ZUL PROTOCOL · TECHNICAL WHITEPAPER

Zul

A Privacy Layer 2 for the Solana Virtual Machine — real SVM execution, native settlement, and a zero-knowledge shielded pool, on the road to privacy-wrapping the entire SVM.

SVM · Groth16/BN254 · Poseidon · blake3 SMT · optimistic settlement

VERSION	DATE	STATUS
1.0 — Mainnet edition	June 2026	Live on Solana mainnet-beta

ABSTRACT

Public blockchains are accidental surveillance systems. Every balance, counterparty, and amount is published to everyone, forever — not because openness requires it, but because the dominant designs equate *verifiability* with *visibility*. **Zul** is a Layer 2 for the Solana Virtual Machine (SVM) built to separate the two: it executes ordinary Solana transactions, commits its state to a succinct cryptographic root, settles to Solana L1, and carries a native zero-knowledge shielded pool in which value can be held, transferred, and withdrawn with amounts and the transaction graph hidden behind proofs rather than merely left unindexed.

Zul runs unmodified SVM execution by embedding Anza's `solana-svm` processor over a custom account store, so existing programs, wallets, and the Solana JSON-RPC interface work without change. State is committed through a depth-256 blake3 sparse Merkle tree whose root anchors every block and backs withdrawal proofs verified on L1. The shielded pool is *enshrined* in the node — executed outside the SVM's compute-unit budget — using Poseidon note commitments, a two-level commitment that makes deposits proofless yet sound, an incremental commitment tree, a persistent nullifier set, and Groth16 proofs over the BN254 curve verified natively with arkworks. A bidirectional bridge moves SOL and arbitrary SPL tokens between L1 and L2, binding each withdrawal to the asset it may release.

We give the full construction — execution, state commitment, settlement, bridge, and the shielded circuits — together with security definitions and proof sketches for balance integrity, double-spend resistance, and transaction unlinkability, reduced to the knowledge-soundness of Groth16, the collision resistance of Poseidon and blake3, and the hardness of discrete logarithms on BN254. We are equally explicit about what is *not* yet trustless: today a single honest-but-accountable sequencer orders blocks and fraud proofs are procedural rather than enforced. Finally, we describe the project's north star — generalizing the one enshrined shielded circuit into a programmable privacy layer, and ultimately a zero-knowledge virtual machine for the SVM itself — and argue that the near rung is the ladder to the far one. Zul is live on Solana mainnet-beta.

Curve BN254 · **Proof system** Groth16 · **Hash (in-circuit)** Poseidon · **Hash (state)** blake3 · **State tree** SMT depth 256 · **Note tree** depth 26 · **Withdrawals tree** depth 32 · **Settlement** optimistic · **Status** mainnet-beta

Contents

01 — FOUNDATIONS

Introduction

Most of what a public blockchain learns about you, it tells everyone, forever. Zul inverts that default.

Public ledgers achieved something genuinely new: agreement without an authority. Anyone can verify that a transaction is valid, that no coin was spent twice, that the rules were followed — and they can do it without trusting a bank, a government, or each other. This is the great result of Bitcoin [3] and its successors, the programmable chains [4]. But the dominant designs purchased that verifiability with a second, unadvertised property: total transparency. Every balance, every counterparty, every amount, and the exact order in which you acted are written into a permanent public record. Surveillance became the default and privacy the exception — not because the mathematics demanded it, but because it was the easiest way to let everyone check everyone else's work.

It did not have to be this way, and it does not have to stay this way. The case for financial privacy is older than the blockchain that erased it:

> *Privacy is necessary for an open society in the electronic age. … We cannot expect governments, corporations, or other large, faceless organizations to grant us privacy out of their beneficence.*
>
> — A Cypherpunk's Manifesto, 1993 [2]

Thirty years on, that sentence is still mostly unbuilt. The tools we have for financial privacy on public ledgers are narrow and, almost without exception, take the shape of a single program: a mixer, a shielded coin, a confidential-balance token [7][8][17]. These are boxes you put value into and take value out of. They are useful, and they are far too small. Privacy should not be one application you visit; it should be a property the execution environment itself can offer.

1.1 What Zul is

Zul is a Solana Layer 2 that runs real SVM execution, produces its own blocks, and settles to Solana. It embeds Anza's `solana-svm` transaction processor [6] — the same execution pipeline a Solana validator uses — over a custom account store, so standard programs (System, the SPL Token and Token-2022 programs, the Associated-Token, Memo, and Compute-Budget programs, and user-deployed BPF) run unchanged, and the Solana JSON-RPC surface lets existing wallets and the `solana` CLI talk to the L2 with no modification. It has a native gas token, **ZUL**, and a working zero-knowledge shielded pool in which a user can hold value, send it to someone, and withdraw it, with the amounts and the links between parties hidden behind proofs. That part is not a roadmap item. It runs on mainnet today.

Zul is also honest about what it is not yet. A single sequencer orders the blocks; we do not dress that up. A privacy project that misrepresents its own trust model has already failed at the one thing it sells. Throughout this document we state precisely how much trust each component requires, and we design toward needing less of it, not more (§3, §13).

1.2 Why “a property, not an app”

Privacy, almost everywhere it exists on-chain, is a single hardcoded program. Zul ships one of those too — an enshrined shielded pool — and it is good, and it is not the point. The point is that privacy should be something a program *opts into*, the way it opts into any other property: a sealed-bid auction where the bids stay sealed; a vote whose ballots stay secret while the tally stays provable; an order book that does not leak your position the instant you take it. None of these should require a new chain. They should be programs running in an environment where *no one can see the inputs and everyone can verify the result*. Zul's enshrined pool is the first, fully working instance of that environment; the architecture is built so the same settlement, proof-verification, and hidden-state machinery generalizes to the rest (§12).

1.3 Contributions

This paper documents a complete, deployed system. Its contributions are as much in integration as in any single primitive:

CONTRIBUTIONS

1. **A genuine SVM rollup.** A from-scratch Rust node that drives Anza's `solana-svm` over a redb-backed account store, reproducing enough of a validator's runtime — sysvars, builtins, fee handling, blockhash queue — that unmodified Solana tooling treats the L2 as a Solana cluster (§4).
2. **A dual state-commitment design.** A depth-256 blake3 sparse Merkle tree commits all account state into each block's root, paired with a deliberately shallow depth-32 withdrawals tree whose inclusion proofs fit inside Solana's per-transaction compute budget on L1 (§5, §6).
3. **An enshrined, multi-asset shielded pool.** A UTXO shielded pool processed *outside* the SVM and therefore free of any compute-unit ceiling on proof verification, using Poseidon commitments, a two-level commitment that makes deposits proofless yet sound, and Groth16/BN254 proofs verified natively in-node (§8, §9).
4. **A composable shielded ↔ transparent boundary.** A bidirectional bridge for SOL and arbitrary SPL tokens whose withdrawals bind an asset identifier, so a private balance can become a public one — and vice versa — without the boundary becoming a leak or a theft vector (§7).
5. **Security analysis with explicit trust accounting.** Definitions and proof sketches for balance integrity, double-spend resistance, and unlinkability, together with a frank threat model that names every property not yet enforced in cryptography (§10).
6. **A staged path to a private SVM.** A concrete plan to generalize the single enshrined circuit into a programmable privacy layer, and ultimately a zk-VM for the SVM, reusing the very components this paper describes (§12).

1.4 Roadmap of this document

Section 2 situates Zul among rollups and privacy systems. Section 3 gives the architecture and the trust model. Sections 4–7 build the transparent machine: execution, state commitment, settlement and data availability, and the bridge. Sections 8–9 develop the shielded pool and its circuits in full. Section 10 states and argues the security properties; Section 11 reports the implementation and its measurements. Sections 12–13 set out the north star and the trust-minimization roadmap. Appendices collect the protocol's exact parameters, constants, and interfaces.

02 — CONTEXT

Background & Related Work

What Zul borrows, what it rejects, and where it sits among rollups and privacy systems.

2.1 Solana and the SVM

Solana [5] couples a high-throughput account model with Proof of History, a verifiable delay sequence used to order events before consensus. Its execution layer — the Solana Virtual Machine — runs programs compiled to a Berkeley Packet Filter (BPF/SBF) bytecode against explicitly declared accounts, which permits parallel execution of non-conflicting transactions. Anza's `solana-svm` crate [6] exposes this execution layer as a reusable component: a `TransactionBatchProcessor` that, given a callback supplying account state, loads programs, charges fees, and executes transactions exactly as a validator would. Zul builds directly on this API (§4), which is what lets the L2 inherit Solana's program ecosystem rather than reimplement it. A custom virtual machine would not run a single existing Solana program; embedding the real one runs all of them.

2.2 Rollups and state commitments

A rollup executes transactions off the base layer and publishes enough information for the base layer to hold it accountable. *Optimistic* rollups [35] post state roots and assume them valid unless challenged within a window by a fraud proof; *validity* rollups post a succinct proof of correct execution with every batch. Both require two things the SVM does not provide natively: a commitment to the entire state (Solana validators agree on account state without a global state trie), and a public data-availability path so anyone can reconstruct that state [36]. Zul supplies the first with a sparse Merkle tree [31][32] over all accounts (§5) and the second by compressing batch data onto the Solana ledger (§6). Its settlement is optimistic and, candidly, Stage 0: the data to challenge an invalid root is public, but on-chain re-execution is not yet enforced (§10).

2.3 Privacy on public ledgers

Shielded pools. The lineage runs back to Chaum's untraceable digital cash [1]; on public ledgers, Zerocash [7] and its production realization in Zcash's Sapling [8] established the template Zul's pool follows: value lives in *notes* whose commitments accumulate in a Merkle tree; spending reveals a *nullifier* that prevents double-spending without revealing which note was spent; and a zero-knowledge proof attests that a spend is well-formed and value-preserving. Zul adopts this UTXO/commitment–nullifier structure, swaps Sapling's curve and proof system for Groth16 over BN254 [9][13], and uses the Poseidon hash [14] throughout so that commitments, nullifiers, and Merkle nodes are cheap to prove in-circuit.

Mixers. Tornado Cash [17] popularized fixed-denomination mixing on a smart-contract chain; its Nova variant [18] added arbitrary amounts and shielded internal transfers via a two-input/two-output join-split. Zul's transfer circuit is the same join-split shape (§9), including the dummy-input handling that lets a single circuit serve one-, two-, and (via change) value-splitting spends.

Confidential balances. The Token-2022 confidential-transfer extension [33] encrypts amounts while leaving the participant graph public. This hides *how much* but not *who*. Zul hides both: inside a private transfer, neither the amounts, the asset, nor the link between input and output notes is revealed.

Private computation. Beyond value transfer, Zexe [25] and its realization in Aleo, Aztec's PLONK-based private rollup [24], and Penumbra's shielded application layer [26] pursue general private state transitions. These systems are the closest kin to Zul's north star (§12): privacy as a programmable property rather than a single coin. On Solana specifically, Light Protocol [27] and the Elusiv→Arcium lineage [28] attacked adjacent problems — ZK compression and MPC-based confidentiality — and both narrowed away from the maximalist "make the whole VM private" goal. Zul's wager is that the way to reach that goal is to ship the enshrined pool first and generalize it, so every rung stands on a working artifact.

2.4 Proof systems and arithmetization-friendly hashes

Zul uses Groth16 [9], the pairing-based SNARK — descended from Pinocchio [10] — with the smallest proofs (three group elements) and cheapest verification known, at the cost of a per-circuit trusted setup. We accept that cost because verification is the hot path: every shielded transaction is verified by the node, and Groth16's constant-size, constant-time check is ideal for an enshrined verifier. Transparent alternatives — PLONK [11] with a universal setup, or STARKs [12] with none, both made non-interactive by the Fiat–Shamir transform [34] — remain on the table for the programmable layer, where a universal setup is worth more than minimal proof size. In-circuit, Zul hashes with Poseidon [14], a successor to MiMC [15] and a sponge over the proof field designed so that one hash costs a handful of constraints instead of the tens of thousands a bit-oriented hash like SHA-256 would impose; for state commitments outside any circuit, it uses blake3 [16], which is fast in native code and exposed as a Solana syscall so the L1 settlement program can recompute the same roots.

Table 1. Where Zul sits among on-chain privacy designs. "Graph" = hides the link between sender and receiver; "Amounts" = hides values; "General" = privacy for arbitrary program state, not only value.

