GLYPH Formal Proof Pack Appendix

Christopher Schulze

February 4, 2026

This appendix reproduces the formal proof pack in `docs/proofs/` verbatim and accompanies the GLYPH whitepaper dated February 4, 2026.

1 Overview

```
# Formal Proof Pack - Overview

## Purpose
This pack provides a complete internal proof chain for GLYPH with explicit assumptions, lemmas,
and a proof-to-code map. It is designed to withstand scrutiny from cryptography reviewers without
relying on external audits.

## Security Model
We assume:
1. Random Oracle Model for Fiat-Shamir transforms.
2. Collision resistance of Keccak256.
3. Soundness of the GKR protocol.
4. Binding and correctness of the PCS commitment and opening scheme.
5. Correctness of adapter verification logic for each supported receipt format.

## Adversary Model
We consider a polynomial time adversary who:
- Controls all prover inputs, including transcripts and intermediate values.
- Can choose statements adaptively and submit arbitrary proofs.
- Has oracle access to the Fiat-Shamir transcript.
- Is bounded by standard cryptographic security parameters.

We target negligible soundness error in the security parameter lambda, with
explicit epsilon bounds derived in `06_end_to_end.md`.

## Formal Assumptions Registry
We reference the following assumptions in all proofs:
- A1: Fiat Shamir Random Oracle Model. Transcript outputs are indistinguishable from random.
- A2: Keccak256 collision resistance. Finding collisions is infeasible.
- A3: GKR soundness. A prover cannot convince the verifier of a false claim except with negligible
probability.
- A4: PCS binding. Commitment and opening do not allow equivocation.
- A5: PCS correctness. Openings verify iff the committed polynomial evaluates to the claimed value.
- A6: Adapter upstream verifier correctness. Each receipt format verifies the intended statement.
- A7: UCIR/VM correctness. UCIR matches defined gates; VM/Merkle matches hashing and padding rules.
```

We explicitly do not assume any cryptographic property beyond A1..A7. All proofs must reduce to these assumptions or to standard algebraic facts about finite fields.

Security Parameters
- Field size: |F| = 2^128 - 159 (packed verifier field).
- Sumcheck rounds: r = number of packed rounds derived from calldata length.
- Transcript security: modeled as a random oracle with domain separation tags.

All explicit bounds are computed for the above defaults and parameterized for alternative configurations.

Notation
- F: 128-bit prime field used by the packed verifier (p = 2^128 - 159).
- H: Keccak256.
- S: sumcheck claim.
- T: transcript state.
- C: PCS commitment.
- O: PCS opening.
- Tag: artifact tag = H(commitment_tag || point_tag).

Note: Goldilocks appears in off-chain prover and adapter receipts, but the packed on-chain sumcheck uses the 128-bit field above.

Dependency Graph
01_sumcheck depends on A1.
02_gkr_binding depends on A1, A2, A3.
03_pcs_basefold depends on A1, A4, A5.
04_ucir_correctness depends on A6, A7.
05_state_diff_binding depends on A1, A2, A7.
06_end_to_end depends on all A1..A7 and the composition of their bounds.

System Statement
For any accepted proof, the on-chain verifier accepts only if the statement hash and bound artifact tags correspond to a valid execution of the intended verification logic and constraints. This is formalized in `06_end_to_end.md`.

Artifacts
- UCIR encoding and invariants: `docs/specs/ucir_spec.md`
- Adapter IR encoding and kernel routing: `docs/specs/adapter_ir_spec.md`
- Custom gate IDs, gating, and payloads: `docs/specs/custom_gates_spec.md`
- Verifier calldata and memory spec: `docs/specs/verifier_spec.md`
- Artifact tag and chain binding: `docs/specs/artifact_tag_spec.md`
- Canonical STARK receipt and VK encoding: `docs/specs/stark_receipt_spec.md`
- State transition VM spec: `docs/specs/state_transition_vm_spec.md`

Proof Pack Index
1. `01_sumcheck.md`
2. `02_gkr_binding.md`
3. `03_pcs_basefold.md`
4. `04_ucir_correctness.md`
5. `05_state_diff_binding.md`
6. `06_end_to_end.md`
7. `07_mechanized_proof_plan.md`

Mechanized Proof Scope
The written proofs are complete. Mechanized proofs are planned for the core theorems using a proof assistant; the roadmap is documented separately.

Proof-to-Code Map
The detailed mapping is in `06_end_to_end.md` and repeated in each chapter.

