Full L1 On-Chain ZK-STARK+PQC Verification on Solana: A Measurement Study

Jotaro Yano
Independent Researcher, Japan
jotaro.yano@jotaro-yano.org

September 23, 2025

Abstract

Blockchains preserve public data indefinitely, creating tension between verifiability today and secrecy decades hence. In particular, pairing-based SNARKs (e.g., Groth16, PLONK) rely on discrete-log assumptions that are structurally vulnerable to Shor-type quantum attacks, motivating hash-based alternatives. This work investigates whether a *fully on-chain* pipeline that verifies both a ZK-STARK and a post-quantum signature can operate within Solana L1's compute and memory constraints. Our prototype adapts Winterfell 0.12 with a dedicated SHA-256 `hashv` syscall path to reduce hashing overhead, suppresses inlining in FRI hotspots to respect SBF (Solana BPF) stack limits, and uses a custom bump allocator synchronized with requested heap frames. Artifacts are uploaded in ≤ 900 byte chunks under a rolling hash chain and finalized in two steps: (i) SLH-DSA (SPHINCS+) signature verification over a length-delimited transcript, and (ii) STARK verification bound to SHA256(`cipher`) via public inputs. The result is an L1 verifier that is CPI-friendly, reproducible from a public repository, and engineered for predictable cost. On devnet, across $n = 100$ runs, we measure mean *finalize_sig* cost of 5.01×10^5 CU (median 4.999×10^5) and mean *verify_stark* cost of 1.10×10^6 CU (median 1.11×10^6), with maxima below 1.19×10^6 CU; all runs fit within Solana's 1.4×10^6 CU transaction budget. At representative sizes, the derived intensities are ≈ 63.8 CU/Sig-byte (7,856 B) and ≈ 248.9 CU/Proof-byte (4,437 B), and verification scales approximately linearly with proof bytes under a fixed FRI policy. We systematize DoS and fee controls (fixed-offset appends, rolling-hash checks), justify the binding of public inputs to the ciphertext, and outline engineering levers (single-call hashing, stack discipline, phase separation) that make full L1 STARK+PQC verification practical at $\approx$ 128-bit settings.

Keywords: Solana, Zero-Knowledge proof, STARK, Post-Quantum Cryptography, SLH-DSA (SPHINCS+), ML-KEM (Kyber), On-Chain Verification

1 Introduction & Motivation

Blockchains are tamper-evident, append-only ledgers whose contents remain public indefinitely. This permanence creates a tension between *verifiability now* and *secrecy later*: artifacts published today—proofs, encrypted payloads, signatures—may be harvested, indexed, and revisited by future adversaries endowed with stronger algorithms or hardware. In particular, pairing-based SNARKs such as Groth16 and PLONK [6, 7] rely on assumptions (discrete logarithms in pairing-friendly groups) that are vulnerable in a post-quantum setting to Shor-type algorithms [1, 2]. Even before fully practical quantum computers, the recording-adversary model—record now, decrypt or break later (RNDL)—pressures systems that bet on long-term classical hardness.

STARKs offer a pragmatic counterpoint. They are transparent (no trusted setup), post-quantum oriented (hash-based security), and come with robust soundness amplification techniques (FRI and refinements) [3–5]. The principal cost is volume: proofs and verification work are typically larger than succinct pairing-based SNARKs. Whether a *full L1* on-chain STARK verification pipeline is viable on a production L1 depends not only on cryptographic parameters but also on the target chain's execution and memory model.

At a high level, Figure 1 summarizes the trade-off that motivates our measurement: *SNARKs are lightweight today but face long-term PQ risk*, whereas *STARK+PQC is PQ-friendly but heavier and less mature*; this study measures the feasibility envelope for the latter on Solana L1.

