GLYPH: A Universal Transparent Verification Layer for Heterogeneous Zero-Knowledge Proof Systems on Ethereum

Christopher Schulze
`schulze.christopher@icloud.com`

February 4, 2026

Abstract

GLYPH is a transparent verification layer for Ethereum that enables trustless on-chain verification of heterogeneous proof systems by executing upstream verification inside GLYPH-Prover, covering pairing-based SNARKs and hash-based STARKs without requiring protocol-specific verifiers or any GLYPH-specific trusted setup. Proofs imported from external systems retain their original trust assumptions. The architecture combines a universal adapter framework with a packed GKR sumcheck protocol, compressing verification logic into a compact, round-dependent on-chain check. The on-chain verifier executes a packed arity-8 sumcheck protocol over the 128-bit prime field ($p = 2^{128} - 159$), with execution gas estimates in the low thousands and total tx gas measured at 29.45k on Sepolia and Hoodi receipts; see Appendix B for testnet evidence. The *GLYPH Artifact Boundary* provides a stable, cryptographic interface between off-chain provers and the blockchain, verified via a single contract deployment. Binding includes chain id and verifier address, preventing cross-chain replay and enabling a single stateless verifier across adapter families without redeployment. This paper describes the system architecture, security analysis, and the benchmark evidence for a unified verification layer in the Ethereum modular stack.

1 Introduction

Zero-knowledge proofs are the cornerstone of Ethereum's scaling roadmap, yet the ecosystem remains fragmented across proof systems and verifier deployments. Every new proof system (Groth16, PLONK, Halo2, STARKs, Folding Schemes) introduces its own verification complexity, distinct on-chain contracts, and potential security vulnerabilities. Developers are forced to choose between efficiency (trusted setups) and transparency (STARKs with high on-chain verification costs), while managing multiple incompatible proof formats and bespoke verifier deployments. GLYPH addresses this fragmentation by introducing a *universal verification layer*. Instead of implementing yet another proving scheme, GLYPH compiles heterogeneous upstream verification relations into a single, canonical, gas-efficient on-chain check. This architecture is not limited to L2 rollups; any application that needs a compact, transparent verification step, from bridges to audit layers to specialized app chains, can use GLYPH as a verification layer.

Name and metaphor. A *GLYPH* (IPA: [glif]) evokes a primordial sign carved into stone, a durable symbol that compresses layered meaning into a form, enduring by design. GLYPH applies the same principle to verification, distilling heterogeneous proofs into a compact, immutable on-chain artifact.

Reference implementation available at `https://github.com/Christopher-Schulze/glyph-zk`.

1.1 Design Goals

GLYPH is designed with a focus on stability and security:

1. **Trustless Validity Chain:** GLYPH proves verification. An on-chain GLYPH artifact implies, under standard assumptions and correct adapters, that the proof satisfied its constraints when the adapter verifies it in GLYPH-Prover. The IVC adapter enforces a transparent PCS opening for BaseFold and a transparent R1CS receipt for Nova-family formats.[22, 18, 19, 20, 21] Nova and SuperNova RecursiveSNARK proofs are verified against the canonical receipt; HyperNova or Sangria proofs use CompressedSNARK. Trustlessness depends on adapter correctness; security reduces to UCIR equivalence, sumcheck, and PCS binding.

2. **Transparency by Design:** The GLYPH verifier and proving system require no trusted setup or structured reference string (SRS) in the GLYPH core; wrapping trusted-setup proof systems preserves their original trust assumptions.

3. **Universal Adapter Surface:** A stabilized interface supporting everything from pairing-based SNARKs (Groth16, KZG, PLONK) to hash-based STARKs and IVC/folding schemes.

4. **Interoperability Without Lock-In:** GLYPH does not impose a new proving language or runtime. Adapters compile existing proof systems into UCIR, allowing projects to retain their toolchains while gaining a common settlement layer.

5. **Gas-Efficient Verification:** Verification cost is minimized via packing (single-digit thousands of execution gas for typical round counts, and 29.45k total tx gas on Sepolia and Hoodi for artifact-bound layouts; see Appendix B); the on-chain cost is independent of upstream proof size and scales with rounds and calldata length.

Breaking the ZK Trilemma. Ethereum ZK has historically faced a trade-off: *low-cost verification* typically relies on trusted setups (Groth16), while transparent verification tends to incur higher on-chain costs due to large proofs and calldata pricing (STARKs).[7, 9, 16] GLYPH breaks this trade-off by shifting expensive upstream verification off-chain and verifying only a compact, low-cost sumcheck proof on-chain. The on-chain cost scales with the number of sumcheck rounds and calldata length, but is independent of upstream proof size. The result is transparency *and* efficiency, previously considered mutually exclusive.

1.2 Threat Model and Assumptions

GLYPH is designed to be secure against a strong adversary. We assume:

- The adversary controls the upstream prover, adapter inputs, and calldata, and can supply malformed proofs or receipts.
- The adversary can observe, replay, or reorder transactions but cannot break Ethereum consensus or execution correctness.
- The adversary seeks calldata accepted by `GLYPHVerifier` without a valid upstream proof.
- Correct execution of the EVM and Keccak256.
- Keccak256 behaves as a random oracle for the transcript and domain separation.
- Soundness of packed sumcheck over the 128-bit prime field $p = 2^{128} - 159$.
- Soundness of BaseFold PCS, LogUp lookup arguments, and upstream verification checks implemented inside adapters and receipt verifiers.
- Upstream cryptographic assumptions for wrapped proofs and correctness of adapter implementations and canonical byte encodings; otherwise GLYPH soundness can fail for the affected adapter family.

1.3 Contributions

This paper formalizes the following contributions to the Ethereum ZK stack:

- **Transparent settlement layer:** The GLYPH verifier is SRS-free and uses standard Ethereum primitives (Keccak256 and EVM arithmetic over a 128-bit prime field), enabling a single transparent contract deployment for heterogeneous proof systems. This turns one transparent contract into a common settlement anchor.

- **Gas Efficiency:** GLYPH verification costs a few thousand execution gas in local benchmarks and 29.45k total tx gas on Sepolia and Hoodi (Appendix B). In the same receipts, Groth16 verification with 3 public inputs costs 227,128 total tx gas. For pairing-based verifiers, precompile costs follow EIP-196 and EIP-197 (repriced by EIP-1108), while calldata pricing follows EIP-2028; larger proof sizes therefore increase total gas.[13, 14, 15, 16] Savings scale with upstream proof size because on-chain work is fixed by sumcheck rounds.

- **Packed GKR Sumcheck:** A gas-optimized arity-8 sumcheck verifier tailored for the EVM, utilizing packed arithmetic over a 128-bit prime field. This keeps transparent verification viable at low gas.