2 Sumcheck

```
# Formal Proof Pack - Sumcheck

## Definitions
Let ‘f‘ be the packed GLYPH statement polynomial over the 128-bit prime field ‘F‘
(p = 2^128 - 159), defined on the arity-8 domain ‘{0..7}^r‘.

The packed sumcheck protocol proves a claim ‘S = sum_{x in {0..7}^r} f(x)‘ by reducing it to a
sequence of univariate claims. Here r is the packed round count determined by the statement length
in calldata, and the verifier runs exactly r rounds.

Let F be the 128-bit prime field. Let r be the number of packed rounds.

For round i, the prover constructs a univariate polynomial g_i(t) such that:
g_i(t) = sum_{x in {0..7}^{r-i-1}} f(r_0, ..., r_{i-1}, t, x)
where r_0..r_{i-1} are transcript challenges. This is the partial sum of f over the remaining
coordinates, leaving t as the only free variable.

In GLYPH, each round sends (c0, c1) and recovers c2 by enforcing the arity-8 sum constraint.

The verifier checks a degree-2 polynomial g_i(t) = c0 + c1*t + c2*t^2 over t in {0..7}, and it
rejects immediately if c0 or c1 is non-canonical. Only (c0, c1) are transmitted, so the arity-8
sum check fully determines the quadratic.

## Protocol (Non-Interactive via Fiat-Shamir)
For i = 0..r-1:

1. Prover computes g_i(t) such that: g_i(t) = sum_{x in {0..7}^{r-i-1}} f(r_0, ..., r_{i-1}, t, x)

2. Prover sends coefficients (c0, c1).

3. Verifier enforces the arity-8 sum constraint:
   g_i(0) + g_i(1) + ... + g_i(7) = claim_i,
   recovers c2, and rejects if c0 or c1 is non-canonical.

4. Verifier samples r_i = H(transcript).

5. Updates claim_{i+1} = g_i(r_i).

Final check: claim_r equals f(r_0..r_{r-1}).

## Adversary Model
The prover is any PPT adversary that chooses f and the round polynomials g_i adaptively, with
access to the Fiat-Shamir transcript oracle.

The verifier is deterministic given transcript outputs and treats the transcript as a random
oracle under A1.

## Formal Definitions
Let f be the packed statement polynomial over {0..7}^r. Let claim_0 be the initial claim from the
packed proof header.

Define the partial sum polynomial at round i:
g_i^*(t) = sum_{x in {0..7}^{r-i-1}} f(r_0,...,r_{i-1},t,x)

The prover sends (c0, c1) defining g_i(t). The verifier enforces:
sum_{t=0..7} g_i(t) = claim_i,
recovers c2, and sets claim_{i+1} = g_i(r_i).
```

Lemmas
1. **Round Consistency**: For each round `i`, the verifier checks sum_{t=0..7} g_i(t) = claim_i.
2. **Claim Reduction**: The next claim is `claim_{i+1} = g_i(r_i)` where `r_i` is derived from the transcript.
3. **Degree Bound**: In GLYPH, g_i is degree <= 2, because it is quadratic.
4. **Soundness**: If `f` is not consistent with the claimed sum, acceptance probability is bounded by (deg(g) / |F|) per round.

Theorem 1: Sumcheck Soundness
Assuming A1, if a prover convinces the verifier of an incorrect claim, the probability of acceptance is at most 2r / |F| for r packed rounds over field F.

For standard sumcheck, the bound is r * deg(g) / |F|. In GLYPH's packed arity-8 check, each round uses a quadratic polynomial, so the bound is r * 2 / |F|.

Proof (Formal Sketch with Explicit Bound)
Define the ideal polynomial g_i^*(t) derived from the true f and the challenger prefix r_0..r_{i-1}. The prover sends g_i(t). If g_i != g_i^*, then h(t) = g_i(t) - g_i^*(t) is non-zero with degree <= 2.

The verifier checks sum_{t=0..7} g_i(t) = claim_i. If this passes, then the only way for a cheating prover to continue is for h(r_i) = 0, where r_i is uniformly random in F under A1. By Schwartz Zippel, Pr[h(r_i)=0] <= deg(h)/|F| <= 2/|F|.

By union bound over r rounds, the total soundness error is <= 2r/|F|.

Fully Expanded Proof
If the claim_0 is false, then at least one round i must have g_i != g_i^*. Define h_i = g_i - g_i^*. Since g_i is degree <= 2 by construction, h_i has degree <= 2 and is non-zero.

The verifier accepts round i only if h_i(r_i) = 0. By Schwartz-Zippel:
Pr[h_i(r_i)=0] <= deg(h_i)/|F| <= 2/|F|.
By union bound across r rounds, the total failure probability is <= 2r/|F|.

Formal Bound Statement
Let Adv_sc be the adversary advantage to make the verifier accept an incorrect claim.
Then: Adv_sc <= 2r/|F|.

This is the exact bound used in the end-to-end composition.

Assumptions
- Fiat-Shamir transcript behaves as a random oracle.
- Field size is large enough for the required soundness error.

Quantitative Bound (Default Parameters)
Let |F| = 2^128 - 159. For r packed rounds, the soundness error is:
epsilon_sumcheck <= 2 * r / |F|.
This bound is explicit and composed in `06_end_to_end.md`. For typical r up to 64, the bound is far below 2^-80.