System	Form	Amounts	Graph	General	Proof system	Setting
Zcash (Sapling) [8]	Shielded pool	Yes	Yes	No	Groth16 / BLS12-381	Own L1
Tornado Nova [18]	Pool / mixer	Yes	Yes	No	Groth16 / BN254	EVM contract
Token-2022 CT [33]	Confidential balance	Yes	No	No	Sigma / Bulletproofs	Solana L1
Aztec [24]	Private rollup	Yes	Yes	Yes	PLONK / BN254	Ethereum L2
Aleo / Zexe [25]	Private execution	Yes	Yes	Yes	Marlin / varuna	Own L1
Zul	**Enshrined pool → programmable**	**Yes**	**Yes**	**Roadmap**	**Groth16 / BN254**	**Solana L2**

Zul's distinguishing choice is the combination: a Zcash-style shielded pool, verified *natively* rather than inside a metered smart contract, enshrined in a real SVM L2, and architected from the start so

that the pool is the first instance of a general mechanism rather than the whole product.

03 — SYSTEM

Architecture & Trust Model

Three machines in one node: a transparent SVM, a settlement boundary to Solana, and an enshrined shielded pool — and an honest account of what each requires you to trust.

Zul is a single Rust process — the *node* — that plays the role a whole validator network plays on an L1, plus the role of a rollup's prover/poster. It is useful to read it as three cooperating machines that share one account store and one block stream:

- a **transparent execution machine** that runs ordinary SVM transactions and commits the resulting state (§4, §5);
- a **settlement boundary** that anchors state roots and data to Solana and moves assets across the L1↔L2 seam (§6, §7);
- an **enshrined shielded pool** that processes private value transfer outside the SVM, with native proof verification (§8).

The composition is the design. The shielded pool is not a smart contract bolted onto the chain; it is a builtin the node executes directly, so it pays no compute-unit tax for pairing-heavy verification and can keep its own commitment tree and nullifier set as first-class state. The bridge is not an afterthought; it is the seam that lets value cross between the transparent and shielded worlds, and getting that seam right — binding each withdrawal to exactly the asset it may release — is what keeps composition from becoming a vulnerability.

3.1 Components

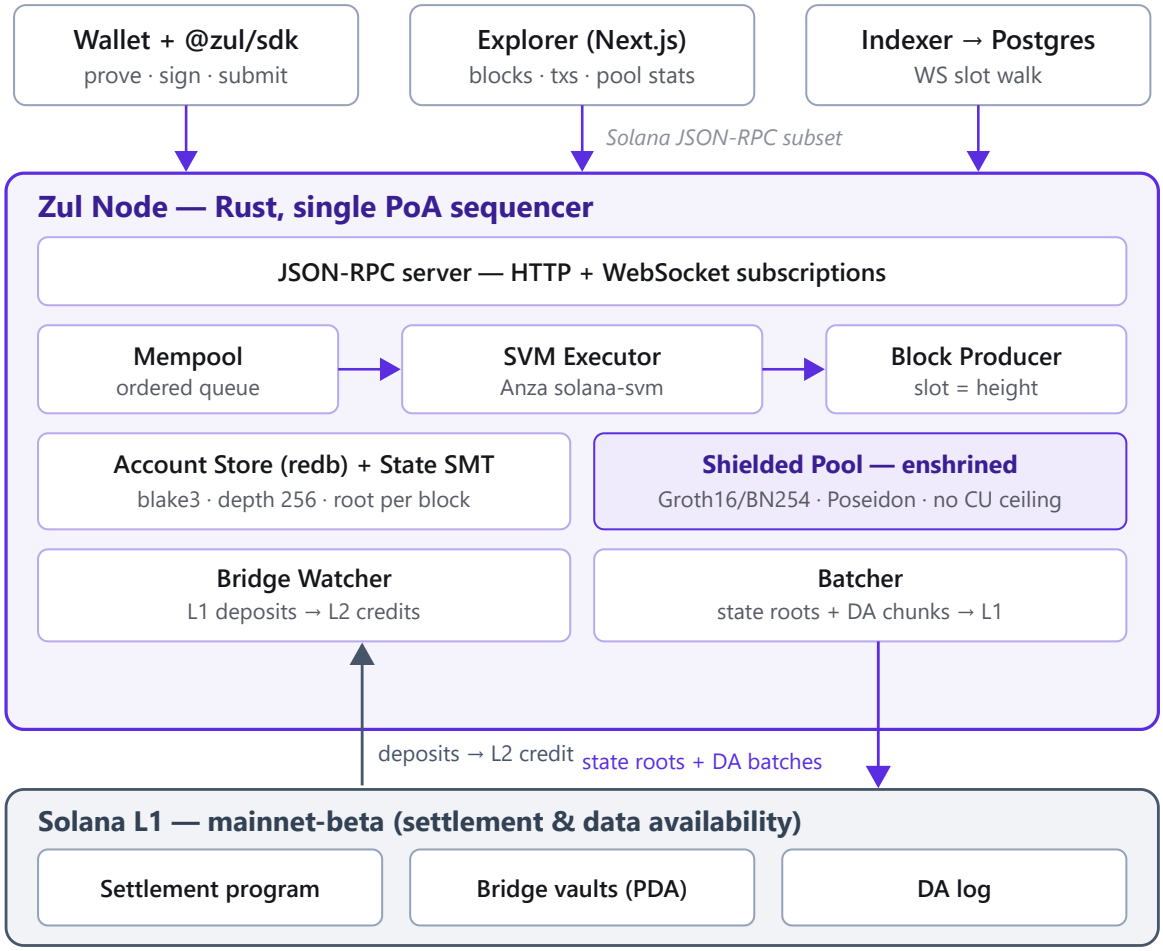

Figure 1. Zul system architecture. Clients speak the Solana JSON-RPC subset to the node; the node executes SVM transactions, commits state through the blake3 SMT, runs the enshrined shielded pool, and exchanges deposits, state roots, and data-availability batches with Solana L1.

Figure 1 shows the pieces. The **node** contains: a JSON-RPC server (a subset of Solana's HTTP and WebSocket API, Appendix C); a mempool feeding the **SVM executor** built on `solana-svm`; a block producer that seals a block each slot when there is work or a heartbeat elapses; an account store on `redb` carrying the depth-256 state SMT; the shielded-pool builtin; a bridge watcher that turns L1 deposits into L2 credits; and a batcher that posts state roots and compressed data to L1. Around the node sit the **clients** — a TypeScript SDK and wallet, a Next.js explorer, and an indexer that mirrors the chain into Postgres — and beneath it the **L1 programs**: a settlement program holding the bridge vaults and verifying withdrawal claims, and a data-availability log.

3.2 Transaction lifecycles

Four flows exercise the whole system; each is detailed in its own section.

— **Public transfer.** A client submits a normal Solana transaction; the executor runs it; the block producer seals a block; the touched accounts update the state SMT (§4).

— **Deposit (L1→L2).** A user locks SOL or an SPL token in the L1 vault; the watcher observes the event and credits native ZUL or a deterministic wrapped token on L2 (§7).
— **Shielded operation.** A client builds a shield, private transfer, or unshield, generates a Groth16 proof in the browser, and submits it; the node verifies natively, updates the commitment tree, and records nullifiers (§8).
— **Withdraw (L2→L1).** A burn on L2 enters the shallow withdrawals tree; once its root is posted, anyone can present an inclusion proof to the settlement program to release the asset on L1 (§6).

3.3 Trust model

We separate two kinds of guarantee. A property is *enforced* when violating it requires breaking a cryptographic assumption; it is *procedural* when violating it is detectable by anyone with the public data but is not yet automatically rejected. Zul today is a Stage-0 rollup: its privacy and value-conservation guarantees are enforced, while its execution-integrity guarantee is procedural.

WHAT THE SEQUENCER CAN AND CANNOT DO

Can reorder or censor pending transactions	liveness / fairness assumption
Can halt block production	liveness assumption
Can post a state root not produced by honest execution	procedural — publicly detectable, not yet rejected on-chain
Cannot create value in the shielded pool or forge a spend	enforced by Groth16 soundness + value conservation
Cannot double-spend a note	enforced by the nullifier set
Cannot learn the contents of a shielded transfer	enforced by zero-knowledge
Cannot release a bridged asset without a valid inclusion proof	enforced by the L1 settlement program

This is the one place a privacy project must not equivocate. The sequencer is trusted for *ordering and execution integrity* and is *not* trusted for *privacy or solvency*. A dishonest sequencer can stall the chain or attempt an invalid state transition — and because all batch data is public (§6), such an attempt is detectable by reconstruction — but it cannot mint hidden value, cannot unmask a private transfer, and cannot drain the L1 vaults, because those barriers are cryptographic rather than behavioral. Section 10 makes these statements precise; Section 13 describes how the procedural guarantees become enforced ones.

3.4 Threat model

We assume a probabilistic polynomial-time adversary that may submit arbitrary transactions, observe all public chain data and network traffic, and act as an ordinary user of the bridge and pool. The adversary may also *be* the sequencer. We do not assume a trusted third party beyond the per-circuit

common reference string produced by the trusted setup (§10.5), whose compromise we analyze explicitly. Table 2 maps the principal threats to the mechanism that addresses each.

Table 2. Threats and the mechanism that addresses each. "E" = enforced by cryptography; "P" = procedural / disclosed.

Threat	Mechanism	§	Class
Inflate value in the shielded pool	In-circuit value conservation + Groth16 soundness	9	E
Spend a note twice	Deterministic nullifier + persistent nullifier set	8.3	E
Spend a note you do not own	Ownership constraint (spending key → public key)	9	E
Forge an inclusion proof against an old root	64-root ring + Poseidon Merkle tree	8.4	E
Mint a wrapped asset without a deposit	Keyless mint authority; supply grows only via vault lock	7.2	E
Release a bridged asset without burning on L2	L1 inclusion proof against posted withdrawals root	6.3	E
De-anonymize a private transfer	Zero-knowledge of Groth16; encrypted note discovery	10.4	E
Front-run a public unshield to a new recipient	Recipient bound into the proof	9.2	E
Post an invalid state root	Public DA + reconstruction (challenge not yet enforced)	6	P
Censor or reorder transactions	Single sequencer; decentralization on roadmap	13	P
Link the public fee payer of a private transfer	v1 public fee payer; relayer planned	13	P

04 — TRANSPARENT MACHINE

The SVM Execution Layer

Running the real Solana runtime over a custom store, so every existing program and wallet works unchanged — and so the chain can verify arbitrary zero-knowledge proofs on-chain.

The execution layer's job is to be *boringly* Solana. If the L2 executed transactions even slightly differently from a validator, the entire program ecosystem and tooling base would break, and the leverage of building on the SVM would evaporate. Zul therefore does not reimplement the runtime; it embeds Anza's `solana-svm` [6] and supplies only the parts a rollup must own: where account state comes from, when blocks are sealed, and how state is committed.

4.1 Embedding `solana-svm`

The core of the executor is a `TransactionBatchProcessor` driven through the `TransactionProcessingCallback` trait. The callback is Zul's adapter between the SVM and its own storage: when the processor needs an account, the callback answers from a two-layer view — a write overlay holding accounts touched earlier in the same block, falling back to the committed `redb` store. After a batch executes, the resulting account writes are collected from the overlay, applied to the store, and folded into the state commitment (§5). Because Zul is a single sequencer with a linear history, the program cache's fork graph is trivial and every account is reported at one slot, which keeps loader and cache behavior deterministic.

Per-slot pipeline. Each block is produced by the following steps, all within the node:

1. drain ordered transactions from the mempool;
2. apply any pending L1 deposits as direct credits (§7.3);
3. sanitize and signature-verify each transaction, and check its blockhash against the recent-blockhash queue;
4. charge fees and run compute-budget instructions, then `load_and_execute_transactions` through the SVM;
5. collect account writes; update the state SMT with exactly the touched accounts;
6. seal the block header with the new `state_root`, and make non-empty data available to the batcher (§6).

4.2 Builtins and preloaded programs

The processor is configured with the standard builtin programs — the System program, the BPF loaders (including loader-v4), and the Compute-Budget program — and the common SPL programs are preloaded so they exist from genesis: the SPL Token and Token-2022 programs, the Associated-Token-Account program, and Memo. Crucially, the set of builtins also includes Zul's own **shielded-pool program** at a reserved address. It is registered as a builtin precisely so it executes in native Rust

rather than as metered BPF; this is what frees proof verification from the compute-unit budget (§8). User-deployed BPF programs work exactly as on Solana, through the upgradeable loader.

4.3 Sysvars, features, and determinism

The executor maintains the sysvars programs expect. The `Clock` is refreshed every slot with the block's height and timestamp; `Rent` and the other static sysvars are seeded at genesis. The runtime is configured with the full feature set enabled, matching a modern Solana cluster, so feature-gated program behavior matches mainnet. Determinism comes from three facts: a single ordering authority, a fixed per-slot sysvar update, and a content-addressed account store whose Merkleization is insertion-order independent (Theorem 2).