- **Universal Adapter Architecture:** A formal specification unifying incompatible proof formats. The adapter framework supports the following families:
  - Pairing-based SNARKs and KZG: Groth16, PLONK, Halo2-KZG, KZG commitments, IPA (BN254, BLS12-381), SP1 receipts (Groth16/Plonk BN254), and PLONK receipts (BN254 gnark, BLS12-381 dusk).[7, 8, 4]
  - IVC and folding: Nova, HyperNova, external Nova and SuperNova proofs, HyperNova and Sangria via CompressedSNARK.
  - Keccak hash adapter (Keccak merge).
  - Binius constraint-system receipts.
  - STARK receipts:
    - Standard FRI BabyBear (SHA3, Blake3, Poseidon, Rescue) and RISC Zero.
    - Winterfell receipts: f128/f64 do_work/fibonacci/tribonacci (SHA3, Blake3).
    - Circle STARK receipts: M31, BabyBear, KoalaBear (SHA3, Blake3, Poseidon, Rescue).
    - Goldilocks receipts: Plonky2 Goldilocks SHA3 (STARK Goldilocks enabled), Plonky3 Goldilocks (Poseidon2, Poseidon, Rescue, Blake3), Miden Goldilocks (Blake3-192/256, RPO256, RPX256, Poseidon2, no precompile requests).
    - M31 receipts: Stwo M31 (Blake2s), Plonky3 M31 (Keccak).
    - Cairo receipts: Starknet Prime with Starknet-with-Keccak (keccak-160-lsb, Stone6).

    [9, 25, 24, 23, 27, 29, 28]

  Trustless verification is implemented for Groth16, KZG, IPA, STARK, the Keccak hash adapter, IVC, and Binius via transparent PCS and transparent R1CS receipts. The architecture is designed for extensibility: new adapter families can be added without modifying the core verifier, preserving backward compatibility. Existing toolchains require no changes - developers integrate GLYPH as an additional verification layer. A unified interface preserves ecosystem diversity: projects retain preferred proof systems while converging on a common settlement standard. The design deploys in environments with comparable hash and 128-bit prime-field arithmetic.

- **State Diff Binding Layer:** Optional state-diff tooling binds transition diffs into the statement hash, enabling auditability and deterministic root updates without modifying the on-chain verifier.

2 System UCIR Architecture

GLYPH implements a layered "Compiler-Verifier" architecture separating upstream proof complexity from on-chain verification. Adapters compile upstream verification relations into UCIR; GLYPH-Prover proves UCIR constraints and emits a GLYPH artifact bound to the packed on-chain sumcheck verifier. This separation minimizes on-chain gas while preserving off-chain rate.

Hybrid Field Architecture. GLYPH matches the field to each workload stage:

- **Host arithmetic:** Goldilocks ($2^{64} - 2^{32} + 1$) for off-chain sumcheck and adapter computation.
- **PCS commitments:** BaseFold over binary tower fields for hashing and packing.
- **Settlement:** packed arity-8 sumcheck over $p = 2^{128} - 159$ for EVM-friendly arithmetic.

Claims are ring-switched from Goldilocks into p before artifact packing; see `docs/specs/` for the exact derivation.

2.1 Architectural Layers

The system comprises four layers: (1) **Adapter Layer**, wrappers that accept upstream proof bytes and public inputs, verify them off-chain, and produce canonical UCIR constraints; (2) **Constraint Compilation (UCIR)**, a unified representation for arithmetic gates, copy constraints, custom gates, and lookup tables; (3) **Prover Core (GLYPH-Prover)**, integrating witness generation, constraint evaluation, LogUp product-tree GKR, BaseFold PCS commitments, Goldilocks sumcheck, and packed arity-8 proof derivation; and (4) **On-Chain Verifier (GLYPHVerifier)**, a single Solidity contract implementing packed arity-8 sumcheck verification over the 128-bit prime field $p = 2^{128} - 159$, calldata-only with no storage writes.

Assumption (2.1 and 2.2). Upstream cryptographic assumptions, adapter correctness, and canonical byte encodings; otherwise GLYPH soundness can fail for the affected adapter family.

2.2 The GLYPH Artifact Boundary

The *GLYPH artifact boundary* links the off-chain prover and on-chain verifier.

Specification reference. The canonical artifact binding layout and derivations are specified in `docs/specs/artifact_tag_spec.md`.

Definition 1 (GLYPH Artifact). The GLYPH artifact is the tuple $(\mathsf{commitment\_tag}, \mathsf{point\_tag}, \mathsf{claim128}, \mathsf{initial\_claim})$ where $\mathsf{commitment\_tag} \in \{0,1\}^{256}$ is the domain-separated hash of the BaseFold commitment and optional ZK bindings; $\mathsf{point\_tag} \in \{0,1\}^{256}$ is derived from commitment and evaluation point via domain-separated hashing; $\mathsf{claim128} \in [0, p)$ (with $p = 2^{128} - 159$) is the evaluated claim at the evaluation point, encoded as a 128-bit integer; and $\mathsf{initial\_claim} \in [0, p)$ is the initial claim bound into the transcript for binding.

Packed header and artifact tag. On-chain calldata packs `artifact_tag`, `claim128`, `initial_claim`. The tag is `artifact_tag = Keccak256(commitment_tag|| point_tag)`. The on-chain verifier recomputes chain binding from `chainid`, `address(this)`, `artifact_tag`, `claim128`, and `initial_claim`, and enforces `claim128` $< p$ and `initial_claim` $< p$. In calldata, `claim128` is the high 16 bytes and `initial_claim` the low 16 bytes of one 32-byte word, both big-endian. This keeps verifier logic fixed across adapter families; upstream proof system changes require no redeployment.

The GLYPH Settlement Invariant. Assuming adapter-defined validity and upstream assumptions, an accepted GLYPH artifact implies the off-chain relation for GLYPH-Prover-verified families, independent of proof size. This yields a settlement layer: (1) adapter verifies upstream proof or receipt, compiles UCIR constraints; (2) GLYPH-Prover generates a witness, runs Goldilocks sumcheck, emits a packed arity-8 LogUp/BaseFold proof; (3) the artifact boundary binds to the on-chain header via `artifact_tag` and chain binding; and (4) `GLYPHVerifier` enforces packed sumcheck and header checks.

2.3 Adapter Families

GLYPH defines adapter families as a stable protocol surface:

Groth16 SNARK. Handles Groth16 and other pairing-based proofs on BN254 or BLS12-381 curves.[7] These non-transparent systems can benefit by reduced on-chain gas.