	STARK + PQC (this work)	**Pairing-based SNARKs**
Long-term security & setup	Hash-based, post-quantum oriented; transparent (no trusted setup).	Classical discrete-log assumptions (at risk under Shor); often require SRS/trusted setup.
Near-term cost on Solana SVM	Heavier today; verifier fits in $\sim$1.1M CU at $\approx$128-bit settings (measured); larger byte budgets.	Lightweight where pairing precompiles exist; software-only pairings/MSMs are costly on SVM.
Proof size & scaling	Larger proofs; verifier CU $\approx$ linear in proof bytes (fixed FRI).	Succinct proofs; cost tied to pairings/MSMs count and implementation details.
Deployment today	PQ-friendly direction; integration maturity is early—this work evaluates feasibility.	Widely deployed and optimized; strong tooling, but long-term PQ risk remains.

Figure 1: Motivation trade-off: SNARKs are lighter today, while STARK+PQC is PQ-friendly but heavier. This work measures whether full L1 verification is feasible on Solana.

We select Solana for this measurement because of its SVM/BPF execution, explicit compute accounting in *compute units* (CU), bounded per-transaction budgets, and explicit heap frames requested via the Compute Budget Program [11, 13, 14]. These features force engineering choices that are easy to measure and reason about:

- **CU budget and predictability.** A transaction either fits the CU envelope or fails deterministically; there is little ambiguity about whose cost is being paid and where.
- **Heap versus stack on SBF.** A relatively tight stack plus BPF calling conventions motivate inlining suppression and heap-aware designs; the program can mirror the requested heap size to

avoid allocator surprises.

- **I/O granularity.** Bounded account sizes and transaction limits push toward chunked uploads and on-chain streaming with constant-work rejection for malformed input [15].

Against this background, we implement and evaluate a pipeline that verifies *both* a STARK and a post-quantum signature (SLH-DSA per FIPS 205 [10]) entirely on Solana L1. Our prototype adapts Winterfell 0.12 [22]: SHA-256 is routed to a single `hashv` syscall per serialized transcript to reduce hashing overhead; critical FRI components are annotated to suppress inlining to respect SBF stack limits; memory is managed by a custom bump allocator synchronized with the transaction's requested heap frame. The message is uploaded as a single on-chain account that contains the ciphertext, Kyber768 KEM ciphertext, and the STARK proof; uploads are bounded to ≤ 900 byte chunks and protected by a rolling hash chain. Finalization proceeds in two steps: (1) verify and persist the SLH-DSA signature of the application transcript; (2) derive public inputs $(\text{seed}, \text{inc})$ from SHA256(`cipher`) and verify the embedded STARK.

Our experimental question is specific and practical: *can a full L1 STARK+PQC verifier fit within the typical transaction CU budget at around 128-bit security on a modern L1, and which parameters matter most?* We cast this as the following hypotheses:

H1 (budget feasibility)
L1 verification of STARK+PQC fits within 1.4×10^6 CU at $\approx$ 128-bit verifier settings.

H2 (scaling law)
With fixed FRI query counts and blowup, verifier CU scales roughly linearly with proof bytes, yielding a stable CU/Proof-byte ratio.

H3 (robust streaming)
Chunking at ≤ 900 bytes with a rolling hash chain enables constant-work rejection and predictable write costs, bounding DoS exposure.

The novelty is not a new proof system, but a *measured* end-to-end pipeline that (i) binds the STARK to the ciphertext via public inputs derived from SHA256(`cipher`), (ii) couples a PQ signature to a length-delimited application transcript, (iii) respects SVM constraints with explicit heap and stack-aware engineering, and (iv) exposes a CPI-ready surface suitable for reuse by other programs. We view this as a reference point for L1 designs that want post-quantum friendly, hash-based verification semantics today. It complements L2 and zk-rollup directions [16–20] by showing that, with careful engineering, full verification can live at L1 for small but nontrivial AIRs.

Finally, we emphasize scope and limits. The evaluated AIR is intentionally simple (a one-column affine counter); our goal is not expressiveness but a clean baseline with clear scaling behavior. We use MinConjecturedSecurity(127) at the verifier, following Winterfell's conservative accounting (a minus-one convention that caps at ≈ 127 bits under SHA-256 collision resistance), which aligns with the intuition of targeting "128-bit" security [5, 22]. The results and design lessons should transfer to richer settings, but the absolute numbers will change with AIR shape, hasher, and FRI policy.