Proof-to-Code Map
- Packed sumcheck rounds and arity-8 constraint: `src/glyph_gkr.rs`
- Packed verifier implementation: `contracts/GLYPHVerifier.sol`

Implementation Invariants
- Round polynomial is quadratic with c2 recovered from the arity-8 sum constraint via `INV140` in `src/glyph_gkr.rs`.
- Claim update uses `g_i(r_i)` with transcript challenges derived from Keccak.
- Each round validates sum_{t=0..7} g_i(t) equals the current claim.

3 GKR Binding

```
# Formal Proof Pack - GKR Binding

## Definitions
The GLYPH artifact contains `(commitment_tag, point_tag, claim128, initial_claim)`. The artifact
tag is `artifact_tag = keccak256(commitment_tag || point_tag)`.

The packed proof binds the artifact tag and claim values into the transcript before the final
sumcheck polynomial is produced.

Let Tag = H(commitment_tag || point_tag). Let the transcript include Tag, claim128, and
initial_claim before challenge sampling. Let Proof be the packed GKR artifact proof.

Let FS be the Fiat-Shamir transcript function that maps absorbed bytes to challenges. The
transcript input stream is fixed and domain-separated.

Statement binding for L2 updates uses the extended statement hash produced by
`statement_hash_extended` in `src/l2_statement.rs`. The exact preimage is:
`L2_STATE_DOMAIN || u256_be(chainid) || contract_addr || old_root || new_root || da_commitment ||
batch_id_be || extra_commitment || extra_schema_id` where `contract_addr` is 20 bytes and
`batch_id_be` is a big-endian u64. The minimal flow uses statement_hash_minimal and omits
`extra_commitment` and `extra_schema_id`.

Tags are derived as:
`commitment_tag = H(L2_COMMIT_DOMAIN || statement_hash)`,
`point_tag = H(L2_POINT_DOMAIN || commitment_tag)`,
`artifact_tag = H(commitment_tag || point_tag)`.

## Lemmas
1. **Artifact Tag Integrity**: If `artifact_tag` does not match `commitment_tag` and `point_tag`,
the verifier rejects.
2. **Statement Binding**: The transcript absorbs the statement-derived tags. A proof generated for
one statement cannot be replayed for another without breaking the hash or GKR soundness.
3. **Non-Malleability**: For any two distinct statements s != s', the derived tags differ except
with negligible probability under A2. Therefore a proof bound to s cannot verify under s'.

## Theorem 2: Artifact Binding
Assuming A1, A2, A3, any proof that verifies on-chain implies that Tag matches the commitment and
point tags, and that the claim was generated under the same transcript.

## Proof (Formal Sketch)
Let s be the statement and (ct, pt) the derived tags. The on-chain verifier checks:
1) artifact_tag == H(ct || pt), and
2) the packed GKR proof is valid under the transcript that absorbed Tag and claim.

If an adversary attempts to use a proof for s' under s, then either:
- H(ct || pt) collides, violating A2, or
- the transcript challenges differ, and by A3 the proof is rejected.

By ROM (A1), challenges are uniformly random conditioned on transcript state, and any deviation
from the correct Tag changes the transcript inputs, making acceptance probability negligible.

## Game-Hopping Outline
Game 0: Real transcript with Tag and claim absorbed.
Game 1: Replace transcript hash with a random oracle (A1).
Game 2: Adversary forges Tag collision to reuse proof (A2).
Game 3: Adversary forges a valid GKR proof under a mismatched transcript (A3).

Each transition is negligible under the corresponding assumption, yielding binding.
```

Formal Reduction Statement
Let Adv_bind be the advantage of producing a proof that verifies under a mismatched statement.
Then: Adv_bind <= Adv_RO + Adv_Hash + Adv_GKR
where:
- Adv_RO is the advantage of distinguishing the transcript from random (A1).
- Adv_Hash is the advantage of finding a Keccak256 collision on Tag (A2).
- Adv_GKR is the advantage of breaking GKR soundness under the transcript (A3).

Domain Separation Justification
All transcript inputs are tagged by domain bytes:
- DOMAIN_SUMCHECK and DOMAIN_SUMCHECK_MIX separate sumcheck stages.
- L2_COMMIT_DOMAIN and L2_POINT_DOMAIN separate statement binding tags.
- Artifact tag derivation is applied before challenge sampling.
Under A1, the transcript behaves as a random oracle per domain. This prevents cross-protocol collisions between sumcheck and GKR stages.

ROM Proof Sketch
We model the transcript as an oracle H. The prover's view is a sequence of oracle queries determined by absorbed tags and claims.
1) Replace H with a uniformly random oracle (A1).
2) Any change in Tag or claim changes the query inputs.
3) Therefore challenges are independent of adversarial choices except with negligible probability.
4) Under these conditions, a false claim reduces to a violation of A2 or A3.