KZG SNARK. Supports PLONK and other polynomial commitment schemes using KZG commitments.[4, 8] Commonly used in Ethereum-native rollups.

IVC/Folding. Covers incremental verifiable computation formats including the CCS-based Nova family (SuperNova, HyperNova), Sangria, and the BaseFold PCS.[18, 19, 20, 21, 22] The current implementation verifies a transparent PCS opening for BaseFold and a transparent R1CS receipt for Nova-family formats. Nova and SuperNova external RecursiveSNARK proofs are verified against the canonical receipt, and HyperNova/Sangria proofs are verified via CompressedSNARK. Proof fields are deterministic functions of the receipt hash.

IPA. Handles Halo2-IPA and Bulletproofs-style inner product arguments [5].

STARK. The universal STARK entry point, accepting receipts from multiple implementations: Winterfell do_work/fibonacci/tribonacci (f128/f64 with SHA3 or Blake3), Circle STARK (M31/BabyBear/KoalaBear, SHA3, Blake3, Poseidon, or Rescue), Stwo, Plonky2 Goldilocks SHA3 (STARK Goldilocks enabled), Plonky3 (Poseidon2, Poseidon, Rescue, Blake3 for BabyBear, KoalaBear, Goldilocks; Keccak for M31), Standard FRI BabyBear (SHA3, Blake3, Poseidon, or Rescue, RISC Zero compatible), Cairo Starknet Prime with Starknet-with-Keccak (keccak-160-lsb, Stone6 monolith), and Miden Goldilocks (Blake3-192/256, RPO256, RPX256, Poseidon2, no precompile requests).[25, 24, 23, 27, 29, 28] STARK-specific logic is encoded in receipt, VK, and program bytes for supported canonical formats; adding new STARK systems requires defining a compatible receipt format and verifier integration.

Hash Proof. Keccak256 merge proofs for hash-based statements. Enables efficient verification of Merkle-style commitments and hash chains.

SP1. SP1 Groth16/Plonk receipts over BN254. Receipts are verified off-chain with the SP1 verifier keys and bound into the GLYPH artifact.[26]

PLONK. PLONK receipts verified off-chain using the gnark BN254 verifier or the dusk BLS12-381 verifier, then bound into the GLYPH artifact.[8]

Family	Proof Family	Supported Systems
Groth16	Pairing SNARK	Groth16 (BN254, BLS12-381)
KZG	KZG SNARK	PLONK and other KZG-based schemes
IVC/Folding	IVC/Folding	Nova, SuperNova, HyperNova, Sangria, BaseFold
IPA	IPA	Halo2-IPA, Bulletproofs
STARK	STARK	Winterfell, Circle STARK, Stwo, Plonky2 (STARK Goldilocks enabled), Plonky3, Standard FRI (RISC Zero compatible), Cairo, Miden
Hash	Hash Proof	Keccak256 merge proofs
SP1	SP1	SP1 Groth16/Plonk (BN254)
PLONK	PLONK	PLONK (BN254 gnark, BLS12-381 dusk)

Table 1: GLYPH adapter families and supported proof systems.

Note. The IVC adapter verifies transparent PCS opening for BaseFold and transparent R1CS receipts for Nova-family formats. External Nova and SuperNova proofs are verified, and HyperNova/Sangria proofs are verified via CompressedSNARK against the canonical receipt.

2.4 Prover Pipeline

The GLYPH-Prover executes the following phases:

1. **Adapter Compilation:** The adapter verifies the upstream proof and emits UCIR constraints.
2. **Witness Generation:** Compute wire values satisfying all constraints, including lookup multiplicities.
3. **LogUp Proof:** Generate product-tree GKR proofs for all lookup tables.
4. **PCS Commitment:** Commit to the witness polynomial via BaseFold (matrix encoding with Merkle tree).
5. **Sumcheck Rounds:** Run off-chain Goldilocks sumcheck rounds on the combined constraint polynomial.
6. **PCS Opening:** Open the polynomial commitment at the transcript-derived evaluation point.
7. **Artifact Derivation:** Compute the GLYPH artifact boundary and emit packed arity-8 calldata for on-chain verification.

The prover supports two modes: `ZkMode` (default) uses salted PCS commitments and blinding rows for zero-knowledge, while `FastMode` omits blinding for development and testing.

3 Cryptographic Construction

GLYPH is built on standard cryptographic building blocks: the Fiat–Shamir transform [6] instantiated with Keccak256 [12, 11], BaseFold PCS commitments over binary tower fields, GKR-style sumcheck protocols [1], and LogUp lookup arguments [2].

3.1 Universal Constraint IR (UCIR)

The constraint system is represented in UCIR, a unified intermediate representation supporting three gate types:

Definition 1 (UCIR Gate Types)**.** • ***Arithmetic Gate*** *(tag 0x01): Enforces* $q_m \cdot a \cdot b + q_l \cdot a + q_r \cdot b + q_o \cdot c + q_c = 0$ *for selector coefficients* q_* *and wire values* a, b, c*.*
- ***Copy Gate*** *(tag 0x02): Enforces* left = right *for two wire references.*
- ***Custom Gate*** *(tag* $\geq$ *0x80): Family-specific operations including curve and field operations for upstream verifiers, Keccak merge, and adapter verify gates.*

The witness buffer is segmented into: (1) public inputs, (2) wire values, (3) lookup multiplicities, and (4) blinding values (ZK mode only).

3.2 LogUp Lookup Argument

Lookups are implemented via LogUp [2] using product-tree GKR. For a table $T = \{t_1, \ldots, t_m\}$ and lookup values $\{v_1, \ldots, v_n\}$ with multiplicities $\{m_1, \ldots, m_m\}$, the argument proves:

$$\prod_{i=1}^{n} (\beta - v_i) = \prod_{j=1}^{m} (\beta - t_j)^{m_j}$$

for a random challenge β derived from the transcript.

The products are computed via binary product trees where $L_{k+1}[i] = L_k[2i] \cdot L_k[2i+1]$. Layer constraints are folded into the packed sumcheck, avoiding separate opening proofs.

Canonical table IDs include: `TABLE_RANGE8` (values $[0, 256)$), `TABLE_RANGE16` (values $[0, 65536)$), `TABLE_BIT` ($\{0, 1\}$), and `TABLE_CHI5` (Chi-5 distribution).

3.3 BaseFold PCS Commitment

GLYPH uses BaseFold PCS over binary tower fields for polynomial commitments.[22] Commitments and openings are verified off-chain, while the commitment tag and point tag are bound into the on-chain artifact via the packed sumcheck. BaseFold encodes the witness polynomial as a matrix and commits via a Merkle tree over binary tower fields; the resulting commitment tag is the on-chain anchor for the PCS state.