2 Threat Model, Goals, and Binding Mechanism

We consider a two-phase adversary $\mathcal{A}$ that records the full ledger at t_0 and gains advanced (possibly quantum) capabilities at $t_1 > t_0$. We target four properties and formalize games. Let $\text{Acc}(\cdot)$ denote on-chain acceptance by the deployed verifier.

2.1 Security Games

Soundness G^{snd}: $\mathcal{A}$ outputs $(\texttt{cipher}, \texttt{proof}, \texttt{meta})$ such that $\text{Acc}(\texttt{cipher}, \texttt{proof}, \texttt{meta})=1$ without a corresponding honest pipeline run. Advantage Adv^{snd} is the success probability.
Unforgeability G^{uf}: With access to verification keys but not signing keys, $\mathcal{A}$ outputs $(\texttt{msg}, \texttt{sig})$ accepted by on-chain signature verification where `msg` was not authorized by the signer. Advantage Adv^{uf} is the success probability.
Binding G^{bind}: $\mathcal{A}$ outputs $(\texttt{cipher}_1, \texttt{cipher}_2, \texttt{proof})$ with $\texttt{cipher}_1 \neq \texttt{cipher}_2$ such that the verifier accepts the proof bound to $\texttt{cipher}_1$ while the signed transcript commits to $\texttt{cipher}_2$. Advantage Adv^{bind} is the success probability.
Confidentiality G^{conf}: The challenger samples $b \in \{0,1\}$ and encrypts m_b into `cipher` using ML-KEM-768 and AES-256-GCM (key via HKDF-SHA256); $\mathcal{A}$ outputs b' from recorded on-chain artifacts. Advantage $\text{Adv}^{\text{conf}} := |\Pr[b' = b] - \frac{1}{2}|$.

2.2 Target Properties and Mechanisms

Soundness (hash-based STARK at $\approx$ 128-bit verifier setting [3, 5]); *Unforgeability* (SLH-DSA SHA2-128s over a length-delimited transcript [10]); *Binding* (public inputs derived from d=SHA256(`cipher`)); *Confidentiality* (Kyber768 KEM, HKDF-SHA256, AES-256-GCM [8]). DoS attempts are bounded by chunk-level constant work and account-size caps.

2.3 Public-Input Binding (Derivation)

We compute $d = \text{SHA256}(\texttt{cipher})$ and set

$$\text{seed} = \text{LE_u64}(d[0..8)) \quad \text{and} \quad \text{inc} = \text{LE_u64}(d[8..16)).$$

These public inputs for the affine-counter AIR prevent replay of proofs against a different ciphertext without detection: the verifier asserts x_0=seed and $x_{\ell-1}$=seed $+ (\ell - 1) \cdot$ inc for trace length ℓ.

3 Architecture and Implementation

3.1 Data Flow and State Machine

End-to-end sequence (line-broken to fit width):

$$\begin{aligned}\texttt{init_buffer} &\rightarrow \texttt{upload_body}(\texttt{cipher} \parallel \texttt{kem} \parallel \texttt{proof}) \\ &\rightarrow \texttt{init_signature} \rightarrow \texttt{upload_signature}(7{,}856\text{ B}) \\ &\rightarrow \texttt{finalize_sig} \rightarrow \texttt{verify_stark}.\end{aligned}$$

Two-phase finalization verifies and persists the signature (*FinalizeSig*) before proof verification (*VerifyStark*), enabling early rejection with bounded CU.

