

Address binding and key rotation. The first transaction from an account binds the account address to the initial key via BLAKE3($algo\_id \| public\_key$). After the account object exists, the address is the stable account identifier and validation uses the account state's stored public-key hash or custom validator code. Key rotation therefore updates the account's validation material; it does not rederive a new address from the replacement key. This distinction avoids a contradiction between key-derived first-use addresses and stable account identity.

Why eliminate the EOA concept entirely? The EOA/contract distinction in Ethereum forces all innovation in account security through ERC-4337 or similar application-layer overlays, which still depend on a base-layer authentication primitive (classically, ECDSA). Native AA places authentication logic in code—where it is inspectable, upgradeable, and governed by the same rules as all other on-chain logic. For a PQ-native chain starting from genesis, there is no existing EOA base to maintain compatibility with; native AA costs nothing in terms of migration burden while providing a strictly more powerful account model.

6.4 Instruction Set

PQVM-1 ISA retains the full EVM arithmetic, memory, storage, and control-flow opcode set for developer familiarity, removes cryptographically compromised instructions, and adds three PQ-native opcodes.

PQVM specifies an Ethereum-Cancun-compatible ISA minus blob opcodes (`BLOBHASH`, `BLOBBASEFEE`), plus PQ-native opcodes `PQVERIFY` (0xB0), `PQHASH` (0xB1), `PQADDR` (0xB2). New Cancun opcodes `PUSH0`, `MCOPY`, `TLOAD`, `TSTORE` are retained.

Retained opcodes (EVM-identical semantics). All of the following opcodes are retained with unchanged semantics:

- **Arithmetic**: `ADD`, `MUL`, `SUB`, `DIV`, `SDIV`, `MOD`, `SMOD`, `ADDMOD`, `MULMOD`, `EXP`, `SIGNEXTEND`
- **Comparison/bitwise**: `LT`, `GT`, `SLT`, `SGT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, `NOT`, `BYTE`, `SHL`, `SHR`, `SAR`
- **Hashing**: `SHA3`/`KECCAK256` (retained for Solidity storage slot compatibility)
- **Environment**: `ADDRESS` (32 B), `BALANCE`, `ORIGIN` (32 B), `CALLER` (32 B), `CALLVALUE`, `CALLDATALOAD`, `CALLDATASIZE`, `CALLDATACOPY`, `CODESIZE`, `CODECOPY`, `GASPRICE`, `EXTCODESIZE`, `EXTCODECOPY`, `RETURNDATASIZE`, `RETURNDATACOPY`, `EXTCODEHASH`
- **Block**: `BLOCKHASH`, `COINBASE` (32 B), `TIMESTAMP`, `NUMBER`, `DIFFICULTY`/`PREVRANDAO`, `GASLIMIT`, `CHAINID`, `SELFBALANCE`, `BASEFEE`
- **Memory**: `MLOAD`, `MSTORE`, `MSTORE8`, `MSIZE`
- **Storage**: `SLOAD`, `SSTORE`
- **Control**: `JUMP`, `JUMPI`, `PC`, `GAS`, `JUMPDEST`, `PUSH1`–`PUSH32`, `DUP1`–`DUP16`, `SWAP1`–`SWAP16`
- **Logging**: `LOG0`–`LOG4`
- **System**: `STOP`, `RETURN`, `REVERT`, `INVALID`, `CREATE`, `CREATE2`, `CALL`, `DELEGATECALL`, `STATICCALL`

Removed opcodes. `SELFDESTRUCT` is eliminated in the PQVM design, extending the Cancun EIP-6780 policy: there is no configuration under which `SELFDESTRUCT` has destructive effect in PQVM. `CALLCODE` is also removed: its legacy call semantics predate modern contract patterns and provide no necessary capability once `DELEGATECALL` and native account abstraction exist. All classic Ethereum cryptographic precompiles are absent from the PQVM precompile surface; low addresses `0x0001`–`0x0006` are reserved for the Shell PQ suite described below.

New PQ opcodes. Table 8 specifies the three new PQ-native opcodes.
PQVERIFY enables contracts to verify PQ signatures inline, without a precompile call. The `algo_id` argument uses the same byte encoding as `sig_type`: `0x00` for Dilithium3, `0x01` for ML-DSA-65, `0x02` for SLH-DSA-SHA2-256f. Gas cost is proportional to verification latency (see §6.7).

PQHASH exposes BLAKE3-256 natively within the PQVM, enabling contracts to perform address derivation, Merkle tree construction, and other hash-based operations using ShellChain's native hash function. The gas cost ($30 + 6$ per 32-byte word) reflects that BLAKE3 is faster than Keccak-256 in practice.

PQADDR derives a `PQAddress` from a public key, enabling contracts to reconstruct addresses for verification without off-chain computation.

Table 8: PQVM-1 New Opcodes

Opcode	Byte	Stack I/O	Description
`PQVERIFY`	`0xB0`	`sig_ptr, sig_len, pk_ptr, pk_len, msg_ptr, msg_len, algo_id` → `result`	PQ signature verification. Returns 1 (valid) or 0 (invalid/error).
`PQHASH`	`0xB1`	`data_ptr, data_len, out_ptr` → (side effect)	BLAKE3-256 hash; writes 32 bytes to memory at `out_ptr`.
`PQADDR`	`0xB2`	`algo_id, pk_ptr, pk_len` → `address`	Derive PQ-native 32-byte address from a public key.

6.5 Transaction Format

PQVM defines a new transaction type `PQTx` that carries PQ signature material as explicit typed fields:

```
PQTx {
  chain_id:               u64        // replay protection
  nonce:                  u64        // account nonce
  to:                     PQAddress? // null = contract creation
  value:                  U256       // native token (wei)
  data:                   bytes      // calldata or init code
  gas_limit:              u64        // execution gas limit
  max_fee_per_gas:        u64        // EIP-1559 max fee (wei per gas)
  max_priority_fee_per_gas: u64      // EIP-1559 priority fee (wei per gas)
  tx_type:                u8         // 0 legacy, 1 access, 2 EIP-1559, 3 blob
  // blob-only fields (tx_type == 3):
  max_fee_per_blob_gas:   Option<u64>
  blob_versioned_hashes:  Option<[Hash]>
  // PQ authentication fields:
  sig_type:   u8            // 0x00 Dilithium3, 0x01 ML-DSA-65, 0x02 SLH-DSA
  public_key: Option<bytes> // first use only
  signature:  bytes         // 3309 B lattice, 49856 B SLH-DSA
}
```

Canonical encoding and signing envelope. Consensus serialization for `PQTx` is byte-exact: integer fields are unsigned big-endian fixed-width values (`u64` = 8 bytes, `u8` = 1 byte, `U256` = 32 bytes); `bytes` fields are encoded as `u32_be length || raw bytes`; optional fields are encoded as a one-byte tag (`0x00` absent, `0x01` present) followed by the value when present; arrays are encoded as `u32_be count` followed by each fixed-size element. The signed preimage is the canonical encoding of:

$$\begin{aligned}\mathsf{PQTX\_SIGNING\_V1} \| &\mathit{chain\_id} \| \mathit{nonce} \| \mathit{to} \| \mathit{value} \| \mathit{data} \| \mathit{gas\_limit} \| \\ &\mathit{max\_fee\_per\_gas} \| \mathit{max\_priority\_fee\_per\_gas} \| \mathit{tx\_type} \| \mathit{sig\_type} \| \\ &\mathit{max\_fee\_per\_blob\_gas}? \| \mathit{blob\_versioned\_hashes}?.\end{aligned}$$

The *signed payload* is BLAKE3(*canonical_signed_preimage*). For EIP-4844 blob transactions (*tx_type* = 3), the two blob-only fields above must be present; for all other transaction types they must be absent. Binding `chain_id` prevents cross-chain replay; binding `sig_type` prevents algorithm substitution attacks (a signature produced under Dilithium3 cannot be replayed as ML-DSA-65 or SLH-DSA-SHA2-256f). Here `sig_type` serves as the

`algo_id` domain separator. The `public_key` and `signature` fields are outside the signed payload. They are authentication material: on first use the public key must derive the sender address BLAKE3(*sig_type*‖*public_key*), and for returning accounts it must match the public-key hash stored in account state. This separation keeps the transaction hash stable across witness availability while still binding the signature to the authenticated key.

Transaction validation flow.

1. Deserialize and size-check `public_key` and `signature` against algorithm-specific safe upper bounds (`MAX_ML_DSA_65_SIG_BYTES` = 4,096 B; `MAX_SIGNATURE_BYTES` = 51,200 B for SLH-DSA-SHA2-256f).
2. Derive the sender address: *sender* = BLAKE3(*sig_type*‖*public_key*) (full 32-byte output, no truncation), where `sig_type` serves as the `algo_id` domain separator.
3. Look up the sender account; check nonce and balance. If the sender address is empty, initialize it with the reference `PQAccount` code hash after the signature check succeeds.
4. If sender has code, call `sender.validateSignature(tx_hash, sig_type, public_key, signature)`. For first-use accounts, the protocol applies the same `PQAccount` validation logic to the supplied public key before installing the account code hash.
5. On success, execute the transaction body with `CALLER` set to the derived sender address.

Wire size. A `PQTx` with Dilithium3 or ML-DSA-65 carries 1,952 + 3,309 = 5,261 bytes of key/signature data on first use, and 3,309 bytes for returning accounts. With SLH-DSA-SHA2-256f, first-use overhead is 64 + 49,856 = 49,920 bytes. These costs are directly reflected in the transaction base gas cost (§6.7). The `public_key` field is present on first use (when the account has not previously appeared on-chain) and omitted for returning accounts where the key is already stored in state, reducing transaction size.

6.6 PQ Precompile Suite

Shell-Chain exposes a six-precompile PQVM suite covering verification, hashing, and address derivation. Table 9 records the implemented clean-slate PQVM precompile surface.