Definition 2 (BaseFold Commitment Tag)**.** *Given a BaseFold commitment object, GLYPH derives:*

1. *Compute* `base_tag` *from the BaseFold commitment (implementation-defined).*
2. *Compute* `commitment_tag = Keccak256(PCS_COMMIT_DOMAIN || base_tag || [salt_commitment] || [mask_commitment])`*.*

The PCS uses domain separation tags for commitments, openings, ring-switch binding, and ZK mask binding. The canonical tag list is documented in the project documentation. Tags are fixed labels that prevent cross-context reuse of commitments and openings. Commitment tags are deterministic for a fixed commitment and mask configuration. The shared tag registry prevents cross-domain ambiguity across adapters.

3.4 Packed Sumcheck Protocol

The core verification mechanism is a packed arity-8 sumcheck protocol. Given a multivariate polynomial $f(x_0, \ldots, x_{R-1})$ of degree 2 in each variable, the prover demonstrates that f evaluates to a claimed value at a random point. All verifier arithmetic is performed modulo the 128-bit prime $p = 2^{128} - 159$. Packed arithmetic evaluates the round constraint over $t \in \{0, \ldots, 7\}$ with two coefficients, keeping the protocol low-degree while reducing calldata and verifier work.
In each round i, the prover sends the coefficients (c_0, c_1) of the univariate polynomial

$$g_i(t) = c_0 + c_1 t + c_2 t^2$$

where c_2 is recovered from the sumcheck constraint. The verifier checks the transition constraint $g_i(0) + \cdots + g_i(7) = \mathsf{claim}_i$ and derives the next challenge r_i via Fiat–Shamir. After R rounds, the verifier computes the expected final claim from an artifact-defined public polynomial and checks equality.

3.5 Transcript and Domain Separation

All Fiat–Shamir challenges use Keccak256 with explicit domain separation:

- **Binding domains:** `GLYPH_GKR_BIND_UNBOUND`, `GLYPH_GKR_BIND_STATEMENT`, `GLYPH_GKR_BIND_STMT_POLY`
- **Linear coefficient domain:** `GLYPH_GKR_ARTIFACT_LIN`
- **PCS domains:** `GLYPH_PCS_*`
- **Adapter domains:** `GLYPH_ADAPTER_*`

The transcript state is a 32-byte seed updated by hashing with each protocol message. Challenge scalars are derived by hashing and reducing the 256-bit output modulo $p = 2^{128} - 159$. Each round hashes the prior transcript with the round coefficients, and the binding domain commits to artifact-bound metadata to prevent transcript splicing or replay. Transcript labels are fixed ASCII strings hashed into field elements, ensuring explicit domain separation across adapter families. The transcript commits to round coefficients and adapter metadata so any mutation of proof content changes the derived challenges. Challenge derivation is deterministic for a given artifact and calldata layout, supporting reproducible verification.

4 Implementation

The reference implementation consists of an off-chain Rust prover and a single Solidity verifier contract.

4.1 Prover Stack (Rust)

The Rust crate (`glyph`) implements the complete prover: adapter import and canonical parsing, UCIR compilation, witness generation, LogUp lookup GKR, BaseFold PCS commitments and openings, packed sumcheck proving, Keccak256 Fiat–Shamir transcript, and SIMD-accelerated field arithmetic (AVX-512, AVX2, NEON) with optional CUDA backends. The prover supports configurable acceleration via environment variables (`GLYPH_CUDA=1`, `GLYPH_ACCEL_PROFILE`) and defaults to CPU-only with automatic SIMD detection. Performance-critical kernels use targeted SIMD paths with portable fallbacks to ensure identical arithmetic across platforms.

4.2 Adapter Architecture

Adapters share a canonical IR and digest pipeline: they verify upstream proofs off-chain, emit UCIR constraints, and bind results into the GLYPH artifact. Family modules cover Groth16, KZG and PLONK, Halo2-KZG, IPA, SP1 receipts, PLONK receipts, IVC and folding formats, Binius receipts, the Keccak hash adapter, and STARK receipts. STARK support is data-driven via the `CanonicalStarkReceipt` format; receipts encode field and hash IDs, and unsupported profiles are rejected. Supported systems are enumerated in Section 2.3 and Table 1.[25, 24, 23, 27, 29, 28] Adapter inputs are canonicalized and malformed encodings are rejected before binding into the artifact. Adapters enforce strict parsing and domain-separated hashes before any binding step, keeping the artifact fail-closed under malformed inputs. Feature flags allow trimming unused adapter families for smaller, reproducible builds. Receipt formats include explicit field and hash identifiers to prevent cross-family misinterpretation.

4.3 On-Chain Verifier

The Solidity verifier contract `GLYPHVerifier` implements the packed sumcheck verifier as a selectorless fallback function. It uses inline assembly for gas optimization and supports a single packed layout:

The byte-accurate calldata and memory layout is documented in `docs/specs/verifier_spec.md`.

- **Artifact-bound layout:** 64-byte header (`artifact_tag` and `claim128 || initial_claim`) followed by 32 bytes per round (`c0 || c1`); `c2` is recovered from the sumcheck constraint over $t \in \{0, \ldots, 7\}$.

Statement-bound layouts are defined in off-chain Rust tooling for generation and testing, but are not accepted by the on-chain verifier contract.

The verifier performs no storage writes (`SSTORE`), uses no dynamic verification keys, and executes in constant gas for a given round count. Calldata is interpreted deterministically from the header, and the artifact layout keeps verification inputs minimal and uniform across adapter families. Truncated or malformed calldata is rejected before any field arithmetic. Length checks also enforce the expected round count.

The implementation is modular by construction: adapters can be extended without modifying the on-chain verifier, and the prover stack follows the cryptographic construction in Section 3. This separation keeps deployment stable while enabling new proof systems to integrate through the off-chain pipeline.

5 Security Analysis

The security of GLYPH rests on standard cryptographic assumptions and careful protocol design. This section analyzes soundness, zero-knowledge, and transparency properties.

5.1 Security Model and Assumptions

GLYPH's security relies on:

1. **Random Oracle Model:** Keccak256 behaves as a random oracle for Fiat–Shamir challenge derivation.
2. **Collision Resistance:** Keccak256 is collision-resistant for domain separation and commitment hashing.
3. **Upstream Assumptions:** Pairing-based proofs inherit DLP hardness on their curves, and other proof systems retain their original assumptions.