Streaming and DoS control. Uploads use ≤ 900 byte chunks with a rolling hash:

$$H_{i+1} = \text{SHA256}(H_i \parallel \text{chunk}_i), \qquad H_0 = 0^{32}.$$

Each write must satisfy the following admissibility checks (constant work per chunk):
(i) fixed-offset append: offset = length,
(ii) chunk-size bound: $|\text{chunk}| \leq 900$,

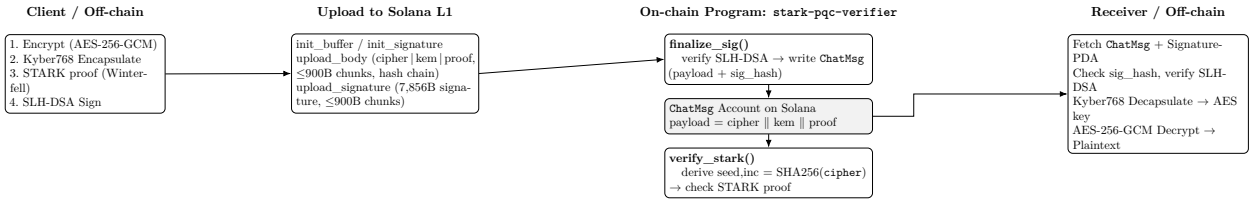

Figure 2: End-to-end flow from client to on-chain verification and the receiver.

Table 1: Key account sizes and limits.

Account / Item	**Limit / Size**
Max account bytes (`MAX_ACCOUNT_BYTES`)	10,240 B
`ChatMsg.payload` (cipher ‖ kem ‖ proof)	$\leq$ 10,068 B
`BufferPda` (staging)	10,232 B
`Signature-PDA` payload cap	$\leq$ 10,156 B
SLH-DSA SHA2-128s signature length	7,856 B
Chunk size (body/signature)	$\leq$ 900 B
Verifier acceptance policy	MinConjecturedSecurity(127)

(iii) total-length bound: payload size $\leq$ account cap.

These checks allow early rejection on mismatch (no proof work) and keep per-write CU predictable (cf. Solana transaction and compute constraints [13, 15]).

3.2 Accounts, Limits, and Transcript

Signature transcript. $T_{\text{sig}} :=$ `cipher` ‖ `kem` ‖ $\texttt{nonce}_{12}$ ‖ $\texttt{slot}_{\text{LE8}}$. Variable-length parts are delimited by (`cipher_len`, `kem_len`). A domain-separation tag (DST) is reserved for future versions (empty in this prototype).

STARK-binding transcript. $T_{\text{stark}} :=$ `cipher`; public inputs are derived from d=SHA256(T_{stark}) per Section 2.3 (cf. [8, 10]).

3.3 Verifier and Runtime Techniques

Winterfell 0.12 on SBF. We use Solana's `hashv` syscall to hash serialized transcripts in a single call, annotate critical FRI components to suppress inlining (SBF stack limits), and synchronize a bump allocator with `requestHeapFrame` (128/256 KiB) [14, 22, 25, 26]. The prover uses representative options (queries $\approx$ 30, blowup $\approx$ 16, grinding $\approx$ 8, folding factor 4); the verifier enforces MinConjecturedSecurity(127). These settings fell within the CU envelope in Section 4.2 and revealed stable scaling trends [3–5].