Table 9: PQVM Precompile Suite

Address	Name	Gas	Description
`0x00..01`	ML-DSA-65 Verify	46,000	Verify one ML-DSA-65 signature
`0x00..02`	SLH-DSA-SHA2-256f Verify	2,300,000	Verify one SLH-DSA-SHA2-256f signature
`0x00..03`	ML-DSA-65 Batch Verify	$12{,}000 \times n$	Verify n ML-DSA-65 sigs in parallel
`0x00..04`	BLAKE3-256	$30 + 6$/word	BLAKE3 hash to 256 bits
`0x00..05`	BLAKE3-512	$30 + 6$/word	BLAKE3 hash to 512 bits
`0x00..06`	PQAddress Derive	200	Derive 32-byte address from PQ public key

Precompile `PQAddresses` occupy the reserved range $[0, 2^{32})$; in full 32-byte form, precompile with ID N occupies the address whose bytes are the big-endian 32-byte encoding of N, rendered as `0x`-prefixed 64-character hex.

Precompile call ABI. All PQ precompiles are deterministic and side-effect free. Invalid input does not revert caller state; the precompile returns an explicit failure value unless calldata is malformed beyond decoding. Verification precompiles consume:

```
u32_be pk_len || pk || u32_be msg_len || msg || sig
```

and return exactly 32 bytes: all zero for invalid/error and `0x...01` for valid. Batch verification at `0x00..03` uses `u32_be count` followed by that tuple repeated `count` times and returns the same 32-byte boolean. Hash precompiles return the raw digest padded/truncated to the advertised length (32 bytes for BLAKE3-256, 64 bytes for BLAKE3-512). The address precompile consumes `u8 algo_id || pubkey` and returns the 32-byte address BLAKE3($algo\_id \| pubkey$). Oversized inputs above algorithm-specific bounds fail closed and return zero.

Design rationale. The precompile set covers the three primitive operations required by the PQ account model: verification (0x0001, 0x0002, 0x0003), hashing (0x0004, 0x0005), and address derivation (0x0006). Batch verification (0x0003) is a performance-critical primitive for multisig contracts: its $12{,}000 \times n$ cost reflects parallel execution on multi-core validators, approximately 26% of the serial per-signature cost, consistent with measured parallel speedup on reference hardware (§11.3). A single batch call is capped at **256 signatures** (`MAX_BATCH_SIGNATURES`); inputs exceeding this limit are rejected with an out-of-gas result to bound denial-of-service risk.

Precompile 0x0001 algorithm dispatch. Precompile 0x0001 implements the ML-DSA family: it attempts ML-DSA-65 verification first (primary path, FIPS 204) and falls back to Dilithium3 for legacy-compatible signatures. This dual-path behaviour is intentional and consistent with the active algorithm set defined in §5.8 (`ALLOWED_ALGORITHMS = {Dilithium3, MlDsa65, SphincsSha2256f}`). Dilithium3 remains an active algorithm until governance formally retires it via the algorithm registry (`proposeAlgorithmActivation` / deactivation path).

Absent Ethereum precompiles. Ethereum precompiles 0x01–0x09 (`ecrecover`, SHA-256, RIPEMD-160, identity, modexp, BN256Add, BN256ScalarMul, BN256Pairing, BLAKE2f) are removed. Contracts depending on BN256 pairings (e.g., Groth16 ZK-proof verifiers deployed on Ethereum) therefore require migration to a PQ-safe proof system.

6.7 Gas Model

PQVM inherits the EVM gas schedule for arithmetic, memory, storage, logging, and call operations without change. PQ-specific operations receive calibrated costs reflecting their actual latency on reference hardware. Table 10 summarizes the Shell-Chain PQVM gas schedule.

Calibration methodology. PQ-specific gas costs are calibrated from relative measured execution time on reference hardware (Apple M2 Pro, Rust 1.87 `--release`) using `criterion` [Hc19] microbenchmarks. Gas is an abstract resource unit rather than a direct wall-clock conversion: expensive PQ operations are priced high enough to bound denial-of-service risk while preserving headroom for mixed workloads under the 50,000,000-gas block limit.

Block gas limit implications. The hard cap of 500 transactions per block is the binding ceiling for any realistic block composition: 500 simple ML-DSA-65 transfers consume $500 \times 67{,}000 = 33{,}500{,}000$ gas, well within the 50,000,000-gas block limit. The gas limit (≈ 746 simple transfers) provides additional headroom for mixed workloads. For contracts making additional `PQVERIFY` calls (e.g., multisig wallets), throughput is proportionally lower but bounded. The 2,300,000-gas cost of SLH-DSA-SHA2-256f single verification limits SLH-DSA to high-assurance

Table 10: PQVM PQ-Specific Gas Costs

Operation	Gas	Rationale
`PQVERIFY` (ML-DSA-65)	46,000	$\approx$46 μs single verify
`PQVERIFY` (SLH-DSA-SHA2-256f)	2,300,000	$\approx$2.3 ms single verify
Precompile 0x0001 (ML-DSA-65 single)	46,000	Same cost as inline `PQVERIFY`
Precompile 0x0002 (SLH-DSA-SHA2-256f)	2,300,000	Same cost as inline `PQVERIFY`
Precompile 0x0003 (ML-DSA-65 batch, per sig)	12,000	Parallel; $\approx$12 μs amortised
`PQHASH` (base + per 32-byte word)	$30 + 6$	BLAKE3 faster than Keccak-256
`PQADDR`	200	Address derivation (BLAKE3)
Tx base cost (ML-DSA-65)	67,000	21,000 base + 46,000 ML-DSA verify precompile
Tx base cost (SLH-DSA-SHA2-256f)	2,321,000	21,000 base + 2,300,000 sig validation

operations; a block at the current gas limit can contain at most approximately 21 SLH-DSA-SHA2-256f transactions.

Gas Model Scope. Gas measures *PQVM execution cost* only: opcode execution, memory, storage, and precompile invocations. Block proposer signature verification, BFT attestation collection, and P2P message passing occur outside the gas model, analogous to Ethereum's separation of consensus overhead from EVM gas metering. The 2-second slot target encompasses end-to-end block import time (network propagation + signature verification + PQVM execution + storage), not PQVM gas alone.

6.8 PQABI Encoding

PQABI extends the Ethereum ABI specification with a single change to the `address` type.

The address type. In the Ethereum ABI, `address` is a 20-byte value zero-padded to a 32-byte slot. In PQABI, `address` is a full 32-byte value occupying the entire slot with no padding:

Table 11: ABI Slot Encoding: Ethereum vs. PQABI

Specification	`address` encoding	Slot
Ethereum ABI	12 zero bytes $\|\|$ 20-byte value	32 bytes
PQABI	32-byte `PQAddress` value (no padding)	32 bytes

Unchanged elements. All other ABI types are identical to Ethereum ABI: `uint`N/`int`N (1–32 bytes, zero-padded), `bytes`/`string` (dynamic, length-prefixed), `bytes`N (fixed), `bool`, `tuple`, static and dynamic arrays. Function selectors remain the 4-byte keccak256(*signature*) scheme, maintaining compatibility with Foundry, Hardhat, and other existing tooling. Function selectors in PQABI retain Keccak-256 for tool-chain compatibility (Foundry, Hardhat, cast) while all other hashing uses BLAKE3. This selective retention limits Keccak-256 exposure to the 4-byte selector namespace.

Migration impact. Contracts that do not use the `address` type require no ABI changes. Non-address storage slots (`uint256`, `bytes32`, etc.) are unchanged. Within PQVM, storage slots

of `address` type hold 32-byte `PQAddress` values, and contracts with `address` parameters use encoders that emit 32-byte (not 12-zero + 20-byte) values. All contracts use 32-byte `PQAddress` from the outset. Function selectors remain identical since keccak256 of the type signature is unchanged.

6.9 Smart Contract Migration

PQVM is *not* bytecode-compatible with the EVM. All contracts must be recompiled. However, the migration cost is bounded and predictable: the required source-level changes are minimal for contracts that do not use classical crypto primitives.

Required source changes.

1. **Address type**: Solidity `address` (20 bytes) becomes a 32-byte PQVM `address`. This is a type-size change; contract logic (comparisons, mappings, event emission) is unaffected.
2. **`ecrecover` calls**: Replace with a call to precompile `0x00..01` (ML-DSA-65 Verify) or the inline `PQVERIFY` opcode. Libraries such as OpenZeppelin's `ECDSA.sol` require a PQ-equivalent replacement.
3. **BN256 precompile calls**: Contracts calling 0x06–0x08 (BN256 pairings, used in Groth16 verifiers) must migrate to PQ-safe proof systems. STARK verifiers using Keccak-256 or BLAKE3 do not require changes.
4. **Recompile with PQVM target**: All contracts must be compiled with the PQVM-targeting compiler flag (`--pqvm`) that enforces 32-byte address encoding and emits PQABI-conformant output.

What does not change. Contract logic, state layout (storage slot assignments are unchanged since they use `keccak256` of slot keys), event definitions, function selectors, and the calling convention are all preserved. Contracts not using `address` comparisons, `ecrecover`, or BN256 precompiles may migrate by recompile with zero logic changes. The 256-bit word arithmetic, memory model, and gas accounting for non-PQ operations are identical to the EVM.

Comparison with EVM-patching. The migration burden of PQVM is comparable to—and in practice no greater than—the burden of deploying on a heavily patched EVM with 32-byte address extensions. A patched EVM would require the same Solidity address-type change, the same `ecrecover` replacement, and the same ABI encoding change, *plus* the additional complexity of maintaining backward compatibility with 20-byte address tooling. PQVM avoids this complexity entirely by making the 32-byte address the only address type from genesis.

7 Consensus Mechanism

Classical BFT consensus—PBFT [CL99], Casper FFG [BHK+20]—assumes that validator identity proofs cannot be forged by a computationally bounded adversary. Under a quantum adversary, this assumption fails: ECDSA validator keys are recoverable in polynomial time by Shor's algorithm.

Shell-Chain ensures that every protocol message—block proposals, attestations, equivocation proofs—is authenticated with a PQ signature. The mechanism has two distinct layers that solve two distinct sub-problems.

Shell-Chain's target finality protocol is wPoA. Plain PoA remains a simpler compatibility and block-production mode, but deterministic single-block finality is provided by the wPoA

path, which consists of two components: (1) the *Weighted Round-Robin (WRR) leader schedule*, which determines the proposer for each slot in proportion to validator weights; and (2) the *BFT Attestation Finality layer*, which provides safety and liveness guarantees independent of the proposer schedule.