5.2 Soundness

Theorem 1 (Sumcheck Soundness)**.** *Let $f : \mathbb{F}^R \to \mathbb{F}$ be a degree-d multivariate polynomial. If the prover sends round polynomials $g_i(t)$ such that the verifier accepts, then with probability $1 - R \cdot d/|\mathbb{F}|$ for degree $d \leq 2$, the claimed evaluation $f(r_0, \ldots, r_{R-1})$ equals the initial claim.*

The packed sumcheck inherits soundness from the standard sumcheck protocol [1]. The on-chain verifier enforces the transition constraint $g_i(0) + \cdots + g_i(7) = \mathsf{claim}_i$ for each round, recovers the quadratic coefficient, and derives challenges via Fiat–Shamir. Packed arithmetic is a verifier-side optimization and does not alter the soundness bound.

Trustless Validity. A critical property: if adapter witness generation enforces upstream verification, then an on-chain accepted GLYPH proof implies that the bound upstream instance verified successfully. The adapter does not merely bind digests; it proves verification. This is enforced by:

- Custom verify gates (`CUSTOM_GATE_{IVC,STARK,IPA,SP1,PLONK,BINIUS}_VERIFY`) that encode upstream verification logic as UCIR constraints.
- Witness generation that fails if upstream verification fails.
- Binding meta that commits to the full artifact including upstream statement hashes.
- Transparent PCS and transparent R1CS receipts for IVC families.

BaseFold PCS binding ties commitment openings to the transcript challenge, and the artifact boundary binds the PCS commitment and evaluation point into the on-chain verification context.

Theorem 2 (Adapter Soundness Chain)**.** *Assume the adapter-specific verification checks implemented inside GLYPH-Prover are sound (pairing/KZG security, IPA and STARK soundness, hash collision resistance, SP1 and PLONK verification, and transparent PCS + transparent R1CS verification for IVC). Then any on-chain acceptance by `GLYPHVerifier` implies that the corresponding upstream statement is valid, except with negligible probability in the security parameters. For IVC Nova-family formats, the statement is defined by a transparent R1CS receipt and deterministic proof-field derivations; external Nova and SuperNova proofs are verified, and HyperNova/Sangria proofs are verified via CompressedSNARK against the canonical receipt.*

5.3 Zero-Knowledge

In `ZkMode` (default), the prover achieves computational zero-knowledge:

- **Salted PCS:** Commitments include random salt values that hide the underlying evaluations.
- **Blinding Rows:** The witness buffer includes random blinding values that mask intermediate computations.
- **Hiding Commitments:** Merkle roots reveal no information about the committed polynomial beyond evaluation queries.

5.4 Transparency

All public parameters are derived deterministically:

- Generator points via SHA-256 try-and-increment hash-to-curve from fixed labels (`GLYPH_G_{i}`, `GLYPH_H_{i}`, `GLYPH_U`); BLS12-381 points are cofactor-cleared, and BN254 uses the prime-order G_1 subgroup.
- Domain separation tags from ASCII strings via Keccak256.
- No structured reference string, no toxic waste, no trusted setup in the GLYPH core. Upstream proofs retain their original trust assumptions.

Any party can independently derive all parameters from the specification.
GLYPH provides transparency for the settlement verifier itself. When wrapping upstream proofs that require trusted setups, those proofs retain their original assumptions.

5.5 Testing and Verification Strategy

To ensure the robustness of the GLYPH verifier and adapters, GLYPH employs a comprehensive testing strategy that goes beyond standard unit tests:

- **Property-Based Fuzzing:** The codebase uses `proptest` and `cargo-fuzz` to subject the decoding and verification logic (adapter families) to high-volume randomized inputs, aiming to detect panic conditions and invalid state transitions.
- **Corpus and Dictionary Seeding:** Fuzz targets are seeded with minimal corpora and domain-specific dictionaries for adapter and STARK decoding, enabling repeatable coverage of structured formats.
- **Differential Roundtrip Checks:** Adapter IR encode and decode roundtrips are fuzzed to ensure canonical stability under adversarial inputs.
- **Tamper and Negative Tests:** Deliberate corruption of proofs, receipts, and calldata is used to ensure fail-closed behavior across adapters and on-chain verification.
- **Snapshot Isolation:** All integration tests are bound to deterministic snapshots, ensuring that any behavioral change in the prover or verifier is explicitly detected and reviewed.

5.6 Verifier Properties

The on-chain verifier is intentionally minimal and fail-closed:

- **Single contract:** A single deployed verifier handles all adapter families.
- **Constant-size header:** A packed artifact header binds to chain id and contract address.
- **No storage writes:** Verification is stateless and uses only calldata.
- **Deterministic parsing:** All byte layouts are fixed and length-checked.
- **Chain binding:** Every accepted proof is bound to the target chain and verifier address.

5.7 On-Chain Verifier Security

The Solidity verifier is hardened against:

- **Malformed calldata:** Strict length checks; invalid layouts revert.
- **Coefficient overflow:** Require $c_i < q$ with $q = 2^{128} - 159$.
- **Claim overflow:** Require `claim128` $< q$ and `initial_claim` $< q$.
- **Binding forgery:** Recompute chain binding from `chainid`, `address(this)`, `artifact_tag`, `claim128`, and `initial_claim`; any mismatch reverts.

5.8 Failure Modes and Impact

GLYPH provides strong guarantees under its assumptions, but the following failure modes must be understood by integrators:

- **Adapter bug:** If an adapter or receipt verifier accepts an invalid upstream proof, GLYPH can accept invalid statements for that family. Mitigation: strict decoding and fuzzing.
- **Upstream break:** If a proof system or its assumptions break, GLYPH inherits the risk.
- **Hash collision or EVM fault:** A Keccak collision or EVM arithmetic fault could break binding or soundness; assumed infeasible under current Ethereum security assumptions.
- **Encoding drift:** Non-canonical encodings can cause false negatives or unintended statements; mitigated by canonical formats.
- **DA unavailability:** Loss of DA payloads affects auditability only.

Theorem 3 (On-Chain Verifier Soundness)**.** *Under the random oracle model and the proof pack assumptions, if `GLYPHVerifier` accepts calldata, then with overwhelming probability there exists a valid GLYPH-Prover execution that produced the artifact.*

6 Related Work

GLYPH builds on prior work in transparent zero-knowledge proofs.

Sumcheck Protocols. The sumcheck [3] and GKR [1] protocols underpin GLYPH, which adopts a packed arity-8 variant optimized for EVM gas efficiency.

Polynomial Commitments. GLYPH uses BaseFold PCS over binary tower fields for transparent commitments. Unlike KZG [4], only the commitment tag is bound on-chain.

Lookup Arguments. LogUp [2] provides lookup arguments via logarithmic derivatives. GLYPH uses product-tree GKR to avoid inversion-heavy costs in the target field.