Expanded Reduction
Assume an adversary outputs a proof that verifies under statement s but was constructed for statement s'. If s != s', then the statement hash differs unless Keccak collision occurs. If the statement hash differs, then tags differ.
Case 1: Tag collision. This implies a collision in H(commitment_tag || point_tag), breaking A2.
Case 2: Tag mismatch without collision. The transcript input stream differs at the tag absorption step. Under A1, the resulting challenges are independent of the adversary's prior view. The adversary must produce a valid GKR proof under the wrong transcript, which breaks A3.

Proof Sketch
The verifier checks the tag consistency and the sumcheck chain. Since the transcript is derived from the tags and claim values, a prover cannot change the statement without invalidating the proof.

Assumptions
- Keccak256 is collision resistant.
- GKR soundness holds under Fiat-Shamir.
- Transcript challenges are modeled under ROM.

Proof-to-Code Map
- Artifact construction: `src/glyph_gkr.rs`
- Packed calldata: `src/glyph_gkr.rs`, `src/glyph_core.rs`
- On-chain verification: `contracts/GLYPHVerifier.sol`
- Calldata and memory layout: `docs/specs/verifier_spec.md`
- Artifact tag and chain binding: `docs/specs/artifact_tag_spec.md`
- Extended statement binding tags: `src/l2_statement.rs`

Implementation Invariants
- Tag computed as keccak256 of commitment_tag and point_tag.
- Transcript absorbs Tag and claim in a fixed order.
- On-chain verifier recomputes tag and rejects mismatches.
- Extended statement binding includes chainid and verifier address to prevent cross-domain replay.

4 PCS BaseFold

```
# Formal Proof Pack - PCS Commitment (BaseFold)

## Definitions
The PCS commitment binds an evaluation table to a commitment and opening proof at a verifier
point derived from sumcheck challenges. The evaluation table is the binary-field encoding of
the Goldilocks evaluations used by the packed prover, and the commitment is computed over it.
Let C be the commitment to polynomial P. Let r be the evaluation point derived from the transcript
after absorbing the PCS commitment and any masking data. Let O be the opening proof for P(r).

Binding definition:
A PCS is binding if no PPT adversary can produce (C, r, v, v', O, O') such that v != v' and both
openings verify for the same commitment C and point r, except with negligible probability.

Correctness definition:
For any committed polynomial P and point r, the verifier accepts the opening O if and only if O
proves P(r).

## Lemmas
1. **Commitment Binding**: A commitment binds the evaluation table to a unique polynomial.
2. **Opening Correctness**: The opening proof verifies that the committed polynomial evaluates to
the claimed value at the verifier point.
3. **ZK Mode Hiding**: When ZK mode is enabled, the masking rows and salt prevent leakage of
witness values.
4. **Transcript Binding**: The evaluation point r is derived from sumcheck challenges, which are
transcript-bound under A1.

## Theorem 3: PCS Binding
Assuming A1, A4, A5, a prover cannot open a commitment to two different values at
the same point except with negligible probability.

## Proof (Reduction Sketch)
Suppose an adversary produces two valid openings for the same commitment C and
point r. Then either:
1) It breaks PCS binding (A4), or
2) It breaks PCS correctness (A5), or
3) It exploits a transcript deviation to force r, contradicting A1.
Thus the adversary advantage is bounded by the sum of advantages against A4, A5, and A1.

## Game-Hopping Outline
Game 0: Real PCS commitment and opening with transcript-derived r.
Game 1: Replace transcript hash with random oracle (A1).
Game 2: If the adversary opens C to two values, reduce to A4.
Game 3: If opening verifies but value is incorrect, reduce to A5.

## Formal Reduction Statement
Let Adv_pcs be the adversary advantage to open a commitment to an incorrect value.
Then: Adv_pcs <= Adv_RO + Adv_Bind + Adv_Corr
where:
- Adv_RO bounds adversary influence on r under A1.
- Adv_Bind bounds PCS binding under A4.
- Adv_Corr bounds PCS correctness under A5.

## Expanded Reduction
Assume an adversary outputs (C, r, v, O) such that verify(C, r, v, O) = true but v != P(r).
Then either:
1) C does not bind P to a unique polynomial, breaking A4, or
2) The verification accepts an incorrect value, breaking A5, or
3) The adversary biases r by influencing the transcript beyond A1.

These cases are mutually exclusive by definition of the PCS interface and the transcript binding
logic.
```

```
## Proof Sketch
The prover commits to the evaluation table and opens at a deterministic point r
derived from the transcript challenges. The verifier checks the opening using
BaseFold, and binding prevents a different polynomial from matching the opening
without breaking the PCS binding property. When ZK mode is enabled, the masking
rows and salt are absorbed before r is sampled.

## Assumptions
- Binding and correctness properties of the PCS scheme.
- Fiat-Shamir for deriving evaluation points from transcript challenges.

## Proof-to-Code Map
- PCS commitment and opening: `src/glyph_pcs_basefold.rs`
- Common PCS helpers: `src/pcs_common.rs`
- Transcript binding: `src/glyph_transcript.rs`

## Implementation Invariants
- Evaluation point derived from sumcheck challenges.
- ZK mode uses masking rows and salt for hiding.
- Opening verification returns false and callers treat it as failure.
- BaseFold commit and open are the only commitment interfaces used on the critical path.
```