4.4 Native verification of zero-knowledge proofs

Because the full feature set is enabled, the runtime exposes the `alt_bn128` syscalls — addition, scalar multiplication, and the pairing check on the BN254 curve — and the `poseidon` syscall. The immediate consequence is mundane and the eventual one is the entire thesis of this project. Mundanely, the enshrined pool does not even need the syscalls, because it verifies in native arkworks code (§8.6). The eventual consequence is that *an ordinary BPF program running on Zul can itself verify a Groth16 or PLONK proof on-chain*, within its compute budget, using the same pairing the pool uses. That capability — arbitrary, user-deployed programs verifying succinct proofs natively — is the seed of the programmable privacy layer (§12). The execution layer is built today in the shape of what it must become.

BLOCK

```
// sealed once per slot; the state_root anchors all account state
BlockHeader {
  slot:              u64,        // = chain height
  parent_hash:       H256,
  state_root:        H256,       // blake3 SMT root over all accounts (§5)
  transactions_root: H256,
  timestamp_ms:      u64,
  transaction_count: u32,
}
```

4.5 Fees and the gas token

Zul's lamport unit is **ZUL**. SOL does not exist natively on the L2; it arrives only as a bridged, wrapped asset (§7). Fees use the SVM's standard handling — a base fee plus compute-budget-driven priority — paid in ZUL and swept to a collector account. Native ZUL is fully backed: with zero genesis pre-mine on mainnet, every native ZUL lamport corresponds to a ZUL-SPL token locked in the L1 bridge vault (§7.3), so the gas supply cannot be conjured by the sequencer any more than user value can.

4.6 RPC compatibility

The node answers a subset of the Solana JSON-RPC over HTTP and WebSocket (Appendix C): the read and submit methods wallets and the `solana` CLI use — `sendTransaction`, `simulateTransaction`, `getAccountInfo`, `getLatestBlockhash`, `getSignaturesForAddress`, `getTransaction`, token-account queries — plus slot, signature, and account subscriptions, and one Zul-specific method, `getWithdrawalProof`, that returns an SMT inclusion proof for a bridge claim (§6.3). Reporting an accurate `solana-core` version and the legacy status fields is enough that off-the-shelf clients treat Zul as a Solana cluster.

05 — COMMITMENT

State Commitment: the blake3 Sparse Merkle Tree

A succinct, binding commitment to all account state, with inclusion and absence proofs that the L1 settlement program can re-verify.

The SVM agrees on account state without ever building a global state trie: a Solana validator does not need one, because consensus is over the bank rather than a single root. A rollup does need one. To settle to L1 and to back withdrawals, Zul must commit *all* of its account state to one short value per block and produce proofs about individual accounts against it. Zul does this with a sparse Merkle tree [31][32] over $(\mathsf{pk} \mapsto \text{account hash})$, hashed with blake3 [16] under domain separation.

5.1 Construction

Let $\mathcal{H} = \mathsf{blake3}$ and fix four disjoint domain tags — the byte strings `PL2:smt:key`, `PL2:smt:leaf`, `PL2:smt:node`, `PL2:smt:empty` — written $D_\kappa, D_\lambda, D_\eta, D_\varepsilon$. The tree has fixed depth $N = 256$. An account with public key pk occupies the leaf addressed by

$$\kappa(\mathsf{pk}) \;=\; \mathcal{H}\big(D_\kappa \,\|\, \mathsf{pk}\big) \;\in\; \{0,1\}^{256},$$

a uniformly distributed path, so keys do not collide except with negligible probability and an adversary cannot steer accounts to a chosen subtree. Internal and leaf nodes are

$$\eta(L, R) = \mathcal{H}\big(D_\eta \,\|\, L \,\|\, R\big), \qquad \lambda(\mathsf{pk}, a) = \mathcal{H}\big(D_\lambda \,\|\, \mathsf{pk} \,\|\, a\big),$$

where $a = \alpha(\mathsf{acct})$ is the account hash of §5.2. Empty subtrees are precomputed once:

$$\varepsilon_N = \mathcal{H}(D_\varepsilon), \qquad \varepsilon_d = \eta(\varepsilon_{d+1}, \varepsilon_{d+1}) \;\; (0 \le d < N).$$

The commitment to a finite account map M is the value at depth 0, the **state root** $R(M)$. Because every absent leaf equals ε_N and every empty subtree equals its precomputed constant, the tree is stored sparsely: only nodes on a path to a non-empty leaf are materialized.

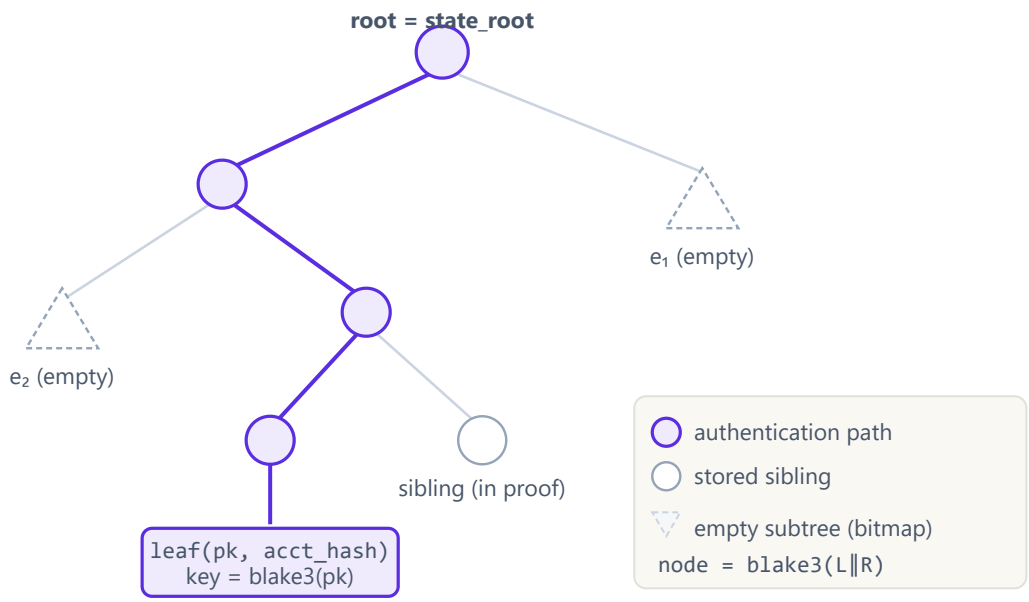

Figure 2. The state SMT (schematic; true depth 256). An account hashes to a leaf keyed by $\kappa(\mathsf{pk}) = \mathsf{blake3}(\mathsf{pk})$; internal nodes are domain-separated blake3 of their children. Empty subtrees collapse to precomputed constants ε_d and are elided from proofs via a 256-bit bitmap, so a proof lists only the non-empty siblings on the authentication path.

5.2 Account hashing

An account's hash binds every field that affects execution:

$$\alpha(\mathsf{acct}) = \mathcal{H}\big(D_\alpha \,\|\, \mathsf{pk} \,\|\, \ell \,\|\, o \,\|\, x \,\|\, \rho \,\|\, |d| \,\|\, d\big),$$

over the domain `PL2:account:v1`, where ℓ is the lamport balance, o the owner program, x the executable flag, ρ the rent epoch, and d the account data with its length $|d|$ bound explicitly to prevent length-extension ambiguity. Any change to any field changes α, hence the leaf, hence the root.

5.3 Incremental updates and proofs

Only the leaves an executed block actually touches are updated. For each touched account the node recomputes the leaf, then walks the 256 levels to the root, reading each sibling from a write overlay or, failing that, the committed store or the empty constant. A deletion sets the leaf to ε_N, which transparently restores the subtree to its empty form. A proof for pk is compact: a 256-bit *bitmap* records which siblings along the path are non-empty, and only those siblings are listed; empty siblings are recovered by the verifier from the constants. The same structure proves *absence* — that no account exists at $\kappa(\mathsf{pk})$ — by authenticating the empty leaf, which is what lets the bridge prove a withdrawal has not yet been claimed.

5.4 Security of the commitment

> **Theorem 1 (Binding).** *Model $\mathcal{H}$ as collision resistant. For any PPT adversary $\mathcal{A}$, the probability that $\mathcal{A}$ outputs two distinct account maps $M \neq M'$ with $R(M) = R(M')$ is* negl.

Proof (sketch). Suppose $M \neq M'$ yet $R(M) = R(M')$. There is a key pk at which the maps differ, so the two leaves $\lambda(\mathsf{pk}, \alpha) \neq \lambda(\mathsf{pk}, \alpha')$ differ (else, by collision resistance of λ, the account hashes agree and the maps agree at pk). Walk from this leaf to the root. The leaf values differ but the roots coincide, so along the path there is a first level where two distinct child pairs $(L, R) \neq (L', R')$ hash to the same parent under η. That is a blake3 collision, contradicting collision resistance. The domain tags ensure a collision cannot be forged *across* the leaf, node, and empty constructions. ∎

> **Theorem 2 (Determinism).** *The root $R(M)$ depends only on the final account map M, not on the order in which leaves were inserted, updated, or deleted.*

Proof (sketch). Each node value is a pure function of the values of its two children, and each leaf is a pure function of (pk, α); absent leaves are the fixed constant ε_N. Thus the value at every position is determined by M alone via the fixed recurrence, independent of evaluation order. The implementation's overlay-then-commit update computes exactly these values, and restoring a leaf to ε_N on deletion yields the same node values as never having inserted it. ∎

> **Theorem 3 (Proof soundness).** *Under collision resistance of $\mathcal{H}$, if $\mathsf{Verify}(R, \mathsf{pk}, a, \pi) = 1$ then the committed leaf at $\kappa(\mathsf{pk})$ equals $\lambda(\mathsf{pk}, a)$ (for inclusion) or ε_N (for absence, $a = \perp$). No PPT adversary produces a root and two accepting proofs for contradictory statements about the same key except with probability* negl.

Proof (sketch). Verification recomputes the path from the claimed leaf to R using the bitmap to supply empty constants and the listed siblings otherwise, accepting iff it reaches R and consumes exactly the provided siblings. Two accepting proofs for the same key with different committed leaves give two distinct paths to the same root, hence (as in Theorem 1) a node-level blake3 collision. The exact-consumption check rules out padding a proof with spurious siblings. ∎

5.5 Agreement with L1, and the shallow withdrawals tree

The settlement program re-implements this exact construction using Solana's blake3 syscall, with the same four domain tags, so the two implementations agree byte-for-byte; the agreement is pinned by shared test vectors — for instance the empty-tree root $R(\varnothing) =$`eb350f1a…0ef42abb` is asserted on both sides. This matters because the L1 program verifies proofs the L2 produces.

There is a deliberate split. The **state** SMT is depth 256 — it must address the entire 256-bit pubkey space. But verifying a depth-256 path on Solana would cost hundreds of blake3 invocations per claim. Zul therefore commits withdrawals in a *separate* SMT of depth **32**, of identical construction, whose inclusion proofs are short enough to verify within Solana's per-transaction compute-unit budget (§6.3). The state root proves *what the L2 believes*; the shallow withdrawals root proves *exactly which exits are authorized*, cheaply, on-chain.

06 — SETTLEMENT BOUNDARY

Blocks, Data Availability & Settlement

Making the L2's history public on Solana, and turning a shallow withdrawals root into the one thing the L1 must trust.

Settlement is where the L2 becomes accountable to the L1. Two artifacts cross the boundary every batch: a commitment to the L2's state, and enough data for anyone to reconstruct that state and check it. A third artifact — a shallow withdrawals root — is what actually authorizes assets to leave the bridge. Figure 3 shows the two flows.