STARK Systems. GLYPH's STARK adapter provides a canonical interface to supported STARK receipts, including Ben-Sasson et al. [9] systems, Circle STARKs, and Stwo. GLYPH aggregates supported STARK proofs for gas-efficient settlement.

Pairing-Based SNARKs. Groth16 [7] and PLONK [8] offer efficient verification but require trusted setup. GLYPH wraps these via Groth16 and KZG adapters, preserving compatibility with existing deployments. KZG-based SNARKs inherit the KZG trust assumptions [4].

Folding Schemes. Nova [18] and related IVC schemes enable efficient recursive proof composition. GLYPH's IVC adapter supports Nova, SuperNova, HyperNova, Sangria, and BaseFold PCS formats, verifying transparent PCS openings and transparent R1CS receipts for Nova-family formats. External Nova and SuperNova proofs are verified; HyperNova and Sangria use CompressedSNARK against the canonical receipt. These lines motivate a unified settlement layer.

7 Benchmarks and Gas Costs

This section evaluates GLYPH on two axes: on-chain cost and prover overhead.

7.1 On-Chain Verification Gas

The `GLYPHVerifier` was benchmarked in Foundry on local Anvil and on Sepolia and Hoodi testnets; Appendix B lists tx-hash validation. *Execution gas* is EVM computation only; *Total L1 gas* includes calldata (16 non-zero, 4 zero).[16]

Layout	Calldata Bytes	Total Tx Gas (Anvil)
Artifact-bound packed	224	$\approx$ 29k

Table 2: GLYPHVerifier gas costs (local Anvil bench, packed arity-8 layout).

The packed layout uses 32 bytes per round and a 64-byte artifact-bound header. Total gas is dominated by calldata size and per-round hashing.

7.2 Comparison with Other Systems

System	Setup	Transparent	Total Tx Gas	Calldata
Groth16 (BN254)	Trusted	No	227,128 (3 publics, this work)	356 B
PLONK (KZG)	Trusted	No	$\approx$ 300k–1M	670–900 B
FRI STARK (on-chain)	None	Yes	$\approx$ 1M–6M	50–200 kB
GLYPH (packed)	None	Yes	29,450 (this work)	$64 + 32R$ B (artifact)

Table 3: Comparison of verification systems (R = number of sumcheck rounds).

Note. Groth16 and GLYPH values are measured receipts from this work (Appendix B); PLONK and STARK ranges are external estimates.[33, 34, 35] PLONK proof sizes follow Ethereum research and public proof format documentation; STARK proof sizes reflect reported benchmark sizes.[30, 32, 25, 31] Calldata cost per byte follows EIP-2028, and pairing precompile costs follow EIP-196 and EIP-197 with repricing in EIP-1108.[16, 13, 14, 15] Starknet cost notes indicate multi-million gas per proof train, so the STARK range is representative and batching dependent.[35, 34] For the fixed on-chain parameters used here, the verifier executes in the low thousands of execution gas and total tx gas stays at 29.45k in the recorded Sepolia and Hoodi cases. Total L1 gas depends on calldata pricing and the chosen header variant; exact results are captured by the reproducible benchmark harness and cross-checked on local and testnet environments.

7.3 Prover Performance

The GLYPH-Prover adds minimal overhead; for most adapter families, cost is dominated by upstream verification (e.g., Groth16 proof or STARK receipt). Benchmarks use the repository harness (Foundry and Anvil) and are cross-checked with Sepolia and Hoodi testnet transactions (Appendix B); JSON sidecars record round counts and calldata zero or nonzero breakdowns. Runtime is hardware- and adapter-dependent and is reported via harness outputs rather than fixed statements in this paper.

8 Integration and Adoption

GLYPH is designed for straightforward integration into existing ZK infrastructure.

8.1 Integration Steps

To integrate GLYPH, protocol designers follow these steps:

1. **Select adapter family:** Choose a supported adapter family based on the upstream proof system (Groth16, KZG, IVC, IPA, STARK, or hash).
2. **Prepare upstream proof:** Generate the proof using the existing upstream prover and encode it in the canonical adapter format.
3. **Run GLYPH-Prover:** Invoke the Rust prover via CLI (`glyph_adapt_batch`) or library API to compile the upstream proof into a GLYPH artifact and packed calldata.
4. **Submit on-chain:** Send the packed calldata to `GLYPHVerifier` via a raw call (selectorless fallback).
5. **Handle result:** On success, the fallback returns `1`; on failure, it reverts.

8.2 Tooling

The reference implementation provides CLI binaries (`glyph_adapt_batch`, `glyph_import_*`) and a Rust library API for integrating GLYPH into existing infrastructure. Detailed usage is documented in the repository.

8.3 Optional Data Availability and State Diff Tooling

GLYPH includes optional tooling for data availability and state diff workflows. These utilities are decoupled from the core verifier and do not affect on-chain soundness. They provide operational convenience for teams that want to retain proof payloads or state diffs outside the execution layer.

- **DA strategy:** Optional data availability workflows support blob posting, Arweave storage, and EigenDA dispersal. DA envelopes may contain multiple commitments for the same payload, enabling redundant retrieval and verification. GLYPH is DA-neutral and the envelope interface is provider-agnostic. Availability varies by network and backend. Celestia Blobstream integration is a roadmap item and remains optional.[36]
- **State diff layer:** Snapshot-based diffs can be generated, canonicalized, hashed, and bound to GLYPH artifacts for auditability. The diff can optionally be submitted with proof metadata to anchor a root update. This is a diff layer, not a full state transition system or consensus replacement.

8.4 Deployment Considerations

Single Verifier Model. A single `GLYPHVerifier` deployment can serve multiple protocols. Each protocol's proofs are distinguished by their artifact boundary (`commitment_tag`, `point_tag`), which includes domain-specific hashes. This reduces deployment costs and simplifies auditing.

Migration Path. Existing Groth16-based systems can adopt GLYPH incrementally: wrap Groth16 proofs via the Groth16 adapter for new features while maintaining existing verifiers for backward compatibility. Over time, the ecosystem converges on the transparent GLYPH layer.

9 Limitations and Future Work

9.1 Current Limitations

Upstream Curve Security. GLYPH's on-chain verifier does not depend on elliptic-curve precompiles. Pairing-based upstream proofs (Groth16, KZG, PLONK with BN254) retain their original curve security assumptions, such as the ≈ 100-bit classical security of BN254 [17, 10]. GLYPH preserves these assumptions.