5 UCIR Correctness

```
# Formal Proof Pack - UCIR Correctness

## Definitions
UCIR is the Universal Constraint IR. The UCIR decoder enforces structure, bounds, and canonical
field elements.

Let D be the UCIR decoder. Let C be the UCIR compiler for a given adapter family. Let E be the
UCIR execution engine (witness evaluation). Let V be the upstream verifier for a receipt.

Define the UCIR semantics as a relation:
UCIR_SAT(ucir, w) == all gates evaluate to zero over the witness w, with the
witness layout constraints satisfied.

Define adapter correctness as:
ADAPTER_OK(receipt) == V(receipt) == true.

Gate semantics:
- Arithmetic gate enforces: q_mul*a*b + q_l*a + q_r*b + q_o*c + q_c = 0.
- Copy gate enforces: left == right.
- Lookup gate enforces: witness value is in the specified table.
- Custom gates encode adapter specific verifier relations.

## Lemmas
1. **Decoder Safety**: `Ucir2::from_bytes` rejects malformed encodings and out-of-range values.
2. **Compiler Correctness**: Each adapter compiler produces UCIR that faithfully represents the
upstream verification logic for that receipt family.
3. **Execution Soundness**: UCIR execution enforces all constraints; any mismatch between public
inputs and adapter proof logic yields a non-zero constraint and cannot be satisfied.
4. **Gate Semantics Preservation**: Each gate type in UCIR implements its algebraic meaning
exactly as specified in `docs/specs/ucir_spec.md`.

## Theorem 4: UCIR Adapter Equivalence
Assuming A6 and A7, if a receipt verifies under the upstream verifier V, then the UCIR produced by
compiler C is satisfiable under E. If V rejects, the UCIR constraints are unsatisfiable.
```

Proof (Formal Sketch)
For a given adapter family F, the compiler C_F emits UCIR constraints encoding the verifier equations of V_F. By A6, V_F is correct, so any accepting receipt satisfies those equations.
By Lemma 4, each UCIR gate enforces the exact algebraic semantics.
Therefore there exists a witness w derived from the receipt such that UCIR_SAT(ucir, w) holds.

Conversely, if V_F rejects, then the verifier equations are unsatisfied, and any UCIR witness must violate at least one gate, so UCIR_SAT is false. This establishes equivalence.

Adapter-specific obligations:
- Groth16 and PLONK adapters: pairing and transcript equations must be mapped to UCIR gates.
- STARK adapters: FRI and constraint evaluation equations must be mapped to UCIR gates.
- IVC and Binius adapters: recursive verification relations must be mapped to UCIR gates.
- BLS12-381 SNARK/KZG adapters: commitment and pairing checks must be mapped to UCIR gates.

Proof Obligations Checklist
For each adapter family F:
1) Receipt parsing yields the same public inputs as upstream.
2) UCIR emission matches verifier algebraic checks.
3) Witness layout respects UCIR invariants.
4) All custom gate payloads are validated and length checked.

Expanded Equivalence Argument
For each adapter family F, define predicate R_F where R_F(receipt, public_inputs) holds iff upstream V_F accepts.
The compiler C_F emits UCIR such that: UCIR_SAT(ucir, w) <-> R_F(receipt, public_inputs).

This equivalence is established by:
1) Parsing correctness: receipt bytes map to the same public inputs as upstream.
2) Gate correctness: each UCIR gate enforces the same algebraic constraint.
3) Witness completeness: the witness produced by the adapter contains all intermediate values required by the constraints. Therefore UCIR_SAT holds iff the upstream verifier relation holds.

Assumptions
- Upstream verifier logic is correct for its receipt format. UCIR execution is faithful to the defined gate semantics.

Proof-to-Code Map
- UCIR encoding and invariants: ‘docs/specs/ucir_spec.md‘, ‘src/glyph_ir.rs‘
- Adapter IR byte encoding and routing: ‘docs/specs/adapter_ir_spec.md‘, ‘src/adapter_ir.rs‘
- Custom gate IDs, gating, and payloads: ‘docs/specs/custom_gates_spec.md‘, ‘src/glyph_ir.rs‘
- Canonical STARK receipt encoding: ‘docs/specs/stark_receipt_spec.md‘, ‘src/stark_receipt.rs‘
- Adapter compilation: ‘src/glyph_ir_compiler.rs‘
- UCIR execution: ‘src/glyph_witness.rs‘
- Equivalence tests: ‘scripts/tests/rust/ucir_compiler_equivalence.rs‘

Implementation Invariants
- UCIR decoder rejects trailing bytes and non canonical field elements. Custom gate payload sizes are validated. Witness evaluation returns zero for all constraints on valid receipts.