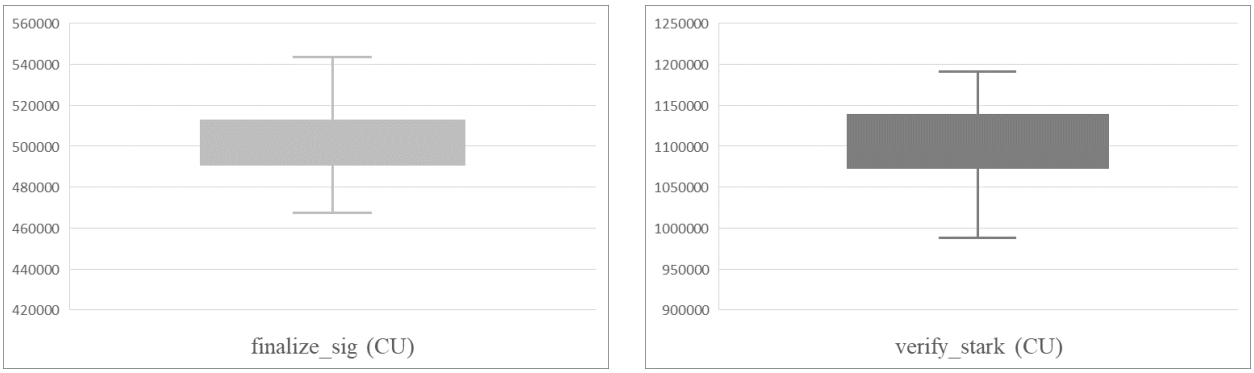

Figure 3: Compute-units distributions across n=100 runs (box-and-whisker). Left: `finalize_sig`. Right: `verify_stark`. Boxes show IQR with median; whiskers indicate the conventional 1.5×IQR extent; outliers appear as individual marks if present.

4 Evaluation and Measurement Results

4.1 Methodology and Metrics

We run the pipeline on Solana *devnet* with fixed RPC/commitment and repeat n=100 times. The primary metric is `computeUnitsConsumed`. Derived metrics are

$$\text{CU/Proof-byte} := \frac{\text{CU}_{\text{verify}}}{\text{proof_len}}, \qquad \text{CU/Sig-byte} := \frac{\text{CU}_{\text{finalize}}}{\text{sig_len}}, \qquad \text{CU ratio} := \frac{\text{CU}}{1.4 \times 10^6}.$$

Representative sizes: proof ≈ 4,437 B; signature = 7,856 B. Environment details and compute budget/heap semantics follow Solana documentation [13, 14].

4.2 Baseline Results ($n = 100$)

Finalize phase (`finalize_sig`). Mean 501,263 CU; median 499,946; min 467,360; max 543,534. Derived CU/Sig-byte ≈ 63.8; mean CU ratio ≈ 35.8%.
Verification phase (`verify_stark`). Mean 1,104,510 CU; median 1,110,510; min 988,312; max 1,190,982. Derived CU/Proof-byte ≈ 248.9; mean CU ratio ≈ 78.9%. All runs are under 1.4×10^6 CU, supporting H1.

4.3 Scaling, Sensitivity, and Failure Envelope

FRI sensitivity (RQ2). Empirically, CU fits CU $\approx a \cdot |\text{proof}| + b \cdot \text{queries} + c$ with $|\text{proof}|$ as the dominant factor, consistent with linear scaling in proof bytes (H2). The observed dispersion of CU under fixed settings is summarized in Figure 3; this distributional view complements per-byte intensities reported above.

AIR dependence (RQ3). Moving from affine to higher-degree AIRs increases constraint count/degree; CU tracks both the number of assertions and their algebraic degree. A single-column affine counter offers a clean baseline for isolating STARK overhead from app logic.

Chunking trade-offs (RQ4). 900-byte chunks reduce transaction count and amortize fixed per-tx costs; 600-byte chunks improve retransmission locality under failures. The rolling hash enforces constant-work rejection at chunk granularity, capping adversarial waste.

Failure boundary (RQ5). A sharp threshold emerges: signature verification is stable by $\approx 0.55\times10^6$ CU request; STARK verification succeeds around 1.1×10^6 CU with a 256 KiB heap. Below these budgets, failures concentrate in allocation pressure and Merkle/FRI stages.

5 Reproducibility and Artifact Metadata

Code and version. Repository: `pqzk-labs/solana-pqzk-fullchain` [21]. Version `v0.1.0`.

Program ID (devnet). The active program ID is documented in the project's `README.md` file.

Toolchain (as tested). Rust/Cargo 1.88.0; Solana CLI 2.2.20 (Agave); Node.js v24.4.0; npm 11.4.2; `wasm-pack` 0.13.1; Anchor `anchor-lang` 0.31.1; `solana-program` 2.3.0 [25, 26].

Execution environment. Linux and macOS hosts, plus Windows Subsystem for Linux (WSL2). Native Windows is not supported by `solana-cli`.

How to run.

```
npm -prefix examples/cli-chat run setup
npm -prefix examples/cli-chat run keys
npm -prefix examples/cli-chat run upload
npm -prefix examples/cli-chat run finalize
npm -prefix examples/cli-chat run receive
```

Pinned IDL/types under `fixed/` allow running without an Anchor build.