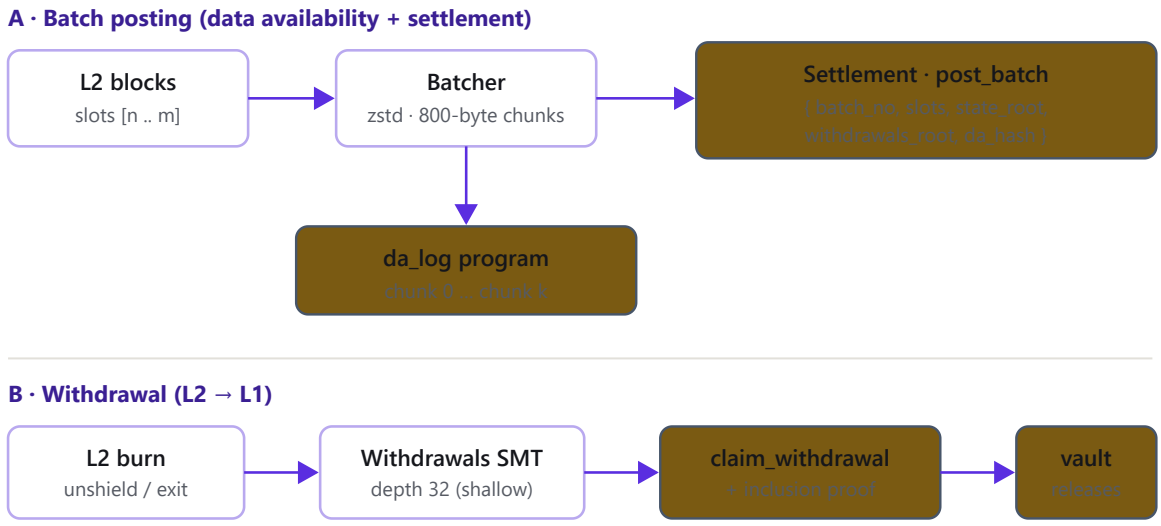

Figure 3. (A) The batcher compresses a range of L2 blocks, chunks the bytes into the data-availability log, and posts a settlement record binding the state root, withdrawals root, and a hash of the data. (B) A withdrawal burned on L2 enters the depth-32 withdrawals tree; once its root is posted, an inclusion proof lets anyone claim the asset from the L1 vault.

6.1 Block production

The sequencer seals a block every slot in which there is work, and otherwise on a roughly ten-second heartbeat so that time advances and recent blockhashes rotate even when the chain is idle. The slot number is the chain height. Only blocks carrying real data are batched to L1; empty heartbeat blocks stay local. The recent-blockhash queue keeps the last 150 blockhashes, matching Solana's window so that client-side transaction expiry behaves identically.

6.2 Data availability

For a batch spanning slots $[n, m]$, the batcher serializes the block and transaction data, compresses it with zstd, and splits the result into roughly 800-byte chunks carried as instruction data to the data-availability log program — a noop-style program whose only purpose is to make bytes permanently retrievable from the Solana ledger. The batch's *data hash* is

$$\mathsf{da\_hash} \;=\; \mathsf{blake3}\big(D_{\mathsf{da}} \,\|\, \mathsf{zstd}(\mathrm{batch})\big), \qquad D_{\mathsf{da}} = \texttt{PL2:da:batch:v1},$$

and is bound into the settlement record so the posted state root and the published bytes are inseparable. The node also serves batches directly over RPC, so reconstruction does not depend on a single archival source. This is the cheapest way to make the L2's history public, and publicity is the whole basis of the optimistic guarantee: anyone can download the data, replay it, and recompute the root the sequencer claimed.

6.3 Settlement records and withdrawal claims

Each batch is recorded on L1 as:

SETTLEMENT RECORD (L1)

```
Batch {
  batch_no:         u64,    // must equal the program's batch_count — strictly sequential
  first_slot:       u64,
  last_slot:        u64,
  state_root:       [u8;32],   // blake3 SMT root over all L2 accounts (§5)
  withdrawals_root: [u8;32],   // depth-32 SMT root over authorized exits
  da_hash:          [u8;32],
  chunk_count:      u32,
  posted_at:        i64,
}
```

Posting is strictly sequential — the program accepts batch k only when its internal counter is exactly k — so the settled history is a single, gapless chain of roots. A **withdrawal claim** is the security-critical operation: it releases real value on L1. To claim, a user presents the withdrawal's parameters and an inclusion proof against a posted `withdrawals_root`. The settlement program recomputes the depth-32 leaf and walks the proof with the same blake3 construction as §5; if it reaches the posted root, it releases the asset from the vault. The withdrawal leaf binds an *asset identifier*:

$$\mathsf{leaf}_{\mathsf{wd}} = \mathsf{blake3}\big(D_{\lambda} \,\|\, \mathsf{recipient} \,\|\, \mathsf{asset\_id} \,\|\, \mathsf{amount} \,\|\, \mathsf{nonce}\big),$$

with $\mathsf{asset\_id} = \mathbf{0}^{32}$ for native SOL/ZUL and the L1 mint address for an SPL token. Binding the asset into the leaf is what makes `claim_withdrawal` and `claim_withdrawal_spl` safe: a proof authorizes releasing exactly one asset in exactly one amount to exactly one recipient, and nothing else (§7). The depth-32 tree is the reason this verifies cheaply enough to fit Solana's per-transaction compute budget; the full depth-256 state root would not.

6.4 The optimistic model, stated honestly

Zul's settlement is optimistic and currently Stage 0. The challenge window is procedural: because every batch's data is public and every state root is recomputable from it, an invalid root is *detectable* by anyone who replays the data — but the settlement program does not yet reject invalid roots on its own. We state this plainly rather than implying a guarantee we do not yet enforce. What *is* already enforced on L1 is solvency at the bridge: no asset leaves a vault without a valid inclusion proof against

a posted withdrawals root, regardless of anything the sequencer claims about state. Section 13 describes the path from a procedural challenge window to enforced fraud or validity proofs.

07 — THE SEAM

The Bridge

Moving SOL and arbitrary SPL tokens across the L1↔L2 boundary so that value — public or private — can cross without the boundary becoming a leak or a mint.

A privacy L2 is only as useful as its connection to the assets people actually hold. Zul's bridge is bidirectional and asset-general: it moves SOL and any SPL token (classic or Token-2022) in both directions, and every bridged asset can subsequently be shielded (§8). The design discipline throughout is conservation: the L2 may credit a wrapped asset only when the original is genuinely locked on L1, and the L1 may release the original only against a proof that the wrapped asset was genuinely burned on L2.

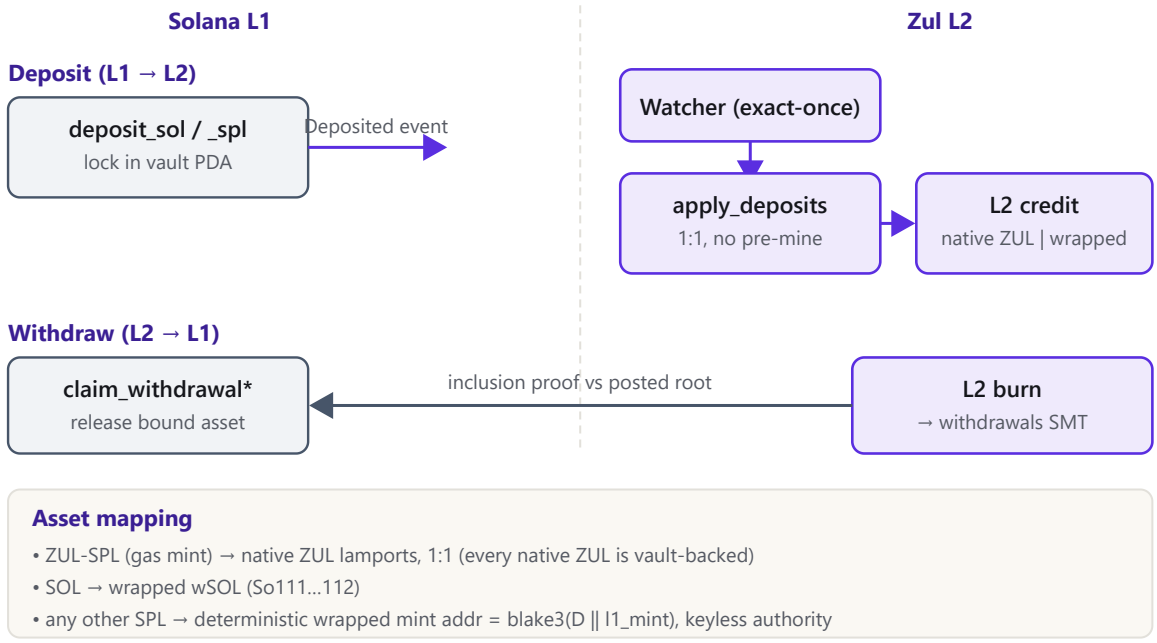

Figure 4. Bridge flows and asset mapping. Deposits lock value in an L1 vault and emit an event the watcher turns into an exactly-once L2 credit; withdrawals burn on L2 and release the bound asset from the vault against an inclusion proof. Native ZUL is fully backed by locked ZUL-SPL; other assets become deterministic wrapped tokens.

7.1 The asset model

On L2 there is exactly one native asset, ZUL, used for gas. Everything else is a *wrapped* token. Three rules cover every case:

— **Gas.** Depositing the designated gas mint — the ZUL-SPL token on L1 — credits native ZUL lamports one-for-one. With zero pre-mine at genesis, this is the *only* way native ZUL comes into existence, so the entire gas supply is backed by locked ZUL-SPL.

— **SOL.** Depositing native SOL credits a wrapped wSOL token at the canonical address `So111…112`, so SOL behaves on L2 as an ordinary SPL token.

— **Any other SPL.** Depositing an SPL mint credits a *deterministic* wrapped token whose mint address is $\mathsf{blake3}(D_{\mathsf{wm}} \,\|\, \mathsf{l1\_mint})$, with decimals copied from the original.

7.2 Why wrapped supply cannot be forged

Each wrapped mint's authority is a *keyless reserved address* — a pubkey with no corresponding secret key, off the Ed25519 curve. No party can sign a `MintTo` for it, so a wrapped token's supply can grow only through the node's own deposit-crediting path, never through an SVM instruction. Combined with deterministic addressing, this means the wrapped representation of an L1 mint is unique and its supply tracks exactly the value locked in that mint's vault.

The deposit path also screens the asset itself. The L1 program's `require_safe_mint` check rejects Token-2022 extensions that would make a locked balance unsafe or non-conservative — transfer hooks, permanent delegates, non-transferable or default-frozen states, and confidential-transfer configurations — while permitting benign ones. For a mint carrying a transfer fee, the bridge credits the amount the vault *actually received* after the fee, never the requested amount, so a fee-on-transfer token cannot be used to over-credit the L2.

7.3 Deposit: exactly once

A deposit on L1 (`deposit_sol` or `deposit_spl`) locks value in the vault and emits a `Deposited` event carrying the depositor, amount, L2 recipient, and — for SPL — the mint and its decimals. The node's watcher polls for these events on a short interval and applies them as direct credits at the top of the next block. The danger with any deposit watcher is double-crediting; Zul forecloses it with a *deposit receipt*: a reserved account keyed by the L1 transaction signature is written the first time a deposit is credited, and the credit is skipped whenever that receipt already exists, whether earlier in the same block or in any committed block. Crediting is therefore idempotent in the L1 signature.

7.4 Withdraw: bound to one asset

A withdrawal burns the wrapped (or native) asset on L2 and inserts a leaf into the depth-32 withdrawals tree (§6.3). Once the root is posted, anyone holding the witness can call `claim_withdrawal` (native) or `claim_withdrawal_spl` (token), which verifies the inclusion proof and releases the asset from the vault — native lamports from the vault PDA, or tokens from the per-mint vault associated-token account it controls. Because the asset identifier is hashed into the withdrawal leaf, a valid proof can only ever release the specific asset the L2 burned: the claim cannot be redirected to a different, more valuable token in the vault.

7.5 Crossing into the shielded world

The bridge and the pool compose. A bridged asset is identified on L2 by its mint; inside the shielded pool that same identity becomes a field element — $\mathsf{asset\_id} = 0$ for native ZUL, or the mint's bytes reduced modulo the field prime for any wrapped token (§8.1). So the full lifecycle — deposit an SPL token, shield it, transfer it privately, unshield it, withdraw it back to L1 — is available for any asset the bridge supports, with the asset's identity carried coherently across every stage and revealed only where a public vault movement reveals it anyway.

08 — PRIVATE MACHINE

The Shielded Pool: Cryptographic Core

Notes, commitments, nullifiers, and an incremental Merkle tree — the UTXO machinery that hides amounts and the transaction graph while keeping value provably conserved.

The shielded pool is where Zul stops merely *not indexing* private data and starts making it cryptographically unavailable. It is a UTXO system in the Zerocash lineage [7][8]: value lives in *notes*, a note's existence is announced only as an opaque *commitment* in a Merkle tree, and spending a note reveals only a *nullifier* that prevents reuse without revealing which note was spent. Everything else — amounts, the asset, the link between a spend and the note it consumes — is hidden behind a zero-knowledge proof. The pool is enshrined in the node (§4.2), so verifying those proofs costs no compute units and the pool keeps its commitment tree and nullifier set as native state.