7.1 Layer 1: Weighted Proof-of-Authority for Block Production

Why wPoA and not PoS or DPoS? Three consensus models were considered for the initial deployment:

- **Proof-of-Stake (PoS)**: permissionless but requires slashing economics, stake liquidity management, and Sybil-resistance via staking — adding significant complexity to an already-novel PQ cryptographic layer.
- **Delegated PoS (DPoS)**: reduces validator count but introduces delegation governance complexity and is vulnerable to stake concentration attacks.
- **Proof-of-Authority (PoA) / wPoA**: requires a trusted validator authority for set management. This is appropriate for an initial deployment where the validator set is known, audited, and contractually bound. The permissioned model simplifies the security analysis and allows the PQ cryptography layer to be the primary engineering challenge.

The design makes this trade-off explicitly: wPoA is the right choice for a *launch network* with a known validator set. Permissionless validator admission (via PQ-secured staking) is planned for a future network upgrade. Proof-of-Work is additionally incompatible with a permissioned validator set: it provides Sybil resistance via energy expenditure, which is unnecessary when validators are identity-verified and pre-registered. The *weighted* variant—wPoA—adds governance-assigned block-frequency proportional to each validator's weight, allowing the consortium to reflect trust asymmetry without a secondary token market. Weights are set at genesis and adjustable via on-chain governance.

Validator lifecycle. Validators are identified on-chain by their post-quantum public key. Each validator is assigned a weight $w_i \geq 1$ at genesis (or via on-chain governance), reflecting its authority within the permissioned validator set. Let $W = \sum_{i=1}^{N} w_i$ be the total weight of the N active validators.

Weighted round-robin proposer selection. Block b is proposed by the validator v_j whose cumulative weight window contains $(b \bmod W)$:

$$\sum_{i=1}^{j-1} w_i \;\leq\; (b \bmod W) \;<\; \sum_{i=1}^{j} w_i \tag{3}$$

This deterministic mapping is weight-proportional: validator v_j proposes a fraction w_j/W of all blocks. WRR provides *deterministic, publicly verifiable* proposer ordering—appropriate for a permissioned consortium where the validator set is trusted. Unlike VRF-based leader selection, WRR is fully predictable, simplifying auditing but enabling long-horizon scheduling analysis by adversaries. The deterministic nature of WRR allows a sophisticated adversary to predict the proposer schedule and pre-position targeted attacks. A per-epoch randomised shuffle using a committed VRF output would mitigate long-horizon prediction; this is a known limitation of the current design (see §14).

The design is consistently weight-based: validator *weights* govern both proposer frequency and BFT attestation quorum. A quorum requires attestations whose total active weight exceeds $2W/3$ (see §7.2), preventing any coalition with less than one third of total weight from finalising conflicting blocks. Selection requires only an $O(N)$ scan over cumulative weights—practical for $N \leq 1{,}000$ (the protocol ceiling; initial deployments target $N = 100$).

Block sealing. The proposer seals the block by attaching a PQ signature $\sigma_{seal} \leftarrow \mathsf{Sign}(sk_j, \mathsf{Hash}(H))$ over the block header hash H. Validators reject any block whose `proposer` field does not match Equation (3) or whose seal fails PQ verification. The sealed block hash is computed as $\mathsf{BLAKE3}(\mathit{canonical\_encode}(\mathit{header}))$ where $\mathit{canonical\_encode}$ serializes all header fields in a fixed, length-prefixed format ensuring unambiguous binding.

Equivocation detection. A proposer that signs two distinct blocks at the same height produces a pair of valid PQ signatures on conflicting messages—an equivocation proof that is itself quantum-resistant. The slashing module accepts such proofs and reduces the offending validator's weight.

Proposer timeout and view-change. If the slot's proposal deadline passes without a valid block, validators enter view-change rather than unilaterally finalising a substitute block. The implementation uses a conservative timeout derived from the slot duration and deployment profile; the protocol requirement is simply that honest validators apply the same timeout rule for a given height and view. Timeout-based rotation keeps the chain live when individual proposers are unreachable, subject to the partial-synchrony assumptions in Theorem 3.

Slot lifecycle. Each 2-second slot proceeds: (1) 0–100 ms: proposer broadcasts block; (2) 100 ms–1.5 s: validators verify and broadcast attestations; (3) 1.5 s–2.0 s: any node may assemble attestations whose total weight exceeds $2W/3$ into a QC; (4) if no valid proposal is observed by the proposal deadline, validators start the view-change state machine for the same height rather than producing an uncoordinated local fork.

View-change state machine. For each height h, validators track a monotonically increasing *view* number r. The initial proposer is selected by Equation (3); later views rotate through the same ordered active validator set using $(\mathit{wrr\_index}(h)+r) \bmod N$. Weights affect quorum, not the number of messages required. The state machine is:

$$\textsc{Normal}(h,r) \to \textsc{Timeout}(h,r) \\ \to \textsc{ViewChange}(h,r{+}1) \to \textsc{NewView}(h,r{+}1) \to \textsc{Propose}(h,r{+}1).$$

A validator enters Timeout only after its local timeout expires without a valid proposal or QC for (h,r). It then broadcasts a signed `ViewChangeMessage`$(h, r{+}1, \mathit{last\_finalized}, \mathit{highest\_qc})$ over GossipSub. Once messages for the same $(h,r{+}1)$ exceed the weighted quorum $\lceil 2W/3 \rceil$, the new proposer may issue a block for the next view. Messages bind `chain_id`, height, view, sender address, and highest known QC to prevent replay across chains or heights.

The safety argument is the same quorum-intersection argument used for finality. Two conflicting QCs for the same height, even in different views, would need two sets of weight $> 2W/3$; their intersection has more than $W/3$ weight. At least one honest validator in that intersection would then have signed conflicting attestations for the same height, which honest validators never do. View-change restores liveness without weakening finalized-block safety, though it does not address the WRR schedule-predictability limitation discussed in §14.

7.2 Layer 2: BFT Attestation for Deterministic Finality

Why not probabilistic finality? Nakamoto-style probabilistic finality ("wait k confirmations") provides no formal BFT safety guarantee. Under a quantum adversary capable of rapidly forging validator keys, a block that has received $k-1$ confirmations might still be reorged if the adversary can forge a competing chain. Deterministic BFT finality closes this attack window: once finalized, a block cannot be reversed regardless of subsequent chain extension, even by an adversary with unlimited signing power.

Attestation protocol. After accepting a block, each validator broadcasts an `Attestation` message over GossipSub, signed with its PQ key. Attestation messages are broadcast over GossipSub topics (see §8); today those connections still run over standard libp2p Noise/XX, with migration to PQ Noise deferred to the planned transport upgrade (§8.1). The attestation signing payload is:

$$att_i = \mathsf{Sign}\big(sk_i,\ \mathsf{H}(chain\_id \parallel epoch \parallel parent\_hash \parallel block\_hash \parallel height \parallel round)\big)$$

Binding `chain_id` and `parent_hash` prevents cross-fork and cross-chain replay attacks. The `FinalityState` tracker records received attestations:

Definition 4 (Weighted quorum)**.** *Given active validators with total weight W, the* weighted quorum *threshold is any attestation set with total weight strictly greater than $2W/3$. A block is finalized when valid attestations exceeding that threshold have been submitted for it.*

Any validator that has collected a quorum of valid attestations may assemble the Quorum Certificate (QC) and propagate it over GossipSub. Any node can independently verify the QC from the validator bitmap and individual PQ signatures.

Proposition 1 (Weighted BFT Safety)**.** *If Byzantine validators control total weight strictly less than $W/3$, no two conflicting blocks can both achieve finalization under the $> 2W/3$ weighted quorum rule.*

Proof sketch. Suppose conflicting blocks B and B' at the same height both receive weighted quorum certificates. Each certificate has weight greater than $2W/3$, so their intersection has weight greater than $W/3$. If Byzantine weight is strictly less than $W/3$, at least one honest validator by weight must be in the intersection and therefore attested to both conflicting blocks—a contradiction, since honest validators attest at most once per height. □

Because every attestation carries a PQ signature, no CRQC can forge an attestation from an honest validator. This is a simplified *single-phase BFT protocol*: a proposer broadcasts a block; validators attest; any node assembling valid attestations whose total weight exceeds $2W/3$ holds a Quorum Certificate (QC) and triggers finalization. This is distinct from two-phase protocols like PBFT [CL99] and includes a view-change mechanism for Byzantine proposer resilience (§7).

Synchrony and liveness scope. The finality safety argument does not require synchrony: two conflicting finalized blocks would violate weighted quorum intersection. Liveness is qualified. After GST, if message delays are bounded and view-change messages for a timed-out proposer reach weighted quorum, the protocol rotates until an honest active proposer can produce a valid block. The view-change mechanism for Byzantine leader resilience is implemented: validators broadcast signed `ViewChangeMessage`s, and rotation proceeds when the weighted quorum $\lceil 2W/3 \rceil$ is reached. (The liveness analysis assumes a partially synchronous network model [DLS88]; see §14 for the network partition assumption.)

No in-protocol classical fallback. In the production protocol, ECDSA is fully eliminated and consensus messages never admit a classical-signature alternative. Off-chain adapters may help legacy tooling construct PQ-native transactions, but such adapters terminate before consensus and do not widen the set of cryptographically valid on-chain messages. This preserves the quantum-safe-genesis property argued in Section 2.

7.3 Deployment Parameters

The following genesis parameters are hard-coded in the default `ChainConfig` (testnet / initial mainnet) and may be revised via on-chain governance:

The 2-second block time target is a performance goal that holds in normal operation post-GST. Network partitions or Byzantine proposers may delay finality until synchrony is restored.