Data for figures. Bench CSVs under `examples / benchmarks/`: `results . csv`, `results-finalize-only.csv`, `results-verify-only.csv`. Figure 3 is derived from n=100 runs; the PNGs `finalize_sig.png`, `verify_stark.png` were generated from these CSVs. The bench runner is `examples/benchmarks/run-bench.sh`.

Parameters and settings. STARK verifier target: MinConjecturedSecurity(127) (Winterfell). Prover options: queries $\approx$ 30, blowup $\approx$ 16, grinding $\approx$ 8, folding = 4. Heap frames: 128 KiB (`finalize_sig`) and 256 KiB (`verify_stark`). Chunk size limit: $\leq$ 900 B. Maximum `ChatMsg.payload` $\leq$ 10,068 B.

6 Discussion: Design Implications and Limitations

(1) **Budget for bytes.** Treat proof length as the primary driver (H2); once within budget, adjust FRI queries for security margin.
(2) **Make rejection cheap.** Enforce fixed-offset appends and rolling-hash checks to bound adversarial burn per chunk (H3).
(3) **Right-size heap.** Request $\sim$128 KiB for signature finalization and $\sim$256 KiB for STARK verification; mirror limits on chain to avoid allocator surprises [14].
(4) **Separate phases.** Persist signature results prior to proof verification to avoid expensive STARK work on bad payloads.
(5) **Interfaces enable reuse.** A CPI-friendly verifier with pinned IDL/types lowers adoption cost for other programs.

6.1 Limitations and Future Work

Our AIR is intentionally simple; richer AIRs increase CU with constraint degree/width. Devnet figures may shift under mainnet load; expanding sampling will strengthen external validity. Future directions: (i) AIR templates for common app logics, (ii) parameter tightening at fixed security, (iii) PQC diversification (e.g., ML-DSA per FIPS 204 [9]), (iv) security review/audit, (v) a shared verifier interface hardened for CPI by third-party programs.

7 Conclusion

The central question of this work is whether a *full L1* pipeline that verifies both a hash-based ZK-STARK and a post-quantum signature can operate within the practical constraints of a modern L1, without privileged precompiles or external trust, and with predictable cost. Our results on Solana L1 answer in the affirmative for a representative security target and a minimal but nontrivial AIR.

Beyond the headline CU numbers—$\approx 5.01 \times 10^5$ CU for signature finalization and $\approx 1.10 \times 10^6$ CU for STARK verification at the evaluated sizes—the study surfaces design levers that generalize:

- **Hashing path matters.** A single `hashv` over serialized transcripts substantially reduces syscall and copy overhead compared to incremental hashing. This is especially impactful for Merkle commitments and random coins inside STARK verification.
- **Stack discipline matters.** SBF's tight stack motivates inlining suppression in FRI hotspots and favors designs that move large temporaries to heap. An explicit, synchronized heap model (via `requestHeapFrame` mirrored in-program) avoids allocator surprises.
- **Bytes dominate budgets.** With a fixed FRI policy, verifier CU tracks proof bytes closely (H2). Engineering effort should first minimize proof width/length consistent with the desired security, then tune query counts.
- **Make failure cheap.** A two-step finalization that persists signature results before proof work, combined with chunked uploads and rolling-hash checks, ensures malformed inputs are rejected at low, constant cost (H3).
- **Interfaces enable reuse.** A CPI-friendly verifier with pinned IDL/types lowers the barrier for other programs to adopt post-quantum, hash-based verification semantics without re-implementing the stack.

It complements L2 and zk-rollup directions [16–20] by showing that, with careful engineering, full verification can live at L1 for small-to-moderate statements today, while sharing design levers useful at L2.

References

[1] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Review*, 41(2):303–332, 1999. doi:10.1137/S0036144598347011.