8.1 Field, hash, and notes

All shielded arithmetic is over $\mathbb{F}_p$, the scalar field of the BN254 curve [13] (a 254-bit prime). The only hash is Poseidon [14], written $\mathcal{H}_k : \mathbb{F}_p^k \to \mathbb{F}_p$ for arity k; the implementation uses the circom-compatible parameters so the in-node Rust hasher and the in-circuit hasher agree exactly [22]. A **note** is

$$\nu \;=\; (\,\mathsf{asset\_id},\; v,\; \mathsf{pk},\; \mathsf{blind}\,), \qquad v \in [0,\, 2^{64}),$$

where $\mathsf{asset\_id} \in \mathbb{F}_p$ is 0 for native ZUL or the L1 mint reduced into $\mathbb{F}_p$ for a bridged token, v is the amount, pk the owner's public key, and blind a random blinding factor that makes commitments hiding.

8.2 The two-level commitment

Keys and commitments are built from a single spending secret s in two layers (Figure 5):

$$\mathsf{pk} = \mathcal{H}_1(s), \qquad \mathsf{secret} = \mathcal{H}_2(\mathsf{pk},\, \mathsf{blind}), \qquad \mathsf{cm} = \mathcal{H}_3(\mathsf{asset\_id},\, v,\, \mathsf{secret}).$$

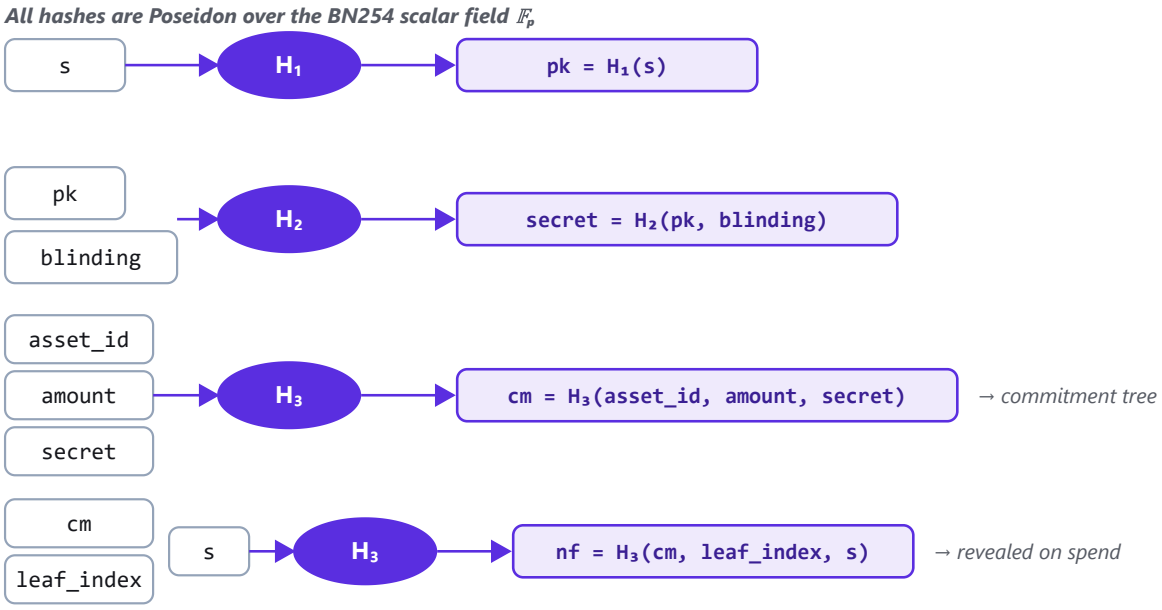

Figure 5. The note commitment and nullifier. The public key derives from the spending secret; a *secret* hides the owner and blinding; the commitment binds the public $(\mathsf{asset\_id}, v)$ to that secret; and the nullifier ties a commitment, its tree position, and the spending key together.

The two-level structure is not decoration; it is what makes a *deposit need no proof* while remaining sound. At shield time the depositor reveals the opaque secret (which still hides pk and blind by Poseidon's preimage resistance) together with the *public* $(\mathsf{asset\_id}, v)$. The node recomputes $\mathsf{cm} = \mathcal{H}_3(\mathsf{asset\_id}, v, \mathsf{secret})$ itself and checks it against the deposited funds. A flat commitment $\mathcal{H}(\mathsf{asset\_id}, v, \mathsf{pk}, \mathsf{blind})$ could not be recomputed on-chain without revealing pk and blind, which would either break privacy or let a deposit of 1 mint a note worth 1000. The two-level commitment dissolves that dilemma (Theorem 6).

8.3 Nullifiers

Spending a note at tree position i reveals

$$\mathsf{nf} \;=\; \mathcal{H}_3(\mathsf{cm},\, i,\, s).$$

The nullifier is *deterministic* in the note and key, so the same note always yields the same nullifier and cannot be double-spent; yet without the spending key s it is computationally unlinkable to the commitment cm, so observers cannot tell which note a spend consumed. Including the leaf index i domain-separates otherwise-identical notes at different positions. The node keeps a persistent **nullifier set** — a store table mapping each spent nullifier to the slot at which it was spent — and rejects any transaction reusing one (Theorem 5).

8.4 The commitment tree

Commitments accumulate in an *incremental* Merkle tree of depth 26 with Poseidon nodes $\eta(L, R) = \mathcal{H}_2(L, R)$, giving capacity $2^{26} \approx 6.7 \times 10^7$ notes. New commitments are appended at the next free leaf; empty positions use a fixed empty-leaf constant. To spend, a prover shows Merkle membership of the input note's commitment under *some* recent root. Because the root changes with every insertion,

the pool retains a ring of the **64 most recent roots**; a proof is accepted against any root in the ring, which lets a wallet build a proof without racing the very next deposit. A spend against a root older than the ring is rejected, bounding how stale an accepted proof may be.

8.5 The three operations

The pool exposes exactly three instructions; Figure 6 traces a note's life through them, and §9 (Figure 7) details the circuits. Here is the pool's view of each.

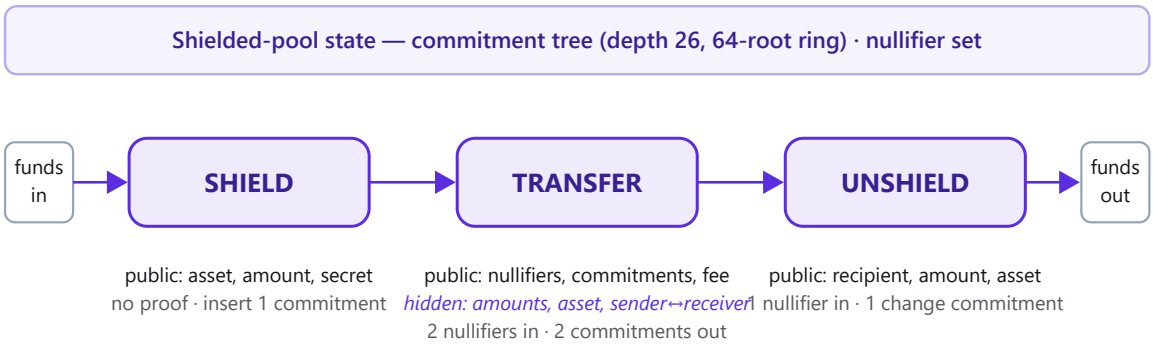

Figure 6. The note lifecycle. Value enters by *shield*, moves privately by *transfer*, and leaves by *unshield*; each spend reveals only a nullifier and a proof, while amounts, asset, and the sender↔receiver link stay hidden.

— **Shield.** Public $(\mathsf{asset\_id}, v, \mathsf{secret})$ plus an encrypted note. The node moves v of the asset into the pool vault, recomputes cm, and appends it. No proof is required.

— **Transfer.** A 2-input/2-output join-split with a Groth16 proof. It reveals two nullifiers and two output commitments and a public fee, and hides the amounts, the asset, and which notes were consumed. Its public input vector has exactly six elements, $[\,\mathsf{root}, \mathsf{nf}_0, \mathsf{nf}_1, \mathsf{cm}_0, \mathsf{cm}_1, \mathsf{fee}\,]$.

— **Unshield.** Spends one note to a public recipient and amount, with a private change note for the remainder, again with a six-element public input vector $[\,\mathsf{root}, \mathsf{nf}, \mathsf{cm}_{\mathrm{chg}}, \mathsf{asset\_id}, \mathsf{amount}, \mathsf{recipient}\,]$. The node verifies the proof, releases amount of the asset from the vault to the recipient, records the nullifier, and appends the change commitment.

8.6 Native verification

Transfer and unshield proofs are Groth16 over BN254, generated in the browser with snarkjs [21] and verified in the node with arkworks (`ark-groth16`, `ark-bn254`) [23]. Verification is a constant three-pairing check against a circuit-specific verifying key (§9.3); because it runs as a native builtin rather than metered BPF, the pairing cost is irrelevant to any compute-unit budget. The cross-language binding is tested directly: a snarkjs-generated proof is required to verify in the arkworks verifier, so the prover and verifier cannot silently diverge.

8.7 Note discovery

A recipient must find notes paid to them without an index that would leak the graph. Each output note is encrypted to its recipient and attached to the transaction: the sender derives a shared secret by X25519 ECDH [29] and seals the note's fields under XChaCha20-Poly1305 [30]. A wallet trial-decrypts the ciphertexts of new transactions; the ones it can open are its incoming notes. Discovery therefore costs a scan but reveals nothing to anyone lacking the viewing key.

8.8 Pool state and reserved addresses

The pool occupies reserved, keyless addresses so its accounts cannot be spoofed by a signed transaction: the pool program at `[05 21 00…00]`, its state account at `…01`, and its vault at `…02`. The state account holds the commitment tree's frontier, the 64-root ring, and the next free leaf index; the vault holds native ZUL lamports, and a per-mint associated-token account holds each wrapped asset that shield and unshield move in and out. Keeping this as native state — rather than inside a metered program account — is what lets the pool scale its tree and nullifier set without contorting around account-size and compute limits.

09 — CONSTRAINTS

The Shielded Circuits

Two circom circuits — a join-split transfer and an unshield — whose constraints are exactly the statements "this spend is well-formed, authorized, and value-preserving."

Both circuits are written in circom 2.1.6 [21] over the BN254 scalar field, using circomlib's Poseidon, comparators, and bit-decomposition gadgets [22]. A subtle but important convention governs the public interface: circom orders public signals as outputs first and then inputs, so the circuits declare their public values as *inputs* in a fixed order and constrain them to the values the circuit derives. This makes the public-input vector deterministic and exactly matches the order the node's verifier reads (Appendix E).

9.1 The transfer circuit

Figure 7 shows the join-split. For inputs $i \in \{0,1\}$ with amounts v_i, spending keys s_i, blindings b_i, positions ι_i, and outputs $j \in \{0,1\}$ with amounts v'_j, recipient keys pk'_j, blindings b'_j, sharing one private $\mathsf{asset\_id} = \mathfrak{a}$ and a public fee, the circuit enforces:

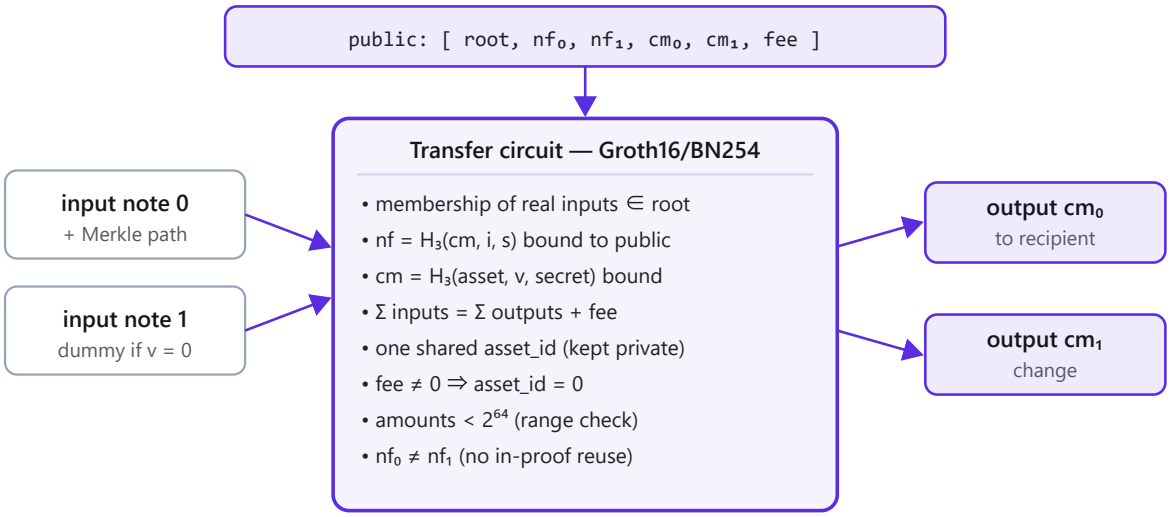

Figure 7. The 2-in/2-out transfer circuit. Real inputs are proven members of a recent root; nullifiers and output commitments are bound to the public signals; value is conserved over a single, private asset; and amounts are range-checked so sums cannot wrap the field.