Parameter	Value	Notes
Block interval (active)	2 s	Testnet / mainnet default when transactions are executable
Heartbeat interval (idle)	configurable	Empty slots are skipped; `max_idle_interval` controls heartbeat emission
Finality quorum	weighted $> 2W/3$	WPoA finality threshold over active validator weight
Max validators (deploy)	100	Deployment target; protocol ceiling `MAX_VALIDATORS`= 1,000
Max tx per block	500	`max_tx_per_block` (testnet / mainnet)
Max block gas	50,000,000	Current testnet / mainnet cap
Max wire message	50 MiB	`MAX_MESSAGE_SIZE` in `shell-network`
Witness pruning default	immediate on accepted proof	STARK-enabled nodes default to `proof_replacement_grace = 0`; retention value `0` means retain all

Table 12: Deployment parameters (current testnet / initial mainnet).

Fork-choice rule. During the pre-finalization window, nodes follow the fork-choice rule: select the branch with the highest finalised height; if heights are equal, prefer the branch with the greater cumulative validator weight; if still tied, use the lowest block hash as the deterministic final fallback. This matches the current `ForkChoice` implementation.

Quorum Certificate bandwidth. Each QC contains a validator bitmap (125 bytes at the `MAX_VALIDATORS` ceiling of $N = 1{,}000$) and individual ML-DSA-65 signatures (3,309 bytes each). In an equal-weight $N = 1{,}000$ deployment, a minimal weighted quorum needs 667 signatures and occupies approximately 2.2 MB. This estimate assumes all validators use ML-DSA-65. A mixed validator set with SLH-DSA-SHA2-256f validators would increase QC size proportionally, so the network should prefer ML-DSA-65 for attestations. This overhead is incurred once per finalized block and is not on the critical path for block propagation.

At mainnet launch the validator set will be bootstrapped from a permissioned genesis list; governance expansion is planned for a subsequent protocol upgrade.

8 Network Layer

All layers of Shell-Chain—application, consensus, and transport—target post-quantum security.

8.1 Post-Quantum Network Transport (PQ Noise)

Deployment boundary: Nodes use libp2p with the classical Noise/XX pattern (X25519 + Ed25519). Migration to PQ Noise is planned when the libp2p upstream adds ML-KEM-768 support [Pro20].

Protocol choice. Shell-Chain's P2P layer (`shell-network`) is built on `libp2p` [Pro20] with the following configuration:

- **Transport**: TCP + Noise handshake (classical Noise/X25519 currently; PQ Noise target) + Yamux multiplexing.
- **Broadcast**: GossipSub with per-peer scoring to mitigate eclipse and amplification attacks.

- **Discovery**: mDNS (local) + Kademlia DHT (wide-area) + boot-node seeding.
- **Connection control**: limit policies, DCUTR/AutoNAT NAT hole-punching, relay support.
- **Bandwidth accounting**: per-peer ingress/egress tracking for rate limiting.

Rationale for PQ Noise over alternatives. Shell-Chain evaluated three transport alternatives: (1) *Custom bespoke protocol* (PQTP-1 design): provides full control but requires independent formal analysis and long-term maintenance burden; (2) *Hybrid TLS 1.3 + ML-KEM*: production-ready and IETF-standardised [TT21, SFB23], but retains classical key exchange in the handshake and therefore is not purely post-quantum; (3) *PQ Noise over libp2p*: builds on the existing libp2p ecosystem Shell-Chain already depends on, inherits the Noise protocol's compositional security framework [Per18], and the PQ-Noise formal extension [ADH+22] provides a security reduction to ML-KEM-768 IND-CCA2 and ML-DSA-65 EUF-CMA. Shell-Chain adopts PQ Noise as the superior option: community-maintained infrastructure, formal security model, and pure post-quantum primitives.

Why transport-layer PQ security matters. Application-layer security—transaction and block signatures—is fully post-quantum. However, a cryptographically relevant quantum computer (CRQC) that records today's network traffic could *retroactively* recover classical Noise session keys from stored handshake transcripts, exposing peer identities and network topology metadata. Although this does not compromise transaction or block integrity (PQ signatures are permanently on-chain), it constitutes a live classical attack surface in validator P2P communication. PQ Noise closes this gap.

PQ Noise framework. Rather than designing a bespoke transport protocol, Shell-Chain adopts the *PQ Noise* framework proposed by Angel, Dowling, Hülsing, Schwabe, and Weber [ADH+22]: a formal extension of the Noise Protocol Framework [Per18] that replaces Diffie–Hellman operations with KEM encapsulation. In PQ Noise, the Noise_KK or Noise_XX handshake pattern is re-cast as a KEM pattern where each DH call is replaced by an ML-KEM-768 encapsulation/decapsulation, while static peer authentication uses ML-DSA-65 signatures. This preserves the compositional security proofs of the Noise framework while substituting quantum-safe primitives throughout.

Shell-Chain targets the **Noise_XX_MLKEM768_MLDSA65** pattern:

- **Ephemeral key exchange**: ML-KEM-768 [Nat24a] encapsulation replaces X25519; provides IND-CCA2-secure forward secrecy (MLWE hardness assumption).
- **Static peer authentication**: ML-DSA-65 [Nat24b] signatures replace Ed25519; reuses the validator's consensus identity key—one key pair for both consensus attestation and P2P transport.
- **Session encryption**: ChaCha20-Poly1305 [NL18] with directional keys derived via `BLAKE3.derive_key` [OANWO20].

Peer identity.

$$PeerID = \text{BLAKE3.derive\_key}(\texttt{"shell-peer-id-v1"},\ ml\_dsa\_65\_pubkey)[0{:}32]$$

Peer IDs are 32-byte post-quantum identifiers sharing the same 32-byte width as PQAddresses (§5.4), but derived via `BLAKE3.derive_key` with a dedicated context string and without the `algo_id` prefix—ensuring peer IDs are distinct from on-chain addresses. **Deployment boundary:** Nodes use the standard libp2p `PeerId` derived from an Ed25519 keypair. The 32-byte BLAKE3-derived PeerID is the PQ Noise target.

Security properties.

1. **Post-quantum confidentiality.** ML-KEM-768 session key establishment is secure against a CRQC (MLWE hardness).
2. **Forward secrecy.** Per-session ephemeral ML-KEM-768 key pairs are discarded after the handshake; past sessions are secure even if long-term identity keys are later compromised.
3. **Post-quantum authentication.** ML-DSA-65 static key signatures prevent man-in-the-middle attacks by adversaries with classical or quantum resources.
4. **Symmetric quantum security.** ChaCha20-Poly1305 with 256-bit keys provides ≥128-bit quantum security (Grover: 2^{128} operations).
5. **Compositional security.** The PQ Noise framework inherits the Noise protocol's compositional security structure; security of the full handshake reduces to the IND-CCA2 security of ML-KEM-768 and the EUF-CMA security of ML-DSA-65 [ADH+22].

Handshake overhead. The Noise_XX_MLKEM768 handshake involves one ML-KEM-768 encapsulation (ek: 1,184 bytes; ct: 1,088 bytes) and two ML-DSA-65 authentication signatures (3,309 bytes each), giving a total handshake overhead of approximately **8,890 bytes**—a one-time cost per connection. For long-lived validator P2P sessions (typical duration: hours to days), this overhead is negligible when amortized over thousands of messages.

Table 13: Transport Protocol Comparison: Classical Noise vs. PQ Noise

Property	Classical Noise_XX	PQ Noise (target)
Key exchange	X25519 ECDH (quantum-broken)	ML-KEM-768 (quantum-safe)
Peer authentication	Ed25519 (quantum-broken)	ML-DSA-65 (quantum-safe)
Session encryption	ChaCha20-Poly1305	ChaCha20-Poly1305
Forward secrecy	Yes	Yes
Retroactive CRQC decryption	**Yes**	No
Handshake size	≈200 bytes	≈8.9 KB
Handshake latency (LAN)	<1 ms	≈3–10 ms
Formal security model	Noise framework [Per18]	PQ Noise framework [ADH+22]

The ≈8.9 KB handshake overhead is a one-time cost per connection, negligible for long-lived validator sessions.

Comparison: classical Noise_XX vs. PQ Noise.

Integration architecture. PQ Noise is integrated as a libp2p transport upgrade, replacing the classical `NoiseTransport`:

TCP → PQ Noise handshake → Yamux multiplexer → GossipSub / Kademlia / RPC

Existing libp2p protocols (GossipSub, Kademlia DHT, request-response) run unchanged above the PQ Noise layer.

End-to-end post-quantum stack. Once PQ Noise is deployed, Table 14 summarizes the targeted end-to-end post-quantum security stack across all protocol layers:

Table 14: Shell-Chain End-to-End Post-Quantum Security Stack

Layer	Protocol	PQ Algorithm	Quantum Security
Application (transactions)	Shell-Chain signed tx	ML-DSA-65 primary; Dilithium3 legacy-compatible; SLH-DSA-SHA2-256f fallback	≥128-bit target
Application (blocks)	PoA / optional wPoA + finality	ML-DSA-65 / Dilithium3 active allowlist	≥128-bit target
Transport (session)	libp2p Noise/XX today; PQ Noise target	ChaCha20-Poly1305 (256-bit)	128-bit symmetric target
Transport (key exchange)	libp2p Noise/XX today; PQ Noise target	X25519 today; ML-KEM-768 planned	current transport not yet end-to-end PQ
Transport (authentication)	libp2p Noise/XX today; PQ Noise target	Ed25519 today; ML-DSA-65 planned	current transport not yet end-to-end PQ
Storage (keys)	PQ keystore	Argon2id + XChaCha20-Poly1305	Memory-hard; limited quantum speedup

8.2 JSON-RPC Interface

Shell-Chain's `shell-rpc` layer maintains Ethereum-shaped JSON-RPC compatibility for read-oriented tooling, while signing, transaction construction, and 32-byte address handling require the Shell SDK, wallet, or plugin layer:

- **Read namespaces** (`eth_`, `web3_`, `net_`, `debug_`, `trace_`): Compatible at the JSON-RPC shape level with tools such as `cast` (Foundry) and block explorers. Standard Ethereum wallets require Shell-specific address/signing support.
- **Write methods** (`eth_signTransaction`, `eth_sign`): Return an error, since all signing must occur client-side with PQ key material.
- **`shell_` namespace**: PQ key operations, proof queries, and wPoA state inspection.
- **`evm_` namespace**: EVM execution and testing helpers exposed by the current RPC surface.