[2] J. Proos and C. Zalka. Shor's discrete logarithm quantum algorithm for elliptic curves. *arXiv:quant-ph/0301141*, 2004. `https://arxiv.org/abs/quant-ph/0301141`.

[3] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *ICALP 2018*, LIPIcs 107, 14:1–14:17. doi:10.4230/LIPIcs.ICALP.2018.14.

[4] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf. DEEP-FRI: Sampling outside the box improves soundness. *arXiv:1903.12243*, 2019. https://arxiv.org/abs/1903.12243.

[5] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018/046. https://eprint.iacr.org/2018/046.

[6] J. Groth. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016*, pp. 305–326. doi:10.1007/978-3-662-49896-5_11.

[7] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *Cryptology ePrint Archive*, 2019/953. https://eprint.iacr.org/2019/953.

[8] NIST. Module-Lattice-Based Key-Encapsulation Mechanism Standard (FIPS 203). 2024. doi:10.6028/NIST.FIPS.203. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.203.pdf.

[9] NIST. Module-Lattice-Based Digital Signature Standard (FIPS 204). 2024. doi:10.6028/NIST.FIPS.204. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.204.pdf.

[10] NIST. Stateless Hash-Based Digital Signature Standard (FIPS 205). 2024. doi:10.6028/NIST.FIPS.205. https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.205.pdf.

[11] A. Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. Whitepaper, 2018.

[12] Light Protocol Team. Scaling the Design Space for On-Chain Applications with ZK Compression. Whitepaper, 2025. https://www.zkcompression.com/references/whitepaper. (accessed Sep. 21, 2025)

[13] Solana Foundation. Transaction Fees — Compute Units and Limits. Docs, 2025. https://solana.com/docs/core/fees. (accessed Sep. 21, 2025)

[14] Solana Foundation. How to Request Optimal Compute Budget. Docs, 2025. https://solana.com/developers/guides/advanced/how-to-request-optimal-compute. (accessed Sep. 21, 2025)

[15] Solana Foundation. Transactions and Instructions — Transaction Size Limit. Docs, 2025. https://solana.com/docs/core/transactions. (accessed Sep. 21, 2025)

[16] StarkWare Industries. Starknet: A Validity Rollup Powered by STARK Proofs. Official documentation, 2025. https://starkware.co/starknet/. (accessed Sep. 21, 2025)

[17] Aztec Labs. Aztec Protocol Documentation. 2025. https://docs.aztec.network/. (accessed Sep. 21, 2025)

[18] Polygon Labs. Polygon zkEVM: Scalable EVM-equivalent Zero-Knowledge Rollup. 2025. https://polygon.technology/polygon-zkevm. (accessed Sep. 21, 2025)

[19] Scroll Foundation. Scroll zkEVM Documentation. 2025. https://scroll.io/blog/zkEVM. (accessed Sep. 21, 2025)

[20] Cronos Labs. Cronos zkEVM Litepaper. 2025. https://docs-zkevm.cronos.org/getting-started/litepaper. (accessed Sep. 21, 2025)

[21] PQZK Labs. *solana-pqzk-fullchain: Full On-Chain ZK-STARK + PQC Verification on Solana.* v0.1.0, 2025. https://github.com/pqzk-labs/solana-pqzk-fullchain.

[22] Meta (Facebook). Winterfell: A STARK prover and verifier for arbitrary computations. v0.12.0, 2025. https://crates.io/crates/winterfell.

[23] RustCrypto. slh-dsa: Pure Rust implementation of SLH-DSA (SPHINCS+) as described in the FIPS-205 standard. v0.0.3, 2025. https://crates.io/crates/slh-dsa.

[24] rustpq. pqcrypto-kyber: Post-Quantum Key-Encapsulation Mechanism kyber (Kyber/ML-KEM). v0.8.1, 2024. https://crates.io/crates/pqcrypto-kyber.

[25] Solana Foundation. anchor-lang: Solana Sealevel eDSL. v0.31.1, 2025. https://crates.io/crates/anchor-lang.

[26] Anza. solana-program: Solana Program. v2.3.0, 2025. https://crates.io/crates/solana-program.