6 State Diff Binding

```
# Formal Proof Pack - State Diff Binding

## Definitions
State diffs are serialized into bytes and committed via a Merkle root. The root is bound into the
statement hash as an extra commitment in the extended binding flow. This extra commitment is
carried as ‘extra_commitment‘ in the statement and is included in the chain binding.

Let D be the ordered diff list. Let R = MerkleRoot(D). Let H be Keccak256. Let S be the statement
hash computed by the L2 statement logic. Let Tag be the artifact tag derived from S.

State model: A state is a mapping from keys to values, with keys for account nonces, balances,
code hashes, and storage slots. Keys and values are encoded as 32-byte words. A transaction batch
induces a transition from old_root to new_root with a diff set D that records all changed leaves.

Diff extraction function (VM update flow):
1) Produce a deterministic ordered list of updates during VM execution.
2) For each update, append ‘key || old_value || new_value‘ (each 32 bytes) to the byte stream.
3) Chunk the byte stream into 32-byte leaves, padding the final chunk with zeros.
4) If the byte stream is empty, use a single zero leaf.
5) Pad the leaf list with zero leaves to the next power of two.
The ordering is the VM update order and is deterministic for a fixed execution trace.

Diff extraction function (JSON state diff flow):
1) Build the diff JSON as emitted by ‘glyph_state_diff‘, with top-level ‘"version": 1‘
   and an ‘accounts‘ array ordered by address. Each account’s ‘storage‘ array is ordered by slot.
2) Canonicalize by sorting all object keys lexicographically. Array order is preserved.
3) Serialize the canonical JSON to bytes.
4) Apply the same 32-byte chunking and padding rules as above.

Merkle root definition:
- Let L_0 be the list of leaves (padded to power of two).
- For each level k, define L_{k+1}[i] = H(L_k[2i] || L_k[2i+1]).
- The root R is the single element of the final level.

## Lemmas
1. **Diff Root Determinism**: The diff root is deterministic given the ordered byte stream and
padding rules.
2. **Statement Binding**: The extended statement hash binds the diff root into the transcript.
3. **Replay Resistance**: A proof bound to one diff root cannot be reused for another.
4. **Diff Extraction Correctness**: The diff extraction produces a unique byte stream for the
transition from old_root to new_root: VM updates or canonical JSON diff.

## Theorem 5: State Diff Binding
Assuming A1 and A2, if a proof verifies under the extended statement hash, then the diff
root R is the one provided to the prover, and cannot be swapped without invalidating the proof.

## Theorem 5b: State Diff Circuit Correctness
Assuming A7, if the VM executes a batch from old_root to new_root and produces diff list D, then
the circuit output state_diff_root equals MerkleRoot(D) as defined by ‘src/state_diff_merkle.rs‘.

## Proof (Formal Sketch)
1. Diff extraction is deterministic by construction. VM updates use execution order, while JSON
diffs are canonicalized with sorted object keys and deterministic account and storage ordering.
Both are encoded into 32-byte leaves with zero padding to the next power of two.
2. The Merkle root function in the circuit matches the Rust implementation on each level,
including padding and hashing rules.
3. Therefore the computed root in the circuit matches MerkleRoot(D), and the bound statement
ensures the proof ties to this root.
```

Expanded Correctness Argument
Let exec(old_root, txs) -> (new_root, updates) be the VM execution. The update order is deterministic for a fixed execution trace. The byte stream is formed by concatenating ‘key || old_value || new_value‘ for each update. The leaf encoding is fixed-length (32-byte chunks) and deterministic. The Merkle tree construction uses zero padding to the next power of two and Keccak hashing on each internal node.

The circuit implements the same leaf encoding and the same Merkle reduction on the produced byte stream. By induction on tree depth, each internal node in the circuit matches the corresponding internal node in the Rust implementation. Therefore the final root matches MerkleRoot(D).

Proof Sketch
The prover computes the diff root and includes it as ‘extra_commitment‘. The statement hash and artifact tags include this value, and the schema id is hashed alongside to prevent cross-schema replays. For state diff flow, ‘extra_schema_id = keccak256("GLYPH_STATE_DIFF_MERKLE_V1")‘. Any modification to the diff root changes the artifact tag, causing verification to fail.

Assumptions
- Fiat-Shamir transcript behaves as a random oracle (A1).
- Keccak256 collision resistance (A2).
- Correctness of statement hash derivation.

Proof-to-Code Map
- Diff root computation: ‘src/state_diff_merkle.rs‘
- Statement hash derivation: ‘src/bin/glyph_l2_statement.rs‘
- Extended binding verifier: ‘contracts/GLYPHRootUpdaterExtended.sol‘
- State transition VM: ‘src/state_transition_vm.rs‘

Implementation Invariants
- Merkle root uses zero padding to power of two.
- Statement hash includes extra_commitment and schema id.
- Verifier checks artifact tag derived from the statement hash.
- Diff list ordering and leaf encoding are canonical and deterministic.