9 Node Roles and Deployment

Shell-Chain decouples three operational responsibilities—block production, attestation, and STARK proving—into distinct node roles. This separation allows operators to right-size their hardware investment and lets the network scale proving capacity independently of its validator set.

9.1 Role Definitions

The `NodeRole` enum in `shell-node` defines three roles:

`Validator` **(default):** Participates in wPoA block production (signs and proposes blocks when elected), validates imported blocks, broadcasts `Attestation` messages, and maintains BFT finality state. Does *not* run the STARK prover service. This is the minimum role required to be part of the active validator set.

`Prover`**:** Runs the STARK prover service exclusively. Does not hold a proposer keypair, does not produce blocks, and does not broadcast attestations. Its sole function is to drain the proof backlog and respond to `ProofChallenge` messages with `ChallengeResponse` proofs. Multiple Prover nodes can operate in parallel to scale proving throughput horizontally.

`ValidatorProver`**:** Everything a Validator does, plus the asynchronous STARK prover service. The prover pipeline runs continuously in the background, parallel to block production and attestation; it drains the proof backlog and responds to `ProofChallenge` messages regardless of whether the node is in a proposer slot or idle. Suitable for validators with spare CPU capacity who wish to contribute proof work.

9.2 Why Separate Roles?

Incompatible latency profiles. A Validator must respond to its proposer slot within the 2 s block interval: collect transactions, execute them through the PQVM execution layer, and broadcast the sealed block with minimal delay. STARK proving, by contrast, is a batch job that tolerates minutes of latency—proofs are not required for block acceptance, only for re-verification efficiency and challenge responses. Running both workloads on the same thread pool causes head-of-line blocking. The ValidatorProver role uses an asynchronous prover pipeline to isolate these workloads; the Prover role separates them entirely.

Hardware decentralisation. Generating a STARK proof for a near-full batch of 500 signatures requires substantially more computation than verifying them. Requiring every validator to also be a prover would raise the minimum hardware bar, reducing decentralisation. The Prover role lets dedicated machines handle proving while lightweight validators focus on consensus.

Horizontal scaling of proof throughput. STARK proof generation is embarrassingly parallelizable across non-overlapping block ranges. Multiple Prover nodes each claim a disjoint window of the proof backlog and generate proofs concurrently. The `shell-stark-prover` backlog scheduler coordinates assignment, ensuring each block range is proved exactly once.

Role decomposition alternatives. An alternative design assigns all roles to every node (monolithic), as in Ethereum pre-Merge. The monolithic design simplifies the protocol but forces every node to bear the full CPU cost of STARK proof generation—a burden that is *non-trivial* for the current accumulator circuit (see §11) and will grow with block size. The three-role decomposition (Validator / Prover / ValidatorProver) allows operators to match hardware to role: validators need fast CPUs for signature verification; provers need sustained CPU throughput for STARK generation; full-participation nodes (ValidatorProver) accept both burdens.

9.3 L2 STARK Mode

Orthogonal to the node role, the `L2StarkMode` setting governs how aggressively STARK infrastructure is engaged:

`Disabled` **(default):** No input index, no proof scheduler, no L2 settlements. Safe for all current testnet deployments.

`Scaffold`: Input index and job tracking are active; scheduling windows are computed and emitted as metrics; but recursive proving does *not* execute. Use this for operational visibility (gap detection, latency histograms) without committing to proof generation.

`Active`: Full recursive aggregation: input index, job store, scheduler, and the recursive prover all run. Requires the `recursive` Cargo feature.

This three-mode progression allows operators to adopt proving infrastructure incrementally: `Disabled` for initial deployment, `Scaffold` for observability, `Active` for full production proving.

9.4 Storage Profiles

PQ signatures impose a qualitatively different storage burden than ECDSA. Table 15 shows the per-transaction overhead.

Table 15: Per-transaction storage overhead: PQ vs. ECDSA

Field	ECDSA (Ethereum)	ML-DSA-65
Signature	64 bytes	3,309 bytes
Public key (first use)	—	1,952 bytes
First-use total	64 bytes	5,261 bytes
Returning-account total	64 bytes	3,309 bytes

The 52× signature size increase means a full 500-transaction block carries approximately ≈1.61 MB of raw witness data (500 × 3,309 bytes for returning accounts). Shell-Chain addresses this with two mechanisms.

StrippedBlock design. Each block body is structurally separated into *execution data* (transaction payloads, EVM results) and *witness data* (PQ signatures and public keys). This separation allows witness data to be pruned or replaced independently of the execution record.

Proof-replacement model. Once a STARK batch proof is recorded for a block, it certifies that the published accumulator root was computed from the declared witness-entry list. It does *not* by itself prove that all n signatures were valid. Signature validity is established by proposer-side PQ verification and can be challenged through ProofChallenge (§11.4). After the proof is accepted and the configured retention window has passed, raw ML-DSA-65 signatures may be deleted according to the node's storage profile. `proof_replacement_grace` sets that delay; the default, 0, makes witnesses eligible for replacement as soon as an accepted proof arrives.

Three named `StorageProfile` presets cover the operational spectrum (Table 16):

Table 16: Storage Profile Parameters (from `pruning.rs`)

Profile	`body_ret.`	`witness_ret.`	`proof grace`	`keep_recent`
`Archive`	∞ †	∞ †	never	∞ †
`Full`	∞ †	128 blk	0 (imm.)	—
`Pruned (Rolling)`	4 096 blk	64 blk	0 (imm.)	4 096 blk

† In code, retention value `0` means "retain all" (equivalent to ∞), not zero-byte retention; for `Full`, this applies to body retention specifically (keep-all).

`Archive`: All data—transaction bodies, PQ witness bundles, and STARK proofs—retained forever. `proof_replacement_grace = u64::MAX` prevents witness pruning even when a

STARK proof is present. Intended for block explorers, forensic auditors, and research nodes.

`Full` **(recommended for validators):** All transaction bodies retained indefinitely (body_retention = 0 means keep-all). PQ witness bundles are retained for `DEFAULT_WITNESS_RETENTION` = 128 blocks; when an accepted STARK proof arrives for a block, its witnesses become eligible for release after `proof_replacement_grace` blocks. The default grace value is 0, so release may occur immediately after proof acceptance, subject to local archival policy. Post-replacement, a STARK proof (≈43 KB at $n = 100$; ≈87 KB at $n \approx 500$) substitutes for up to 1.61 MB of raw signatures.

`Pruned (Rolling)` **(formerly** `Light`**):** Rolling window of the most recent 4 096 blocks (≈2.3 h at 2 s/block). Bodies are retained for 4 096 blocks and witnesses for 64 blocks; older data is pruned. Suitable for resource-constrained nodes that participate in consensus but do not need deep historical access.[2]

Profile–role interaction. Storage profile and node role are orthogonal but combine predictably: `Archive` nodes are typically read-only observers; validator nodes default to `Full`; edge participants use `Pruned (Rolling)`. When `L2StarkMode = Active`, proof availability enables proof-replacement pruning: witness bundles may be released once the corresponding STARK proof is accepted and the configured `proof_replacement_grace` has elapsed. This is independent of the special retention value `0`, which means "retain all" for retention windows rather than immediate deletion.

Part IV presents security and empirical evaluation (§§10 and 13), related work (§12), known limitations (§14), and conclusions (§15).

10 Security Analysis

The security properties achieved map directly to the three challenges of Section 3.2.

10.1 C1: Authentication Security

Theorem 1 (PQ Authentication)**.** *Under the Module-LWE / Module-SIS hardness assumptions (ML-DSA-65) or the PRF security, collision resistance, second-preimage resistance, and undetectability assumptions of SLH-DSA (FIPS 205), no QPT (quantum polynomial-time) adversary can forge a valid transaction signature or spoof a registered sender address in Shell-Chain.*

Proof sketch. All transaction signatures use ML-DSA-65 or SLH-DSA, both EUF-CMA in the QROM [KLS18]. EUF-CMA in the QROM implies unforgeability against a QPT adversary that queries hash functions in quantum superposition. Sender address derivation uses a 32-byte BLAKE3 output prefix. Targeted preimage and second-preimage attacks against a specified address require roughly 2^{128} quantum queries under Grover's algorithm; arbitrary address collisions are governed by the lower BHT bound of roughly 2^{85} quantum queries [BHT97]. The *algo_id* binding provides domain separation rather than a mathematical guarantee of disjoint address sets. Deserialization is bounded before any cryptographic operation: 4,096 bytes for ML-DSA-65 and 51,200 bytes for SLH-DSA (`MAX_ML_DSA_65_SIG_BYTES` and `MAX_SIGNATURE_BYTES` in `shell-crypto`), preventing allocation-overflow attacks. □

The retrospective key forgery threat is addressed: Shell-Chain contains no ECDSA private keys, so historical transactions archived under classical keys yield nothing useful to a future

[2] A true header-only light client (holding only block headers and state proofs) is planned for a future development milestone.

CRQC. Key storage uses Argon2id with 64 MiB state; its memory-hard structure primarily resists GPU/ASIC acceleration, and any quantum speedup is expected to be limited rather than eliminated.

STARK accountability assumption. The STARK accumulator provides *post-hoc* accountability under the assumption that the BFT proposer operates correctly during block finalization. A Byzantine proposer can finalize blocks with invalid batch roots before a STARK challenge resolves; the `ProofChallenge` mechanism provides deterrence and auditability, not prevention. This trust assumption is appropriate for a permissioned wPoA network where proposers are identity-verified and economically accountable.