Adapter Completeness. While eight adapter families are defined, only the explicitly supported proof systems and receipt profiles are production-ready. IVC relies on a transparent PCS opening (BaseFold) and transparent R1CS receipts, and verifies external Nova, SuperNova, HyperNova, and Sangria proofs against the canonical receipt. The STARK adapter accepts only specific canonical receipt profiles (field, hash, and commitment combinations); new STARK systems require explicit receipt format support.

State Diff Scope. The state diff tooling is a diff layer for auditing and data binding. It does not implement a full state transition system or data availability guarantees. Users must ensure state updates are derived from a valid execution environment and required availability guarantees are enforced by their own infrastructure.

EVM Gas Schedule and Calldata. GLYPH's on-chain proof verification cost depends on calldata length and the number of sumcheck rounds. Any future repricing of Keccak or calldata costs would affect total gas, even though the verifier logic is fixed.

Sumcheck Determinism. The packed GKR verifier relies on deterministic EVM arithmetic (constant limb width, no overflow exceptions) and exact rounding behaviour of `ADDMOD`/`MULMOD`. Porting to other VMs requires revalidating these assumptions.

Audit Workflow. The audit process documents verifier bytecode and adapter semantics, but it still relies on reproducible tooling to regenerate calldata, proofs, and KPIs. End-to-end reproducibility requires running those tools and verifying outputs against recorded JSON artifacts.

No Post-Quantum Security. GLYPH's core verifier is hash and field based, but upstream proofs may rely on elliptic-curve cryptography. Those proof families are not post-quantum secure. For applications requiring long-term security guarantees, STARK-based systems with hash-based commitments may be preferable as the primary proof layer.

9.2 Future Directions

Additional Adapter Families. Future adapters may support emerging proof systems.

Receipt Standardization. Converge on GLYPH-compatible canonical receipts to reduce adapter effort and speed adoption.

Formal Verification. Formalizing the protocol in a proof assistant and proving end-to-end knowledge soundness for adapter compositions would strengthen security guarantees.

Verifier Formal Audit. Prove key on-chain verifier invariants and minimize the code surface.

Post-Quantum Transition. For long-term security, GLYPH can prioritize hash-based proof families and adopt post-quantum primitives as they mature.

Multi-Chain Settlement. The packed sumcheck verifier needs only a hash function and 128-bit prime-field arithmetic, so it ports to chains with comparable primitives without curve precompiles. A multi-chain deployment lets proof systems settle across supported chains, positioning GLYPH as a universal cross-chain ZK settlement layer.

10 Conclusion

GLYPH introduces a unified approach to zero-knowledge proof verification on Ethereum. Instead of defining yet another proof system, it provides a transparent verification layer that accepts heterogeneous upstream proofs and reduces them to a single compact on-chain verification surface. The contribution is architectural. Upstream verification runs off-chain inside adapters and is bound into a fixed, chain-bound artifact, so the on-chain verifier stays small, deterministic, and gas-efficient while the off-chain pipeline absorbs proof-system complexity. The UCIR compiler, adapter families, and artifact boundary form a stable interface that decouples protocol evolution from on-chain deployment risk. GLYPH is transparent by default and preserves upstream assumptions rather than obscuring them. The verifier is stateless and calldata-only, with strict length and field checks that fail closed on malformed inputs. Testnet receipts show low-thousands execution gas and 29.45k total tx gas on Sepolia and Hoodi, independent of upstream proof size (Appendix B). Trustless validity reduces to UCIR equivalence, sumcheck soundness, and PCS binding for the adapter path. This collapses the trust boundary to the adapter path and its canonical receipts. The artifact boundary binds verification to the chain and contract address, while canonical receipts enforce non-malleable encodings across adapter families. For integrators, GLYPH is a settlement layer, not a new proving DSL. Teams keep existing proof systems, reuse existing verifiers inside the adapter pipeline, and adopt GLYPH as a uniform on-chain endpoint. This lowers operational cost, improves audit stability, and avoids a proliferation of bespoke verifiers because the on-chain verifier is shared and fixed. Upgrades remain off-chain in adapters and receipts, while the on-chain surface stays stable and minimal, concentrating audit scope on a single on-chain surface while allowing proof-system evolution off-chain.

The key innovations are:

- A packed arity-8 sumcheck protocol that keeps the on-chain verifier compact and predictable while decoupling gas cost from upstream proof size.
- A verification transpiler that compiles external verification logic into UCIR while preserving existing toolchains.
- The GLYPH artifact boundary providing a minimal, stable interface between off-chain proving and on-chain verification with explicit chain binding and canonical receipts.
- Optional state-diff binding for auditability and deterministic root updates.
- Complete tooling: a SIMD/CUDA-accelerated Rust prover, canonical receipts, deterministic parsing, and comprehensive testing and fuzzing.

Together, the formal proof pack and Appendix B connect the model to reproducible receipts and on-chain evidence. The spec pack fixes canonical encodings and adapter policies, keeping the evidence aligned with the implementation. Canonical receipts provide a stable, auditable interface across adapters. This keeps auditability and reproducibility tied to the same canonical receipt surface. Fail-closed parsing and canonical encodings ensure malformed inputs are rejected before on-chain verification.

By consolidating verification into a single transparent contract, GLYPH reduces bespoke verifiers, lowers gas costs across applications, and avoids trusted setups for the settlement verifier while preserving upstream trust assumptions. It provides a practical, neutral substrate for transparent verification in Ethereum's modular stack.

A Formal Proof Pack Index and Scope

The formal proof pack for GLYPH is maintained in the repository under `docs/proofs/`; this appendix provides a concise map for reviewers.

- **00__overview.md**: Assumptions A1..A7, adversary model, and dependency graph.
- **01__sumcheck.md**: Sumcheck protocol, soundness bounds, and adversary model.
- **02__gkr__binding.md**: Artifact-tag binding, ROM argument, and unforgeability reductions.
- **03__pcs__basefold.md**: PCS binding and correctness reductions for BaseFold.
- **04__ucir__correctness.md**: UCIR semantics, adapter equivalence, and proof obligations.
- **05__state__diff__binding.md**: State-diff binding and circuit correctness.
- **06__end__to__end.md**: End-to-end soundness composition and explicit error bounds.
- **07__mechanized__proof__plan.md**: Mechanized proof roadmap and theorem list.

The proof pack is parameterized; numerical bounds use a non-normative reference configuration to preserve protocol flexibility.