7 End to End Soundness

Formal Proof Pack - End to End Soundness

Theorem
If the prover generates a GLYPH proof accepted by ‘GLYPHVerifier.sol‘, then the statement encoded by ‘(commitment_tag, point_tag, claim128)‘ corresponds to a valid execution of the intended verification logic and bound inputs, except with negligible probability under the stated assumptions.

Formally, for any accepted proof P and statement S, under A1..A7, there exists a witness W such that UCIR constraints hold and the bound commitments correspond to the same statement S. The on-chain header also checks claim128 and initial_claim are canonical field elements and binds them into the chain hash.

Proof Outline
1. By ‘01_sumcheck.md‘, the sumcheck chain is sound under Fiat-Shamir.
2. By ‘03_pcs_basefold.md‘, the PCS opening binds the evaluation table to the derived point.
3. By ‘02_gkr_binding.md‘, the artifact tag is bound into the transcript and cannot be altered.
4. By ‘04_ucir_correctness.md‘, adapter compilers and UCIR execution correctly encode upstream verification logic.
5. By ‘05_state_diff_binding.md‘, the state diff root is bound into the statement hash and artifact tags. Therefore, any accepted proof implies a valid statement consistent with the bound inputs.

Expanded Proof
Assume a proof is accepted on-chain. The verifier recomputes the artifact tag
from commitment_tag and point_tag and verifies the packed GKR proof. By Theorem
2, the tag and transcript binding are correct, so the statement hash is fixed.
By Theorem 1, the sumcheck chain is sound for the claimed evaluation.
By Theorem 3, PCS openings bind the evaluation table to the transcript-derived
point. By Theorem 4, UCIR satisfiability is equivalent to upstream verification.
By Theorem 5, the state diff root bound into the statement corresponds to the
actual VM transition. Therefore the accepted proof implies a valid statement.
Transcript domains are fixed, so all challenges and tags are domain separated.

Assumptions Registry
- Random Oracle Model for Fiat-Shamir.
- Collision resistance of Keccak256.
- Soundness of GKR and PCS binding.
- Correctness of adapter verification logic for each supported receipt format.
- UCIR and state-diff execution correctness.

Soundness Error
Let epsilon_sumcheck be the sumcheck soundness error. Let epsilon_gkr be the GKR
soundness error. Let epsilon_pcs be the PCS binding error. The total soundness
error is bounded by:
epsilon_total <= epsilon_sumcheck + epsilon_gkr + epsilon_pcs + epsilon_hash
where epsilon_hash is negligible under A2.

Explicit Error Composition
Define failure events:
- E_sumcheck: sumcheck accepts with a false claim.
- E_pcs: PCS opening verifies for an incorrect evaluation.
- E_gkr: GKR artifact proof verifies for a false transcript claim.
- E_ucir: UCIR constraints accept a receipt that upstream verifier rejects.
- E_state_diff: state_diff_root is bound but does not correspond to the real transition.
- E_hash: a collision in Keccak256 breaks binding.

Then:
Pr[accepts false statement] <= Pr[E_sumcheck] + Pr[E_pcs] + Pr[E_gkr] + Pr[E_ucir] +
Pr[E_state_diff] + Pr[E_hash]

Under A1..A7, each event is negligible and the sum is negligible.

Assumption Mapping Table
Failure event	Bound term	Depends on
E_sumcheck	2r/|F|	A1
E_pcs	Adv_PCS_Binding + Adv_PCS_Corr	A1, A4, A5
E_gkr	Adv_GKR + Adv_RO + Adv_Hash	A1, A2, A3
E_ucir	Adv_Adapter	A6, A7
E_state_diff	Adv_Diff_Correctness	A7
E_hash	Adv_Hash	A2

Quantitative Bounds (Default Parameters)
Let |F| = 2^128 - 159 and r be the number of packed rounds.
- epsilon_sumcheck <= 2r / |F|
- epsilon_pcs <= Adv_PCS_Binding
- epsilon_gkr <= Adv_GKR + Adv_RO + Adv_Hash
- epsilon_ucir <= Adv_Adapter
- epsilon_state_diff <= Adv_Diff_Correctness
- epsilon_hash negligible under A2

For default configs, epsilon_sumcheck is dominated by 2r/|F| and is far below 2^-80 for typical r.
The remaining terms are cryptographic assumptions. Example numeric bounds:
- r = 32: epsilon_sumcheck <= 1.88079096131566e-37
- r = 64: epsilon_sumcheck <= 3.76158192263132e-37

Reproducible Calculation Method
To reproduce epsilon_sumcheck: 1) Set |F| = 2^128 - 159. 2) Set r from calldata
length. 3) Compute epsilon_sumcheck = 2r / |F|.

No other terms have concrete numeric values without selecting explicit
cryptographic security parameters for A1..A7.

Reference Configuration (Non-Normative)
To provide a concrete numeric bound without constraining the protocol, we define
one reference instance for reporting:
- r_ref = 32 (reporting baseline).
- N_ref = 8^r_ref.
- epsilon_sumcheck_ref = 2 * r_ref / |F|.