10.2 C2: Consensus Security

Theorem 2 (wPoA BFT Safety)**.** *If Byzantine validators control total weight strictly less than $W/3$, the Shell-Chain attestation protocol never finalizes two conflicting blocks at the same height under the $> 2W/3$ weighted quorum rule.*

Proof. Suppose, for contradiction, that two conflicting blocks B and B' at the same height both collect weighted quorum certificates. Since attestations carry PQ signatures, no forged attestation is accepted (Theorem 1). Each quorum certificate has total weight greater than $2W/3$, so the two attestation sets overlap in weight greater than $W/3$. Any validator in the overlap submitted valid attestations for *both* B and B' at the same height. Since honest validators attest at most once per height, the entire overlap would need to be Byzantine, contradicting the assumption that Byzantine weight is strictly less than $W/3$. □ □

Proposition 2 (Prefix Safety)**.** *If block B_k is finalized at height k, then for all heights $h < k$, block B_h is also finalized and immutable.*

Proof sketch. By induction: finalization of B_k requires finalization of B_{k-1} (its parent must have been finalized, or validators would not attest). By the BFT safety guarantee, a finalized block cannot be reversed. □

Theorem 3 (wPoA Liveness)**.** *If faulty validators control total weight strictly less than $W/3$, the network is eventually synchronous, all honest validators apply the same timeout/view-change rule, and the active view eventually selects an honest proposer, then a valid block for that height will be finalized.*

Proof sketch. Before an honest proposer is selected, timed-out views can be advanced only by signed `ViewChangeMessage`s whose total weight exceeds $2W/3$; with faulty weight below $W/3$, honest validators control enough weight to form that quorum post-GST. Once an honest proposer is active, the sealed block reaches all honest validators within a finite round-trip time. Each honest validator independently verifies the PQ proposer seal and, if the block is valid, broadcasts a signed `Attestation`. Since faulty weight is strictly less than $W/3$, honest validators control weight greater than $2W/3$. Therefore a weighted quorum of valid attestations will eventually be collected by the `FinalityState` tracker, triggering finalization post-GST. □ □

Because all validator messages carry PQ signatures, a CRQC cannot forge an attestation from an honest validator, manufacture an equivocation proof, or impersonate a registered proposer. wPoA is permissioned: Sybil resistance is provided by the genesis authority list and on-chain weight governance rather than proof-of-work.

10.3 Additional Classical Security Properties

- **Signature unforgeability**: EUF-CMA under Module-LWE/SIS (ML-DSA-65) and the PRF-security, collision-resistance, second-preimage-resistance, and undetectability assumptions of SLH-DSA [DKL+18, BHK+19].
- **Denial-of-service**: Per-sender rate limiting and fee-floor admission in the mempool; GossipSub peer scoring penalizes invalid-message senders.
- **Eclipse attacks**: Kademlia DHT routing table diversification; connection limits bound malicious peer flooding.
- **ecrecover disabled**: Applications depending on classical ECDSA recovery fail loudly at integration time rather than silently at security-critical moments.

11 Performance Viability

PQ cryptography carries a real performance cost relative to ECDSA. The central question for a PQ-native blockchain design is therefore: *are the chosen algorithms fast enough to meet production performance targets?* This section answers that question by stating the targets, presenting reference measurements, and showing that the three architectural design choices close the gap.

Design performance targets.

- **Throughput:** ≥ 500 transactions per block at a 2 s block interval.
- **Validator import latency:** block import (including all signature verification) must complete well within the block interval on commodity validator hardware.
- **Re-verification cost:** as the validator set scales, the per-validator re-verification burden must not become the throughput bottleneck.
- **Storage:** the default (Full) profile must be sustainable on commodity NVMe storage without unbounded growth.

11.1 Reference Measurements

The following measurements were obtained on Apple M2 Pro (12-core, 16 GB) with Rust 1.87 (`--release`), and serve as a *reference implementation baseline*. Production hardware performance will vary; these numbers establish that the design targets above are achievable on commodity developer hardware.

11.2 Baseline Overhead

Table 17 reports the reference microbenchmark results used throughout the gas and throughput discussion.

Reference measurements confirm the design targets are met: at a 2 s block time with a 500-transaction block cap, serial signature verification takes $500 \times 0.046\,\text{ms} = 23\,\text{ms}$, well within the block interval. However, in a network of V validators each independently re-verifying all n signatures per block, the *network-wide* cost is $V \times n$ ML-DSA-65 verifications—a quantity that grows with both network size and block fullness. For $V = 100$ validators and $n = 500$ transactions, the network performs 50,000 verifications per block, totalling ≈2.3 s of *aggregate CPU-seconds* (i.e., wall-clock time on a single core; with parallel batch verification, each validator completes in roughly ≈5.8 ms). As the validator set scales, this motivates the three design choices below.

Table 17: Signature Operation Latency (Apple M2 Pro (12-core))

Operation	ECDSA (secp256k1)	ML-DSA-65	SLH-DSA-SHA2-256f
Key generation	< 0.1 ms	0.068 ms	< 1 ms
Sign	$\approx$0.08 ms	0.127 ms	84 ms
Verify	$\approx$0.04 ms	0.046 ms	2.3 ms
Sign + Verify	$\approx$0.12 ms	0.173 ms	86 ms
100× Verify (serial)	$\approx$4 ms	4.6 ms	230 ms
100× Verify (parallel, 12-core)	$\approx$0.5 ms	$\approx$1.1 ms	$\approx$30 ms
PQ-secure	No	Yes	Yes

11.3 Design Choice 1: Parallel Batch Verification

The batch verification design parallelizes verification across all available cores via Rayon's work-stealing thread pool. Reference measurements confirm that 100 ML-DSA-65 verifications complete in $\approx$1.1 ms vs. 4.6 ms serially on 12-core hardware—a $\approx$4× speedup that keeps block import latency well within the 2 s block interval.

11.4 Design Choice 2: Batch-Commitment STARK Pipeline

Why STARK and not zk-SNARK or Bulletproofs? The design requires a proof system that: (i) requires no trusted setup, (ii) is post-quantum safe (security from hash collision resistance only), and (iii) has fast verification. Table 18 compares the main alternatives.

STARK, not zk-STARK. Shell-Chain uses a plain *STARK* (Scalable Transparent ARgument of Knowledge), not a *zk-STARK*. The zero-knowledge property hides the witness from the verifier; it is required only when the witness contains private data. Here the witness consists of `(msg_hash, pk_hash)` pairs that are already permanently and publicly recorded on-chain. Hiding them from validators would be meaningless. Omitting the zero-knowledge layer avoids the overhead of blinding polynomials and simplifies the security argument: the proof's soundness rests solely on hash collision resistance, with no additional hiding assumption required.

Table 18: Proof system trade-offs for batch verification

System	Trusted setup	PQ-safe	Verify time	Proof size
STARK (Winterfell)	None	Yes (hash)	$\approx$100–300 μs	$\approx$40–90 KB
Groth16 (zk-SNARK)	Required	No (pairing)	$\approx$1 ms	$\approx$200 B
PLONK	Universal SRS	No (pairing)	$\approx$2 ms	$\approx$1 KB
Bulletproofs	None	No (DLOG)	$O(\log n)$	$\approx$1 KB

STARK proofs are among the strongest available transparent (no trusted setup) proof systems satisfying all three requirements simultaneously [BSBHR19]. The proof size overhead (40–90 KB vs. < 1 KB for SNARKs) is acceptable because proofs are generated once per block and verified by each validator in $\approx$100–300 μs — far faster than re-verifying the underlying PQ signatures.

Motivation. The batch verification design (§11.3) reduces per-node import latency but leaves the network-wide cost of $V \times n$ verifications unchanged. A STARK proof collapses this to V constant-time checks: the proof is short and fast to verify, and its soundness rests on hash collision resistance—a post-quantum-safe assumption with no trusted setup.

Why not full in-circuit ML-DSA-65? The ideal solution would be to encode the full ML-DSA-65 verification algorithm into a STARK circuit. Doing so requires expressing NTT operations over the ring $\mathbb{Z}_q[x]/(x^{256}+1)$ as arithmetic constraints, which generates ≈50,000–100,000 trace rows per signature. At current STARK prover throughput this approach would require several seconds of proving time per transaction—far longer than the 2 s block interval. Full in-circuit verification is a mid-term research goal (Section 14); it is not viable today.

The batch-commitment design. Instead, Shell-Chain uses a *batch-commitment STARK*: the proposer performs native ML-DSA-65 verification (fast, outside the circuit), then commits to the set of verified pairs via a lightweight STARK-proven accumulator. **Important trust assumption**: the STARK proof demonstrates that the accumulator trace over the committed $(\mathsf{msg\_hash}_i, \mathsf{pk\_hash}_i)$ pairs is arithmetically consistent—it does *not* re-execute the underlying ML-DSA-65 verifications. Validators therefore trust the proposer's claimed verifications when importing a block; the proof challenge protocol (below) provides a post-hoc accountability mechanism but does not eliminate this trust assumption at block-import time. The design is appropriate for a permissioned wPoA network where proposers are identity-verified and economically accountable.

Concretely:

1. The proposer verifies all n signatures natively (at ≈46 μs each via `shell-crypto`).
2. For each verified pair $(\mathsf{msg\_hash}_i, \mathsf{pk\_hash}_i)$, a 256-bit entry is derived:

$$e_i = \text{BLAKE3}(\mathsf{msg\_hash}_i \| \mathsf{pk\_hash}_i)$$

   Each entry has full 256-bit collision resistance.
3. The STARK circuit commits to an ordered Merkle tree of entries $(e_1, \ldots, e_n)$, where the root R is the block's signature accumulator. The proposer publishes R and the sorted list of entries. The STARK proof certifies that R is correctly computed from the ordered list.
4. A STARK proof over the Merkle commitment trace produces a `batch_root`, stored in the block header's `sig_aggregate_proof` field and computed from the Merkle root R.
5. Each additional validator re-verifies the short STARK proof (≈52 μs for $n = 1$; ≈140 μs for $n = 100$; ≈292 μs for $n \approx 500$, measured on Apple M2 Pro) rather than re-executing all n ML-DSA-65 verifications (proof sizes above reflect JSON-encoded proofs for tooling compatibility; binary-encoded STARK proofs are approximately 40% smaller).

Proof challenge protocol. If a validator doubts the proposer's claimed verifications—for example, if it suspects the proposer included an invalid signature without checking—it can broadcast a `ProofChallenge` message over GossipSub. Any node running the prover service responds with a `ChallengeResponse` carrying the full STARK proof for the challenged block. This mechanism provides accountability without requiring every node to do full re-verification by default: the common case is cheap, and the adversarial case is auditable.