1. **Derivation and binding.** For each input, $\mathsf{pk}_i = \mathcal{H}_1(s_i)$, $\mathsf{cm}_i = \mathcal{H}_3(\mathfrak{a}, v_i, \mathcal{H}_2(\mathsf{pk}_i, b_i))$, and $\mathsf{nf}_i = \mathcal{H}_3(\mathsf{cm}_i, \iota_i, s_i)$ with the public nullifier constrained $\mathsf{nf}_i \overset{!}{=} \mathsf{nullifier}[i]$. For each output, $\mathsf{cm}'_j = \mathcal{H}_3(\mathfrak{a}, v'_j, \mathcal{H}_2(\mathsf{pk}'_j, b'_j))$ with $\mathsf{cm}'_j \overset{!}{=} \mathsf{out\_commitment}[j]$.
2. **Membership with dummy handling.** Each input proves Merkle inclusion of cm_i at ι_i under the public root, gated by $\mathsf{enabled}_i = 1 - \mathsf{isZero}(v_i)$; a zero-amount input is a *dummy* whose membership check is skipped, exactly as in Tornado Nova [18]. Concretely the constraint is $(\widehat{\mathsf{root}}_i - \mathsf{root}) \cdot \mathsf{enabled}_i = 0$, where $\widehat{\mathsf{root}}_i$ is the root recomputed from the supplied path.
3. **Value conservation.** $v_0 + v_1 = v'_0 + v'_1 + \mathsf{fee}$ over a single asset.
4. **Asset gating.** All four notes share $\mathfrak{a}$ (which is never made public), and the fee may be non-zero only in the native asset: $\mathsf{fee} \cdot \big(1 - \mathsf{isZero}(\mathfrak{a})\big) = 0$.
5. **Range.** Every amount is decomposed into 64 bits ($\mathsf{Num2Bits}(64)$), so $0 \le v < 2^{64}$ and no sum can wrap the 254-bit field.
6. **Distinct nullifiers.** $\mathsf{nf}_0 \neq \mathsf{nf}_1$, closing the "spend the same note twice within one proof" attack.

The public-input vector is $[\,\mathsf{root}, \mathsf{nf}_0, \mathsf{nf}_1, \mathsf{cm}_0, \mathsf{cm}_1, \mathsf{fee}\,]$ — six elements, and nothing about the amounts, the asset, the owners, or which leaves were spent appears in it.

9.2 The unshield circuit

Unshield spends one input note to a public $(\mathsf{recipient}, \mathsf{amount})$ in a stated $\mathsf{asset\_id}$, returning the remainder to a private change note. With input amount v, key s, blinding b, position ι, and change $(v_c, \mathsf{pk}_c, b_c)$, it enforces:

1. **Spend authorization.** $\mathsf{pk} = \mathcal{H}_1(s)$, $\mathsf{cm} = \mathcal{H}_3(\mathsf{asset\_id}, v, \mathcal{H}_2(\mathsf{pk}, b))$, Merkle inclusion of cm under root (always enabled), and $\mathsf{nf} = \mathcal{H}_3(\mathsf{cm}, \iota, s) \overset{!}{=} \mathsf{nullifier}$.
2. **Change well-formedness.** $\mathsf{cm}_{\mathrm{chg}} = \mathcal{H}_3(\mathsf{asset\_id}, v_c, \mathcal{H}_2(\mathsf{pk}_c, b_c)) \overset{!}{=} \mathsf{change\_commitment}$, sharing the input's asset.
3. **Conservation.** $v = \mathsf{amount} + v_c$, with all three amounts range-checked to 64 bits.
4. **Recipient binding.** A constraint $\mathsf{recipient}^2 = \mathsf{recipient} \cdot \mathsf{recipient}$ forces the recipient signal into the constraint system so the solver cannot optimize it away. This is what prevents a public unshield from being *front-run* to a different address: the recipient is part of what the proof attests, so a relayer cannot rewrite it.

The public-input vector is $[\,\mathsf{root}, \mathsf{nf}, \mathsf{cm}_{\mathrm{chg}}, \mathsf{asset\_id}, \mathsf{amount}, \mathsf{recipient}\,]$ — again six elements. Here the asset and amount *are* public, because the vault movement they trigger reveals them anyway; what stays hidden is the input note, its owner, and the change.

9.3 Groth16 instantiation

Each circuit compiles to a rank-1 constraint system, and Groth16 [9] turns the R1CS into a proving key and a verifying key via a per-circuit trusted setup (§10.5). A proof is three group elements $(A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$, and verification is the single pairing-product equation

$$e(A, B) \;=\; e(\alpha, \beta) \cdot e\Big(\textstyle\sum_{k=0}^{\ell} a_k \, L_k,\; \gamma\Big) \cdot e(C, \delta),$$

where $(a_0, \ldots, a_\ell)$ are the public inputs (with $a_0 = 1$) and $L_k, \alpha, \beta, \gamma, \delta$ come from the verifying key. For Zul both circuits have $\ell = 6$ public inputs in the orders above. The node performs this check natively (§8.6); its cost is constant and independent of circuit size, which is precisely why Groth16 suits an enshrined verifier.

9.4 Design notes

Three small choices carry weight. Poseidon's low multiplicative complexity makes each commitment, nullifier, and Merkle node cost a handful of constraints, so a full join-split is a small circuit and proving stays seconds-scale in the browser. The Merkle gadget routes left/right children with a multiplexer driven by the index bits, matching the incremental tree's index arithmetic so the in-node tree and the in-circuit tree address leaves identically. And the 64-bit range checks are not cosmetic: without them an adversary could choose amounts near the field modulus so that $v_0 + v_1$ wraps, manufacturing value out of modular arithmetic. Bounding amounts to 2^{64} keeps every sum honest in $\mathbb{Z}$ (Theorem 4).

10 — GUARANTEES

Security Analysis

What Zul guarantees, reduced to standard assumptions — and, with equal care, what it does not yet guarantee.

We state Zul's security as a set of properties, each reduced to a named assumption. The integrity properties (balance, double-spend resistance, shield soundness) hold against an adversary that may even be the sequencer; the privacy property holds against any external observer of the chain and network. We then analyze the trusted setup and close by listing, without softening, the properties that remain procedural.

10.1 Assumptions

- **(A1) Knowledge soundness of Groth16** [9]. For each circuit's CRS, any PPT prover that produces an accepting proof for public input x admits an efficient extractor yielding a witness w with $(x, w) \in \mathcal{R}$, except with negl probability; and Groth16 is statistically zero-knowledge.
- **(A2) Poseidon** [14] is collision- and preimage-resistant over $\mathbb{F}_p$.
- **(A3) blake3** [16] is collision-resistant.
- **(A4) BN254** [13] is a pairing-friendly curve on which discrete logarithms in $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are hard (we treat its security level as ≈ 100 bits and discuss the implication in §11).
- **(A5) Authenticated encryption.** XChaCha20-Poly1305 is IND-CCA secure and X25519 key agreement is secure under the Diffie–Hellman assumption on Curve25519 [29][30].

10.2 Definitions

Fix an asset. Let In be the total amount shielded into the pool and Out the total unshielded out, over the history so far. We say the pool is *solvent* if its vault holds at least $\mathsf{In} - \mathsf{Out}$ of the asset, and *balanced* if no sequence of operations makes the claimable shielded value exceed $\mathsf{In} - \mathsf{Out}$. *Double-spend resistance* requires that the value of any one note be removable from the pool at most once. *Unlinkability* requires that the public record of a transfer be simulatable without its witness, so it reveals nothing efficiently distinguishable about amounts, asset, owners, or the input↔output map.

10.3 Integrity

> **Theorem 4 (Balance / no inflation).** *Under (A1) and (A2), for every asset the total claimable shielded value never exceeds* $\mathsf{In} - \mathsf{Out}$*; consequently the pool stays solvent and no vault can be over-drawn.*

Proof (sketch). Consider the per-asset total of live note values. A *shield* of public amount v moves exactly v into the vault and adds a note of value v (Theorem 6), increasing both sides equally. A *transfer* produces, by knowledge soundness (A1), a witness satisfying the R1CS, hence $v_0 + v_1 = v_0' + v_1' + \mathsf{fee}$; because all amounts are constrained to $[0, 2^{64})$, this identity holds over $\mathbb{Z}$ and cannot be forged by field wraparound, so a transfer preserves the per-asset total (the fee is paid from it to the collector,

not minted). An *unshield* releases exactly its public amount from the vault and reduces shielded value by the same, since $v = \mathsf{amount} + v_c$. Commitments bind their openings by (A2), so no operation can later reinterpret a note's value. By induction over the operation sequence, claimable shielded value equals $\mathsf{In} - \mathsf{Out}$ at all times, never more; the vault, which receives every shield and funds every unshield, therefore never under-draws. ∎

> **Theorem 5 (Double-spend resistance).** *Under (A1) and (A2), the value of a note can be removed from the pool at most once.*

Proof (sketch). Every spend reveals $\mathsf{nf} = \mathcal{H}_3(\mathsf{cm}, \iota, s)$, and the circuit binds this public value to the one derived from the spent note (A1). The nullifier is a deterministic function of the note and its key, so re-spending the same note at its position reproduces the same nf; the node's persistent nullifier set rejects any repeat. To spend the same note under a *different* nullifier would require a second preimage of $\mathcal{H}_3$ with the bound commitment and key, contradicting (A2). Within a single proof, the explicit constraint $\mathsf{nf}_0 \neq \mathsf{nf}_1$ blocks consuming one note twice. ∎

> **Theorem 6 (Shield soundness).** *Under (A2), a shield of public* $(\mathsf{asset\_id}, v)$ *with revealed* secret *creates a note that opens only to value* v *in* $\mathsf{asset\_id}$*, without revealing the owner.*

Proof (sketch). The node computes $\mathsf{cm} = \mathcal{H}_3(\mathsf{asset\_id}, v, \mathsf{secret})$ from the public asset and amount and the revealed secret, and locks exactly v in the vault. By collision resistance (A2) no $(\mathsf{asset\_id}', v') \neq (\mathsf{asset\_id}, v)$ yields the same cm except with negl probability, so any later spend — which re-derives cm from $(\mathsf{asset\_id}, v)$ inside the circuit — is forced to the same value. Meanwhile $\mathsf{secret} = \mathcal{H}_2(\mathsf{pk}, \mathsf{blind})$ hides pk and blind by preimage resistance, so revealing secret at shield time binds the amount without exposing the owner. This is the property a flat commitment cannot provide. ∎

10.4 Privacy

> **Theorem 7 (Unlinkability).** *Under (A1), (A2), and (A5), the public record of a transfer is simulatable from its public inputs alone; an external observer cannot efficiently determine the amounts, the asset, the owners, or the map from input to output notes beyond the anonymity set.*

Proof (sketch). The public record is the proof, the root, two nullifiers, two commitments, the fee, and two note ciphertexts. The proof is zero-knowledge (A1): a simulator reproduces it from the public inputs without the witness, so it leaks nothing about amounts, asset, keys, or Merkle paths. Output commitments are hiding (a fresh blinding inside Poseidon, A2), hence pseudorandom and independent of the amounts and owners. Nullifiers are pseudorandom functions of the spending key (A2) and are unlinkable to their commitments without that key. The ciphertexts are authenticated-encryption outputs (A5), indistinguishable from random to anyone without the recipient's viewing key. Thus the entire record is simulatable, and distinguishing real from simulated reduces to breaking one of (A1), (A2), (A5). Privacy is therefore *relative to the anonymity set* — the set of notes a spend could plausibly have consumed — which grows with pool usage. ∎

HONEST CAVEAT — THE FEE PAYER

In v1, the account that pays a private transfer's transaction fee is public, which can correlate an otherwise-unlinkable transfer with an identity. This is a metadata leak *around* the proof, not a break of it, and it is removed by the planned relayer with an in-circuit fee output (§13). We list it here rather than leave it for a reader to discover.