Theorem Summaries

Sumcheck Soundness. Let f be a degree-d multivariate polynomial over a field F (on-chain $p = 2^{128} - 159$), and r the number of sumcheck rounds. Under ROM, the soundness error is at most $r \cdot d/|F|$ for $d \leq 2$.
GKR Artifact Binding. Let $Tag = H(commitment\_tag || point\_tag)$. Under ROM, collision resistance of Keccak, and GKR soundness, a proof bound to one statement cannot verify against a different statement except with negligible probability.
PCS Binding. Under PCS binding and correctness, a commitment cannot be opened to two distinct values at the same evaluation point except with negligible probability.
UCIR Equivalence. For each adapter family, UCIR satisfiability is equivalent to upstream verifier acceptance under the adapter correctness assumption.
State-Diff Correctness. If the VM executes from old_root to new_root producing diff set D, the state diff circuit outputs $MerkleRoot(D)$ with the same padding and hash rules.
End-to-End Soundness. The total soundness error is bounded by the sum of component errors: $\epsilon_{total} \leq \epsilon_{sumcheck} + \epsilon_{pcs} + \epsilon_{gkr} + \epsilon_{ucir} + \epsilon_{\text{state-diff}} + \epsilon_{hash}$

B Testnet Evidence (Tx Hashes)

Representative testnet transactions for GLYPH artifact verification and Groth16 direct verification are listed below. These hashes correspond to the cases referenced in Section 7 and are cross-checked against the harness outputs. The full set, including Hoodi, is available in the repository.

Network	Case	Tx Hash	Total Tx Gas
Sepolia	GLYPH artifact__full	0x7d37fc1722e1172d968d750db94fda10812536bd26255c40e17c5d3a498acd3b	29,450
Sepolia	Groth16 verify (3 publics)	0x522222efb008569c6a43f612ce3c1ce365138507d6ea7ab4a9b542f19b2de31f	227,128

Table 4: Representative GLYPH and Groth16 verification tx hashes.

C Competitor Comparison

System	Model	Cost (gas)	Notes
GLYPH	Packed GKR sumcheck	≈30k per proof	Transparent; no setup
Aligned Layer	Off-chain verify; BLS + recursive agg	sig ≈113k + agg ≈300k	Groth16, Plonk, SP1, Risc0, Circom
NEBRA UPA	Permissioned agg; untrusted	≈350k/batch (32); ≈18k amortized	UPA v1 Groth16 (batch 32)

Table 5: Comparison with aggregation systems (vendor-reported protocol documentation sources).[33, 38, 37]

References

[1] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating Computation: Interactive Proofs for Muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, 2008.

[2] U. Habock. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Report 2022/1530.

[3] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS)*, 1990.

[4] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology – ASIACRYPT 2010.*

[5] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[6] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology – CRYPTO '86.*

[7] J. Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016.*

[8] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Report 2019/953.

[9] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, Transparent, and Post-Quantum Secure Computational Integrity. Cryptology ePrint Archive, Report 2018/046.

[10] P. S. L. M. Barreto and M. Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. Cryptology ePrint Archive, Report 2005/133.

[11] National Institute of Standards and Technology (NIST). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, 2015.

[12] G. Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper). `https://ethereum.github.io/yellowpaper/paper.pdf`.

[13] Ethereum Improvement Proposal Authors. EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. `https://eips.ethereum.org/EIPS/eip-196`.

[14] Ethereum Improvement Proposal Authors. EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. `https://eips.ethereum.org/EIPS/eip-197`.

[15] Ethereum Improvement Proposal Authors. EIP-1108: Reduce alt_bn128 precompile gas costs. `https://eips.ethereum.org/EIPS/eip-1108`.

[16] Ethereum Improvement Proposal Authors. EIP-2028: Transaction data gas cost reduction. `https://eips.ethereum.org/EIPS/eip-2028`.

[17] R. Barbulescu and S. Duquesne. Updating key size estimations for pairings. Cryptology ePrint Archive, Report 2017/334.

[18] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. Cryptology ePrint Archive, Report 2021/370.

[19] A. Kothapalli and S. Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Report 2022/1758. `https://eprint.iacr.org/2022/1758`.

[20] A. Kothapalli and S. Setty. HyperNova: Recursive arguments for customizable constraint systems. Cryptology ePrint Archive, Report 2023/573. `https://eprint.iacr.org/2023/573`.

[21] N. Mohnblatt. Sangria: a Folding Scheme for PLONK (technical note). `https://github.com/geometryresearch/technical_notes/blob/main/sangria_folding_plonk.pdf`.

[22] H. Zeilberger, B. Chen, and B. Fisch. BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes. Cryptology ePrint Archive, Report 2023/1705. `https://eprint.iacr.org/2023/1705`.

[23] Polygon Labs. Plonky proving system documentation (Plonky2 and Plonky3 overview). `https://docs.polygon.technology/zk/zkEVM/protocol/proving-system/plonky/`.

[24] StarkWare. S-two documentation (Stwo book). `https://docs.starknet.io/learn/S-two-book/why-stwo`.

[25] Meta (Facebook). Winterfell STARK prover implementation (repository). `https://github.com/facebook/winterfell`.

[26] Succinct Labs. SP1 documentation. `https://docs.succinct.xyz/docs/sp1/introduction`.

[27] RISC Zero. Receipts. `https://dev.risczero.com/api/zkvm/receipts`.

[28] Polygon Miden. Miden VM documentation. `https://docs.miden.xyz/miden-vm/`.

[29] E. Ben-Sasson et al. Cairo: a Turing-complete STARK-based virtual machine. Cryptology ePrint Archive, Report 2021/1063. `https://eprint.iacr.org/2021/1063`.

[30] I. Rubin. SLONK (PLONK proof size in bytes). Ethereum Research forum post, 2020. `https://ethresear.ch/t/slonk-plonk-proof-size-in-bytes/8100`.

[31] J. Poon. Distaff VM now with Deep-FRI (proof size benchmarks). Ethereum Research forum post, 2020. `https://ethresear.ch/t/distaff-vm-now-with-deep-fri/7459`.

[32] ZKM. ZK Prover documentation (proof size example for PLONK). `https://docs.zkm.io/zkvm/prover/`.

[33] Aligned Layer. FAQ. `https://docs.alignedlayer.com/introduction/3_faq`.

[34] Aligned Layer. Aligned testnet: connecting 7 protocols to Ethereum in 15 days. May 29, 2024. `https://www.alignedlayer.com/blog/aligned-testnet`.

[35] Starknet Community. Starknet costs and fees: Part 1 - L1 costs. Sep 14, 2022. `https://community.starknet.io/t/starknet-costs-and-fees-part-1-l1-costs/231`.

[36] Celestia. Blobstream documentation. `https://docs.celestia.org/how-to-guides/blobstream`.

[37] NEBRA. Gas costs on L1s. `https://docs.nebra.one/developer-guide/gas-costs-on-l1s`.

[38] NEBRA. UPA protocol specification. `https://docs.nebra.one/upa-protocol-specification`.