Proposer Trust Assumption. The STARK proof certifies that the published accumulator root R is correctly derived from the declared entry list $(e_1, \ldots, e_n)$. It does *not* prove that each entry corresponds to a valid post-quantum signature. Signature validity is enforced by the full per-signature PQVM precompile verification that the block proposer runs before constructing the accumulator. This is a *trust-but-verify with accountability* design: any validator may issue a `ProofChallenge` that demands the proposer supply the full witness list and corresponding signatures within $T_{\text{challenge}}$ blocks. A proposer who cannot respond is slashed.

A `ProofChallenge` follows the state machine: `OPEN` → (valid proof submitted within T_c = 7,200 blocks) → `RESOLVED` / (timeout) → `SLASHED`. If no valid proof is submitted before T_c, the challenged validator is penalized according to the slashing rules.

Accumulator soundness. The Merkle-tree accumulator inherits its binding property from BLAKE3 collision resistance: each entry $e_i = \text{BLAKE3}(\mathsf{msg\_hash}_i \| \mathsf{pk\_hash}_i)$ is a 256-bit value, and the root R is a standard binary Merkle root over the ordered entry list. Finding two distinct entry lists that produce the same root R requires finding a BLAKE3 collision, which requires 2^{85} quantum queries (BHT algorithm [BHT97]); finding a second preimage for a specific committed root R requires 2^{128} quantum queries (Grover's algorithm). Both bounds substantially exceed the adversary's computational budget. The overall STARK soundness (i.e., that the declared entries match the committed root) targets approximately 100-bit conjectured soundness with the chosen FRI parameters (see §14). Accordingly, the STARK layer should be read as a practical amortisation and accountability mechanism, not as a substitute for a full proof of transaction-signature validity.

Cost model. This design has two distinct impacts when `L2StarkMode = Active`: (1) *Re-verification speedup*: for $n = 100$ transactions, re-verification cost falls from $100 \times 46\,\mu\text{s} = 4.6\,\text{ms}$ to $\approx 140\,\mu\text{s}$—a reduction of $\approx 33\times$ per validator per block. For a full 500-transaction block, the speedup is approximately $\approx 79\times$ ($500 \times 46\,\mu\text{s} \approx 23.0\,\text{ms}$ vs. $\approx 292\,\mu\text{s}$ STARK verify). (2) *Witness pruning*: once a `ProofAmendment` settles, Full-mode nodes replace the $\approx 1.61\,\text{MB}$ raw witness bundle (500 ML-DSA-65 signatures) with the $\approx 87\,\text{KB}$ STARK proof payload ($\approx 19\times$ smaller for a 500-transaction block; $\approx 43\,\text{KB}$ for $n = 100$); the full raw signatures are retained only by Archive-mode nodes. During the window before proof settlement, Full-mode nodes retain the raw witness bundle (within the 128-block grace period) and it is therefore not yet pruned.

`ProofAmendment` records are gossiped and stored in the state trie. Witness pruning is safe only after the amendment has been finalized for at least `MIN_AMENDMENT_DEPTH` = 128 blocks, preventing reorg-based double-spending of the amendment.

11.5 Design Choice 3: Witness Pruning

The storage overhead of PQ signatures and the proof-replacement model that mitigates it are described in Section 9.4. The `StrippedBlock` design separates witness data from execution data; once a STARK batch proof settles, the $\approx 1.61\,\text{MB}$ witness bundle for a 500-transaction block can be replaced by a `ProofAmendment` payload of $\approx 87\,\text{KB}$—roughly a $19\times$ reduction in long-term witness storage cost per block. The `sig_aggregate_proof` field in the block header stores a compact commitment to the proof; the full `ProofAmendment` is propagated over GossipSub and stored separately on proof-capable nodes.

12 Related Work

Post-quantum blockchain proposals. Fernández-Caramés and Fraga-Lamas [FCFL20] survey PQ blockchain approaches. QRL [QRL18] deploys XMSS [HBG+18] as a native PQ L1 but without an EVM-familiar execution environment. NIST PQC standardization has spurred proposals to retrofit Ethereum with PQ signatures via account abstraction or hard fork [But22], all subject to the coordination challenges described in Section 2. Most PQ-overlay chains retain 20-byte Ethereum-style addresses for compatibility; Shell-Chain instead uses a clean-slate 32-byte `0x`-encoded address space from genesis. Hybrid classical/PQ signature schemes [BBF+19] offer a migration path but compound complexity and typically become permanent. Existing PQ blockchain proposals also include QRL (XMSS signatures, hash-based, no smart contracts) [QRL18], Ethereum PQ migration proposals such as EIP-7212 / EIP-7702-style overlays,

and PBFT-style post-quantum adaptations [CL99]. Shell-Chain differentiates itself by being protocol-native PQ from genesis (no migration path required), by using a clean-slate 32-byte `0x`-encoded address space from genesis, by targeting PQVM rather than patching the EVM, and by embedding algorithm-agility governance directly into the protocol.

PQ-native systems. Prior PQ-native designs include the Open Quantum Safe (OQS) project [SM16], which provides reference implementations of NIST candidates; the NIST PQC standardisation process [Nat24e] which specified PQ-native API requirements; and recent IETF drafts for PQ-safe TLS [SFB23] and SSH. Shell-Chain applies these principles at the blockchain protocol layer.

BFT consensus. PBFT [CL99] established practical Byzantine fault tolerance. Casper FFG [BHK+20] adapts BFT finality for Ethereum's Nakamoto-style chain. The partial synchrony model of Dwork, Lynch, and Stockmeyer [DLS88] grounds our liveness analysis, while FLP impossibility [FLP85] justifies the honest-proposer and partial-synchrony assumptions required for liveness.

STARK systems. Ben-Sasson et al. [BSBHR19] provide the foundational STARK construction. Shell-Chain uses the Winterfell library [Pol21] for its FRI-based batch-commitment accumulator.

Algorithm Agility Standards. Cryptographic algorithm agility has been formalised by the IETF in RFC 7696 [Hou15], which provides guidelines for designing systems that support multiple cryptographic algorithms without requiring protocol redesign. Shell-Chain's Algorithm Agility mechanism (§5.8) applies these principles at the blockchain governance layer.

PQ key infrastructure. The NIST PQC standards ML-DSA (FIPS 204) [Nat24b], SLH-DSA (FIPS 205) [Nat24c], and ML-KEM (FIPS 203) [Nat24a] form the cryptographic foundation. Argon2id [BDK16] provides memory-hard key derivation for key storage.

13 Evaluation

13.1 Quantum Security

Table 19 compares Shell-Chain's post-quantum security posture with representative deployed systems.

13.2 Execution Environment Comparison

Table 20 summarizes the execution-environment trade-offs between Shell-Chain and representative EVM-derived systems.

13.3 Trade-off Analysis

Shell-Chain is the only system in the comparison that deploys post-quantum signatures from genesis together with a clean-slate PQVM execution model. Shell-Chain uses a clean-slate PQVM execution model with 32-byte `0x`-encoded addresses from genesis. The immediate cost is a permanent increase in per-transaction signature size: 3,309 bytes (Dilithium3 / ML-DSA-65 class) vs. 64 bytes (ECDSA), a 52× factor. We argue this trade-off is justified for three reasons:

Table 19: Post-Quantum Security Comparison

Blockchain	Signature	PQ Security	Migration Required
Bitcoin	ECDSA (secp256k1)	0-bit†	Yes
Ethereum	ECDSA (secp256k1)	0-bit†	Yes
Solana	Ed25519	0-bit†	Yes
QRL	XMSS (RFC 8391)	128-bit (hash-based)	No (native PQ L1)
Shell-Chain	**ML-DSA-65 primary; Dilithium3 legacy-compatible; SLH-DSA-SHA2-256f fallback**	**≥128-bit PQ target**‡	**No**

†"0-bit" PQ security means a CRQC running Shor's algorithm breaks the scheme in polynomial time; no quantum hardness bound remains.
‡Dilithium3 and ML-DSA-65 target the Level-3 security class, while SLH-DSA-SHA2-256f targets Level 5; all are used in the paper as part of Shell-Chain's ≥128-bit post-quantum target.

Table 20: Execution Environment Comparison

Platform	Execution model	Tooling	Migration Effort
Ethereum	Native	Full	None
Polygon	EVM byte-compatible	High	Low
Binance Smart Chain	EVM byte-compatible	High	Low
Arbitrum	EVM-derived	High	Medium
Optimism	EVM-derived	High	Medium
Shell-Chain	**PQVM clean-slate execution**	**High (EVM-familiar opcodes)**	**Contract recompile for 32-byte addresses**

1. **Cost is bounded; benefit is durable.** The overhead is a fixed engineering cost known at deployment time. The security benefit—security under the conjectured quantum hardness of MLWE, MLSIS, and SLH-DSA assumptions—provides durable protection against retrospective attacks once the full stack is deployed.
2. **Mitigations are effective.** The STARK pipeline (when `L2StarkMode = Active`) reduces re-verification cost by ≈33× for 100-transaction blocks and ≈79× for 500-transaction blocks; witness pruning achieves a ≈19× reduction in long-term witness storage once proof settlement occurs; parallel batch verification recovers import latency.
3. **The alternative is a coordination problem, not an engineering one.** A hard-fork migration requires network-wide agreement under adversarial conditions. Shell-Chain shifts the cost from an uncertain future coordination event to a predictable present engineering cost.

SLH-DSA block throughput. The SLH-DSA-SHA2-256f signature size of ≈49 KB constrains single-transaction block capacity to approximately 20 SLH-DSA transactions per megabyte. In practice, SLH-DSA is reserved for high-assurance use cases; the ML-DSA-65/Dilithium3 3.3 KB signature profile enables approximately 300 transactions per megabyte.