10.5 The trusted setup

Groth16 requires a per-circuit common reference string whose toxic waste, if retained, would let its holder forge proofs — though never break privacy or steal funds outside the pool's own conservation. Zul's CRS is built in the standard two phases [19][20]: a universal Powers-of-Tau phase 1 (the Hermez perpetual ceremony transcript) followed by a per-circuit phase 2, finalized with a public beacon so the result is non-malleable. The mainnet ceremony combined the Hermez phase-1 parameters, a local secret contribution, and a Bitcoin block-hash beacon; the resulting `zkey` verifies, the node cross-checks against it, and the two verifying keys are committed in the repository.

HONEST CAVEAT — 1-OF-1 PHASE 2

Groth16's phase-2 is secure if *at least one* contributor is honest and destroys their entropy. Zul's mainnet phase-2 was a single local contribution: the guarantee is therefore "this machine was honest and its randomness was destroyed," plus the beacon. That is a real, named trust assumption. The ceremony tooling supports a multi-party run, and a high-value deployment should re-run phase-2 with independent contributors and redeploy the verifying keys; doing so strictly strengthens (A1) for this system and requires no protocol change.

10.6 What is not yet enforced

For completeness, the procedural items from Table 2, restated plainly: **(i)** execution integrity — an invalid state root is publicly detectable by replaying the data-availability stream but is not yet rejected on-chain; **(ii)** liveness and ordering — a single sequencer can stall, censor, or reorder; **(iii)** fee-payer linkage, above. None of these let an adversary inflate value, double-spend, unmask a transfer, or drain a vault — those are the enforced guarantees of §§10.3–10.4. They are the agenda of §13.

11 — REALITY CHECK

Implementation & Evaluation

A deployed system, the tests that hold its pieces together, and an honest account of its performance and its limits.

Zul is not a specification awaiting implementation; it is an implementation this document describes. The node is a Rust workspace driving Anza's `solana-svm`; the L1 programs are Anchor; the circuits are circom with a snarkjs proving path; the client stack is a TypeScript SDK, a Next.js explorer, and an indexer. This section reports what exists, how it is verified, and how it performs.

11.1 Components and verification

Each component is covered by unit and integration tests; the cross-language bindings — the seams most likely to drift silently — are pinned by explicit cross-tests.

Table 3. Workspace components and how each is verified. The chain workspace alone carries well over one hundred Rust tests.

Component	Role	Verified by
`zul-primitives`	hashing, blocks, genesis, constants	field/hash/Merkle vectors
`zul-store`	redb account store, SMT, nullifiers	SMT proofs, atomic commit, order-independence
`zul-executor`	solana-svm integration	real SVM transfers; SPL mint runs in-VM
`zul-privacy`	shielded pool, Groth16, Poseidon	proof verify, pool service, snarkjs cross-test
`zul-rpc`	Solana JSON-RPC subset	method-level tests against fixtures
`zul-bridge`	deposit watcher, batcher, DA	batcher + deposit flow (L1 client mockable)
`zul-node`	block loop, gas, e2e	block production, shield e2e, restart safety
programs/ (Anchor)	settlement, bridge, da_log	BPF build + host tests; SMT cross-pinned to L2
circuits/	transfer, unshield (circom)	compile + snarkjs↔arkworks proof cross-test
sdk/ & indexer/	TS client, Postgres indexer	bincode cross-test with Rust; idempotent slot-walk

11.2 Cross-language soundness

Three bindings could break consensus if any side drifted, so each is asserted directly:

- a snarkjs-generated Groth16 proof must verify in the arkworks node verifier, so prover and verifier cannot diverge;
- the TypeScript SDK's bincode encoding must equal Rust's serde output byte-for-byte, so the client and node agree on every wire structure;
- the L1 (Anchor) blake3 SMT must match the L2 store SMT, pinned by shared constants such as the empty-tree root, so the program verifies exactly the proofs the node produces.

11.3 Performance

The shielded design is deliberately sized for client-side proving. A 2-in/2-out join-split over Poseidon is a small circuit, so a browser generates a transfer proof with snarkjs in a few seconds on commodity hardware — fast enough that a wallet need not run a prover sidecar. Verification is the opposite extreme: Groth16's check is a constant three-pairing equation independent of circuit size, completing on the order of a millisecond in native arkworks code. Because the pool is an enshrined builtin (§4.2), that verification pays *no* compute-unit cost — the decisive advantage of native verification over running a pairing inside metered BPF, where a single shielded transaction could exhaust a block's budget. The state SMT updates touch only the accounts a block actually wrote, so commitment cost scales with activity, not with the size of the account set.

11.4 Mainnet status

Zul is live on Solana mainnet-beta. The settlement and data-availability programs are deployed and initialized; the batcher posts non-empty batches on a periodic sweep; and the end-to-end shielded flow has been exercised on mainnet with the production ceremony keys: a shield, a real Groth16 private transfer, a double-spend attempt that was *rejected*, and an unshield that paid the recipient. Native gas was seeded by bridging the gas mint one-for-one, and a native L2 transfer confirmed. The exact deployment identifiers are listed in Appendix D. In the interest of accuracy: the outbound-withdraw and arbitrary-SPL-deposit paths are the same bytecode proven on devnet and testnet and were not separately re-exercised on mainnet at launch.

11.5 A note on the curve

BN254 is the pragmatic choice today: it is the curve circom and snarkjs target and the curve Solana's `alt_bn128` syscalls implement, which is what makes both browser proving and a future on-chain verifier path realistic. Its security level is now estimated near 100 bits rather than the 128 once assumed [13]. For the present value at stake this is adequate, and the integrity guarantees of §10 degrade gracefully — an attacker would need to break discrete log on BN254, not merely observe the chain. A migration to a higher-security pairing-friendly curve, or to a transparent proof system for the programmable layer, is a clean future change because verification is isolated behind the verifying-key interface (§9.3).

12 — THE NORTH STAR

Privacy-Wrapping the SVM

The enshrined pool is one private program. The goal is to make privacy a property any program can opt into — and, eventually, a property of the SVM itself.

Everything to this point describes a working privacy *application*: a shielded pool, generalized to multiple assets, verified natively, settled to Solana. It is good, and it is not the destination. The destination is to make privacy a property a program *opts into*, the way it opts into any other — so that a sealed-bid auction, a secret-ballot vote, a hidden-information game, or a non-leaking order book is just a program, running where no one can see the inputs and everyone can verify the result. Zul approaches this on a ladder, not in a leap (Figure 8).

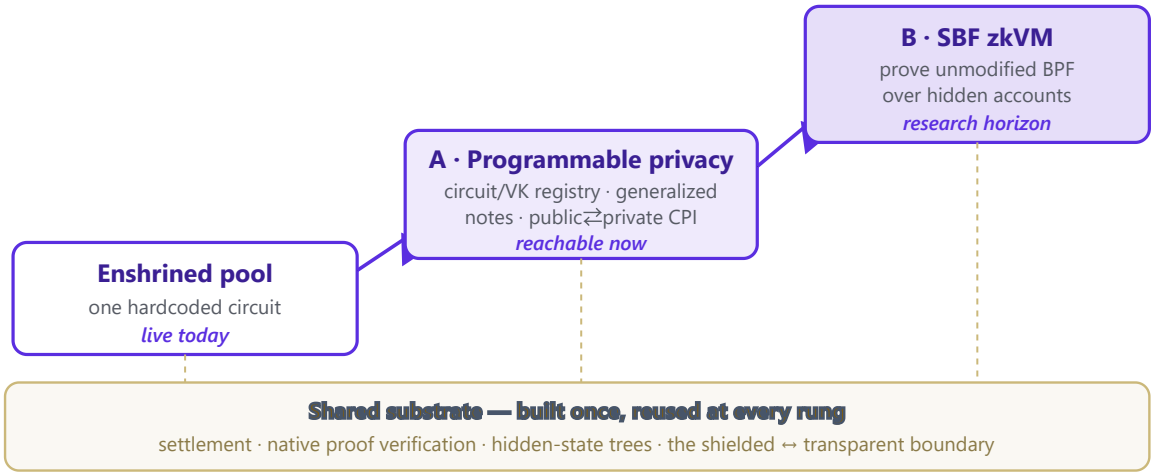

Figure 8. The staged path. Today's enshrined pool is one hardcoded private program. Rung A generalizes it into a programmable privacy layer; rung B proves unmodified BPF execution over hidden state. Both stand on the same substrate this paper builds — settlement, native verification, hidden-state trees, and the shielded↔transparent boundary — so the near rung is genuinely the ladder to the far one.

12.1 Rung A — a programmable privacy layer

The shielded pool is a single circuit hardwired into the node. Rung A removes the "single" and the "hardwired." It generalizes the pool's fixed machinery into four open mechanisms:

- **A circuit / verifying-key registry.** A developer registers a circuit describing a private state transition; the chain stores its verifying key and gives the program a private counterpart. The node already verifies arbitrary Groth16 natively, so the work is opening that door to many circuits rather than one built-in.
- **A generic private-execution engine.** The pool's "verify proof → update tree → record nullifier" loop becomes a general one: verify the registered circuit's proof against its declared public inputs, then apply the declared effects to that program's hidden state.

— **Generalized notes.** Today a note commits to *value*: $(\mathsf{asset\_id}, v, \mathsf{pk}, \mathsf{blind})$. Generalized, a note commits to *arbitrary application state* — an auction bid, a vote, a position — under the same commitment/nullifier discipline, so every dApp gets a private UTXO space of its own shape.

— **A public⇄private CPI boundary.** Today's *unshield* is exactly the special case of a private transition that emits a public effect: it spends a hidden note and triggers a transparent vault payment. Promoted to a first-class boundary, a private program can call a public one (and be called back) — which is what makes private programs compose with the existing SVM instead of living in a sealed annex.

The result is an Aztec/Aleo-style model [24][25] realized on an SVM L2: any dApp that ships a circuit gets a private counterpart, and coverage grows program by program until "most of the SVM can be private" stops being a slogan and becomes a count.

12.2 Rung B — a zkVM for the SBF

Rung A still asks a developer to write a circuit. Rung B removes even that: take an *unmodified* BPF/SBF program and prove its execution over hidden accounts directly, with no bespoke circuit — just the program. This is the SVM analogue of a general-purpose zkVM [12], and it is hard: it is years of work and active research, and we name it as a destination rather than a dated promise. Its value is that it collapses the gap between "a Solana program" and "a private Solana program" to nothing.

12.3 Why A is the ladder, not a detour

The reason to build A first is not merely that it is easier; it is that A and B share their entire foundation. The settlement to Solana, the native proof verification, the hidden-state trees, the nullifier discipline, and above all the seam between the private and public worlds are the same whether the thing being proven is one registered circuit or a whole VM execution. We therefore build the reachable rung in the shape of the unreachable one, so that climbing is never starting over. Each component this paper describes — the SMT, the enshrined verifier, the commitment/nullifier machinery, the bridge boundary — is a load-bearing part of both rungs.

12.4 What this changes for developers

Under rung A, deploying becomes a Zul-native lifecycle rather than a bare program upload: deploy the BPF program, register its circuit and verifying key, and initialize its private-state tree, as one coherent operation. This justifies a first-class `zul` toolchain that owns that lifecycle, much as today's SDK owns the shield/transfer/unshield flow. The through-line from this paper to that future is direct: the enshrined pool is the first program of the programmable layer, and its SDK is the first draft of that toolchain.

THE WAGER

Two serious teams attacked Solana privacy and both narrowed away from the maximalist goal [27][28]. Zul's bet is that the way to actually reach “most of the SVM is private” is to ship the reachable rung first and generalize it, so every step stands on a working artifact and each makes the next one safer. A vision with no running system under it is decoration; this one has a mainnet under it.

13 — TOWARD LESS TRUST

Trust-Minimization Roadmap

Every procedural guarantee in this paper is a place we intend to need less trust, not more. Here is how each becomes enforced.

Section 10.6 named what is not yet enforced. This section turns that list into a direction. The ordering principle is the same one that governs the north star: ship the reachable improvement, and build it in the shape of the eventual one so the work compounds.

13.1 Decentralizing the sequencer

A single sequencer can censor, reorder, or stall. The path is a sequencer set with rotation, and — independent of who sequences — a *forced-inclusion* path on L1 so that a censored user can compel their transaction's inclusion through the settlement contract. Forced inclusion is the property that matters most for users; it can precede full sequencer decentralization.

13.2 From procedural to enforced settlement

Today an invalid state root is detectable from public data but not rejected on-chain. The near step is an enforced fraud proof: a challenger re-executes a disputed batch from the data-availability stream and submits a succinct disagreement the settlement program can adjudicate. The far step is validity — a proof of correct execution posted with each batch. Rung B of §12 is exactly this endgame: an SBF zkVM that can prove execution is simultaneously the tool that makes settlement trustless and the tool that makes execution private. Settlement and privacy converge on the same artifact.