This reference is used only for reporting. The protocol remains parameterized
by N, and the formal bound stays `2 * r / |F|`.

Default Numeric Instantiation (Reference)
To instantiate concrete epsilon values without constraining the protocol, fix
the reference configuration r_ref = 32 and assume 128-bit security for the
cryptographic terms (Keccak256, PCS binding, and GKR transcript binding). Under
A6 and A7, epsilon_ucir and epsilon_state_diff are zero.

- epsilon_sumcheck_ref = 64 / (2^128 - 159) = 1.88079096131566e-37
- epsilon_crypto_ref <= 3 * 2^-128 = 8.81620763116716e-39
- epsilon_total_ref <= 1.96895303762733e-37

For r_ref = 64, epsilon_total_ref <= 3.84974399894299e-37 under the same assumptions.

Parameter Table (Default)
Parameter	Symbol	Default
Field size	|F|	2^128 - 159
Sumcheck rounds	r	number of packed rounds
GKR rounds	g	derived from sumcheck challenges
Transcript hash	H	Keccak256
Statement hash	S	Keccak256

Proof-to-Code Trace
The following invariants are enforced at the code level:
- Transcript absorption order in `src/glyph_transcript.rs`.
- Tag computation in `src/glyph_gkr.rs` and `contracts/GLYPHVerifier.sol`.
- Statement binding in `src/l2_statement.rs` and `contracts/GLYPHRootUpdaterExtended.sol`.

Proof-to-Code Map
- Sumcheck: `src/glyph_core/sumcheck.rs`, `src/glyph_core.rs`, `src/glyph_gkr.rs`
- PCS: `src/glyph_pcs_basefold.rs`, `src/pcs_common.rs`
- GKR artifact: `src/glyph_gkr.rs`, `contracts/GLYPHVerifier.sol`
- UCIR: `src/glyph_ir.rs`, `src/glyph_ir_compiler.rs`, `src/glyph_witness.rs`
- State diff binding: `src/state_diff_merkle.rs`, `src/bin/glyph_l2_statement.rs`
- Spec boundaries: `docs/specs/verifier_spec.md`, `docs/specs/artifact_tag_spec.md`,
`docs/specs/ucir_spec.md`, `docs/specs/adapter_ir_spec.md`, `docs/specs/custom_gates_spec.md`,
`docs/specs/stark_receipt_spec.md`, `docs/specs/state_transition_vm_spec.md`

Implementation Invariants
- Transcript order is fixed for all absorbed domains.
- Public inputs are committed prior to PCS opening.
- Artifact tags are recomputed on-chain.

8 Mechanized Proof Plan

```
# Formal Proof Pack - Mechanized Proof Plan

## Purpose
Provide a machine-checkable proof roadmap without constraining the protocol
implementation. This plan fixes theorem statements, dependencies, and the proof
structure required by a proof assistant.

## Target Assistants
Lean4 primary, Coq fallback.

## Proof Structure
Files: `sumcheck.lean`, `pcs_basefold.lean`, `gkr_binding.lean`,
`ucir_semantics.lean`, `state_diff_correctness.lean`, `end_to_end.lean`.

## Formal Statements (Lean-style sketches)
Sumcheck:
```
theorem sumcheck_soundness (f : F^n -> F) (S : F) :
  Pr[VerifierAcceptsFalse] <= (2 * r) / |F|
```
PCS Binding:
```
theorem pcs_binding (C : Commitment) (r : Point) :
  Adv_open_two_values <= Adv_RO + Adv_Bind + Adv_Corr
```
GKR Binding:
```
theorem gkr_tag_binding (Tag : Hash) (Proof : PackedProof) :
  Adv_mismatch <= Adv_RO + Adv_Hash + Adv_GKR
```
UCIR Equivalence:
```
theorem ucir_equivalence (receipt : Receipt) (ucir : UCIR) :
  VerifierOK(receipt) <-> UCIR_SAT(ucir)
```
State Diff Correctness:
```
theorem state_diff_root_correct (old_root new_root : Hash) :
  VM_exec -> circuit_root = MerkleRoot(diff_set)
```
End-to-End Composition:
```
theorem end_to_end_soundness :
  Adv_total <= sum(Adv_sumcheck, Adv_pcs, Adv_gkr, Adv_ucir, Adv_state_diff, Adv_hash)
```

## Proof Dependencies
A1..A7 from `docs/proofs/00_overview.md`, UCIR gate semantics from
`docs/specs/ucir_spec.md`, state diff rules from
`docs/specs/state_transition_vm_spec.md`, transcript domains from
`src/glyph_transcript.rs`.

## Acceptance Criteria
- Each theorem is encoded with explicit parameters and bound terms.
- Proof assistant builds without external edits to protocol code.
- Each lemma is mapped to the corresponding proof section in `docs/proofs/`.
```