14 Limitations and Open Problems

The following limitations are acknowledged:

- **Liveness under Byzantine proposer**: View-change is implemented, but liveness remains conditional on post-GST message delivery, timeout agreement among honest validators, and eventual rotation to an honest active proposer (Section 7).
- **SLH-DSA block throughput**: Large SLH-DSA-SHA2-256f signatures (≈49 KB) limit throughput for SLH-DSA-dominant workloads (≈20 tx/MB).
- **Permissioned validator set**: wPoA requires a trusted validator authority for set management; permissionless validator admission is not currently supported.
- **STARK prover overhead**: STARK proof generation adds latency to finalization. Proof generation time is not yet optimized for real-time finalization at high TPS.
- **STARK accountability gap**: The current batch-commitment STARK proves accumulator consistency, not full in-circuit transaction-signature validity; invalid proposer claims are addressed through post-hoc challenge and governance, not by the proof alone at block-import time.
- **PQ Noise deployment**: PQ Noise support in libp2p is under active development. Shell-Chain currently uses classical Noise/X25519; migration to PQ Noise will occur as the libp2p ecosystem matures. The end-to-end PQ security stack (Table 14) is the target once PQ Noise is deployed.
- **Algorithm agility rollout**: Governance can activate only algorithms already implemented and audited in released clients; introducing an entirely new primitive still requires a coordinated software rollout.
- **PQVM opcode exposure**: The PQVM execution engine is implemented with a 32-byte address model, a six-precompile PQ suite (`0x0001`–`0x0006`), and a native `PQTx` format. The PQ-native opcodes `PQVERIFY` (0xB0), `PQHASH` (0xB1), and `PQADDR` (0xB2) are defined, gas-priced, and wired into the interpreter dispatch table.

- **View-change implemented**: wPoA includes view-change. When the expected proposer misses its slot, validators broadcast signed `ViewChangeMessage`s; once weighted quorum $\lceil 2W/3 \rceil$ is reached, proposer rotation proceeds. WRR schedule predictability remains a known limitation (see below).
- **Key reuse**: Validator keys are used for consensus attestation and algorithm-governance votes; in the target PQ Noise transport they also act as peer-authentication keys (§8.1). While operationally simple, this key reuse means a compromised validator key has multi-context impact. Dedicated governance and transport keys are planned future improvements.
- **Governance centralisation**: Algorithm-agility governance (§5.8) requires a $\lceil 2N/3 \rceil$ validator quorum for registry changes. In a small validator set, that concentrates governance power; the main mitigation is to grow the validator set over time.
- **PQ Noise formal analysis**: The PQ Noise framework [ADH+22] provides a compositional security proof, but Shell-Chain's specific Noise_XX_MLKEM768_MLDSA65 pattern has not been independently verified. The migration from classical Noise is also pending libp2p upstream ML-KEM-768 support.
- **WRR predictability**: Deterministic WRR lets adversaries know future proposers and pre-position attacks. Per-epoch VRF-based shuffling is the known mitigation, but is not yet implemented.

STARK Soundness Gap

With the current parameters (28 queries, blowup 8, 16 bits of grinding), the STARK batch commitment provides roughly 100 bits of soundness, below the system's 128-bit security target. Raising the query count to 40 and grinding to 20 bits would give roughly 140 bits under the same per-query approximation used above (exact FRI soundness may differ), at the cost of larger proofs. The current setting is a deliberate compactness trade-off; the parameters are governance-upgradable (§5.8), and the gap is treated as a known limitation.

STARK Trust Assumption

The proposer-run STARK batch commitment proves accumulator consistency, not signature validity. Challenges reduce the trust placed in proposer pre-verification, but do not remove it. A colluding supermajority could still publish a forged accumulator without being challenged.

15 Conclusion

Quantum-resistant security cannot be retrofitted onto a deployed blockchain without residual risk—and Shell-Chain demonstrates that building it in from genesis is both technically viable and practically deployable. The case rests on three observations:

1. **The threat is present.** Retrospective key forgery via Shor's algorithm means that on-chain ECDSA public keys can, once quantum resources mature, be used to recover private keys and forge future transactions or authorizations from historically exposed public keys.
2. **Retrofitting fails.** Hard forks, L2 overlays, and hybrid schemes each leave a residual classical attack surface or impose coordination costs that realistic networks cannot reliably meet.

3. **Quantum-safe genesis closes the window.** By integrating post-quantum signatures from genesis—ML-DSA-65 as the primary FIPS 204 path, Dilithium3 as a legacy-compatible active path, and SLH-DSA-SHA2-256f as the conservative hash-based fallback— with no ECDSA fallback for on-chain authentication, Shell-Chain materially reduces retrospective quantum exposure.

Shell-Chain already carries post-quantum security through execution, authentication, and consensus. Full end-to-end coverage still depends on PQ Noise transport and release-hardening the PQ-native opcode surface.

The result remains familiar to Ethereum developers where that familiarity is useful: arithmetic, memory, storage, and control-flow semantics stay EVM-like, while addresses are native 32-byte `0x`-prefixed values from genesis. The cost—52× larger signatures and higher per-verification latency—is managed with parallel batch verification (about 4× speedup on 12-core hardware), a batch-commitment STARK pipeline that cuts re-verification by 33–79× depending on block fullness when `L2StarkMode = Active`, and witness pruning for long-term storage.

The wPoA consensus with BFT attestation finality provides deterministic finality once a single-block QC is assembled, under the weighted Byzantine bound ($< W/3$ faulty weight) and the liveness assumptions stated in §7. Compared with Nakamoto-style probabilistic finality, this gives a stronger finality model, and every consensus message is authenticated by PQ signatures that a CRQC cannot forge.

Shell-Chain's open-source Rust implementation, the `shell-crypto` crate, and the wPoA consensus engine are available at https://github.com/shelldao/shell-chain for independent review and reuse by the broader post-quantum blockchain research community.

References

[ADH+22] Yawning Angel, Benjamin Dowling, Andreas H"ulsing, Peter Schwabe, and Florian Weber. Post-Quantum Noise. IACR ePrint 2022/539, 2022.

[B+23] Vitalik Buterin et al. ERC-4337: Account abstraction using alt mempool. Ethereum Improvement Proposal, 2023.

[BBF+19] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. Hybrid signatures. In *Security Standardisation Research (SSR 2019)*, volume 11036 of *Lecture Notes in Computer Science*, pages 121–141, 2019.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, 2016.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008, 2008.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. SPHINCS+: Stateless hash-based signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2129–2146, 2019.

[BHK+20] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Kanabar, Barnabe Kusmierz, Danny Ryan, Justin Sanders, Hsiao-Wei Wang, and Yan Wang. Combining GHOST and casper. arXiv:2003.03052, 2020.

[BHT97] Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. In *Proceedings of LATIN 1998*, 1997. Extended version in SIGACT News 1997.

[BSBHR19] Eli Ben-Sasson, Iddo Bentov, Yonatan Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. In *Advances in Cryptology – EUROCRYPT 2019*, volume 11476 of *Lecture Notes in Computer Science*, pages 415–444, 2019.

[But22] Vitalik Buterin. How to hard-fork to save most users' funds in a quantum emergency. Ethereum Research Forum, 2022.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, 1999.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[FCFL20] Tiago M. Fernández-Caramés and Paula Fraga-Lamas. Towards post-quantum blockchain: A comprehensive survey. *IEEE Access*, 8:21091–21116, 2020.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[H+22] Andreas H"ulsing et al. SPHINCS+: Submission to the NIST post-quantum cryptography standardization project. https://sphincs.org/data/sphincs+-r3.1-specification.pdf, 2022.

[HBG+18] Andreas H"ulsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: extended merkle signature scheme. RFC 8391, IETF, 2018.

[Hc19] Brook Heisler and contributors. Criterion.rs: Statistics-driven microbenchmarking in Rust. GitHub repository, 2019.

[Hou15] R. Housley. Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms. Best Current Practice 7696, IETF, 2015.

[KLS18] Eike Kiltz, Vadim Lyubashevsky, and Christian Schaffner. A modular analysis of the Fujisaki-Okamoto transformation. In *Advances in Cryptology – EUROCRYPT 2018*, volume 10822 of *Lecture Notes in Computer Science*, pages 341–371, 2018.

[Mos18] Michele Mosca. Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy*, 16(5):38–41, 2018.

[Nat24a] National Institute of Standards and Technology. FIPS 203: Module-lattice-based key-encapsulation mechanism standard (ML-KEM). Technical Report FIPS 203, National Institute of Standards and Technology, 2024.

[Nat24b] National Institute of Standards and Technology. FIPS 204: Module-lattice-based digital signature standard (ML-DSA). Technical Report FIPS 204, National Institute of Standards and Technology, 2024.

[Nat24c] National Institute of Standards and Technology. FIPS 205: Stateless hash-based digital signature standard (SLH-DSA). Technical Report FIPS 205, National Institute of Standards and Technology, 2024.

[Nat24d] National Institute of Standards and Technology. FIPS 206: Fn-dsa digital signature standard. Technical Report FIPS 206, National Institute of Standards and Technology, 2024.

[Nat24e] National Institute of Standards and Technology. Post-quantum cryptography standardization. https://csrc.nist.gov/projects/post-quantum-cryptography, 2024.

[NL18] Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, IETF, 2018.

[OANWO20] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. BLAKE3 one function, fast everywhere. https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf, 2020.

[P+13] Marek Palatinus et al. BIP-39: Mnemonic code for generating deterministic keys. Bitcoin Improvement Proposal, 2013.

[Per18] Trevor Perrin. The noise protocol framework. https://noiseprotocol.org/noise.html, 2018. Revision 34.

[Pol21] Polygon Miden. Winterfell: A stark-based proof system. GitHub repository, 2021.

[Pro20] Protocol Labs. `libp2p`: A modular peer-to-peer networking stack. Technical specification, 2020.

[QRL18] QRL Foundation. The quantum resistant ledger (qrl) technical overview. Technical white paper, 2018.

[SFB23] D. Stebila, S. Fluhrer, and S. Bhatt. Hybrid key exchange in TLS 1.3. IETF Internet-Draft, 2023.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[SM16] Douglas Stebila and Michele Mosca. Post-quantum key exchange for the Internet and the Open Quantum Safe project. IACR ePrint 2016/1017, 2016.

[TT21] M. Thomson and S. Turner. Using TLS to secure QUIC. RFC 9001, IETF, 2021.

[Wui12] Pieter Wuille. BIP-32: Hierarchical deterministic wallets. Bitcoin Improvement Proposal, 2012.