13.3 Private fees

The v1 public fee payer links an otherwise-unlinkable transfer to an identity (§10.4). The fix is a relayer that submits shielded transactions on the user's behalf, paid by an *in-circuit fee output* so the fee is part of what the proof conserves rather than a separate public payment. The transfer circuit already carries a public fee signal; the relayer is the missing client-and-incentive layer around it.

13.4 A multi-party ceremony

The mainnet phase-2 setup was a single honest contribution (§10.5). For higher value, the same tooling runs a multi-party ceremony: as long as one contributor is honest and discards their entropy, the CRS is sound. This strengthens assumption (A1) with no protocol change — only a re-run and a redeploy of the verifying keys.

13.5 Key custody and the encrypted mempool

Upgrade authority over the L1 programs should move from a hot key to a multisig with minimized powers, so no single key can alter settlement or bridge logic. Separately, an encrypted mempool

would deny even the sequencer a view of transaction contents before ordering, closing the residual ordering-based metadata channel.

Table 4. Trust-minimization targets. Each row moves a property from procedural (P) toward enforced (E).

Property	Today	Target	Mechanism
Censorship resistance	P	E	Forced inclusion on L1
Execution integrity	P	E	Fraud proofs → validity proofs (rung B)
Sequencing	single	set	Rotating sequencer set
Fee-payer privacy	P	E	Relayer + in-circuit fee output
Trusted setup	1-of-1	n-of-1	Multi-party phase-2 ceremony
Upgrade custody	hot key	multisig	Threshold authority, minimized powers
Ordering metadata	visible	hidden	Encrypted mempool

None of these are speculative primitives; each is a known technique with a clear integration point in the system this paper describes. The work is to enforce, one property at a time, what is today disclosed — and to do it without ever quietly claiming a guarantee before it holds.

14 — CLOSE

Conclusion

Public ledgers proved you can have trust without authority. The thing left to prove is that you can have privacy without surrendering that trust.

The first generation of public blockchains established a genuinely new fact about the world: that strangers can agree on a shared record without an authority to enforce it. The cost they charged for that fact — total transparency — was never actually part of the result. It was a design choice that made verification easy by making everything visible, and it quietly turned the most important financial infrastructure of a generation into a permanent surveillance archive.

Zul is an argument, in running code, that the cost was optional. It takes the most expressive mainstream smart-contract runtime there is, runs it faithfully as a Layer 2, commits its state succinctly, settles to Solana, and adds a shielded pool in which value moves with its amounts and its graph hidden behind proofs rather than merely left unindexed. The integrity of that pool — that no value is conjured, no note spent twice, no transfer unmasked, no vault drained — rests on cryptography, not on trusting us; and where we still must be trusted, for ordering and execution integrity, we have said so plainly and pointed at how that trust is removed.

We hold to five things. Privacy is a default to defend, not a feature to upsell. Hidden inputs and public correctness are not a trade — you get both, or it does not ship. Anything we cannot yet enforce in cryptography, we say out loud. Privacy must compose, so the boundary between the shielded and the transparent is a first-class part of the design, not an afterthought. And every step ships something real: a vision with no working artifact beneath it is decoration.

The enshrined pool is the first private program of a system meant to make privacy a property the whole SVM can have. The settlement, the verification, the hidden state, and the seam between the private and public worlds are built today in the shape of what they must become tomorrow, so that the climb from one private program to a private virtual machine is never a fresh start. A whole execution environment where you are open about results and private about contents, and never asked to give up one for the other — that the two were never opposed is the thing left to prove.

That is what Zul is for.

BIBLIOGRAPHY

References

[1] D. Chaum. ***Blind signatures for untraceable payments.*** CRYPTO, 1982.

[2] E. Hughes. ***A Cypherpunk's Manifesto.*** 1993.

[3] S. Nakamoto. ***Bitcoin: A Peer-to-Peer Electronic Cash System.*** 2008.

[4] V. Buterin. ***Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.*** 2014.

[5] A. Yakovenko. ***Solana: A new architecture for a high-performance blockchain.*** Whitepaper, 2018.

[6] Anza. ***Agave validator and the `solana-svm` (SVM API) crate.*** github.com/anza-xyz/agave.

[7] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, M. Virza. ***Zerocash: Decentralized Anonymous Payments from Bitcoin.*** IEEE S&P, 2014.

[8] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox. ***Zcash Protocol Specification (Sapling).*** Electric Coin Co., 2016–.

[9] J. Groth. ***On the Size of Pairing-Based Non-interactive Arguments.*** EUROCRYPT, 2016.

[10] B. Parno, C. Gentry, J. Howell, M. Raykova. ***Pinocchio: Nearly Practical Verifiable Computation.*** IEEE S&P, 2013.

[11] A. Gabizon, Z. J. Williamson, O. Ciobotaru. ***PLONK: Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge.*** IACR ePrint 2019/953.

[12] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev. ***Scalable, transparent, and post-quantum secure computational integrity (STARKs).*** IACR ePrint 2018/046.

[13] P. S. L. M. Barreto, M. Naehrig. ***Pairing-Friendly Elliptic Curves of Prime Order.*** SAC, 2005.

[14] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger. ***Poseidon: A New Hash Function for Zero-Knowledge Proof Systems.*** USENIX Security, 2021.

[15] M. Albrecht, L. Grassi, C. Rechberger, A. Roy, T. Tiessen. ***MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity.*** ASIACRYPT, 2016.

[16] J. O'Connor, J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn. ***BLAKE3: One Function, Fast Everywhere.*** 2020.

[17] A. Pertsev, R. Semenov, R. Storm. ***Tornado Cash: Privacy-preserving transactions on Ethereum.*** 2019.

[18] Tornado Cash. ***Tornado Cash Nova: arbitrary-amount shielded transfers.*** 2021.

[19] S. Bowe, A. Gabizon, I. Miers. ***Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model.*** IACR ePrint 2017/1050.

[20] S. Bowe, A. Gabizon, M. Green. ***A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK.*** 2018.

[21] iden3. ***circom 2 and snarkjs.*** github.com/iden3.

[22] iden3. ***circomlib: circuit templates (Poseidon, comparators, bitify).*** github.com/iden3/circomlib.

[23] arkworks contributors. ***arkworks: a Rust ecosystem for zkSNARK programming (`ark-groth16`, `ark-bn254`).*** arkworks.rs.

[24] Z. J. Williamson et al. ***The Aztec Protocol.*** 2019–.

[25] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, H. Wu. ***Zexe: Enabling Decentralized Private Computation.*** IEEE S&P, 2020.

[26] Penumbra Labs. ***Penumbra: a shielded, cross-chain decentralized exchange.*** 2021–.

[27] Light Protocol. ***ZK compression and private state on Solana.*** lightprotocol.com.

[28] Elusiv / Arcium. ***Encrypted computation on Solana.*** arcium.com.

[29] D. J. Bernstein. ***Curve25519: New Diffie-Hellman Speed Records.*** PKC, 2006.

[30] Y. Nir, A. Langley. ***ChaCha20 and Poly1305 for IETF Protocols.*** RFC 8439, 2018 (XChaCha20 extended-nonce variant, CFRG).

[31] R. C. Merkle. ***A Digital Signature Based on a Conventional Encryption Function.*** CRYPTO, 1987.

[32] R. Dahlberg, T. Pulls, R. Peeters. ***Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs.*** NordSec, 2016.

[33] Solana Labs. ***SPL Token-2022: Confidential Transfer extension.*** spl.solana.com.

[34] A. Fiat, A. Shamir. ***How to Prove Yourself: Practical Solutions to Identification and Signature Problems.*** CRYPTO, 1986.

[35] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, E. W. Felten. ***Arbitrum: Scalable, Private Smart Contracts.*** USENIX Security, 2018.

[36] M. Al-Bassam, A. Sonnino, V. Buterin. ***Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities.*** 2018.

APPENDICES

Parameters & Interfaces

The exact constants, domains, interfaces, and deployment identifiers referenced throughout.

A Protocol parameters

Parameter	Value	§
Pairing curve	`BN254 (alt_bn128)`	8.1
Proof system	`Groth16 — proof = (A,B,C), 3 group elements`	9.3
In-circuit hash	`Poseidon (circom params), arities 1·2·3`	8.1
State / settlement hash	`blake3, 256-bit, domain-separated`	5.1
State SMT depth	`256`	5.1
Withdrawals SMT depth	`32`	6.3
Commitment tree depth	`26 (capacity 2²⁶ ≈ 6.7 × 10⁷ notes)`	8.4
Root history ring	`64 most-recent roots`	8.4
Recent-blockhash queue	`150`	6.1
Amount range (in-circuit)	`[0, 2⁶⁴)`	9.1
Transfer public inputs	`6 — [root, nf₀, nf₁, cm₀, cm₁, fee]`	9.1
Unshield public inputs	`6 — [root, nf, cm_chg, asset_id, amount, recipient]`	9.2
DA chunk size / compression	`≈ 800 bytes per chunk · zstd`	6.2
Block cadence	`slot = height · ~10 s idle heartbeat`	6.1
Gas token	`ZUL · 9 decimals · lamport unit of L2`	4.5

B Domain-separation tags

Every hash is prefixed by a constant byte-string so that values from different roles can never collide. The tags retain a historical `PL2:` prefix that is consensus-pinned across the node, the L1 program, and the circuit test vectors; it is an internal identifier, not the protocol name.

Tag	Role			
`PL2:smt:key`	account	→	leaf	path
`PL2:smt:leaf`	state-tree		leaf	node
`PL2:smt:node`	state-tree		internal	node

`PL2:smt:empty`	empty-subtree base
`PL2:account:v1`	account hash
`PL2:block:header:v1`	block-header hash
`PL2:da:batch:v1`	data-availability batch hash
`PL2:bridge:deposit-receipt:v1`	exactly-once deposit receipt
`PL2:bridge:withdraw:v1`	withdrawal leaf
`PL2:bridge:withdraw-addr:v1`	withdrawal address derivation
`PL2:bridge:wrapped-mint:v1`	deterministic wrapped-mint address

C RPC surface

The node answers a subset of the Solana JSON-RPC, so standard clients work unmodified, plus one Zul-specific method.

Reads	`getHealth` · `getVersion` · `getSlot` · `getBlockHeight` · `getEpochInfo` · `getGenesisHash` · `getLatestBlockhash` · `isBlockhashValid` · `getMinimumBalanceForRentExemption` · `getFeeForMessage`
Accounts	`getBalance` · `getAccountInfo` · `getMultipleAccounts` · `getTokenAccountsByOwner` · `getProgramAccounts`
History	`getSignaturesForAddress` · `getSignatureStatuses` · `getTransaction` · `getBlock`
Submit	`sendTransaction` · `simulateTransaction` · `requestAirdrop` (faucet, non-mainnet)
Subscriptions	`slotSubscribe` · `signatureSubscribe` · `accountSubscribe`
Zul-specific	`getWithdrawalProof(recipient, amount, nonce, asset?)` → state root + SMT inclusion proof

D Mainnet deployment

Public, on-chain-verifiable identifiers of the live mainnet-beta deployment.

Chain	id	`zul-mainnet-1`
Genesis	hash	`FeVXkxhHChVk7HXKxfJuDSLMts45e5wkpF2nQ1c6XJ6D`
Native	pre-mine	`zero — all native ZUL is bridge-backed`
Settlement	program	`Po6oySWHr9oxFWVhZJ1Jca2HkvfzXxsK91n3R6mqZUL`
Data-availability	log	`by9fpRVSYN9ib7qwXJuak37d78cKUMFgNAWtjK3pZUL`
Gas mint	(ZUL)	`okTCeDRb7NeExmvf3RZxEs29i9nfw8Cd51oFVB4FZUL`
Gas-mint	properties	`Token-2022 · 9 dec · mint authority revoked`

Pool program / state / vault	`[05 21 00…00] / …01 / …02 (keyless)`

E Circuit public signals

Public signals are declared as circuit inputs in a fixed order so the prover and the native verifier agree on the vector exactly.

Circuit	Public (in order)	Private
Transfer 2-in / 2-out	`root, nullifier[0], nullifier[1], out_commitment[0], out_commitment[1], fee`	`asset_id, in_amount[2], in_s[2], in_blinding[2], in_leaf_index[2], in_path[2][26], out_amount[2], out_pk[2], out_blinding[2]`
Unshield 1-in → public + change	`root, nullifier, change_commitment, asset_id, amount, recipient`	`in_amount, in_s, in_blinding, in_leaf_index, in_path[26], change_amount, change_pk, change_blinding`